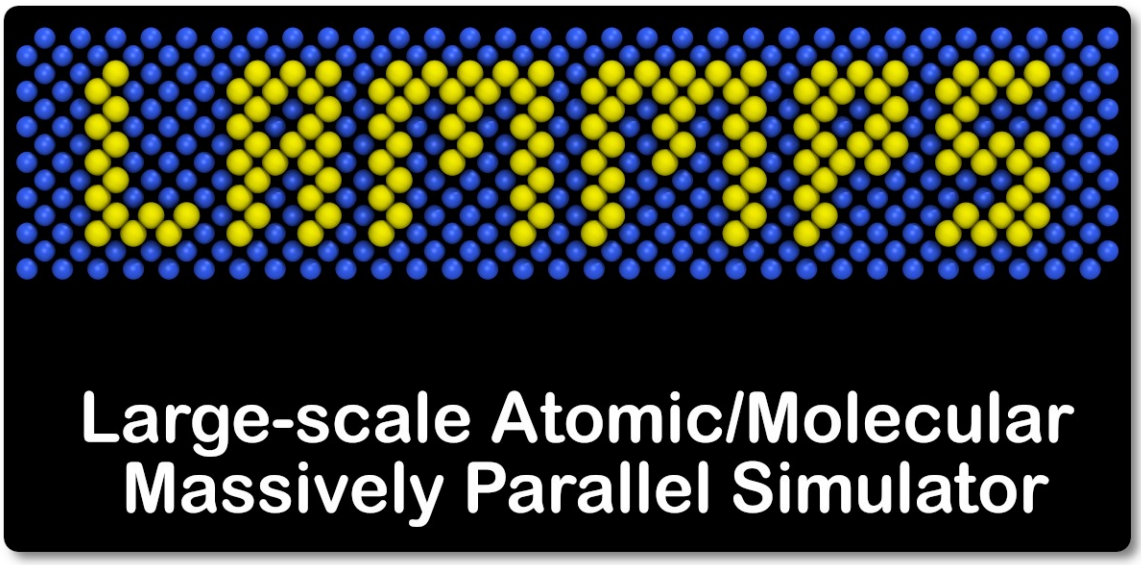



# LAMMPS Documentation

**Release 4 Jul 2026**

**The LAMMPS Developers***

**developers@lammps.org**

* see https://www.lammps.org/authors.html for details

# LAMMPS DOCUMENTATION


About LAMMPS and this manual 1

I User Guide 5

1 Introduction 7
1.1 Overview of LAMMPS . . . 7
1.2 What does a LAMMPS version mean . . . 8
1.2.1 Identifying the Version . . . 8
1.2.2 LAMMPS releases, branches, and tags . . . 8
1.3 LAMMPS features . . . 9
1.3.1 General features . . . 9
1.3.2 Particle and model types . . . 10
1.3.3 Interatomic potentials (force fields) . . . 10
1.3.4 Atom creation . . . 11
1.3.5 Ensembles, constraints, and boundary conditions . . . 12
1.3.6 Integrators . . . 12
1.3.7 Diagnostics . . . 12
1.3.8 Output . . . 12
1.3.9 Multi-replica models . . . 13
1.3.10 Pre- and post-processing . . . 13
1.3.11 Specialized features . . . 13
1.4 LAMMPS non-features . . . 14
1.5 LAMMPS portability and compatibility . . . 15
1.5.1 Programming language standards . . . 15
1.5.2 Build systems . . . 15
1.5.3 Operating systems . . . 16
1.5.4 Compilers . . . 16
1.5.5 CPU architectures . . . 16
1.5.6 Portability compliance . . . 16
1.5.7 Code review and static code analysis . . . 16
1.6 LAMMPS open-source license . . . 17
1.6.1 GPL version of LAMMPS . . . 17
1.6.2 LGPL version of LAMMPS . . . 17
1.7 Authors of LAMMPS . . . 18
1.8 Citing LAMMPS . . . 19
1.8.1 Core Algorithms . . . 19
1.8.2 DOI for the LAMMPS source code . . . 19
1.8.3 Home page . . . 20
1.8.4 Citing contributions . . . 20
1.9 Additional website links . . . 20

**2 Install LAMMPS** **21**
2.1 Download an executable for Linux . . . 21
2.1.1 Pre-built static Linux x86_64 executables . . . 22
2.1.2 Pre-built Ubuntu and Debian Linux executables . . . 22
2.1.3 Pre-built Fedora Linux executables . . . 23
2.1.4 Pre-built EPEL Linux executable . . . 23
2.1.5 Pre-built OpenSuse Linux executable . . . 23
2.1.6 Gentoo Linux executable . . . 24
2.1.7 Archlinux build-script . . . 24
2.2 Download an executable for macOS . . . 25
2.3 Download an executable for Windows . . . 25
2.4 Download an executable for Linux or macOS via Conda . . . 26
2.5 Download source and documentation as a tarball . . . 26
2.6 Download the LAMMPS source with git . . . 27

**3 Build LAMMPS** **31**
3.1 Prerequisites . . . 31
3.1.1 CMake build system . . . 31
3.1.2 GNU build system . . . 31
3.1.3 Compiler and OS compatibility . . . 32
3.2 Build LAMMPS with CMake . . . 32
3.2.1 Advantages of using CMake . . . 32
3.2.2 Getting started . . . 33
3.2.3 Configuration and build options . . . 35
3.2.4 Build targets . . . 35
3.2.5 Multi-configuration build systems . . . 35
3.2.6 Installing CMake . . . 36
3.3 Build LAMMPS with make . . . 36
3.3.1 Make to CMake quick-start guide . . . 37
3.3.2 Basics . . . 37
3.3.3 Requirements . . . 38
3.3.4 Getting started . . . 38
3.3.5 Customized builds and alternate makefiles . . . 39
3.4 Link LAMMPS as a library to another code . . . 39
3.4.1 Link with LAMMPS as a static library . . . 40
3.4.2 Link with LAMMPS as a shared library . . . 41
3.5 Basic build options . . . 42
3.5.1 Serial vs parallel build . . . 42
3.5.2 Choice of compiler and compile/link options . . . 44
3.5.3 Build the LAMMPS executable and library . . . 46
3.5.4 Including or removing debug support . . . 48
3.5.5 Build LAMMPS tools . . . 48
3.5.6 Install LAMMPS after a build . . . 49
3.6 Optional build settings . . . 50
3.6.1 C++17 standard compliance . . . 50
3.6.2 FFT library . . . 50
3.6.3 Size of LAMMPS integer types and size limits . . . 54
3.6.4 Read or write compressed files . . . 55
3.6.5 Support for downloading files from the input . . . 55
3.6.6 Prevent download of large potential files . . . 56
3.6.7 Memory allocation alignment . . . 56
3.6.8 Workaround for long long integers . . . 57
3.6.9 Exception handling when using LAMMPS as a library . . . 57
3.7 Include packages in build . . . 57

3.7.1 Information for both build systems . . . 59
3.7.2 CMake presets for installing many packages . . . 59
3.7.3 Make shortcuts for installing many packages . . . 61
3.8 Packages with extra build options . . . 61
3.8.1 COMPRESS package . . . 62
3.8.2 GRAPHICS package . . . 62
3.8.3 GPU package . . . 64
3.8.4 KIM package . . . 67
3.8.5 KOKKOS package . . . 68
3.8.6 LEPTON package . . . 73
3.8.7 MACHDYN package . . . 73
3.8.8 ML-IAP package . . . 73
3.8.9 OPT package . . . 74
3.8.10 PYTHON package . . . 74
3.8.11 VORONOI package . . . 75
3.8.12 ADIOS package . . . 75
3.8.13 APIP package . . . 76
3.8.14 COLVARS package . . . 76
3.8.15 ELECTRODE package . . . 77
3.8.16 MBX package . . . 77
3.8.17 ML-PACE package . . . 78
3.8.18 ML-POD package . . . 79
3.8.19 ML-QUIP package . . . 79
3.8.20 PLUMED package . . . 80
3.8.21 H5MD package . . . 81
3.8.22 ML-HDNNP package . . . 81
3.8.23 ML-RUNNER package . . . 82
3.8.24 INTEL package . . . 83
3.8.25 MDI package . . . 84
3.8.26 MISC package . . . 84
3.8.27 MOLFILE package . . . 85
3.8.28 NETCDF package . . . 85
3.8.29 OPENMP package . . . 86
3.8.30 QMMM package . . . 86
3.8.31 RHEO package . . . 87
3.8.32 SCAFACOS package . . . 88
3.8.33 VTK package . . . 88
3.9 Build the LAMMPS documentation . . . 89
3.9.1 Build using GNU make . . . 89
3.9.2 Build using CMake . . . 90
3.9.3 Prerequisites for HTML . . . 90
3.9.4 Prerequisites for PDF . . . 91
3.9.5 Prerequisites for ePUB and MOBI . . . 91
3.9.6 Instructions for Developers . . . 91
3.10 Notes for building LAMMPS on Windows . . . 93
3.10.1 General remarks . . . 93
3.10.2 Running Linux on Windows . . . 93
3.10.3 Using a GNU GCC ported to Windows . . . 94
3.10.4 Using Microsoft Visual Studio . . . 94
3.10.5 Using Intel oneAPI Compilers and Libraries . . . 95
3.10.6 Using a cross-compiler . . . 95
3.11 Notes for saving disk space when building LAMMPS from source . . . 95
3.12 Development build options . . . 96
3.12.1 Monitor compilation flags (CMake only) . . . 97

3.12.2 Report missing and unneeded '#include' statements (CMake only) . . . 97
3.12.3 Address, Leak, Undefined Behavior, and Thread Sanitizer Support (CMake only) . . . 97
3.12.4 Code Coverage and Unit Testing (CMake only) . . . 98
3.12.5 Coding style utilities . . . 104
3.12.6 Clang-format support . . . 106
3.12.7 GitHub command-line interface . . . 106

**4 Run LAMMPS** **109**
4.1 Basics of running LAMMPS . . . 109
4.2 Command-line options . . . 110
4.3 Screen and logfile output . . . 118
4.4 Error message output . . . 121
4.4.1 A single line . . . 121
4.4.2 Two lines . . . 121
4.4.3 Three lines . . . 121
4.4.4 Four lines . . . 121
4.5 File formats used by LAMMPS . . . 121
4.5.1 Character Encoding . . . 122
4.5.2 Number Formatting . . . 122
4.5.3 Input File . . . 123
4.5.4 Data File . . . 124
4.5.5 Molecule File . . . 126
4.5.6 Restart File . . . 127
4.5.7 JSON Dump Files . . . 127
4.6 Running LAMMPS on Windows . . . 128

**5 Errors** **131**
5.1 Common issues that are often regarded as bugs . . . 131
5.2 Errors and warnings details . . . 133
5.2.1 General troubleshooting advice . . . 134
5.2.2 Unknown identifier in data file . . . 137
5.2.3 Incorrect format in … section of data file . . . 138
5.2.4 Illegal variable command: expected X arguments but found Y . . . 138
5.2.5 Out of range atoms - cannot compute … . . . 138
5.2.6 Bond (or angle, dihedral, improper, cmap, or shake) atoms missing . . . 138
5.2.7 Non-numeric atom coords or pressure or box dimensions - simulation unstable . . . 139
5.2.8 Fix used in … not computed at compatible time . . . 139
5.2.9 Lost atoms … . . . 139
5.2.10 Too many neighbor bins . . . 139
5.2.11 Unrecognized … style … . . . 139
5.2.12 Energy or stress was not tallied by pair style . . . 140
5.2.13 fmt::format_error . . . 140
5.2.14 Substitution for illegal variable . . . 140
5.2.15 Bond atom missing in image check or box size check . . . 140
5.2.16 Cannot use neighbor bins - box size << cutoff . . . 141
5.2.17 Did not assign all atoms correctly . . . 141
5.2.18 Domain too large for neighbor bins . . . 141
5.2.19 Step X: (h)bondchk failed . . . 141
5.2.20 Numeric index X is out of bounds . . . 141
5.2.21 Compute, fix, or variable vector or array is accessed out-of-range . . . 142
5.2.22 Incorrect args for pair coefficients (also bond/angle/dihedral/improper coefficients) . . . 142
5.2.23 Energy was not tallied on needed timestep (also virial, per-atom energy, per-atom virial) 142
5.2.24 Molecule auto special bond generation overflow . . . 143
5.2.25 Molecule topology/atom exceeds system topology/atom . . . 143

5.2.26 Molecule topology type exceeds system topology type . . . 143
5.2.27 Molecule attributes do not match system attributes . . . 143
5.2.28 Inconsistent image flags . . . 143
5.2.29 No fixes with time integration, atoms won't move . . . 144
5.2.30 System is not charge neutral, net charge = … . . . 144
5.2.31 Variable evaluation before simulation box is defined . . . 144
5.2.32 Invalid thermo keyword 'X' in variable formula . . . 144
5.2.33 One or more atoms are time integrated more than once . . . 145
5.2.34 XXX command before simulation box is defined . . . 145
5.2.35 XXX command after simulation box is defined . . . 145
5.2.36 Error messages ending in 'Please contact the LAMMPS developers' . . . 145
5.2.37 Neighbor list overflow, boost neigh_modify one . . . 145
5.2.38 Variable …: Compute/Fix … does not compute requested property . . . 146
5.2.39 The … style … is no longer available . . . 146
5.3 Reporting bugs . . . 146
5.4 Debugging crashes . . . 147
5.4.1 Using the GDB debugger to get a stack trace . . . 148
5.4.2 Using valgrind to get a stack trace . . . 150
5.5 Debugging when LAMMPS appears to be stuck . . . 151
5.6 Error messages . . . 151
5.7 Warning messages . . . 220

**6 Commands** **231**
6.1 LAMMPS input scripts . . . 231
6.2 Parsing rules for input scripts . . . 232
6.3 Input script structure . . . 234
6.3.1 Initialization . . . 234
6.3.2 System definition . . . 235
6.3.3 Simulation settings . . . 235
6.3.4 Run a simulation . . . 235
6.4 Commands by category . . . 236
6.4.1 Initialization . . . 236
6.4.2 Setup simulation box . . . 236
6.4.3 Setup atoms . . . 236
6.4.4 Force fields . . . 236
6.4.5 Settings . . . 236
6.4.6 Operations within timestepping (fixes) and diagnostics (computes) . . . 236
6.4.7 Output . . . 237
6.4.8 Actions . . . 237
6.4.9 Input script control . . . 237
6.5 General commands . . . 238
6.6 Fix styles . . . 238
6.7 Compute styles . . . 240
6.8 Pair styles . . . 241
6.9 Bond styles . . . 243
6.10 Angle styles . . . 243
6.11 Dihedral styles . . . 244
6.12 Improper styles . . . 244
6.13 KSpace styles . . . 244
6.14 Dump styles . . . 245
6.15 Removed commands and packages . . . 246
6.15.1 amber2lmp tools . . . 246
6.15.2 ATC, AWPMD, and POEMS packages . . . 247
6.15.3 Neighbor style and comm mode multi/old . . . 254

6.15.4 LAMMPS-GUI source code . . . 254
6.15.5 GJF formulation in fix langevin . . . 254
6.15.6 LAMMPS shell . . . 254
6.15.7 i-PI tool . . . 254
6.15.8 USER-REAXC package . . . 255
6.15.9 MPIIO package . . . 255
6.15.10 MSCG package . . . 255
6.15.11 LATTE package . . . 255
6.15.12 Minimize style fire/old . . . 255
6.15.13 Pair style mesont/tpm, compute style mesont, atom style mesont . . . 255
6.15.14 Box command . . . 255
6.15.15 Reset_ids, reset_atom_ids, reset_mol_ids commands . . . 256
6.15.16 MESSAGE package . . . 256
6.15.17 REAX package . . . 256
6.15.18 MEAM package . . . 256
6.15.19 USER-CUDA package . . . 256
6.15.20 Compute atom/molecule . . . 256
6.15.21 Fix ave/spatial and fix ave/spatial/sphere . . . 256
6.15.22 restart2data tool . . . 257

**7 Accelerate performance** **259**
7.1 Benchmarks . . . 259
7.2 Measuring performance . . . 260
7.2.1 Factors that influence performance . . . 260
7.2.2 Examples comparing serial performance . . . 261
7.2.3 Examples comparing parallel performance . . . 262
7.2.4 Measuring performance of your input deck . . . 263
7.3 General tips . . . 264
7.4 Accelerator packages . . . 265
7.4.1 GPU package . . . 265
7.4.2 INTEL package . . . 269
7.4.3 KOKKOS package . . . 275
7.4.4 OPENMP package . . . 282
7.4.5 OPT package . . . 284
7.5 Comparison of various accelerator packages . . . 287

**8 Optional packages** **289**
8.1 Package details . . . 293
8.1.1 ADIOS package . . . 294
8.1.2 AMOEBA package . . . 295
8.1.3 APIP package . . . 295
8.1.4 ASPHERE package . . . 296
8.1.5 BOCS package . . . 297
8.1.6 BODY package . . . 297
8.1.7 BPM package . . . 298
8.1.8 BROWNIAN package . . . 298
8.1.9 CG-DNA package . . . 299
8.1.10 CG-SPICA package . . . 299
8.1.11 CLASS2 package . . . 300
8.1.12 COLLOID package . . . 300
8.1.13 COLVARS package . . . 300
8.1.14 COMPRESS package . . . 301
8.1.15 CORESHELL package . . . 301
8.1.16 DIELECTRIC package . . . 302

8.1.17 DIFFRACTION package . . . 303
8.1.18 DIPOLE package . . . 303
8.1.19 DPD-BASIC package . . . 303
8.1.20 DPD-MESO package . . . 304
8.1.21 DPD-REACT package . . . 304
8.1.22 DPD-SMOOTH package . . . 305
8.1.23 DRUDE package . . . 305
8.1.24 EFF package . . . 306
8.1.25 ELECTRODE package . . . 306
8.1.26 EXTRA-COMMAND package . . . 307
8.1.27 EXTRA-COMPUTE package . . . 307
8.1.28 EXTRA-DUMP package . . . 307
8.1.29 EXTRA-FIX package . . . 308
8.1.30 EXTRA-MOLECULE package . . . 308
8.1.31 EXTRA-PAIR package . . . 308
8.1.32 FEP package . . . 308
8.1.33 GPU package . . . 309
8.1.34 GRAPHICS package . . . 309
8.1.35 GRANULAR package . . . 310
8.1.36 GRANSURF package . . . 311
8.1.37 H5MD package . . . 311
8.1.38 INTEL package . . . 311
8.1.39 INTERLAYER package . . . 312
8.1.40 KIM package . . . 312
8.1.41 KOKKOS package . . . 313
8.1.42 KSPACE package . . . 314
8.1.43 LATBOLTZ package . . . 315
8.1.44 LEPTON package . . . 315
8.1.45 MACHDYN package . . . 316
8.1.46 MANIFOLD package . . . 316
8.1.47 MANYBODY package . . . 317
8.1.48 MBX Package . . . 317
8.1.49 MC package . . . 317
8.1.50 MDI package . . . 318
8.1.51 MEAM package . . . 319
8.1.52 MESONT package . . . 319
8.1.53 MGPT package . . . 320
8.1.54 MISC package . . . 320
8.1.55 ML-HDNNP package . . . 321
8.1.56 ML-RUNNER package . . . 321
8.1.57 ML-IAP package . . . 322
8.1.58 ML-PACE package . . . 322
8.1.59 ML-POD package . . . 323
8.1.60 ML-QUIP package . . . 323
8.1.61 ML-RANN package . . . 324
8.1.62 ML-SNAP package . . . 324
8.1.63 ML-UF3 package . . . 324
8.1.64 MOFFF package . . . 325
8.1.65 MOLECULE package . . . 325
8.1.66 MOLFILE package . . . 326
8.1.67 NETCDF package . . . 326
8.1.68 OPENMP package . . . 327
8.1.69 OPT package . . . 328
8.1.70 ORIENT package . . . 328

8.1.71 PERI package . . . 328
8.1.72 PHONON package . . . 329
8.1.73 PLUGIN package . . . 329
8.1.74 PLUMED package . . . 330
8.1.75 PTM package . . . 330
8.1.76 PYTHON package . . . 330
8.1.77 QEQ package . . . 331
8.1.78 QMMM package . . . 331
8.1.79 QTB package . . . 332
8.1.80 REACTION package . . . 332
8.1.81 REAXFF package . . . 333
8.1.82 REPLICA package . . . 333
8.1.83 RHEO package . . . 334
8.1.84 RIGID package . . . 334
8.1.85 SCAFACOS package . . . 335
8.1.86 SHOCK package . . . 335
8.1.87 SMTBQ package . . . 336
8.1.88 SPH package . . . 336
8.1.89 SPIN package . . . 337
8.1.90 SRD package . . . 337
8.1.91 TALLY package . . . 338
8.1.92 UEF package . . . 338
8.1.93 VORONOI package . . . 339
8.1.94 VTK package . . . 339
8.1.95 YAFF package . . . 339

**9 Auxiliary tools** **341**
9.1 Pre-processing tools . . . 341
9.2 Post-processing tools . . . 341
9.3 Miscellaneous tools . . . 342
9.4 Tool descriptions . . . 342
9.4.1 binary2txt tool . . . 342
9.4.2 ch2lmp tool . . . 342
9.4.3 chain tool . . . 342
9.4.4 LAMMPS coding standard . . . 343
9.4.5 colvars tools . . . 343
9.4.6 createatoms tool . . . 343
9.4.7 drude tool . . . 343
9.4.8 eam database tool . . . 344
9.4.9 eam generate tool . . . 344
9.4.10 eff tool . . . 344
9.4.11 emacs tool . . . 344
9.4.12 fep tool . . . 345
9.4.13 i-PI tool . . . 345
9.4.14 ipp tool . . . 345
9.4.15 JSON support files . . . 345
9.4.16 kate tool . . . 347
9.4.17 LAMMPS-GUI . . . 347
9.4.18 lmp2arc tool . . . 347
9.4.19 lmp2cfg tool . . . 347
9.4.20 Magic patterns for the “file” command . . . 347
9.4.21 matlab tool . . . 348
9.4.22 micelle2d tool . . . 348
9.4.23 moltemplate tool . . . 348

9.4.24 msi2lmp tool . . . 348
9.4.25 Scripts for building LAMMPS when offline . . . 349
9.4.26 phonon tool . . . 350
9.4.27 polybond tool . . . 350
9.4.28 pymol_asphere tool . . . 350
9.4.29 python tool . . . 351
9.4.30 Regression tester tool . . . 351
9.4.31 replica tool . . . 351
9.4.32 smd tool . . . 351
9.4.33 spin tool . . . 352
9.4.34 singularity/apptainer tool . . . 352
9.4.35 stl_bin2txt tool . . . 352
9.4.36 SWIG interface . . . 352
9.4.37 tabulate tool . . . 354
9.4.38 tinker tool . . . 354
9.4.39 valgrind tool . . . 354
9.4.40 vim tool . . . 354
9.4.41 xmgrace tool . . . 354

**10 Howto discussions** **355**
10.1 General howto . . . 355
10.1.1 Restart a simulation . . . 355
10.1.2 Visualize LAMMPS snapshots . . . 356
10.1.3 Basic workflow for loading LAMMPS trajectories in VMD . . . 357
10.1.4 Advanced graphics features in the *dump image* command . . . 358
10.1.5 Run multiple simulations from one input script . . . 374
10.1.6 Multi-replica simulations . . . 375
10.1.7 Library interface to LAMMPS . . . 376
10.1.8 Coupling LAMMPS to other codes . . . 376
10.1.9 Using LAMMPS with the MDI library for code coupling . . . 377
10.2 Settings howto . . . 379
10.2.1 2d simulations . . . 379
10.2.2 Type labels . . . 380
10.2.3 Triclinic (non-orthogonal) simulation boxes . . . 382
10.2.4 Thermostats . . . 387
10.2.5 Barostats . . . 388
10.2.6 Walls . . . 389
10.2.7 NEMD simulations . . . 390
10.2.8 Long-range dispersion settings . . . 391
10.2.9 Convert bulk system to slab . . . 392
10.2.10 Broken Bonds . . . 394
10.3 Analysis howto . . . 395
10.3.1 Output from LAMMPS (thermo, dumps, computes, fixes, variables) . . . 395
10.3.2 Output structured data from LAMMPS . . . 401
10.3.3 Use chunks to calculate system properties . . . 407
10.3.4 Using distributed grids . . . 410
10.3.5 Calculate temperature . . . 412
10.3.6 Calculate elastic constants . . . 412
10.3.7 Calculate thermal conductivity . . . 413
10.3.8 Calculate viscosity . . . 414
10.3.9 Calculate diffusion coefficients . . . 416
10.4 Force fields howto . . . 417
10.4.1 Some general force field considerations . . . 417
10.4.2 CHARMM, AMBER, COMPASS, ClassII-xe, DREIDING, and OPLS force fields . . . 417

10.4.3 AMBER to LAMMPS Tutorial . . . 423
10.4.4 AMOEBA and HIPPO force fields . . . 432
10.4.5 TIP3P water model . . . 437
10.4.6 TIP4P and OPC water models . . . 440
10.4.7 TIP5P water model . . . 445
10.4.8 SPC and SPC/E water model . . . 447
10.4.9 Two-band (and multi-band) EAM potentials with hybrid/overlay . . . 450
10.5 Packages howto . . . 452
10.5.1 Finite-size spherical and aspherical particles . . . 452
10.5.2 Granular models . . . 455
10.5.3 Granular surfaces . . . 457
10.5.4 Body particles . . . 465
10.5.5 Bonded particle models . . . 471
10.5.6 Polarizable models . . . 474
10.5.7 Adiabatic core/shell model . . . 475
10.5.8 Drude induced dipoles . . . 479
10.5.9 Tutorial for Thermalized Drude oscillators in LAMMPS . . . 479
10.5.10 Peridynamics with LAMMPS . . . 486
10.5.11 Manifolds (surfaces) . . . 502
10.5.12 Reproducing hydrodynamics and elastic objects (RHEO) . . . 503
10.5.13 Magnetic spins . . . 504
10.5.14 Adaptive-precision interatomic potentials (APIP) . . . 505
10.5.15 Local Density Dependent Potentials via LDD . . . 510
10.6 Tutorials howto . . . 515
10.6.1 Using CMake with LAMMPS . . . 515
10.6.2 LAMMPS GitHub tutorial . . . 522
10.6.3 Using LAMMPS-GUI . . . 535
10.6.4 Moltemplate Tutorial . . . 535
10.6.5 LAMMPS Python Tutorial . . . 543
10.6.6 Using LAMMPS on Windows 10 with WSL . . . 554

**11 Example scripts** **565**
11.1 Lowercase directories . . . 565
11.2 Uppercase directories . . . 568

**II Programmer Guide** **569**

**1 LAMMPS Library Interfaces** **571**
1.1 LAMMPS C Library API . . . 571
1.1.1 Creating or deleting a LAMMPS object . . . 572
1.1.2 Executing commands . . . 577
1.1.3 System properties . . . 580
1.1.4 Per-atom properties . . . 594
1.1.5 Computes, fixes, variables . . . 595
1.1.6 Scatter/gather operations . . . 606
1.1.7 Neighbor list access . . . 620
1.1.8 Configuration information . . . 622
1.1.9 Utility functions . . . 629
1.1.10 Extending the C API . . . 637
1.2 LAMMPS Python API . . . 638
1.3 LAMMPS Fortran API . . . 638
1.3.1 The *LIBLAMMPS* Fortran Module . . . 638
1.3.2 Creating or deleting a LAMMPS object . . . 639

1.3.3 Executing LAMMPS commands . . . 640
1.3.4 Accessing system properties . . . 641
1.3.5 The *LIBLAMMPS* module API . . . 642
1.4 LAMMPS C++ API . . . 688
1.4.1 Using the C++ API directly . . . 689
1.4.2 Creating or deleting a LAMMPS object . . . 689
1.4.3 Executing LAMMPS commands . . . 689

**2 Use Python with LAMMPS** **691**
2.1 Overview . . . 691
2.2 Installation . . . 692
2.2.1 Installing the LAMMPS Python Module and Shared Library . . . 693
2.2.2 Extending Python to run in parallel . . . 696
2.3 Run LAMMPS from Python . . . 697
2.3.1 Running LAMMPS and Python in serial . . . 697
2.3.2 Running LAMMPS and Python in parallel with MPI . . . 698
2.3.3 Running Python scripts . . . 698
2.3.4 Creating or deleting a LAMMPS object . . . 699
2.3.5 Executing commands . . . 700
2.3.6 System properties . . . 701
2.3.7 Per-atom properties . . . 702
2.3.8 Compute, fixes, variables . . . 703
2.3.9 Scatter/gather operations . . . 704
2.3.10 Neighbor list access . . . 705
2.3.11 Configuration information . . . 707
2.4 The `lammps` Python module . . . 708
2.4.1 The `lammps` class API . . . 708
2.4.2 Additional components of the `lammps` module . . . 736
2.5 Extending the Python interface . . . 738
2.6 Calling Python from LAMMPS . . . 738
2.7 Output Readers . . . 739
2.8 Example Python scripts . . . 739
2.9 Using LAMMPS in IPython notebooks and Jupyter . . . 740
2.9.1 Interactive Python Examples . . . 741
2.10 Handling LAMMPS errors . . . 741
2.11 Troubleshooting . . . 742
2.11.1 Testing if Python can launch LAMMPS . . . 742

**3 Modifying & extending LAMMPS** **743**
3.1 Overview . . . 743
3.2 Submitting new features for inclusion in LAMMPS . . . 744
3.2.1 Communication with the LAMMPS developers . . . 745
3.2.2 Time and effort required . . . 745
3.2.3 Submission procedure . . . 745
3.2.4 External contributions . . . 745
3.2.5 Location of files: individual files and packages . . . 745
3.2.6 Changes to core LAMMPS files . . . 746
3.3 Requirements for contributions to LAMMPS . . . 746
3.3.1 Motivation . . . 746
3.3.2 Licensing requirements (strict) . . . 746
3.3.3 Integration testing (strict) . . . 747
3.3.4 Documentation (strict) . . . 747
3.3.5 Build system (strict) . . . 748
3.3.6 Command or style names, file names, and keywords (strict) . . . 749

3.3.7 Programming style requirements (varied) . . . 749
3.3.8 Examples (preferred) . . . 749
3.3.9 Error or warning messages and explanations (preferred) . . . 749
3.3.10 Citation reminder (optional) . . . 750
3.3.11 Testing (optional) . . . 750
3.4 LAMMPS programming style . . . 750
3.4.1 Include files (varied) . . . 750
3.4.2 Whitespace (preferred) . . . 751
3.4.3 Constants (strongly preferred) . . . 751
3.4.4 Placement of braces (strongly preferred) . . . 752
3.4.5 Miscellaneous standards (varied) . . . 752
3.5 Atom styles . . . 753
3.6 Pair styles . . . 755
3.7 Bond, angle, dihedral, improper styles . . . 757
3.8 Compute styles . . . 758
3.9 Fix styles . . . 759
3.10 Input script command style . . . 762
3.11 Dump styles . . . 762
3.12 Kspace styles . . . 763
3.13 Minimization styles . . . 764
3.14 Region styles . . . 765
3.15 Body styles . . . 765
3.16 Granular Sub-Model styles . . . 766
3.17 Thermodynamic output options . . . 768
3.18 Variable options . . . 769

**4 Information for Developers** **771**
4.1 Source files . . . 772
4.2 Class topology . . . 772
4.3 Code design . . . 775
4.3.1 Object-oriented code . . . 776
4.3.2 I/O and output formatting . . . 779
4.3.3 JSON format input and output . . . 781
4.3.4 Memory management . . . 781
4.4 Parallel algorithms . . . 781
4.4.1 Partitioning . . . 782
4.4.2 Communication . . . 783
4.4.3 Neighbor lists . . . 785
4.4.4 Long-range interactions . . . 787
4.4.5 OpenMP Parallelism . . . 789
4.5 Accessing per-atom data . . . 791
4.5.1 Owned and ghost atoms . . . 791
4.5.2 Atom indexing . . . 791
4.5.3 Atom class versus AtomVec classes . . . 792
4.6 Communication patterns . . . 792
4.6.1 Owned and ghost atoms . . . 792
4.6.2 Higher level communication . . . 793
4.7 How a timestep works . . . 795
4.8 Writing new styles . . . 798
4.8.1 Writing new pair styles . . . 798
4.8.2 Package and build system considerations . . . 798
4.8.3 Case 1: a pairwise additive model . . . 799
4.8.4 Case 2: a many-body potential . . . 812
4.8.5 Case 3: a potential requiring communication . . . 815

4.8.6 Case 4: potentials without a compute() function . . . 816
4.8.7 Writing new bond, angle, dihedral, and improper styles . . . 818
4.8.8 Case 1: Implementing a bond style . . . 818
4.8.9 Case 2: Implementing an angle style . . . 824
4.8.10 Case 3: Implementing a dihedral style . . . 825
4.8.11 Case 4: Implementing an improper style . . . 826
4.8.12 Writing a new fix style . . . 828
4.8.13 Writing new compute styles . . . 831
4.8.14 Case 1: A global scalar compute . . . 832
4.8.15 Case 2: A per-atom compute . . . 836
4.8.16 Notes on flags and output types . . . 838
4.8.17 Writing a new command style . . . 839
4.8.18 Case 1: Implementing the geturl command . . . 839
4.8.19 Creating styles for the OPENMP package . . . 844
4.8.20 Writing or updating documentation . . . 849
4.8.21 Anatomy of a style documentation page . . . 849
4.8.22 Integrating a new style into the command lists . . . 853
4.8.23 Documenting an accelerated variant of an existing style . . . 854
4.8.24 Adding and referencing example inputs . . . 854
4.8.25 Other documentation that may need updating . . . 854
4.8.26 Building and checking the documentation . . . 855
4.9 Notes for developers and code maintainers . . . 855
4.9.1 Reading and parsing of text and text files . . . 855
4.9.2 Requesting and accessing neighbor lists . . . 856
4.9.3 Errors, warnings, and informational messages . . . 858
4.9.4 Choosing between a custom atom style, fix property/atom, and fix STORE/ATOM . . . 860
4.9.5 Fix contributions to instantaneous energy, virial, and cumulative energy . . . 860
4.9.6 KSpace PPPM FFT grids . . . 862
4.10 Notes for updating code written for older LAMMPS versions . . . 863
4.10.1 Setting flags in the constructor . . . 864
4.10.2 Rename of pack/unpack_comm() to pack/unpack_forward_comm() . . . 864
4.10.3 Use ev_init() to initialize variables derived from eflag and vflag . . . 865
4.10.4 Use utils::count_words() functions instead of atom->count_words() . . . 865
4.10.5 Use utils::numeric() functions instead of force->numeric() . . . 866
4.10.6 Use utils::open_potential() function to open potential files . . . 866
4.10.7 Use symbolic Atom and AtomVec constants instead of numerical values . . . 866
4.10.8 Simplify customized error messages . . . 867
4.10.9 Use of “override” instead of “virtual” . . . 867
4.10.10 Simplified function names for forward and reverse communication . . . 868
4.10.11 Simplified and more compact neighbor list requests . . . 868
4.10.12 Split of fix STORE into fix STORE/GLOBAL and fix STORE/PERATOM . . . 869
4.10.13 Rename of fix STORE/PERATOM to fix STORE/ATOM and change of arguments . . . 870
4.10.14 Use Output::get_dump_by_id() instead of Output::find_dump() . . . 870
4.10.15 Refactored grid communication using Grid3d/Grid2d classes instead of GridComm . . . 871
4.10.16 FLERR as first argument to minimum image functions in Domain class . . . 872
4.10.17 Use utils::logmesg() instead of error->warning() . . . 872
4.11 Writing plugins . . . 873
4.11.1 Members of `lammpsplugin_t` . . . 873
4.11.2 Pair style example . . . 874
4.11.3 Fix style example . . . 874
4.11.4 Command style example . . . 875
4.11.5 Additional Details . . . 876
4.11.6 Compiling plugins . . . 877
4.12 Adding tests for unit testing . . . 877

4.12.1 Tests for utility functions . . . 878
4.12.2 Tests for individual LAMMPS commands . . . 880
4.12.3 Tests for the C-style library interface . . . 881
4.12.4 Tests for the Python module and package . . . 882
4.12.5 Tests for the Fortran interface . . . 882
4.12.6 Tests for the C++-style library interface . . . 882
4.12.7 Tests for reading and writing file formats . . . 882
4.12.8 Tests for styles computing or modifying forces . . . 882
4.12.9 Tests for granular (DEM) models . . . 889
4.12.10 Tests for programs in the tools folder . . . 892
4.12.11 Troubleshooting failed unit tests . . . 893
4.13 C++ base classes . . . 894
4.13.1 LAMMPS Class . . . 894
4.13.2 LAMMPS Atom and AtomVec Base Classes . . . 896
4.13.3 LAMMPS Input Base Class . . . 901
4.13.4 Citation management for contributed features . . . 902
4.14 Platform abstraction functions . . . 904
4.14.1 Time functions . . . 904
4.14.2 Platform information functions . . . 904
4.14.3 File and path functions and global constants . . . 905
4.14.4 Standard I/O function wrappers . . . 908
4.14.5 Environment variable functions . . . 909
4.14.6 Dynamically loaded object or library functions . . . 910
4.14.7 Compressed file I/O functions . . . 911
4.15 Utility functions . . . 911
4.15.1 I/O with status check and similar functions . . . 912
4.15.2 String to number conversions with validity check . . . 914
4.15.3 String processing . . . 917
4.15.4 Potential file functions . . . 923
4.15.5 Argument processing . . . 924
4.15.6 Convenience functions . . . 927
4.15.7 Customized standard functions . . . 930
4.16 Special Math functions . . . 930
4.17 Tokenizer classes . . . 932
4.18 Argument parsing classes . . . 937
4.19 Safe pointer classes . . . 940
4.20 File reader classes . . . 942
4.21 Type label support . . . 946
4.21.1 Overview . . . 946
4.21.2 Integration with utils namespace . . . 946
4.21.3 LabelMap class reference . . . 948
4.22 Memory pool classes . . . 953
4.23 Eigensolver functions . . . 957
4.24 Communication buffer coding with *ubuf* . . . 958
4.25 Internal Styles . . . 958
4.25.1 DEPRECATED Styles . . . 959
4.25.2 Internal fix styles . . . 959
4.26 Use of distributed grids within style classes . . . 960
4.26.1 Style commands . . . 961
4.26.2 Grid data allocation and access . . . 961
4.26.3 Grid class constructors . . . 962
4.26.4 Grid class set methods . . . 963
4.26.5 Grid class setup_grid method . . . 964
4.26.6 More grid class set methods . . . 965

4.26.7 Grid class get methods . . . 965
4.26.8 Grid class owned/ghost communication . . . 965
4.26.9 Grid class remap methods for load balancing . . . 967
4.26.10 Grid class I/O methods . . . 968
4.26.11 Style class grid access methods . . . 969
4.26.12 Final notes . . . 970

**III Command Reference 973**

**1 Commands 975**
1.1 angle_coeff command . . . 975
1.1.1 Syntax . . . 975
1.1.2 Examples . . . 975
1.1.3 Description . . . 975
1.1.4 Restrictions . . . 976
1.1.5 Related commands . . . 976
1.1.6 Default . . . 976
1.2 angle_style command . . . 976
1.2.1 Syntax . . . 976
1.2.2 Examples . . . 976
1.2.3 Description . . . 976
1.2.4 Restrictions . . . 978
1.2.5 Related commands . . . 978
1.2.6 Default . . . 978
1.3 angle_write command . . . 978
1.3.1 Syntax . . . 978
1.3.2 Examples . . . 978
1.3.3 Description . . . 979
1.3.4 Restrictions . . . 979
1.3.5 Related commands . . . 979
1.3.6 Default . . . 979
1.4 atom_modify command . . . 979
1.4.1 Syntax . . . 979
1.4.2 Examples . . . 980
1.4.3 Description . . . 980
1.4.4 Restrictions . . . 982
1.4.5 Related commands . . . 982
1.4.6 Default . . . 982
1.5 atom_style command . . . 982
1.5.1 Syntax . . . 982
1.5.2 Examples . . . 983
1.5.3 Description . . . 983
1.5.4 Atom style attributes . . . 983
1.5.5 Particle size and mass . . . 985
1.5.6 Additional information about specific atom styles . . . 985
1.5.7 Restrictions . . . 988
1.5.8 Related commands . . . 988
1.5.9 Default . . . 988
1.6 balance command . . . 988
1.6.1 Syntax . . . 988
1.6.2 Examples . . . 989
1.6.3 Description . . . 989
1.6.4 Restrictions . . . 995

1.6.5 Related commands . . . 995
1.6.6 Default . . . 995
1.7 bond_coeff command . . . 996
1.7.1 Syntax . . . 996
1.7.2 Examples . . . 996
1.7.3 Description . . . 996
1.7.4 Restrictions . . . 997
1.7.5 Related commands . . . 997
1.7.6 Default . . . 997
1.8 bond_style command . . . 997
1.8.1 Syntax . . . 997
1.8.2 Examples . . . 997
1.8.3 Description . . . 997
1.8.4 Restrictions . . . 999
1.8.5 Related commands . . . 999
1.8.6 Default . . . 999
1.9 bond_write command . . . 999
1.9.1 Syntax . . . 999
1.9.2 Examples . . . 999
1.9.3 Description . . . 999
1.9.4 Restrictions . . . 1000
1.9.5 Related commands . . . 1000
1.9.6 Default . . . 1000
1.10 boundary command . . . 1000
1.10.1 Syntax . . . 1000
1.10.2 Examples . . . 1000
1.10.3 Description . . . 1000
1.10.4 Restrictions . . . 1001
1.10.5 Related commands . . . 1001
1.10.6 Default . . . 1002
1.11 change_box command . . . 1002
1.11.1 Syntax . . . 1002
1.11.2 Examples . . . 1002
1.11.3 Description . . . 1003
1.11.4 Restrictions . . . 1006
1.11.5 Related commands . . . 1007
1.11.6 Default . . . 1007
1.12 clear command . . . 1007
1.12.1 Syntax . . . 1007
1.12.2 Examples . . . 1007
1.12.3 Description . . . 1007
1.12.4 Restrictions . . . 1007
1.12.5 Related commands . . . 1007
1.12.6 Default . . . 1007
1.13 comm_modify command . . . 1007
1.13.1 Syntax . . . 1007
1.13.2 Examples . . . 1008
1.13.3 Description . . . 1008
1.13.4 Restrictions . . . 1010
1.13.5 Related commands . . . 1010
1.13.6 Default . . . 1010
1.14 comm_style command . . . 1010
1.14.1 Syntax . . . 1010
1.14.2 Examples . . . 1010

1.14.3 Description . . . 1010
1.14.4 Restrictions . . . 1011
1.14.5 Related commands . . . 1011
1.14.6 Default . . . 1011
1.15 compute command . . . 1011
1.15.1 Syntax . . . 1011
1.15.2 Examples . . . 1011
1.15.3 Description . . . 1011
1.15.4 Restrictions . . . 1018
1.15.5 Related commands . . . 1018
1.15.6 Default . . . 1018
1.16 compute_modify command . . . 1018
1.16.1 Syntax . . . 1018
1.16.2 Examples . . . 1019
1.16.3 Description . . . 1019
1.16.4 Restrictions . . . 1019
1.16.5 Related commands . . . 1019
1.16.6 Default . . . 1019
1.17 create_atoms command . . . 1019
1.17.1 Syntax . . . 1019
1.17.2 Examples . . . 1021
1.17.3 Description . . . 1021
1.17.4 Restrictions . . . 1028
1.17.5 Related commands . . . 1028
1.17.6 Default . . . 1028
1.18 create_bonds command . . . 1028
1.18.1 Syntax . . . 1028
1.18.2 Examples . . . 1029
1.18.3 Description . . . 1029
1.18.4 Restrictions . . . 1031
1.18.5 Related commands . . . 1031
1.18.6 Default . . . 1031
1.19 create_box command . . . 1031
1.19.1 Syntax . . . 1031
1.19.2 Examples . . . 1032
1.19.3 Description . . . 1032
1.19.4 Restrictions . . . 1034
1.19.5 Related commands . . . 1034
1.19.6 Default . . . 1034
1.20 delete_atoms command . . . 1034
1.20.1 Syntax . . . 1034
1.20.2 Examples . . . 1035
1.20.3 Description . . . 1035
1.20.4 Restrictions . . . 1037
1.20.5 Related commands . . . 1037
1.20.6 Default . . . 1037
1.21 delete_bonds command . . . 1037
1.21.1 Syntax . . . 1037
1.21.2 Examples . . . 1038
1.21.3 Description . . . 1038
1.21.4 Restrictions . . . 1039
1.21.5 Related commands . . . 1039
1.21.6 Default . . . 1039
1.22 dielectric command . . . 1040

1.22.1 Syntax . . . 1040
1.22.2 Examples . . . 1040
1.22.3 Description . . . 1040
1.22.4 Restrictions . . . 1040
1.22.5 Related commands . . . 1040
1.22.6 Default . . . 1040
1.23 dihedral_coeff command . . . 1040
1.23.1 Syntax . . . 1040
1.23.2 Examples . . . 1040
1.23.3 Description . . . 1041
1.23.4 Restrictions . . . 1041
1.23.5 Related commands . . . 1042
1.23.6 Default . . . 1042
1.24 dihedral_style command . . . 1042
1.24.1 Syntax . . . 1042
1.24.2 Examples . . . 1042
1.24.3 Description . . . 1042
1.24.4 Restrictions . . . 1044
1.24.5 Related commands . . . 1044
1.24.6 Default . . . 1044
1.25 dihedral_write command . . . 1044
1.25.1 Syntax . . . 1044
1.25.2 Examples . . . 1044
1.25.3 Description . . . 1045
1.25.4 Restrictions . . . 1045
1.25.5 Related commands . . . 1045
1.25.6 Default . . . 1045
1.26 dimension command . . . 1045
1.26.1 Syntax . . . 1045
1.26.2 Examples . . . 1046
1.26.3 Description . . . 1046
1.26.4 Restrictions . . . 1046
1.26.5 Related commands . . . 1046
1.26.6 Default . . . 1046
1.27 displace_atoms command . . . 1046
1.27.1 Syntax . . . 1046
1.27.2 Examples . . . 1047
1.27.3 Description . . . 1047
1.27.4 Restrictions . . . 1048
1.27.5 Related commands . . . 1048
1.27.6 Default . . . 1048
1.28 dynamical_matrix command . . . 1048
1.28.1 Syntax . . . 1048
1.28.2 Examples . . . 1049
1.28.3 Description . . . 1049
1.28.4 Restrictions . . . 1049
1.28.5 Related commands . . . 1049
1.28.6 Default . . . 1050
1.29 echo command . . . 1050
1.29.1 Syntax . . . 1050
1.29.2 Examples . . . 1050
1.29.3 Description . . . 1050
1.29.4 Restrictions . . . 1050
1.29.5 Related commands . . . 1050

1.29.6 Default . . . 1050
1.30 fix command . . . 1050
1.30.1 Syntax . . . 1050
1.30.2 Examples . . . 1051
1.30.3 Description . . . 1051
1.30.4 Restrictions . . . 1061
1.30.5 Related commands . . . 1061
1.30.6 Default . . . 1061
1.31 fix_modify command . . . 1061
1.31.1 Syntax . . . 1061
1.31.2 Examples . . . 1062
1.31.3 Description . . . 1062
1.31.4 Restrictions . . . 1064
1.31.5 Related commands . . . 1064
1.31.6 Default . . . 1064
1.32 fitpod command . . . 1064
1.32.1 Syntax . . . 1064
1.32.2 Examples . . . 1064
1.32.3 Description . . . 1064
1.32.4 POD Potential . . . 1067
1.32.5 Training . . . 1067
1.32.6 Validation . . . 1068
1.32.7 Restrictions . . . 1068
1.32.8 Related commands . . . 1068
1.32.9 Default . . . 1068
1.33 geturl command . . . 1068
1.33.1 Syntax . . . 1068
1.33.2 Examples . . . 1068
1.33.3 Description . . . 1069
1.33.4 Restrictions . . . 1069
1.33.5 Related commands . . . 1069
1.33.6 Default . . . 1070
1.34 group command . . . 1070
1.34.1 Syntax . . . 1070
1.34.2 Examples . . . 1071
1.34.3 Description . . . 1071
1.34.4 Restrictions . . . 1075
1.34.5 Related commands . . . 1075
1.34.6 Default . . . 1075
1.35 group2ndx command . . . 1075
1.36 ndx2group command . . . 1075
1.36.1 Syntax . . . 1075
1.36.2 Examples . . . 1075
1.36.3 Description . . . 1075
1.36.4 File Format . . . 1076
1.36.5 Restrictions . . . 1076
1.36.6 Related commands . . . 1076
1.36.7 Default . . . 1076
1.37 hyper command . . . 1076
1.37.1 Syntax . . . 1076
1.37.2 Examples . . . 1077
1.37.3 Description . . . 1077
1.37.4 Restrictions . . . 1078
1.37.5 Related commands . . . 1078

1.37.6 Default . . . 1078
1.38 if command . . . 1079
1.38.1 Syntax . . . 1079
1.38.2 Examples . . . 1079
1.38.3 Description . . . 1079
1.38.4 Restrictions . . . 1082
1.38.5 Related commands . . . 1082
1.38.6 Default . . . 1082
1.39 improper_coeff command . . . 1082
1.39.1 Syntax . . . 1082
1.39.2 Examples . . . 1082
1.39.3 Description . . . 1082
1.39.4 Restrictions . . . 1083
1.39.5 Related commands . . . 1083
1.39.6 Default . . . 1083
1.40 improper_style command . . . 1083
1.40.1 Syntax . . . 1083
1.40.2 Examples . . . 1083
1.40.3 Description . . . 1083
1.40.4 Restrictions . . . 1084
1.40.5 Related commands . . . 1085
1.40.6 Default . . . 1085
1.41 include command . . . 1085
1.41.1 Syntax . . . 1085
1.41.2 Examples . . . 1085
1.41.3 Description . . . 1085
1.41.4 Restrictions . . . 1085
1.41.5 Related commands . . . 1085
1.41.6 Default . . . 1085
1.42 info command . . . 1085
1.42.1 Syntax . . . 1085
1.42.2 Examples . . . 1086
1.42.3 Description . . . 1086
1.42.4 Restrictions . . . 1087
1.42.5 Related commands . . . 1087
1.42.6 Default . . . 1087
1.43 jump command . . . 1087
1.43.1 Syntax . . . 1087
1.43.2 Examples . . . 1088
1.43.3 Description . . . 1088
1.43.4 Restrictions . . . 1089
1.43.5 Related commands . . . 1089
1.43.6 Default . . . 1089
1.44 kim command . . . 1090
1.44.1 Syntax . . . 1090
1.44.2 Examples . . . 1090
1.44.3 Description . . . 1090
1.44.4 Using OpenKIM IMs with LAMMPS (*kim init*, *kim interactions*) . . . 1091
1.44.5 Using OpenKIM Web Queries in LAMMPS (*kim query*) . . . 1095
1.44.6 Accessing KIM Model Parameters from LAMMPS (*kim param*) . . . 1099
1.44.7 Writing material properties in standard KIM Property Instance format (*kim property*) . . . 1102
1.44.8 Citation of OpenKIM IMs . . . 1108
1.44.9 Restrictions . . . 1108
1.44.10 Related commands . . . 1108

1.45 kspace_modify command . . . 1109
1.45.1 Syntax . . . 1109
1.45.2 Examples . . . 1110
1.45.3 Description . . . 1110
1.45.4 Restrictions . . . 1115
1.45.5 Related commands . . . 1115
1.45.6 Default . . . 1115
1.46 kspace_style command . . . 1116
1.46.1 Syntax . . . 1116
1.46.2 Examples . . . 1118
1.46.3 Description . . . 1118
1.46.4 Restrictions . . . 1124
1.46.5 Related commands . . . 1124
1.46.6 Default . . . 1124
1.47 label command . . . 1125
1.47.1 Syntax . . . 1125
1.47.2 Examples . . . 1125
1.47.3 Description . . . 1125
1.47.4 Restrictions . . . 1125
1.47.5 Related commands . . . 1126
1.47.6 Default . . . 1126
1.48 labelmap command . . . 1126
1.48.1 Syntax . . . 1126
1.48.2 Examples . . . 1126
1.48.3 Description . . . 1126
1.48.4 Restrictions . . . 1127
1.48.5 Related commands . . . 1127
1.48.6 Default . . . 1127
1.49 lattice command . . . 1127
1.49.1 Syntax . . . 1127
1.49.2 Examples . . . 1128
1.49.3 Description . . . 1128
1.49.4 Restrictions . . . 1132
1.49.5 Related commands . . . 1132
1.49.6 Default . . . 1132
1.50 log command . . . 1132
1.50.1 Syntax . . . 1132
1.50.2 Examples . . . 1132
1.50.3 Description . . . 1132
1.50.4 Restrictions . . . 1133
1.50.5 Related commands . . . 1133
1.50.6 Default . . . 1133
1.51 mass command . . . 1133
1.51.1 Syntax . . . 1133
1.51.2 Examples . . . 1133
1.51.3 Description . . . 1133
1.51.4 Restrictions . . . 1134
1.51.5 Related commands . . . 1134
1.51.6 Default . . . 1134
1.52 mdi command . . . 1134
1.52.1 Syntax . . . 1134
1.52.2 Examples . . . 1134
1.52.3 Description . . . 1135
1.52.4 Restrictions . . . 1139

1.52.5 Related commands . . . 1139
1.52.6 Default . . . 1139
1.53 min_modify command . . . 1139
1.53.1 Syntax . . . 1139
1.53.2 Examples . . . 1140
1.53.3 Description . . . 1140
1.53.4 Restrictions . . . 1141
1.53.5 Related commands . . . 1141
1.53.6 Default . . . 1142
1.54 min_style spin command . . . 1142
1.55 min_style spin/cg command . . . 1142
1.56 min_style spin/lbfgs command . . . 1142
1.56.1 Syntax . . . 1142
1.56.2 Examples . . . 1142
1.56.3 Description . . . 1142
1.56.4 Restrictions . . . 1143
1.56.5 Related commands . . . 1143
1.56.6 Default . . . 1143
1.57 min_style cg command . . . 1143
1.58 min_style hftn command . . . 1143
1.59 min_style sd command . . . 1143
1.60 min_style quickmin command . . . 1143
1.61 min_style fire command . . . 1143
1.62 *min_style spin* command . . . 1144
1.63 *min_style spin/cg* command . . . 1144
1.64 *min_style spin/lbfgs* command . . . 1144
1.64.1 Syntax . . . 1144
1.64.2 Examples . . . 1144
1.64.3 Description . . . 1144
1.64.4 Restrictions . . . 1145
1.64.5 Related commands . . . 1145
1.64.6 Default . . . 1146
1.65 minimize command . . . 1146
1.65.1 Syntax . . . 1146
1.65.2 Examples . . . 1146
1.65.3 Description . . . 1146
1.65.4 Restrictions . . . 1150
1.65.5 Related commands . . . 1150
1.65.6 Default . . . 1150
1.66 molecule command . . . 1151
1.66.1 Syntax . . . 1151
1.66.2 Examples . . . 1151
1.66.3 Description . . . 1151
1.66.4 Format of a native molecule file . . . 1154
1.66.5 Format of a JSON molecule file . . . 1162
1.66.6 Restrictions . . . 1168
1.66.7 Related commands . . . 1168
1.66.8 Default . . . 1168
1.67 neb command . . . 1168
1.67.1 Syntax . . . 1168
1.67.2 Examples . . . 1169
1.67.3 Description . . . 1169
1.67.4 Restrictions . . . 1173
1.67.5 Related commands . . . 1173

1.67.6 Default . . . 1173
1.68 neb/spin command . . . 1174
1.68.1 Syntax . . . 1174
1.68.2 Examples . . . 1174
1.68.3 Description . . . 1174
1.68.4 Restrictions . . . 1178
1.68.5 Related commands . . . 1178
1.68.6 Default . . . 1178
1.69 neigh_modify command . . . 1179
1.69.1 Syntax . . . 1179
1.69.2 Examples . . . 1180
1.69.3 Description . . . 1180
1.69.4 Restrictions . . . 1182
1.69.5 Related commands . . . 1182
1.69.6 Default . . . 1183
1.70 neighbor command . . . 1183
1.70.1 Syntax . . . 1183
1.70.2 Examples . . . 1183
1.70.3 Description . . . 1183
1.70.4 Restrictions . . . 1184
1.70.5 Related commands . . . 1184
1.70.6 Default . . . 1184
1.71 newton command . . . 1184
1.71.1 Syntax . . . 1184
1.71.2 Examples . . . 1185
1.71.3 Description . . . 1185
1.71.4 Restrictions . . . 1185
1.71.5 Related commands . . . 1185
1.71.6 Default . . . 1185
1.72 next command . . . 1185
1.72.1 Syntax . . . 1185
1.72.2 Examples . . . 1185
1.72.3 Description . . . 1185
1.72.4 Restrictions . . . 1187
1.72.5 Related commands . . . 1187
1.72.6 Default . . . 1187
1.73 package command . . . 1187
1.73.1 Syntax . . . 1188
1.73.2 Examples . . . 1190
1.73.3 Description . . . 1191
1.73.4 Restrictions . . . 1198
1.73.5 Related commands . . . 1199
1.73.6 Defaults . . . 1199
1.74 pair_coeff command . . . 1199
1.74.1 Syntax . . . 1199
1.74.2 Examples . . . 1200
1.74.3 Description . . . 1200
1.74.4 Restrictions . . . 1201
1.74.5 Related commands . . . 1201
1.74.6 Default . . . 1201
1.75 pair_modify command . . . 1202
1.75.1 Syntax . . . 1202
1.75.2 Examples . . . 1202
1.75.3 Description . . . 1202

1.75.4 Restrictions . . . 1206
1.75.5 Related commands . . . 1206
1.75.6 Default . . . 1206
1.76 pair_style command . . . 1206
1.76.1 Syntax . . . 1206
1.76.2 Examples . . . 1206
1.76.3 Description . . . 1206
1.76.4 Restrictions . . . 1216
1.76.5 Related commands . . . 1216
1.76.6 Default . . . 1216
1.77 pair_write command . . . 1217
1.77.1 Syntax . . . 1217
1.77.2 Examples . . . 1217
1.77.3 Description . . . 1217
1.77.4 Restrictions . . . 1218
1.77.5 Related commands . . . 1218
1.77.6 Default . . . 1218
1.78 partition command . . . 1218
1.78.1 Syntax . . . 1218
1.78.2 Examples . . . 1218
1.78.3 Description . . . 1218
1.78.4 Restrictions . . . 1219
1.78.5 Related commands . . . 1219
1.78.6 Default . . . 1219
1.79 plugin command . . . 1219
1.79.1 Syntax . . . 1219
1.79.2 Examples . . . 1219
1.79.3 Description . . . 1219
1.79.4 Restrictions . . . 1220
1.79.5 Related commands . . . 1220
1.79.6 Default . . . 1220
1.80 prd command . . . 1221
1.80.1 Syntax . . . 1221
1.80.2 Examples . . . 1221
1.80.3 Description . . . 1221
1.80.4 Restrictions . . . 1224
1.80.5 Related commands . . . 1224
1.80.6 Default . . . 1224
1.81 print command . . . 1225
1.81.1 Syntax . . . 1225
1.81.2 Examples . . . 1225
1.81.3 Description . . . 1225
1.81.4 Restrictions . . . 1226
1.81.5 Related commands . . . 1226
1.81.6 Default . . . 1226
1.82 processors command . . . 1226
1.82.1 Syntax . . . 1226
1.82.2 Examples . . . 1227
1.82.3 Description . . . 1227
1.82.4 Restrictions . . . 1230
1.82.5 Related commands . . . 1231
1.82.6 Default . . . 1231
1.83 python command . . . 1231
1.83.1 Syntax . . . 1231

1.83.2 Examples . . . 1232
1.83.3 Description . . . 1232
1.83.4 Restrictions . . . 1240
1.83.5 Related commands . . . 1240
1.83.6 Default . . . 1240
1.84 quit command . . . 1240
1.84.1 Syntax . . . 1240
1.84.2 Examples . . . 1241
1.84.3 Description . . . 1241
1.84.4 Restrictions . . . 1241
1.84.5 Related commands . . . 1241
1.84.6 Default . . . 1241
1.85 read_data command . . . 1241
1.85.1 Syntax . . . 1241
1.85.2 Examples . . . 1242
1.85.3 Description . . . 1242
1.85.4 Reading multiple data files . . . 1243
1.85.5 Format of a data file . . . 1244
1.85.6 Format of the header of a data file . . . 1245
1.85.7 Header specification of the simulation box size and shape . . . 1246
1.85.8 Meaning of other header keywords . . . 1249
1.85.9 Format of the body of a data file . . . 1249
1.85.10 Restrictions . . . 1263
1.85.11 Related commands . . . 1264
1.85.12 Default . . . 1264
1.86 read_dump command . . . 1264
1.86.1 Syntax . . . 1264
1.86.2 Examples . . . 1265
1.86.3 Description . . . 1265
1.86.4 Restrictions . . . 1269
1.86.5 Related commands . . . 1269
1.86.6 Default . . . 1269
1.87 read_restart command . . . 1269
1.87.1 Syntax . . . 1269
1.87.2 Examples . . . 1269
1.87.3 Description . . . 1269
1.87.4 Restrictions . . . 1272
1.87.5 Related commands . . . 1272
1.87.6 Default . . . 1272
1.88 region command . . . 1272
1.88.1 Syntax . . . 1272
1.88.2 Examples . . . 1274
1.88.3 Description . . . 1274
1.88.4 Restrictions . . . 1278
1.88.5 Related commands . . . 1278
1.88.6 Default . . . 1278
1.89 region2vmd command . . . 1278
1.89.1 Syntax . . . 1278
1.89.2 Examples . . . 1279
1.89.3 Description . . . 1279
1.89.4 Restrictions . . . 1280
1.89.5 Related commands . . . 1280
1.89.6 Defaults . . . 1280
1.90 replicate command . . . 1280

1.90.1 Syntax . . . 1280
1.90.2 Examples . . . 1280
1.90.3 Description . . . 1281
1.90.4 Restrictions . . . 1282
1.90.5 Related commands . . . 1282
1.90.6 Default . . . 1282
1.91 rerun command . . . 1282
1.91.1 Syntax . . . 1282
1.91.2 Examples . . . 1283
1.91.3 Description . . . 1283
1.91.4 Restrictions . . . 1285
1.91.5 Related commands . . . 1285
1.91.6 Default . . . 1285
1.92 reset_atoms command . . . 1285
1.92.1 Syntax . . . 1285
1.92.2 Examples . . . 1286
1.92.3 Description . . . 1286
1.92.4 Restrictions . . . 1288
1.92.5 Related commands . . . 1288
1.92.6 Defaults . . . 1288
1.93 reset_timestep command . . . 1288
1.93.1 Syntax . . . 1288
1.93.2 Examples . . . 1289
1.93.3 Description . . . 1289
1.93.4 Restrictions . . . 1289
1.93.5 Related commands . . . 1289
1.93.6 Default . . . 1289
1.94 restart command . . . 1289
1.94.1 Syntax . . . 1289
1.94.2 Examples . . . 1290
1.94.3 Description . . . 1290
1.94.4 Restrictions . . . 1292
1.94.5 Related commands . . . 1292
1.94.6 Default . . . 1292
1.95 run command . . . 1292
1.95.1 Syntax . . . 1292
1.95.2 Examples . . . 1292
1.95.3 Description . . . 1293
1.95.4 Restrictions . . . 1295
1.95.5 Related commands . . . 1295
1.95.6 Default . . . 1295
1.96 run_style command . . . 1295
1.96.1 Syntax . . . 1295
1.96.2 Examples . . . 1296
1.96.3 Description . . . 1296
1.96.4 Restrictions . . . 1299
1.96.5 Related commands . . . 1299
1.96.6 Default . . . 1300
1.97 set command . . . 1300
1.97.1 Syntax . . . 1300
1.97.2 Examples . . . 1303
1.97.3 Description . . . 1303
1.97.4 Restrictions . . . 1309
1.97.5 Related commands . . . 1309

1.97.6 Default . . . 1309
1.98 shell command . . . 1309
1.98.1 Syntax . . . 1309
1.98.2 Examples . . . 1309
1.98.3 Description . . . 1310
1.98.4 Restrictions . . . 1310
1.98.5 Related commands . . . 1311
1.98.6 Default . . . 1311
1.99 special_bonds command . . . 1311
1.99.1 Syntax . . . 1311
1.99.2 Examples . . . 1311
1.99.3 Description . . . 1311
1.99.4 Restrictions . . . 1314
1.99.5 Related commands . . . 1314
1.99.6 Default . . . 1314
1.100 suffix command . . . 1314
1.100.1 Syntax . . . 1314
1.100.2 Examples . . . 1315
1.100.3 Description . . . 1315
1.100.4 Restrictions . . . 1316
1.100.5 Related commands . . . 1316
1.100.6 Default . . . 1316
1.101 tad command . . . 1316
1.101.1 Syntax . . . 1316
1.101.2 Examples . . . 1317
1.101.3 Description . . . 1317
1.101.4 Restrictions . . . 1319
1.101.5 Related commands . . . 1319
1.101.6 Default . . . 1320
1.102 temper command . . . 1320
1.102.1 Syntax . . . 1320
1.102.2 Examples . . . 1320
1.102.3 Description . . . 1320
1.102.4 Restrictions . . . 1322
1.102.5 Related commands . . . 1322
1.102.6 Default . . . 1322
1.103 temper/grem command . . . 1322
1.103.1 Syntax . . . 1322
1.103.2 Examples . . . 1322
1.103.3 Description . . . 1322
1.103.4 Restrictions . . . 1323
1.103.5 Related commands . . . 1323
1.103.6 Default . . . 1323
1.104 temper/npt command . . . 1324
1.104.1 Syntax . . . 1324
1.104.2 Examples . . . 1324
1.104.3 Description . . . 1324
1.104.4 Restrictions . . . 1324
1.104.5 Related commands . . . 1324
1.104.6 Default . . . 1324
1.105 thermo command . . . 1325
1.105.1 Syntax . . . 1325
1.105.2 Examples . . . 1325
1.105.3 Description . . . 1325

1.105.4 Restrictions . . . 1325
1.105.5 Related commands . . . 1325
1.105.6 Default . . . 1325
1.106 thermo_modify command . . . 1325
1.106.1 Syntax . . . 1325
1.106.2 Examples . . . 1326
1.106.3 Description . . . 1326
1.106.4 Restrictions . . . 1329
1.106.5 Related commands . . . 1329
1.106.6 Default . . . 1329
1.107 thermo_style command . . . 1329
1.107.1 Syntax . . . 1329
1.107.2 Examples . . . 1331
1.107.3 Description . . . 1331
1.107.4 Restrictions . . . 1336
1.107.5 Related commands . . . 1336
1.107.6 Default . . . 1336
1.108 third_order command . . . 1336
1.108.1 Syntax . . . 1336
1.108.2 Examples . . . 1336
1.108.3 Description . . . 1336
1.108.4 Restrictions . . . 1337
1.108.5 Related commands . . . 1337
1.108.6 Default . . . 1337
1.109 timer command . . . 1337
1.109.1 Syntax . . . 1337
1.109.2 Examples . . . 1337
1.109.3 Description . . . 1338
1.109.4 Restrictions . . . 1339
1.109.5 Related commands . . . 1339
1.109.6 Default . . . 1339
1.110 timestep command . . . 1339
1.110.1 Syntax . . . 1339
1.110.2 Examples . . . 1339
1.110.3 Description . . . 1339
1.110.4 Restrictions . . . 1339
1.110.5 Related commands . . . 1340
1.110.6 Default . . . 1340
1.111 uncompute command . . . 1340
1.111.1 Syntax . . . 1340
1.111.2 Examples . . . 1340
1.111.3 Description . . . 1340
1.111.4 Restrictions . . . 1340
1.111.5 Related commands . . . 1340
1.111.6 Default . . . 1340
1.112 undump command . . . 1341
1.112.1 Syntax . . . 1341
1.112.2 Examples . . . 1341
1.112.3 Description . . . 1341
1.112.4 Restrictions . . . 1341
1.112.5 Related commands . . . 1341
1.112.6 Default . . . 1341
1.113 unfix command . . . 1341
1.113.1 Syntax . . . 1341

1.113.2 Examples . . . 1341
1.113.3 Description . . . 1341
1.113.4 Restrictions . . . 1341
1.113.5 Related commands . . . 1342
1.113.6 Default . . . 1342
1.114 units command . . . 1342
1.114.1 Syntax . . . 1342
1.114.2 Examples . . . 1342
1.114.3 Description . . . 1342
1.114.4 Restrictions . . . 1346
1.114.5 Related commands . . . 1346
1.114.6 Default . . . 1346
1.115 variable command . . . 1346
1.115.1 Syntax . . . 1346
1.115.2 Examples . . . 1348
1.115.3 Description . . . 1349
1.115.4 Immediate Evaluation of Variables . . . 1367
1.115.5 Variable Accuracy . . . 1368
1.115.6 Restrictions . . . 1369
1.115.7 Related commands . . . 1369
1.115.8 Default . . . 1369
1.116 velocity command . . . 1369
1.116.1 Syntax . . . 1369
1.116.2 Examples . . . 1370
1.116.3 Description . . . 1370
1.116.4 Restrictions . . . 1372
1.116.5 Related commands . . . 1372
1.116.6 Default . . . 1373
1.117 write_coeff command . . . 1373
1.117.1 Syntax . . . 1373
1.117.2 Examples . . . 1373
1.117.3 Description . . . 1373
1.117.4 Restrictions . . . 1373
1.117.5 Related commands . . . 1373
1.118 write_data command . . . 1373
1.118.1 Syntax . . . 1373
1.118.2 Examples . . . 1374
1.118.3 Description . . . 1374
1.118.4 Restrictions . . . 1376
1.118.5 Related commands . . . 1376
1.118.6 Default . . . 1376
1.119 write_dump command . . . 1376
1.119.1 Syntax . . . 1376
1.119.2 Examples . . . 1376
1.119.3 Description . . . 1377
1.119.4 Restrictions . . . 1377
1.119.5 Related commands . . . 1377
1.119.6 Default . . . 1378
1.120 write_molecule command . . . 1378
1.120.1 Syntax . . . 1378
1.120.2 Examples . . . 1378
1.120.3 Description . . . 1378
1.120.4 Output Format . . . 1378
1.120.5 Restrictions . . . 1380

1.120.6 Related commands . . . 1380
1.120.7 Defaults . . . 1380
1.121 write_restart command . . . 1381
1.121.1 Syntax . . . 1381
1.121.2 Examples . . . 1381
1.121.3 Description . . . 1381
1.121.4 Restrictions . . . 1382
1.121.5 Related commands . . . 1382
1.121.6 Default . . . 1382

**2 Fix Styles 1383**
2.1 fix accelerate/cos command . . . 1383
2.1.1 Syntax . . . 1383
2.1.2 Examples . . . 1383
2.1.3 Description . . . 1383
2.1.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1384
2.1.5 Restrictions . . . 1384
2.1.6 Related commands . . . 1384
2.1.7 Default . . . 1384
2.2 fix acks2/reaxff command . . . 1384
2.2.1 Syntax . . . 1384
2.2.2 Examples . . . 1385
2.2.3 Description . . . 1385
2.2.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1385
2.2.5 Restrictions . . . 1386
2.2.6 Related commands . . . 1386
2.2.7 Default . . . 1386
2.3 fix adapt command . . . 1386
2.3.1 Syntax . . . 1386
2.3.2 Examples . . . 1387
2.3.3 Description . . . 1388
2.3.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1394
2.3.5 Restrictions . . . 1394
2.3.6 Related commands . . . 1394
2.3.7 Default . . . 1394
2.4 fix adapt/fep command . . . 1394
2.4.1 Syntax . . . 1394
2.4.2 Examples . . . 1395
2.4.3 Description . . . 1395
2.4.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1398
2.4.5 Restrictions . . . 1398
2.4.6 Related commands . . . 1398
2.4.7 Default . . . 1398
2.5 fix add/heat command . . . 1398
2.5.1 Syntax . . . 1398
2.5.2 Examples . . . 1399
2.5.3 Description . . . 1399
2.5.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1399
2.5.5 Restrictions . . . 1399
2.5.6 Related commands . . . 1400
2.5.7 Default . . . 1400
2.6 fix addforce command . . . 1400
2.6.1 Syntax . . . 1400
2.6.2 Examples . . . 1400

2.6.3 Description . . . 1400
2.6.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1402
2.6.5 Restrictions . . . 1402
2.6.6 Related commands . . . 1402
2.6.7 Default . . . 1402
2.7 fix addtorque/atom command . . . 1402
2.7.1 Syntax . . . 1402
2.7.2 Examples . . . 1403
2.7.3 Description . . . 1403
2.7.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1403
2.7.5 Restrictions . . . 1404
2.7.6 Related commands . . . 1404
2.7.7 Default . . . 1404
2.8 fix addtorque/group command . . . 1404
2.8.1 Syntax . . . 1404
2.8.2 Examples . . . 1404
2.8.3 Description . . . 1404
2.8.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1405
2.8.5 Restrictions . . . 1405
2.8.6 Related commands . . . 1406
2.8.7 Default . . . 1406
2.9 fix alchemy command . . . 1406
2.9.1 Syntax . . . 1406
2.9.2 Examples . . . 1406
2.9.3 Description . . . 1406
2.9.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1407
2.9.5 Restrictions . . . 1408
2.9.6 Related commands . . . 1408
2.9.7 Default . . . 1408
2.10 fix align/self command . . . 1408
2.10.1 Syntax . . . 1408
2.10.2 Examples . . . 1408
2.10.3 Description . . . 1408
2.10.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1409
2.10.5 Restrictions . . . 1409
2.10.6 Related commands . . . 1409
2.10.7 Default . . . 1409
2.11 fix amoeba/bitorsion command . . . 1410
2.11.1 Syntax . . . 1410
2.11.2 Examples . . . 1410
2.11.3 Description . . . 1410
2.11.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1411
2.11.5 Restrictions . . . 1411
2.11.6 Related commands . . . 1411
2.11.7 Default . . . 1412
2.12 fix amoeba/pitorsion command . . . 1412
2.12.1 Syntax . . . 1412
2.12.2 Examples . . . 1412
2.12.3 Description . . . 1412
2.12.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1413
2.12.5 Restrictions . . . 1414
2.12.6 Related commands . . . 1414
2.12.7 Default . . . 1414
2.13 fix append/atoms command . . . 1414

2.13.1 Syntax . . . 1414
2.13.2 Examples . . . 1415
2.13.3 Description . . . 1415
2.13.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1415
2.13.5 Restrictions . . . 1415
2.13.6 Related commands . . . 1415
2.13.7 Default . . . 1415
2.14 fix atc command . . . 1416
2.15 fix atom/swap command . . . 1416
2.15.1 Syntax . . . 1416
2.15.2 Examples . . . 1416
2.15.3 Description . . . 1416
2.15.4 Dump image info . . . 1418
2.15.5 Restart, fix_modify, output, run start/stop, minimize info . . . 1418
2.15.6 Restrictions . . . 1419
2.15.7 Related commands . . . 1419
2.15.8 Default . . . 1419
2.16 fix atom_weight/apip command . . . 1419
2.16.1 Syntax . . . 1419
2.16.2 Examples . . . 1419
2.16.3 Description . . . 1420
2.16.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1420
2.16.5 Restrictions . . . 1421
2.16.6 Related commands . . . 1421
2.16.7 Default . . . 1421
2.17 fix ave/atom command . . . 1421
2.17.1 Syntax . . . 1421
2.17.2 Examples . . . 1421
2.17.3 Description . . . 1422
2.17.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1423
2.17.5 Restrictions . . . 1423
2.17.6 Related commands . . . 1423
2.17.7 Default . . . 1423
2.18 fix ave/chunk command . . . 1423
2.18.1 Syntax . . . 1423
2.18.2 Examples . . . 1425
2.18.3 Description . . . 1425
2.18.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1430
2.18.5 Restrictions . . . 1430
2.18.6 Related commands . . . 1430
2.18.7 Default . . . 1430
2.19 fix ave/correlate command . . . 1430
2.19.1 Syntax . . . 1430
2.19.2 Examples . . . 1431
2.19.3 Description . . . 1431
2.19.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1434
2.19.5 Restrictions . . . 1435
2.19.6 Related commands . . . 1435
2.19.7 Default . . . 1435
2.20 fix ave/correlate/long command . . . 1435
2.20.1 Syntax . . . 1435
2.20.2 Examples . . . 1436
2.20.3 Description . . . 1436
2.20.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1437

2.20.5 Restrictions . . . 1437
2.20.6 Related commands . . . 1437
2.20.7 Default . . . 1437
2.21 fix ave/grid command . . . 1438
2.21.1 Syntax . . . 1438
2.21.2 Examples . . . 1439
2.21.3 Description . . . 1439
2.21.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1443
2.21.5 Restrictions . . . 1444
2.21.6 Related commands . . . 1444
2.21.7 Default . . . 1444
2.22 fix ave/histo command . . . 1444
2.23 fix ave/histo/weight command . . . 1444
2.23.1 Syntax . . . 1444
2.23.2 Examples . . . 1445
2.23.3 Description . . . 1445
2.23.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1448
2.23.5 Restrictions . . . 1449
2.23.6 Related commands . . . 1449
2.23.7 Default . . . 1449
2.24 fix ave/moments command . . . 1449
2.24.1 Syntax . . . 1449
2.24.2 Examples . . . 1450
2.24.3 Description . . . 1450
2.24.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1452
2.24.5 Restrictions . . . 1452
2.24.6 Related commands . . . 1452
2.24.7 Default . . . 1453
2.25 fix ave/spatial command . . . 1453
2.26 fix ave/spatial/sphere command . . . 1453
2.27 fix ave/time command . . . 1453
2.27.1 Syntax . . . 1453
2.27.2 Examples . . . 1454
2.27.3 Description . . . 1454
2.27.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1457
2.27.5 Restrictions . . . 1458
2.27.6 Related commands . . . 1458
2.27.7 Default . . . 1458
2.28 fix aveforce command . . . 1458
2.28.1 Syntax . . . 1458
2.28.2 Examples . . . 1458
2.28.3 Description . . . 1458
2.28.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1459
2.28.5 Restrictions . . . 1459
2.28.6 Related commands . . . 1459
2.28.7 Default . . . 1459
2.29 fix balance command . . . 1460
2.29.1 Syntax . . . 1460
2.29.2 Examples . . . 1460
2.29.3 Description . . . 1461
2.29.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1465
2.29.5 Restrictions . . . 1465
2.29.6 Related commands . . . 1465
2.29.7 Default . . . 1465

2.30 fix baoab command . . . 1465
2.30.1 Syntax . . . 1465
2.30.2 Examples . . . 1466
2.30.3 Description . . . 1466
2.30.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1467
2.30.5 Restrictions . . . 1467
2.30.6 Related commands . . . 1467
2.30.7 Default . . . 1467
2.31 fix bocs command . . . 1467
2.31.1 Syntax . . . 1467
2.31.2 Examples . . . 1468
2.31.3 Description . . . 1468
2.31.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1469
2.31.5 Restrictions . . . 1469
2.31.6 Further information . . . 1469
2.32 fix bond/break command . . . 1469
2.32.1 Syntax . . . 1469
2.32.2 Examples . . . 1470
2.32.3 Description . . . 1470
2.32.4 Dump image info . . . 1471
2.32.5 Restart, fix_modify, output, run start/stop, minimize info . . . 1471
2.32.6 Restrictions . . . 1471
2.32.7 Related commands . . . 1472
2.32.8 Default . . . 1472
2.33 fix bond/create command . . . 1472
2.34 fix bond/create/angle command . . . 1472
2.34.1 Syntax . . . 1472
2.34.2 Examples . . . 1473
2.34.3 Description . . . 1473
2.34.4 Dump image info . . . 1475
2.34.5 Restart, fix_modify, output, run start/stop, minimize info . . . 1475
2.34.6 Restrictions . . . 1476
2.34.7 Related commands . . . 1476
2.34.8 Default . . . 1476
2.35 fix bond/react command . . . 1476
2.35.1 Syntax . . . 1476
2.35.2 Examples . . . 1478
2.35.3 Description . . . 1478
2.35.4 Dump image info . . . 1487
2.35.5 Restart, fix_modify, output, run start/stop, minimize info . . . 1488
2.35.6 Restrictions . . . 1489
2.35.7 Related commands . . . 1489
2.35.8 Default . . . 1489
2.36 fix bond/swap command . . . 1489
2.36.1 Syntax . . . 1489
2.36.2 Examples . . . 1489
2.36.3 Description . . . 1489
2.36.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1492
2.36.5 Restrictions . . . 1492
2.36.6 Related commands . . . 1492
2.36.7 Default . . . 1492
2.37 fix box/relax command . . . 1492
2.37.1 Syntax . . . 1492
2.37.2 Examples . . . 1493

2.37.3 Description . . . 1493
2.37.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1496
2.37.5 Restrictions . . . 1497
2.37.6 Related commands . . . 1497
2.37.7 Default . . . 1497
2.38 fix brownian command . . . 1497
2.39 fix brownian/sphere command . . . 1497
2.40 fix brownian/asphere command . . . 1497
2.40.1 Syntax . . . 1497
2.40.2 Examples . . . 1498
2.40.3 Description . . . 1498
2.40.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1500
2.40.5 Restrictions . . . 1500
2.40.6 Related commands . . . 1501
2.40.7 Default . . . 1501
2.41 fix charge/regulation command . . . 1501
2.41.1 Syntax . . . 1501
2.41.2 Examples . . . 1502
2.41.3 Description . . . 1502
2.41.4 Output . . . 1504
2.41.5 Restrictions . . . 1504
2.41.6 Related commands . . . 1504
2.41.7 Default . . . 1505
2.42 fix cmap command . . . 1505
2.42.1 Syntax . . . 1505
2.42.2 Examples . . . 1505
2.42.3 Description . . . 1505
2.42.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1506
2.42.5 Restrictions . . . 1507
2.42.6 Related commands . . . 1507
2.42.7 Default . . . 1507
2.43 fix colvars command . . . 1507
2.43.1 Syntax . . . 1507
2.43.2 Examples . . . 1508
2.43.3 Description . . . 1508
2.43.4 Restarting . . . 1509
2.43.5 Output . . . 1509
2.43.6 Controlling Colvars via *fix_modify* . . . 1509
2.43.7 Restrictions . . . 1510
2.43.8 Related commands . . . 1510
2.44 fix controller command . . . 1510
2.44.1 Syntax . . . 1510
2.44.2 Examples . . . 1511
2.44.3 Description . . . 1511
2.44.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1513
2.44.5 Restrictions . . . 1513
2.44.6 Related commands . . . 1513
2.44.7 Default . . . 1513
2.45 fix damping/cundall command . . . 1513
2.45.1 Syntax . . . 1513
2.45.2 Examples . . . 1514
2.45.3 Description . . . 1514
2.45.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1515
2.45.5 Restrictions . . . 1515

2.45.6 Related commands . . . 1515
2.45.7 Default . . . 1515
2.45.8 References . . . 1515
2.46 fix deform command . . . 1515
2.46.1 Syntax . . . 1515
2.46.2 Examples . . . 1517
2.46.3 Description . . . 1517
2.46.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1523
2.46.5 Restrictions . . . 1524
2.46.6 Related commands . . . 1524
2.46.7 Default . . . 1524
2.47 fix deform/pressure command . . . 1524
2.47.1 Syntax . . . 1524
2.47.2 Examples . . . 1526
2.47.3 Description . . . 1526
2.47.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1528
2.47.5 Restrictions . . . 1529
2.47.6 Related commands . . . 1529
2.47.7 Default . . . 1529
2.48 fix deposit command . . . 1529
2.48.1 Syntax . . . 1529
2.48.2 Examples . . . 1530
2.48.3 Description . . . 1530
2.48.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1533
2.48.5 Restrictions . . . 1533
2.48.6 Related commands . . . 1533
2.48.7 Default . . . 1534
2.49 fix dpd/energy command . . . 1534
2.49.1 Syntax . . . 1534
2.49.2 Examples . . . 1534
2.49.3 Description . . . 1534
2.49.4 Restrictions . . . 1535
2.49.5 Related commands . . . 1535
2.49.6 Default . . . 1535
2.50 fix edpd/source command . . . 1535
2.51 fix tdpd/source command . . . 1535
2.51.1 Syntax . . . 1535
2.51.2 Examples . . . 1536
2.51.3 Description . . . 1536
2.51.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1536
2.51.5 Restrictions . . . 1536
2.51.6 Related commands . . . 1536
2.51.7 Default . . . 1536
2.52 fix drag command . . . 1537
2.52.1 Syntax . . . 1537
2.52.2 Examples . . . 1537
2.52.3 Description . . . 1537
2.52.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1537
2.52.5 Restrictions . . . 1538
2.52.6 Related commands . . . 1538
2.52.7 Default . . . 1538
2.53 fix drude command . . . 1538
2.53.1 Syntax . . . 1538
2.53.2 Examples . . . 1538

2.53.3 Description . . . 1538
2.53.4 Restrictions . . . 1538
2.53.5 Related commands . . . 1538
2.53.6 Default . . . 1539
2.54 fix drude/transform/direct command . . . 1539
2.55 fix drude/transform/inverse command . . . 1539
2.55.1 Syntax . . . 1539
2.55.2 Examples . . . 1539
2.55.3 Description . . . 1539
2.55.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1541
2.55.5 Restrictions . . . 1541
2.55.6 Related commands . . . 1541
2.55.7 Default . . . 1541
2.56 fix dt/reset command . . . 1541
2.56.1 Syntax . . . 1541
2.56.2 Examples . . . 1542
2.56.3 Description . . . 1542
2.56.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1543
2.56.5 Restrictions . . . 1543
2.56.6 Related commands . . . 1543
2.56.7 Default . . . 1543
2.57 fix efield command . . . 1543
2.58 fix efield/tip4p command . . . 1543
2.58.1 Syntax . . . 1543
2.58.2 Examples . . . 1544
2.58.3 Description . . . 1544
2.58.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1545
2.58.5 Restrictions . . . 1546
2.58.6 Related commands . . . 1546
2.58.7 Default . . . 1546
2.59 fix efield/lepton command . . . 1546
2.59.1 Syntax . . . 1546
2.59.2 Examples . . . 1547
2.59.3 Description . . . 1547
2.59.4 Lepton expression syntax and features . . . 1547
2.59.5 Restart, fix_modify, output, run start/stop, minimize info . . . 1549
2.59.6 Restrictions . . . 1549
2.59.7 Related commands . . . 1549
2.59.8 Default . . . 1549
2.60 fix ehex command . . . 1549
2.60.1 Syntax . . . 1549
2.60.2 Examples . . . 1550
2.60.3 Description . . . 1550
2.60.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1552
2.60.5 Restrictions . . . 1552
2.60.6 Related commands . . . 1552
2.60.7 Default . . . 1552
2.61 fix electrode/conp command . . . 1552
2.62 fix electrode/conq command . . . 1552
2.63 fix electrode/thermo command . . . 1552
2.63.1 Syntax . . . 1552
2.63.2 Examples . . . 1553
2.63.3 Description . . . 1553
2.63.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1556

2.63.5 Restrictions . . . 1557
2.63.6 Default . . . 1557
2.64 fix electron/stopping command . . . 1558
2.65 fix electron/stopping/fit command . . . 1558
2.65.1 Syntax . . . 1558
2.65.2 Examples . . . 1558
2.65.3 Description . . . 1558
2.65.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1560
2.65.5 Restrictions . . . 1560
2.65.6 Default . . . 1561
2.66 fix enforce2d command . . . 1561
2.66.1 Syntax . . . 1561
2.66.2 Examples . . . 1561
2.66.3 Description . . . 1561
2.66.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1562
2.66.5 Restrictions . . . 1562
2.66.6 Related commands . . . 1562
2.66.7 Default . . . 1562
2.67 fix eos/cv command . . . 1562
2.67.1 Syntax . . . 1562
2.67.2 Examples . . . 1562
2.67.3 Description . . . 1562
2.67.4 Restrictions . . . 1562
2.67.5 Related commands . . . 1563
2.67.6 Default . . . 1563
2.68 fix eos/table command . . . 1563
2.68.1 Syntax . . . 1563
2.68.2 Examples . . . 1563
2.68.3 Description . . . 1563
2.68.4 Restrictions . . . 1564
2.68.5 Related commands . . . 1564
2.68.6 Default . . . 1564
2.69 fix eos/table/rx command . . . 1564
2.69.1 Syntax . . . 1565
2.69.2 Examples . . . 1565
2.69.3 Description . . . 1565
2.69.4 Restrictions . . . 1567
2.69.5 Related commands . . . 1567
2.69.6 Default . . . 1567
2.70 fix evaporate command . . . 1567
2.70.1 Syntax . . . 1567
2.70.2 Examples . . . 1568
2.70.3 Description . . . 1568
2.70.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1568
2.70.5 Restrictions . . . 1569
2.70.6 Related commands . . . 1569
2.70.7 Default . . . 1569
2.71 fix external command . . . 1569
2.71.1 Syntax . . . 1569
2.71.2 Examples . . . 1569
2.71.3 Description . . . 1569
2.71.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1571
2.71.5 Restrictions . . . 1571
2.71.6 Related commands . . . 1572

2.71.7 Default . . . 1572
2.72 fix ffl command . . . 1572
2.72.1 Syntax . . . 1572
2.72.2 Examples . . . 1572
2.72.3 Description . . . 1572
2.72.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1573
2.72.5 Restrictions . . . 1573
2.72.6 Related commands . . . 1573
2.73 fix filter/corotate command . . . 1573
2.73.1 Syntax . . . 1573
2.73.2 Examples . . . 1574
2.73.3 Description . . . 1574
2.73.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1574
2.73.5 Restrictions . . . 1574
2.73.6 Related commands . . . 1574
2.73.7 Default . . . 1574
2.74 fix flow/gauss command . . . 1574
2.74.1 Syntax . . . 1574
2.74.2 Examples . . . 1575
2.74.3 Description . . . 1575
2.74.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1576
2.74.5 Restrictions . . . 1576
2.74.6 Related commands . . . 1576
2.74.7 Default . . . 1576
2.75 fix freeze command . . . 1577
2.75.1 Syntax . . . 1577
2.75.2 Examples . . . 1577
2.75.3 Description . . . 1577
2.75.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1577
2.75.5 Restrictions . . . 1578
2.75.6 Related commands . . . 1578
2.75.7 Default . . . 1578
2.76 fix gcmc command . . . 1578
2.76.1 Syntax . . . 1578
2.76.2 Examples . . . 1579
2.76.3 Description . . . 1579
2.76.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1583
2.76.5 Restrictions . . . 1584
2.76.6 Related commands . . . 1584
2.76.7 Defaults . . . 1584
2.77 fix gemc command . . . 1584
2.77.1 Syntax . . . 1584
2.77.2 Examples . . . 1585
2.77.3 Description . . . 1585
2.77.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1586
2.77.5 Restrictions . . . 1587
2.77.6 Related commands . . . 1587
2.77.7 Defaults . . . 1587
2.78 fix gjf command . . . 1587
2.78.1 Syntax . . . 1587
2.78.2 Examples . . . 1587
2.78.3 Description . . . 1588
2.78.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1589
2.78.5 Restrictions . . . 1589

2.78.6 Related commands . . . 1589
2.78.7 Default . . . 1589
2.79 fix gld command . . . 1589
2.79.1 Syntax . . . 1589
2.79.2 Examples . . . 1590
2.79.3 Description . . . 1590
2.79.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1592
2.79.5 Restrictions . . . 1592
2.79.6 Related commands . . . 1592
2.79.7 Default . . . 1592
2.80 fix gle command . . . 1592
2.80.1 Syntax . . . 1592
2.80.2 Examples . . . 1593
2.80.3 Description . . . 1593
2.80.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1593
2.80.5 Restrictions . . . 1594
2.80.6 Related commands . . . 1594
2.81 fix graphics/arrows command . . . 1594
2.81.1 Syntax . . . 1594
2.81.2 Examples . . . 1595
2.81.3 Description . . . 1595
2.81.4 Dump image info . . . 1596
2.81.5 Restart, fix_modify, output, run start/stop, minimize info . . . 1596
2.81.6 Restrictions . . . 1597
2.81.7 Related commands . . . 1597
2.81.8 Default . . . 1597
2.82 fix graphics/chunk command . . . 1597
2.82.1 Syntax . . . 1597
2.82.2 Examples . . . 1597
2.82.3 Description . . . 1598
2.82.4 Usage example . . . 1599
2.82.5 Dump image info . . . 1600
2.82.6 Restart, fix_modify, output, run start/stop, minimize info . . . 1600
2.82.7 Restrictions . . . 1600
2.82.8 Related commands . . . 1600
2.82.9 Defaults . . . 1600
2.83 fix graphics/isosurface command . . . 1601
2.83.1 Syntax . . . 1601
2.83.2 Examples . . . 1601
2.83.3 Description . . . 1601
2.83.4 Dump image info . . . 1602
2.83.5 Example for using STL output in VMD . . . 1603
2.83.6 Restart, fix_modify, output, run start/stop, minimize info . . . 1604
2.83.7 Restrictions . . . 1604
2.83.8 Related commands . . . 1604
2.83.9 Defaults . . . 1604
2.84 fix graphics/labels command . . . 1604
2.84.1 Syntax . . . 1604
2.84.2 Examples . . . 1606
2.84.3 Description . . . 1607
2.84.4 Dump image info . . . 1609
2.84.5 Including a “dynamic” image . . . 1610
2.84.6 Restart, fix_modify, output, run start/stop, minimize info . . . 1612
2.84.7 Restrictions . . . 1612

2.84.8 Related commands . . . 1612
2.84.9 Default . . . 1612
2.85 fix graphics/lines command . . . 1612
2.85.1 Syntax . . . 1612
2.85.2 Examples . . . 1612
2.85.3 Description . . . 1612
2.85.4 Dump image info . . . 1613
2.85.5 Restart, fix_modify, output, run start/stop, minimize info . . . 1614
2.85.6 Restrictions . . . 1614
2.85.7 Related commands . . . 1614
2.85.8 Defaults . . . 1614
2.86 fix graphics/objects command . . . 1614
2.86.1 Syntax . . . 1614
2.86.2 Examples . . . 1615
2.86.3 Description . . . 1615
2.86.4 Dump image info . . . 1616
2.86.5 Restart, fix_modify, output, run start/stop, minimize info . . . 1618
2.86.6 Restrictions . . . 1618
2.86.7 Related commands . . . 1618
2.86.8 Default . . . 1618
2.87 fix graphics/periodic command . . . 1618
2.87.1 Syntax . . . 1618
2.87.2 Examples . . . 1618
2.87.3 Description . . . 1618
2.87.4 Dump image info . . . 1619
2.87.5 Restart, fix_modify, output, run start/stop, minimize info . . . 1619
2.87.6 Restrictions . . . 1619
2.87.7 Related commands . . . 1619
2.87.8 Default . . . 1619
2.88 fix graphics/replica command . . . 1620
2.88.1 Syntax . . . 1620
2.88.2 Examples . . . 1620
2.88.3 Description . . . 1620
2.88.4 Dump image info . . . 1620
2.88.5 Usage example . . . 1621
2.88.6 Restart, fix_modify, output, run start/stop, minimize info . . . 1622
2.88.7 Restrictions . . . 1622
2.88.8 Related commands . . . 1622
2.88.9 Default . . . 1622
2.89 fix gravity command . . . 1622
2.89.1 Syntax . . . 1622
2.89.2 Examples . . . 1623
2.89.3 Description . . . 1623
2.89.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1624
2.89.5 Restrictions . . . 1624
2.89.6 Related commands . . . 1624
2.89.7 Default . . . 1624
2.90 fix grem command . . . 1624
2.90.1 Syntax . . . 1624
2.90.2 Examples . . . 1625
2.90.3 Description . . . 1625
2.90.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1625
2.90.5 Restrictions . . . 1625
2.90.6 Related commands . . . 1626

2.90.7 Default . . . 1626
2.91 fix halt command . . . 1626
2.91.1 Syntax . . . 1626
2.91.2 Examples . . . 1626
2.91.3 Description . . . 1626
2.91.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1629
2.91.5 Restrictions . . . 1629
2.91.6 Related commands . . . 1629
2.91.7 Default . . . 1629
2.92 fix heat command . . . 1629
2.92.1 Syntax . . . 1629
2.92.2 Examples . . . 1629
2.92.3 Description . . . 1629
2.92.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1630
2.92.5 Restrictions . . . 1631
2.92.6 Related commands . . . 1631
2.92.7 Default . . . 1631
2.93 fix heat/flow command . . . 1631
2.93.1 Syntax . . . 1631
2.93.2 Examples . . . 1631
2.93.3 Description . . . 1631
2.93.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1631
2.93.5 Restrictions . . . 1632
2.93.6 Related commands . . . 1632
2.93.7 Default . . . 1632
2.94 fix hmc command . . . 1632
2.94.1 Syntax . . . 1632
2.94.2 Examples . . . 1632
2.94.3 Description . . . 1632
2.94.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1634
2.94.5 Restrictions . . . 1635
2.94.6 Related commands . . . 1635
2.94.7 Default . . . 1635
2.95 fix hyper/global command . . . 1635
2.95.1 Syntax . . . 1635
2.95.2 Examples . . . 1635
2.95.3 Description . . . 1635
2.95.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1637
2.95.5 Restrictions . . . 1638
2.95.6 Related commands . . . 1638
2.95.7 Default . . . 1638
2.96 fix hyper/local command . . . 1639
2.96.1 Syntax . . . 1639
2.96.2 Examples . . . 1639
2.96.3 Description . . . 1639
2.96.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1643
2.96.5 Restrictions . . . 1645
2.96.6 Related commands . . . 1645
2.96.7 Default . . . 1645
2.97 fix ilves command . . . 1646
2.97.1 Syntax . . . 1646
2.97.2 Examples . . . 1646
2.97.3 Description . . . 1646
2.97.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1649

2.97.5 Restrictions . . . 1649
2.97.6 Related commands . . . 1650
2.97.7 Default . . . 1650
2.98 fix imd command . . . 1650
2.98.1 Syntax . . . 1650
2.98.2 Examples . . . 1651
2.98.3 Description . . . 1651
2.98.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1652
2.98.5 Restrictions . . . 1652
2.98.6 Related commands . . . 1652
2.98.7 Default . . . 1652
2.99 fix indent command . . . 1653
2.99.1 Syntax . . . 1653
2.99.2 Examples . . . 1653
2.99.3 Description . . . 1654
2.99.4 Dump image info . . . 1655
2.99.5 Restart, fix_modify, output, run start/stop, minimize info . . . 1656
2.99.6 Restrictions . . . 1656
2.99.7 Related commands . . . 1656
2.99.8 Default . . . 1656
2.100 fix ipi command . . . 1656
2.100.1 Syntax . . . 1656
2.100.2 Examples . . . 1657
2.100.3 Description . . . 1657
2.100.4 Obtaining i-PI . . . 1657
2.100.5 Restart, fix_modify, output, run start/stop, minimize info . . . 1657
2.100.6 Restrictions . . . 1657
2.100.7 Related commands . . . 1658
2.101 fix lambda/apip command . . . 1658
2.101.1 Syntax . . . 1658
2.101.2 Examples . . . 1659
2.101.3 Description . . . 1659
2.101.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1661
2.101.5 Restrictions . . . 1661
2.101.6 Related commands . . . 1661
2.101.7 Default . . . 1661
2.102 fix lambda/la/csp/apip command . . . 1661
2.102.1 Syntax . . . 1661
2.102.2 Examples . . . 1662
2.102.3 Description . . . 1662
2.102.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1663
2.102.5 Restrictions . . . 1664
2.102.6 Related commands . . . 1664
2.102.7 Default . . . 1664
2.103 fix lambda_thermostat/apip command . . . 1664
2.103.1 Syntax . . . 1664
2.103.2 Examples . . . 1664
2.103.3 Description . . . 1664
2.103.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1665
2.103.5 Restrictions . . . 1666
2.103.6 Related commands . . . 1666
2.103.7 Default . . . 1666
2.104 fix langevin command . . . 1666
2.104.1 Syntax . . . 1666

2.104.2 Examples . . . 1667
2.104.3 Description . . . 1667
2.104.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1669
2.104.5 Restrictions . . . 1670
2.104.6 Related commands . . . 1670
2.104.7 Default . . . 1670
2.105 fix langevin/drude command . . . 1670
2.105.1 Syntax . . . 1670
2.105.2 Examples . . . 1671
2.105.3 Description . . . 1671
2.105.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1674
2.105.5 Restrictions . . . 1674
2.105.6 Related commands . . . 1674
2.105.7 Default . . . 1674
2.106 fix langevin/eff command . . . 1674
2.106.1 Syntax . . . 1674
2.106.2 Examples . . . 1675
2.106.3 Description . . . 1675
2.106.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1675
2.106.5 Restrictions . . . 1675
2.106.6 Related commands . . . 1676
2.106.7 Default . . . 1676
2.107 fix langevin/spin command . . . 1676
2.107.1 Syntax . . . 1676
2.107.2 Examples . . . 1676
2.107.3 Description . . . 1676
2.107.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1677
2.107.5 Restrictions . . . 1677
2.107.6 Related commands . . . 1677
2.107.7 Default . . . 1677
2.108 fix lb/fluid command . . . 1677
2.108.1 Syntax . . . 1677
2.108.2 Examples . . . 1678
2.108.3 Description . . . 1679
2.108.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1682
2.108.5 Restrictions . . . 1682
2.108.6 Related commands . . . 1682
2.108.7 Default . . . 1683
2.109 fix lb/momentum command . . . 1683
2.109.1 Syntax . . . 1683
2.109.2 Examples . . . 1683
2.109.3 Description . . . 1683
2.109.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1684
2.109.5 Restrictions . . . 1684
2.109.6 Related commands . . . 1684
2.109.7 Default . . . 1684
2.110 fix lb/viscous command . . . 1684
2.110.1 Syntax . . . 1684
2.110.2 Examples . . . 1684
2.110.3 Description . . . 1684
2.110.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1685
2.110.5 Restrictions . . . 1685
2.110.6 Related commands . . . 1685
2.110.7 Default . . . 1685

2.111 fix lineforce command . . . 1685
2.111.1 Syntax . . . 1685
2.111.2 Examples . . . 1685
2.111.3 Description . . . 1685
2.111.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1686
2.111.5 Restrictions . . . 1686
2.111.6 Related commands . . . 1686
2.111.7 Default . . . 1686
2.112 fix manifoldforce command . . . 1686
2.112.1 Syntax . . . 1686
2.112.2 Examples . . . 1686
2.112.3 Description . . . 1687
2.112.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1687
2.112.5 Restrictions . . . 1687
2.112.6 Related commands . . . 1687
2.113 fix mdi/qm command . . . 1687
2.113.1 Syntax . . . 1687
2.113.2 Examples . . . 1688
2.113.3 Description . . . 1688
2.113.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1690
2.113.5 Restrictions . . . 1690
2.113.6 Related commands . . . 1691
2.113.7 Default . . . 1691
2.114 fix mdi/qmmm command . . . 1691
2.114.1 Syntax . . . 1691
2.114.2 Examples . . . 1691
2.114.3 Description . . . 1691
2.114.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1693
2.114.5 Restrictions . . . 1694
2.114.6 Related commands . . . 1694
2.114.7 Default . . . 1694
2.115 fix meso/move command . . . 1694
2.115.1 Syntax . . . 1694
2.115.2 Examples . . . 1695
2.115.3 Description . . . 1695
2.115.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1697
2.115.5 Restrictions . . . 1697
2.115.6 Related commands . . . 1698
2.115.7 Default . . . 1698
2.116 fix mol/swap command . . . 1698
2.116.1 Syntax . . . 1698
2.116.2 Examples . . . 1698
2.116.3 Description . . . 1698
2.116.4 Dump image info . . . 1699
2.116.5 Restart, fix_modify, output, run start/stop, minimize info . . . 1700
2.116.6 Restrictions . . . 1700
2.116.7 Related commands . . . 1700
2.116.8 Default . . . 1700
2.117 fix momentum command . . . 1700
2.118 fix momentum/chunk command . . . 1701
2.118.1 Syntax . . . 1701
2.118.2 Examples . . . 1701
2.118.3 Description . . . 1701
2.118.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1702

2.118.5 Restrictions . . . 1702
2.118.6 Related commands . . . 1702
2.118.7 Default . . . 1702
2.119 fix move command . . . 1702
2.119.1 Syntax . . . 1702
2.119.2 Examples . . . 1703
2.119.3 Description . . . 1703
2.119.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1705
2.119.5 Restrictions . . . 1706
2.119.6 Related commands . . . 1706
2.119.7 Default . . . 1706
2.120 fix msst command . . . 1706
2.120.1 Syntax . . . 1706
2.120.2 Examples . . . 1706
2.120.3 Description . . . 1706
2.120.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1707
2.120.5 Restrictions . . . 1708
2.120.6 Related commands . . . 1708
2.120.7 Default . . . 1708
2.121 fix mvv/dpd command . . . 1709
2.122 fix mvv/edpd command . . . 1709
2.123 fix mvv/tdpd command . . . 1709
2.123.1 Syntax . . . 1709
2.123.2 Examples . . . 1709
2.123.3 Description . . . 1709
2.123.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1710
2.123.5 Restrictions . . . 1710
2.123.6 Related commands . . . 1710
2.123.7 Default . . . 1710
2.124 fix neb command . . . 1710
2.124.1 Syntax . . . 1710
2.124.2 Examples . . . 1711
2.124.3 Description . . . 1711
2.124.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1713
2.124.5 Restrictions . . . 1713
2.124.6 Related commands . . . 1713
2.124.7 Default . . . 1713
2.125 fix neb/spin command . . . 1714
2.125.1 Syntax . . . 1714
2.125.2 Examples . . . 1714
2.125.3 Description . . . 1714
2.125.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1714
2.125.5 Restrictions . . . 1714
2.125.6 Related commands . . . 1714
2.125.7 Default . . . 1714
2.126 fix neighbor/swap command . . . 1715
2.126.1 Syntax . . . 1715
2.126.2 Examples . . . 1715
2.126.3 Description . . . 1715
2.126.4 Dump image info . . . 1717
2.126.5 Restart, fix_modify, output, run start/stop, minimize info . . . 1718
2.126.6 Restrictions . . . 1718
2.126.7 Related commands . . . 1718
2.126.8 Default . . . 1718

2.127 fix nvt command . . . 1718
2.128 fix npt command . . . 1718
2.129 fix nph command . . . 1719
2.129.1 Syntax . . . 1719
2.129.2 Examples . . . 1720
2.129.3 Description . . . 1720
2.129.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1726
2.129.5 Restrictions . . . 1727
2.129.6 Related commands . . . 1727
2.129.7 Default . . . 1727
2.130 fix nvt/eff command . . . 1728
2.131 fix npt/eff command . . . 1728
2.132 fix nph/eff command . . . 1728
2.132.1 Syntax . . . 1728
2.132.2 Examples . . . 1728
2.132.3 Description . . . 1729
2.132.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1729
2.132.5 Restrictions . . . 1729
2.132.6 Related commands . . . 1730
2.132.7 Default . . . 1730
2.133 fix nvt/uef command . . . 1730
2.134 fix npt/uef command . . . 1730
2.134.1 Syntax . . . 1730
2.134.2 Examples . . . 1730
2.134.3 Description . . . 1731
2.134.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1732
2.134.5 Restrictions . . . 1733
2.134.6 Related commands . . . 1733
2.134.7 Default . . . 1733
2.135 fix nonaffine/displacement command . . . 1733
2.135.1 Syntax . . . 1733
2.135.2 Examples . . . 1734
2.135.3 Description . . . 1734
2.135.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1735
2.135.5 Restrictions . . . 1735
2.135.6 Related commands . . . 1735
2.135.7 Default . . . 1735
2.136 fix nph/asphere command . . . 1735
2.136.1 Syntax . . . 1735
2.136.2 Examples . . . 1736
2.136.3 Description . . . 1736
2.136.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1737
2.136.5 Restrictions . . . 1737
2.136.6 Related commands . . . 1737
2.136.7 Default . . . 1737
2.137 fix nph/body command . . . 1737
2.137.1 Syntax . . . 1737
2.137.2 Examples . . . 1737
2.137.3 Description . . . 1738
2.137.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1738
2.137.5 Restrictions . . . 1739
2.137.6 Related commands . . . 1739
2.137.7 Default . . . 1739
2.138 fix nph/sphere command . . . 1739

2.138.1 Syntax . . . 1739
2.138.2 Examples . . . 1739
2.138.3 Description . . . 1739
2.138.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1740
2.138.5 Restrictions . . . 1741
2.138.6 Related commands . . . 1741
2.138.7 Default . . . 1741
2.139 fix nphug command . . . 1741
2.139.1 Syntax . . . 1741
2.139.2 Examples . . . 1742
2.139.3 Description . . . 1742
2.139.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1743
2.139.5 Restrictions . . . 1744
2.139.6 Related commands . . . 1744
2.139.7 Default . . . 1744
2.140 fix npt/asphere command . . . 1744
2.140.1 Syntax . . . 1744
2.140.2 Examples . . . 1744
2.140.3 Description . . . 1744
2.140.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1745
2.140.5 Restrictions . . . 1746
2.140.6 Related commands . . . 1746
2.140.7 Default . . . 1746
2.141 fix npt/body command . . . 1746
2.141.1 Syntax . . . 1746
2.141.2 Examples . . . 1746
2.141.3 Description . . . 1747
2.141.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1747
2.141.5 Restrictions . . . 1748
2.141.6 Related commands . . . 1748
2.141.7 Default . . . 1748
2.142 fix npt/cauchy command . . . 1748
2.142.1 Syntax . . . 1748
2.142.2 Examples . . . 1749
2.142.3 Description . . . 1749
2.142.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1754
2.142.5 Restrictions . . . 1755
2.142.6 Related commands . . . 1756
2.142.7 Default . . . 1756
2.143 fix npt/sphere command . . . 1756
2.143.1 Syntax . . . 1756
2.143.2 Examples . . . 1756
2.143.3 Description . . . 1756
2.143.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1758
2.143.5 Restrictions . . . 1758
2.143.6 Related commands . . . 1758
2.143.7 Default . . . 1758
2.144 fix numdiff command . . . 1758
2.144.1 Syntax . . . 1758
2.144.2 Examples . . . 1758
2.144.3 Description . . . 1759
2.144.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1759
2.144.5 Restrictions . . . 1760
2.144.6 Related commands . . . 1760

2.144.7 Default . . . 1760
2.145 fix numdiff/virial command . . . 1760
2.145.1 Syntax . . . 1760
2.145.2 Examples . . . 1760
2.145.3 Description . . . 1760
2.145.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1761
2.145.5 Restrictions . . . 1761
2.145.6 Related commands . . . 1761
2.145.7 Default . . . 1761
2.146 fix nve command . . . 1761
2.146.1 Syntax . . . 1761
2.146.2 Examples . . . 1762
2.146.3 Description . . . 1762
2.146.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1762
2.146.5 Restrictions . . . 1762
2.146.6 Related commands . . . 1762
2.146.7 Default . . . 1762
2.147 fix nve/asphere command . . . 1762
2.147.1 Syntax . . . 1763
2.147.2 Examples . . . 1763
2.147.3 Description . . . 1763
2.147.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1763
2.147.5 Restrictions . . . 1763
2.147.6 Related commands . . . 1763
2.147.7 Default . . . 1764
2.148 fix nve/asphere/noforce command . . . 1764
2.148.1 Syntax . . . 1764
2.148.2 Examples . . . 1764
2.148.3 Description . . . 1764
2.148.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1764
2.148.5 Restrictions . . . 1764
2.148.6 Related commands . . . 1764
2.148.7 Default . . . 1764
2.149 fix nve/body command . . . 1765
2.149.1 Syntax . . . 1765
2.149.2 Examples . . . 1765
2.149.3 Description . . . 1765
2.149.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1765
2.149.5 Restrictions . . . 1765
2.149.6 Related commands . . . 1765
2.149.7 Default . . . 1765
2.150 fix nve/bpm/sphere command . . . 1765
2.150.1 Syntax . . . 1765
2.150.2 Examples . . . 1766
2.150.3 Description . . . 1766
2.150.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1766
2.150.5 Restrictions . . . 1766
2.150.6 Related commands . . . 1767
2.150.7 Default . . . 1767
2.151 fix nve/dot command . . . 1767
2.151.1 Syntax . . . 1767
2.151.2 Examples . . . 1767
2.151.3 Description . . . 1767
2.151.4 Restrictions . . . 1767

2.151.5 Related commands . . . 1767
2.151.6 Default . . . 1767
2.152 fix nve/dotc/langevin command . . . 1768
2.152.1 Syntax . . . 1768
2.152.2 Examples . . . 1768
2.152.3 Description . . . 1768
2.152.4 Restrictions . . . 1769
2.152.5 Related commands . . . 1769
2.152.6 Default . . . 1769
2.153 fix nve/eff command . . . 1770
2.153.1 Syntax . . . 1770
2.153.2 Examples . . . 1770
2.153.3 Description . . . 1770
2.153.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1770
2.153.5 Restrictions . . . 1770
2.153.6 Related commands . . . 1770
2.153.7 Default . . . 1770
2.154 fix nve/limit command . . . 1770
2.154.1 Syntax . . . 1771
2.154.2 Examples . . . 1771
2.154.3 Description . . . 1771
2.154.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1771
2.154.5 Restrictions . . . 1772
2.154.6 Related commands . . . 1772
2.154.7 Default . . . 1772
2.155 fix nve/line command . . . 1772
2.155.1 Syntax . . . 1772
2.155.2 Examples . . . 1772
2.155.3 Description . . . 1772
2.155.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1772
2.155.5 Restrictions . . . 1773
2.155.6 Related commands . . . 1773
2.155.7 Default . . . 1773
2.156 fix nve/manifold/rattle command . . . 1773
2.156.1 Syntax . . . 1773
2.156.2 Examples . . . 1773
2.156.3 Description . . . 1773
2.156.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1774
2.156.5 Restrictions . . . 1774
2.156.6 Related commands . . . 1774
2.156.7 Default . . . 1774
2.157 fix nve/noforce command . . . 1774
2.157.1 Syntax . . . 1774
2.157.2 Examples . . . 1774
2.157.3 Description . . . 1775
2.157.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1775
2.157.5 Restrictions . . . 1775
2.157.6 Related commands . . . 1775
2.157.7 Default . . . 1775
2.158 fix nve/sphere command . . . 1775
2.158.1 Syntax . . . 1775
2.158.2 Examples . . . 1776
2.158.3 Description . . . 1776
2.158.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1776

2.158.5 Restrictions . . . 1777
2.158.6 Related commands . . . 1777
2.158.7 Default . . . 1777
2.159 fix nve/spin command . . . 1777
2.159.1 Syntax . . . 1777
2.159.2 Examples . . . 1777
2.159.3 Description . . . 1777
2.159.4 Restrictions . . . 1778
2.159.5 Related commands . . . 1778
2.159.6 Default . . . 1778
2.160 fix nve/tri command . . . 1778
2.160.1 Syntax . . . 1778
2.160.2 Examples . . . 1779
2.160.3 Description . . . 1779
2.160.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1779
2.160.5 Restrictions . . . 1779
2.160.6 Related commands . . . 1779
2.160.7 Default . . . 1779
2.161 fix nvk command . . . 1779
2.161.1 Syntax . . . 1779
2.161.2 Examples . . . 1779
2.161.3 Description . . . 1780
2.161.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1780
2.161.5 Restrictions . . . 1780
2.161.6 Related commands . . . 1780
2.161.7 Default . . . 1780
2.162 fix nvt/asphere command . . . 1780
2.162.1 Syntax . . . 1780
2.162.2 Examples . . . 1781
2.162.3 Description . . . 1781
2.162.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1782
2.162.5 Restrictions . . . 1782
2.162.6 Related commands . . . 1782
2.162.7 Default . . . 1782
2.163 fix nvt/body command . . . 1782
2.163.1 Syntax . . . 1782
2.163.2 Examples . . . 1782
2.163.3 Description . . . 1783
2.163.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1783
2.163.5 Restrictions . . . 1784
2.163.6 Related commands . . . 1784
2.163.7 Default . . . 1784
2.164 fix nvt/manifold/rattle command . . . 1784
2.164.1 Syntax . . . 1784
2.164.2 Examples . . . 1784
2.164.3 Description . . . 1784
2.164.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1785
2.164.5 Restrictions . . . 1785
2.164.6 Related commands . . . 1785
2.165 fix nvt/sllod command . . . 1785
2.165.1 Syntax . . . 1785
2.165.2 Examples . . . 1785
2.165.3 Description . . . 1786
2.165.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1788

2.165.5 Restrictions . . . 1789
2.165.6 Related commands . . . 1789
2.165.7 Default . . . 1789
2.166 fix nvt/sllod/eff command . . . 1789
2.166.1 Syntax . . . 1789
2.166.2 Examples . . . 1789
2.166.3 Description . . . 1790
2.166.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1790
2.166.5 Restrictions . . . 1790
2.166.6 Related commands . . . 1790
2.166.7 Default . . . 1790
2.167 fix nvt/sphere command . . . 1790
2.167.1 Syntax . . . 1791
2.167.2 Examples . . . 1791
2.167.3 Description . . . 1791
2.167.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1792
2.167.5 Restrictions . . . 1792
2.167.6 Related commands . . . 1792
2.167.7 Default . . . 1792
2.168 fix oneway command . . . 1793
2.168.1 Syntax . . . 1793
2.168.2 Examples . . . 1793
2.168.3 Description . . . 1793
2.168.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1793
2.168.5 Restrictions . . . 1794
2.168.6 Related commands . . . 1794
2.168.7 Default . . . 1794
2.169 fix orient/fcc command . . . 1794
2.170 fix orient/bcc command . . . 1794
2.170.1 Syntax . . . 1794
2.170.2 Examples . . . 1794
2.170.3 Description . . . 1794
2.170.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1796
2.170.5 Restrictions . . . 1796
2.170.6 Related commands . . . 1797
2.170.7 Default . . . 1797
2.171 fix orient/eco command . . . 1797
2.171.1 Examples . . . 1797
2.171.2 Description . . . 1798
2.171.3 Restart, fix_modify, output, run start/stop, minimize info . . . 1798
2.171.4 Restrictions . . . 1799
2.171.5 Related commands . . . 1799
2.171.6 Default . . . 1799
2.172 fix pafi command . . . 1800
2.172.1 Syntax . . . 1800
2.172.2 Examples . . . 1800
2.172.3 Description . . . 1800
2.172.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1801
2.172.5 Restrictions . . . 1801
2.172.6 Default . . . 1801
2.173 fix pair command . . . 1801
2.173.1 Syntax . . . 1801
2.173.2 Examples . . . 1801
2.173.3 Description . . . 1802

2.173.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1802
2.173.5 Restrictions . . . 1802
2.173.6 Related commands . . . 1802
2.173.7 Default . . . 1803
2.174 fix phonon command . . . 1803
2.174.1 Syntax . . . 1803
2.174.2 Examples . . . 1803
2.174.3 Description . . . 1803
2.174.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1804
2.174.5 Restrictions . . . 1805
2.174.6 Related commands . . . 1805
2.174.7 Default . . . 1805
2.175 fix pimd/langevin command . . . 1805
2.176 fix pimd/nvt command . . . 1805
2.177 fix pimd/langevin/bosonic command . . . 1805
2.178 fix pimd/nvt/bosonic command . . . 1805
2.178.1 Syntax . . . 1805
2.178.2 Examples . . . 1806
2.178.3 Description . . . 1806
2.178.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1811
2.178.5 Restrictions . . . 1814
2.178.6 Related commands . . . 1814
2.178.7 Default . . . 1814
2.179 fix planeforce command . . . 1815
2.179.1 Syntax . . . 1815
2.179.2 Examples . . . 1815
2.179.3 Description . . . 1815
2.179.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1815
2.179.5 Restrictions . . . 1816
2.179.6 Related commands . . . 1816
2.179.7 Default . . . 1816
2.180 fix plumed command . . . 1816
2.180.1 Syntax . . . 1816
2.180.2 Examples . . . 1816
2.180.3 Description . . . 1816
2.180.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1817
2.180.5 Restrictions . . . 1817
2.180.6 Related commands . . . 1817
2.180.7 Default . . . 1817
2.181 fix poems command . . . 1817
2.182 fix polarize/bem/gmres command . . . 1818
2.183 fix polarize/bem/icc command . . . 1818
2.184 fix polarize/functional command . . . 1818
2.184.1 Syntax . . . 1818
2.184.2 Examples . . . 1818
2.184.3 Description . . . 1818
2.184.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1819
2.184.5 Restrictions . . . 1820
2.184.6 Related commands . . . 1820
2.184.7 Default . . . 1820
2.185 fix pour command . . . 1821
2.185.1 Syntax . . . 1821
2.185.2 Examples . . . 1822
2.185.3 Description . . . 1822

2.185.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1824
2.185.5 Restrictions . . . 1824
2.185.6 Related commands . . . 1824
2.185.7 Default . . . 1824
2.186 fix precession/spin command . . . 1825
2.186.1 Syntax . . . 1825
2.186.2 Examples . . . 1825
2.186.3 Description . . . 1825
2.186.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1827
2.186.5 Restrictions . . . 1827
2.186.6 Related commands . . . 1827
2.186.7 Default . . . 1827
2.187 fix press/berendsen command . . . 1827
2.187.1 Syntax . . . 1827
2.187.2 Examples . . . 1828
2.187.3 Description . . . 1828
2.187.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1830
2.187.5 Restrictions . . . 1830
2.187.6 Related commands . . . 1830
2.187.7 Default . . . 1830
2.188 fix press/langevin command . . . 1830
2.188.1 Syntax . . . 1830
2.188.2 Examples . . . 1831
2.188.3 Description . . . 1831
2.188.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1834
2.188.5 Restrictions . . . 1834
2.188.6 Related commands . . . 1834
2.188.7 Default . . . 1834
2.189 fix print command . . . 1834
2.189.1 Syntax . . . 1834
2.189.2 Examples . . . 1835
2.189.3 Description . . . 1835
2.189.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1836
2.189.5 Restrictions . . . 1836
2.189.6 Related commands . . . 1836
2.189.7 Default . . . 1836
2.190 fix propel/self command . . . 1836
2.190.1 Syntax . . . 1836
2.190.2 Examples . . . 1836
2.190.3 Description . . . 1836
2.190.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1838
2.190.5 Restrictions . . . 1838
2.190.6 Related commands . . . 1838
2.190.7 Default . . . 1838
2.191 fix property/atom command . . . 1838
2.191.1 Syntax . . . 1838
2.191.2 Examples . . . 1839
2.191.3 Description . . . 1839
2.191.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1842
2.191.5 Restrictions . . . 1842
2.191.6 Related commands . . . 1842
2.191.7 Default . . . 1842
2.192 fix python/invoke command . . . 1842
2.192.1 Syntax . . . 1842

2.192.2 Examples . . . 1843
2.192.3 Description . . . 1843
2.192.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1843
2.192.5 Restrictions . . . 1844
2.192.6 Related commands . . . 1844
2.193 fix python/move command . . . 1844
2.193.1 Syntax . . . 1844
2.193.2 Examples . . . 1844
2.193.3 Description . . . 1844
2.193.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1845
2.193.5 Restrictions . . . 1845
2.193.6 Related commands . . . 1845
2.193.7 Default . . . 1845
2.194 fix qbmsst command . . . 1845
2.194.1 Syntax . . . 1845
2.194.2 Examples . . . 1846
2.194.3 Description . . . 1847
2.194.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1848
2.194.5 Restrictions . . . 1849
2.194.6 Related commands . . . 1849
2.194.7 Default . . . 1849
2.195 fix qeq/point command . . . 1849
2.196 fix qeq/shielded command . . . 1849
2.197 fix qeq/slater command . . . 1849
2.198 fix qeq/ctip command . . . 1849
2.199 fix qeq/dynamic command . . . 1849
2.200 fix qeq/fire command . . . 1850
2.200.1 Syntax . . . 1850
2.200.2 Examples . . . 1850
2.200.3 Description . . . 1851
2.200.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1853
2.200.5 Restrictions . . . 1854
2.200.6 Related commands . . . 1854
2.200.7 Default . . . 1854
2.201 fix qeq/comb command . . . 1854
2.201.1 Syntax . . . 1854
2.201.2 Examples . . . 1855
2.201.3 Description . . . 1855
2.201.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1855
2.201.5 Restrictions . . . 1856
2.201.6 Related commands . . . 1856
2.201.7 Default . . . 1856
2.202 fix qeq/reaxff command . . . 1856
2.202.1 Syntax . . . 1856
2.202.2 Examples . . . 1857
2.202.3 Description . . . 1857
2.202.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1857
2.202.5 Restrictions . . . 1858
2.202.6 Related commands . . . 1858
2.202.7 Default . . . 1858
2.203 fix qeq/rel/reaxff command . . . 1858
2.203.1 Syntax . . . 1858
2.203.2 Examples . . . 1859
2.203.3 Description . . . 1859

2.203.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1860
2.203.5 Restrictions . . . 1860
2.203.6 Related commands . . . 1861
2.203.7 Default . . . 1861
2.204 fix qmmm command . . . 1861
2.204.1 Syntax . . . 1861
2.204.2 Examples . . . 1861
2.204.3 Description . . . 1861
2.204.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1861
2.204.5 Restrictions . . . 1862
2.204.6 Related commands . . . 1862
2.204.7 Default . . . 1862
2.205 fix qtb command . . . 1862
2.205.1 Syntax . . . 1862
2.205.2 Examples . . . 1862
2.205.3 Description . . . 1862
2.205.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1864
2.205.5 Restrictions . . . 1864
2.205.6 Related commands . . . 1864
2.205.7 Default . . . 1864
2.206 fix qtpie/reaxff command . . . 1864
2.206.1 Syntax . . . 1864
2.206.2 Examples . . . 1865
2.206.3 Description . . . 1865
2.206.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1866
2.206.5 Restrictions . . . 1866
2.206.6 Related commands . . . 1866
2.206.7 Default . . . 1866
2.207 fix reaxff/bonds command . . . 1867
2.207.1 Syntax . . . 1867
2.207.2 Examples . . . 1867
2.207.3 Description . . . 1867
2.207.4 Dump image info . . . 1868
2.207.5 Restart, fix_modify, output, run start/stop, minimize info . . . 1868
2.207.6 Restrictions . . . 1869
2.207.7 Related commands . . . 1869
2.207.8 Default . . . 1869
2.208 fix reaxff/species command . . . 1869
2.208.1 Syntax . . . 1869
2.208.2 Examples . . . 1870
2.208.3 Description . . . 1870
2.208.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1872
2.208.5 Restrictions . . . 1873
2.208.6 Related commands . . . 1873
2.208.7 Default . . . 1873
2.209 fix recenter command . . . 1873
2.209.1 Syntax . . . 1873
2.209.2 Examples . . . 1874
2.209.3 Description . . . 1874
2.209.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1875
2.209.5 Restrictions . . . 1875
2.209.6 Related commands . . . 1875
2.209.7 Default . . . 1875
2.210 fix restrain command . . . 1875

2.210.1 Syntax . . . 1875
2.210.2 Examples . . . 1876
2.210.3 Description . . . 1876
2.210.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1878
2.210.5 Restrictions . . . 1879
2.210.6 Related commands . . . 1879
2.210.7 Default . . . 1879
2.211 fix rheo command . . . 1879
2.211.1 Syntax . . . 1879
2.211.2 Examples . . . 1880
2.211.3 Description . . . 1880
2.211.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1881
2.211.5 Restrictions . . . 1882
2.211.6 Related commands . . . 1882
2.211.7 Default . . . 1882
2.212 fix rheo/oxidation command . . . 1882
2.212.1 Syntax . . . 1882
2.212.2 Examples . . . 1882
2.212.3 Description . . . 1882
2.212.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1883
2.212.5 Restrictions . . . 1883
2.212.6 Related commands . . . 1883
2.212.7 Default . . . 1883
2.213 fix rheo/pressure command . . . 1883
2.213.1 Syntax . . . 1883
2.213.2 Examples . . . 1883
2.213.3 Description . . . 1884
2.213.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1884
2.213.5 Restrictions . . . 1884
2.213.6 Related commands . . . 1885
2.213.7 Default . . . 1885
2.214 fix rheo/thermal command . . . 1885
2.214.1 Syntax . . . 1885
2.214.2 Examples . . . 1885
2.214.3 Description . . . 1885
2.214.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1886
2.214.5 Restrictions . . . 1886
2.214.6 Related commands . . . 1886
2.214.7 Default . . . 1886
2.215 fix rheo/viscosity command . . . 1886
2.215.1 Syntax . . . 1886
2.215.2 Examples . . . 1887
2.215.3 Description . . . 1887
2.215.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1887
2.215.5 Restrictions . . . 1887
2.215.6 Related commands . . . 1888
2.215.7 Default . . . 1888
2.216 fix rhok command . . . 1888
2.216.1 Syntax . . . 1888
2.216.2 Examples . . . 1888
2.216.3 Description . . . 1888
2.216.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1888
2.216.5 Restrictions . . . 1889
2.216.6 Related commands . . . 1889

2.216.7 Default . . . 1889
2.217 fix rigid command . . . 1889
2.218 fix rigid/nve command . . . 1889
2.219 fix rigid/nvt command . . . 1889
2.220 fix rigid/npt command . . . 1889
2.221 fix rigid/nph command . . . 1889
2.222 fix rigid/small command . . . 1889
2.223 fix rigid/nve/small command . . . 1890
2.224 fix rigid/nvt/small command . . . 1890
2.225 fix rigid/npt/small command . . . 1890
2.226 fix rigid/nph/small command . . . 1890
2.226.1 Syntax . . . 1890
2.226.2 Examples . . . 1891
2.226.3 Description . . . 1891
2.226.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1899
2.226.5 Restrictions . . . 1900
2.226.6 Related commands . . . 1900
2.226.7 Default . . . 1901
2.227 fix rigid/meso command . . . 1901
2.227.1 Syntax . . . 1901
2.227.2 Examples . . . 1901
2.227.3 Description . . . 1902
2.227.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1904
2.227.5 Restrictions . . . 1905
2.227.6 Related commands . . . 1905
2.227.7 Default . . . 1905
2.228 fix rx command . . . 1905
2.228.1 Syntax . . . 1906
2.228.2 Examples . . . 1906
2.228.3 Description . . . 1906
2.228.4 Restrictions . . . 1908
2.228.5 Related commands . . . 1909
2.228.6 Default . . . 1909
2.229 fix saed/vtk command . . . 1909
2.229.1 Syntax . . . 1909
2.229.2 Examples . . . 1909
2.229.3 Description . . . 1909
2.229.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1911
2.229.5 Restrictions . . . 1911
2.229.6 Related commands . . . 1911
2.229.7 Default . . . 1911
2.230 fix set command . . . 1911
2.230.1 Syntax . . . 1911
2.230.2 Examples . . . 1911
2.230.3 Description . . . 1912
2.230.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1913
2.230.5 Restrictions . . . 1913
2.230.6 Related commands . . . 1914
2.230.7 Default . . . 1914
2.231 fix setforce command . . . 1914
2.232 fix setforce/spin command . . . 1914
2.232.1 Syntax . . . 1914
2.232.2 Examples . . . 1914
2.232.3 Description . . . 1914

2.232.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1915
2.232.5 Restrictions . . . 1916
2.232.6 Related commands . . . 1916
2.232.7 Default . . . 1916
2.233 fix settorque/atom command . . . 1916
2.233.1 Syntax . . . 1916
2.233.2 Examples . . . 1916
2.233.3 Description . . . 1916
2.233.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1917
2.233.5 Restrictions . . . 1917
2.233.6 Related commands . . . 1917
2.233.7 Default . . . 1917
2.234 fix sgcmc command . . . 1917
2.234.1 Syntax . . . 1917
2.234.2 Examples . . . 1918
2.234.3 Description . . . 1918
2.234.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1919
2.234.5 Restrictions . . . 1920
2.234.6 Default . . . 1920
2.235 fix shake command . . . 1920
2.236 fix rattle command . . . 1920
2.236.1 Syntax . . . 1920
2.236.2 Examples . . . 1921
2.236.3 Description . . . 1921
2.236.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1924
2.236.5 Restrictions . . . 1924
2.236.6 Related commands . . . 1924
2.236.7 Default . . . 1924
2.237 fix shardlow command . . . 1924
2.237.1 Syntax . . . 1925
2.237.2 Examples . . . 1925
2.237.3 Description . . . 1925
2.237.4 Restrictions . . . 1925
2.237.5 Related commands . . . 1926
2.237.6 Default . . . 1926
2.238 fix smd command . . . 1926
2.238.1 Syntax . . . 1926
2.238.2 Examples . . . 1926
2.238.3 Description . . . 1926
2.238.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1927
2.238.5 Restrictions . . . 1928
2.238.6 Related commands . . . 1928
2.238.7 Default . . . 1928
2.239 fix smd/adjust_dt command . . . 1928
2.239.1 Syntax . . . 1928
2.239.2 Examples . . . 1928
2.239.3 Description . . . 1928
2.239.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1929
2.239.5 Restrictions . . . 1929
2.239.6 Related commands . . . 1929
2.239.7 Default . . . 1929
2.240 fix smd/integrate_tlsph command . . . 1929
2.240.1 Syntax . . . 1929
2.240.2 Examples . . . 1929

2.240.3 Description . . . 1929
2.240.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1929
2.240.5 Restrictions . . . 1930
2.240.6 Related commands . . . 1930
2.240.7 Default . . . 1930
2.241 fix smd/integrate_ulsph command . . . 1930
2.241.1 Syntax . . . 1930
2.241.2 Examples . . . 1930
2.241.3 Description . . . 1930
2.241.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1931
2.241.5 Restrictions . . . 1931
2.241.6 Related commands . . . 1931
2.241.7 Default . . . 1931
2.242 fix smd/move_tri_surf command . . . 1931
2.242.1 Syntax . . . 1931
2.242.2 Examples . . . 1931
2.242.3 Description . . . 1931
2.242.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1932
2.242.5 Restrictions . . . 1932
2.242.6 Related commands . . . 1932
2.242.7 Default . . . 1932
2.243 fix smd/setvel command . . . 1932
2.243.1 Syntax . . . 1932
2.243.2 Examples . . . 1932
2.243.3 Description . . . 1933
2.243.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1933
2.243.5 Restrictions . . . 1933
2.243.6 Related commands . . . 1933
2.243.7 Default . . . 1933
2.244 fix smd/wall_surface command . . . 1933
2.244.1 Syntax . . . 1933
2.244.2 Examples . . . 1934
2.244.3 Description . . . 1934
2.244.4 Dump image info . . . 1934
2.244.5 Restart, fix_modify, output, run start/stop, minimize info . . . 1934
2.244.6 Restrictions . . . 1934
2.244.7 Related commands . . . 1935
2.244.8 Default . . . 1935
2.245 fix sph command . . . 1935
2.245.1 Syntax . . . 1935
2.245.2 Examples . . . 1935
2.245.3 Description . . . 1935
2.245.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1935
2.245.5 Restrictions . . . 1935
2.245.6 Related commands . . . 1935
2.245.7 Default . . . 1936
2.246 fix sph/stationary command . . . 1936
2.246.1 Syntax . . . 1936
2.246.2 Examples . . . 1936
2.246.3 Description . . . 1936
2.246.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1936
2.246.5 Restrictions . . . 1936
2.246.6 Related commands . . . 1936
2.246.7 Default . . . 1936

2.247 fix spring command . . . 1937
2.247.1 Syntax . . . 1937
2.247.2 Examples . . . 1937
2.247.3 Description . . . 1937
2.247.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1938
2.247.5 Restrictions . . . 1939
2.247.6 Related commands . . . 1939
2.247.7 Default . . . 1939
2.248 fix spring/chunk command . . . 1939
2.248.1 Syntax . . . 1939
2.248.2 Examples . . . 1939
2.248.3 Description . . . 1939
2.248.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1940
2.248.5 Restrictions . . . 1940
2.248.6 Related commands . . . 1940
2.248.7 Default . . . 1940
2.249 fix spring/rg command . . . 1940
2.249.1 Syntax . . . 1940
2.249.2 Examples . . . 1941
2.249.3 Description . . . 1941
2.249.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1941
2.249.5 Restrictions . . . 1941
2.249.6 Related commands . . . 1941
2.249.7 Default . . . 1942
2.250 fix spring/self command . . . 1942
2.250.1 Syntax . . . 1942
2.250.2 Examples . . . 1942
2.250.3 Description . . . 1942
2.250.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1942
2.250.5 Restrictions . . . 1943
2.250.6 Related commands . . . 1943
2.250.7 Default . . . 1943
2.251 fix srd command . . . 1943
2.251.1 Syntax . . . 1943
2.251.2 Examples . . . 1944
2.251.3 Description . . . 1944
2.251.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1948
2.251.5 Restrictions . . . 1948
2.251.6 Related commands . . . 1948
2.251.7 Default . . . 1948
2.252 fix store/force command . . . 1949
2.252.1 Syntax . . . 1949
2.252.2 Examples . . . 1949
2.252.3 Description . . . 1949
2.252.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1949
2.252.5 Restrictions . . . 1949
2.252.6 Related commands . . . 1949
2.252.7 Default . . . 1950
2.253 fix store/state command . . . 1950
2.253.1 Syntax . . . 1950
2.253.2 Examples . . . 1951
2.253.3 Description . . . 1951
2.253.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1952
2.253.5 Restrictions . . . 1953

2.253.6 Related commands . . . 1953
2.253.7 Default . . . 1953
2.254 fix surface/global command . . . 1953
2.254.1 Syntax . . . 1953
2.254.2 Examples . . . 1954
2.254.3 Description . . . 1954
2.254.4 Dump image info . . . 1956
2.254.5 Restart, fix_modify, output, run start/stop, minimize info . . . 1957
2.254.6 Restrictions . . . 1958
2.254.7 Related commands . . . 1958
2.254.8 Default . . . 1958
2.255 fix surface/local command . . . 1959
2.255.1 Syntax . . . 1959
2.255.2 Examples . . . 1959
2.255.3 Description . . . 1959
2.255.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1961
2.255.5 Restrictions . . . 1961
2.255.6 Related commands . . . 1961
2.255.7 Default . . . 1961
2.256 fix temp/berendsen command . . . 1961
2.256.1 Syntax . . . 1961
2.256.2 Examples . . . 1962
2.256.3 Description . . . 1962
2.256.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1963
2.256.5 Restrictions . . . 1963
2.256.6 Related commands . . . 1964
2.256.7 Default . . . 1964
2.257 fix temp/csvr command . . . 1964
2.258 fix temp/csld command . . . 1964
2.258.1 Syntax . . . 1964
2.258.2 Examples . . . 1964
2.258.3 Description . . . 1964
2.258.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1965
2.258.5 Restrictions . . . 1966
2.258.6 Related commands . . . 1966
2.258.7 Default . . . 1966
2.259 fix temp/rescale command . . . 1966
2.259.1 Syntax . . . 1966
2.259.2 Examples . . . 1967
2.259.3 Description . . . 1967
2.259.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1968
2.259.5 Restrictions . . . 1968
2.259.6 Related commands . . . 1969
2.259.7 Default . . . 1969
2.260 fix temp/rescale/eff command . . . 1969
2.260.1 Syntax . . . 1969
2.260.2 Examples . . . 1969
2.260.3 Description . . . 1969
2.260.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1969
2.260.5 Restrictions . . . 1970
2.260.6 Related commands . . . 1970
2.260.7 Default . . . 1970
2.261 fix tfmc command . . . 1970
2.261.1 Syntax . . . 1970

2.261.2 Examples . . . 1970
2.261.3 Description . . . 1970
2.261.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1971
2.261.5 Restrictions . . . 1971
2.261.6 Related commands . . . 1972
2.261.7 Default . . . 1972
2.262 fix tgnvt/drude command . . . 1972
2.263 fix tgnpt/drude command . . . 1972
2.263.1 Syntax . . . 1972
2.263.2 Examples . . . 1973
2.263.3 Description . . . 1973
2.263.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1975
2.263.5 Restrictions . . . 1976
2.263.6 Related commands . . . 1976
2.263.7 Default . . . 1976
2.264 fix thermal/conductivity command . . . 1976
2.264.1 Syntax . . . 1976
2.264.2 Examples . . . 1976
2.264.3 Description . . . 1977
2.264.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1978
2.264.5 Restrictions . . . 1978
2.264.6 Related commands . . . 1978
2.264.7 Default . . . 1978
2.265 fix ti/spring command . . . 1978
2.265.1 Syntax . . . 1978
2.265.2 Example . . . 1979
2.265.3 Description . . . 1979
2.265.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1980
2.265.5 Related commands . . . 1980
2.265.6 Restrictions . . . 1980
2.265.7 Default . . . 1980
2.266 fix tmd command . . . 1981
2.266.1 Syntax . . . 1981
2.266.2 Examples . . . 1981
2.266.3 Description . . . 1981
2.266.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1982
2.266.5 Restrictions . . . 1982
2.266.6 Related commands . . . 1982
2.266.7 Default . . . 1982
2.267 fix ttm command . . . 1983
2.268 fix ttm/grid command . . . 1983
2.269 fix ttm/mod command . . . 1983
2.270 fix ttm/thermal command . . . 1983
2.270.1 Syntax . . . 1983
2.270.2 Examples . . . 1984
2.270.3 Description . . . 1984
2.270.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1988
2.270.5 Restrictions . . . 1989
2.270.6 Related commands . . . 1989
2.270.7 Default . . . 1989
2.271 fix tune/kspace command . . . 1989
2.271.1 Syntax . . . 1989
2.271.2 Examples . . . 1990
2.271.3 Description . . . 1990

2.271.4 Restrictions . . . 1990
2.271.5 Related commands . . . 1990
2.271.6 Default . . . 1991
2.272 fix vector command . . . 1991
2.272.1 Syntax . . . 1991
2.272.2 Examples . . . 1991
2.272.3 Description . . . 1991
2.272.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1992
2.272.5 Restrictions . . . 1993
2.272.6 Related commands . . . 1993
2.272.7 Defaults . . . 1993
2.273 fix viscosity command . . . 1993
2.273.1 Syntax . . . 1993
2.273.2 Examples . . . 1993
2.273.3 Description . . . 1994
2.273.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1995
2.273.5 Restrictions . . . 1995
2.273.6 Related commands . . . 1995
2.273.7 Default . . . 1995
2.274 fix viscous command . . . 1995
2.274.1 Syntax . . . 1995
2.274.2 Examples . . . 1996
2.274.3 Description . . . 1996
2.274.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1997
2.274.5 Restrictions . . . 1997
2.274.6 Related commands . . . 1997
2.274.7 Default . . . 1997
2.275 fix viscous/nonlinear command . . . 1997
2.275.1 Syntax . . . 1997
2.275.2 Examples . . . 1997
2.275.3 Description . . . 1998
2.275.4 Restrictions . . . 1998
2.275.5 Restart, fix_modify, output, run start/stop, minimize info . . . 1998
2.275.6 Related commands . . . 1998
2.275.7 Default . . . 1998
2.276 fix viscous/sphere command . . . 1998
2.276.1 Syntax . . . 1999
2.276.2 Examples . . . 1999
2.276.3 Description . . . 1999
2.276.4 Restart, fix_modify, output, run start/stop, minimize info . . . 1999
2.276.5 Restrictions . . . 2000
2.276.6 Related commands . . . 2000
2.276.7 Default . . . 2000
2.277 fix wall/lj93 command . . . 2000
2.278 fix wall/lj126 command . . . 2000
2.279 fix wall/lj1043 command . . . 2000
2.280 fix wall/colloid command . . . 2000
2.281 fix wall/harmonic command . . . 2000
2.282 fix wall/harmonic/outside command . . . 2001
2.283 fix wall/lepton command . . . 2001
2.284 fix wall/morse command . . . 2001
2.285 fix wall/table command . . . 2001
2.285.1 Syntax . . . 2001
2.285.2 Examples . . . 2002

2.285.3 Description . . . 2002
2.285.4 Lepton expression syntax and features . . . 2006
2.285.5 Table file format . . . 2007
2.285.6 Dump image info . . . 2007
2.285.7 Restart, fix_modify, output, run start/stop, minimize info . . . 2008
2.285.8 Restrictions . . . 2009
2.285.9 Related commands . . . 2009
2.285.10 Default . . . 2009
2.286 fix wall/body/polygon command . . . 2009
2.286.1 Syntax . . . 2009
2.286.2 Examples . . . 2009
2.286.3 Description . . . 2010
2.286.4 Dump image info . . . 2010
2.286.5 Restart, fix_modify, output, run start/stop, minimize info . . . 2010
2.286.6 Restrictions . . . 2010
2.286.7 Related commands . . . 2011
2.286.8 Default . . . 2011
2.287 fix wall/body/polyhedron command . . . 2011
2.287.1 Syntax . . . 2011
2.287.2 Examples . . . 2011
2.287.3 Description . . . 2011
2.287.4 Dump image info . . . 2012
2.287.5 Restart, fix_modify, output, run start/stop, minimize info . . . 2012
2.287.6 Restrictions . . . 2012
2.287.7 Related commands . . . 2012
2.287.8 Default . . . 2012
2.288 fix wall/ees command . . . 2013
2.289 fix wall/region/ees command . . . 2013
2.289.1 Syntax . . . 2013
2.289.2 Examples . . . 2013
2.289.3 Description . . . 2013
2.289.4 Restart, fix_modify, output, run start/stop, minimize info . . . 2014
2.289.5 Dump image info . . . 2015
2.289.6 Restrictions . . . 2015
2.289.7 Related commands . . . 2015
2.289.8 Default . . . 2015
2.290 fix wall/flow command . . . 2016
2.290.1 Syntax . . . 2016
2.290.2 Examples . . . 2016
2.290.3 Description . . . 2016
2.290.4 Restart, fix_modify, output, run start/stop, minimize info . . . 2017
2.290.5 Restrictions . . . 2018
2.290.6 Related commands . . . 2018
2.290.7 Default . . . 2018
2.291 fix wall/gran command . . . 2018
2.291.1 Syntax . . . 2018
2.291.2 Examples . . . 2019
2.291.3 Description . . . 2019
2.291.4 Restart, fix_modify, output, run start/stop, minimize info . . . 2021
2.291.5 Dump image info . . . 2022
2.291.6 Restrictions . . . 2022
2.291.7 Related commands . . . 2022
2.291.8 Default . . . 2022
2.292 fix wall/gran/region command . . . 2022

2.292.1 Syntax . . . 2022
2.292.2 Examples . . . 2023
2.292.3 Description . . . 2023
2.292.4 Restart, fix_modify, output, run start/stop, minimize info . . . 2026
2.292.5 Dump image info . . . 2026
2.292.6 Restrictions . . . 2026
2.292.7 Related commands . . . 2027
2.292.8 Default . . . 2027
2.293 fix wall/piston command . . . 2027
2.293.1 Syntax . . . 2027
2.293.2 Examples . . . 2027
2.293.3 Description . . . 2027
2.293.4 Restart, fix_modify, output, run start/stop, minimize info . . . 2028
2.293.5 Restrictions . . . 2028
2.293.6 Related commands . . . 2028
2.293.7 Default . . . 2028
2.294 fix wall/reflect command . . . 2028
2.294.1 Syntax . . . 2028
2.294.2 Examples . . . 2029
2.294.3 Description . . . 2029
2.294.4 Dump image info . . . 2030
2.294.5 Restart, fix_modify, output, run start/stop, minimize info . . . 2031
2.294.6 Restrictions . . . 2031
2.294.7 Related commands . . . 2031
2.294.8 Default . . . 2031
2.295 fix wall/reflect/stochastic command . . . 2031
2.295.1 Syntax . . . 2031
2.295.2 Examples . . . 2032
2.295.3 Description . . . 2032
2.295.4 Dump image info . . . 2032
2.295.5 Restrictions . . . 2033
2.295.6 Related commands . . . 2033
2.295.7 Default . . . 2033
2.296 fix wall/region command . . . 2033
2.296.1 Syntax . . . 2033
2.296.2 Examples . . . 2034
2.296.3 Description . . . 2034
2.296.4 Restart, fix_modify, output, run start/stop, minimize info . . . 2036
2.296.5 Dump image info . . . 2036
2.296.6 Restrictions . . . 2036
2.296.7 Related commands . . . 2037
2.296.8 Default . . . 2037
2.297 fix wall/srd command . . . 2037
2.297.1 Syntax . . . 2037
2.297.2 Examples . . . 2037
2.297.3 Description . . . 2037
2.297.4 Dump image info . . . 2039
2.297.5 Restart, fix_modify, output, run start/stop, minimize info . . . 2039
2.297.6 Restrictions . . . 2040
2.297.7 Related commands . . . 2040
2.297.8 Default . . . 2040
2.298 fix widom command . . . 2040
2.298.1 Syntax . . . 2040
2.298.2 Examples . . . 2040

2.298.3 Description . . . 2041
2.298.4 Restart, fix_modify, output, run start/stop, minimize info . . . 2042
2.298.5 Restrictions . . . 2042
2.298.6 Related commands . . . 2043
2.298.7 Default . . . 2043

**3 Compute Styles 2045**
3.1 compute ackland/atom command . . . 2045
3.1.1 Syntax . . . 2045
3.1.2 Examples . . . 2045
3.1.3 Description . . . 2045
3.1.4 Output info . . . 2046
3.1.5 Restrictions . . . 2046
3.1.6 Related commands . . . 2046
3.1.7 Default . . . 2046
3.2 compute adf command . . . 2046
3.2.1 Syntax . . . 2046
3.2.2 Examples . . . 2047
3.2.3 Description . . . 2047
3.2.4 Output info . . . 2049
3.2.5 Restrictions . . . 2049
3.2.6 Related commands . . . 2049
3.2.7 Default . . . 2049
3.3 compute angle command . . . 2049
3.3.1 Syntax . . . 2049
3.3.2 Examples . . . 2050
3.3.3 Description . . . 2050
3.3.4 Output info . . . 2050
3.3.5 Restrictions . . . 2050
3.3.6 Related commands . . . 2050
3.3.7 Default . . . 2050
3.4 compute angle/local command . . . 2050
3.4.1 Syntax . . . 2050
3.4.2 Examples . . . 2051
3.4.3 Description . . . 2051
3.4.4 Output info . . . 2052
3.4.5 Restrictions . . . 2052
3.4.6 Related commands . . . 2052
3.4.7 Default . . . 2052
3.5 compute angmom/chunk command . . . 2052
3.5.1 Syntax . . . 2052
3.5.2 Examples . . . 2052
3.5.3 Description . . . 2052
3.5.4 Output info . . . 2053
3.5.5 Restrictions . . . 2053
3.5.6 Related commands . . . 2053
3.5.7 Default . . . 2053
3.6 compute ave/sphere/atom command . . . 2053
3.6.1 Syntax . . . 2053
3.6.2 Examples . . . 2054
3.6.3 Description . . . 2054
3.6.4 Output info . . . 2055
3.6.5 Restrictions . . . 2055
3.6.6 Related commands . . . 2055

3.6.7 Default . . . 2055
3.7 compute basal/atom command . . . 2055
3.7.1 Syntax . . . 2055
3.7.2 Examples . . . 2055
3.7.3 Description . . . 2055
3.7.4 Output info . . . 2056
3.7.5 Restrictions . . . 2056
3.7.6 Related commands . . . 2056
3.7.7 Default . . . 2056
3.8 compute body/local command . . . 2056
3.8.1 Syntax . . . 2056
3.8.2 Examples . . . 2056
3.8.3 Description . . . 2056
3.8.4 Output info . . . 2057
3.8.5 Restrictions . . . 2057
3.8.6 Related commands . . . 2057
3.8.7 Default . . . 2057
3.9 compute bond command . . . 2057
3.9.1 Syntax . . . 2057
3.9.2 Examples . . . 2058
3.9.3 Description . . . 2058
3.9.4 Output info . . . 2058
3.9.5 Restrictions . . . 2058
3.9.6 Related commands . . . 2058
3.9.7 Default . . . 2058
3.10 compute bond/local command . . . 2058
3.10.1 Syntax . . . 2058
3.10.2 Examples . . . 2059
3.10.3 Description . . . 2059
3.10.4 Output info . . . 2061
3.10.5 Restrictions . . . 2061
3.10.6 Related commands . . . 2061
3.10.7 Default . . . 2061
3.11 compute born/matrix command . . . 2061
3.11.1 Syntax . . . 2061
3.11.2 Examples . . . 2062
3.11.3 Description . . . 2062
3.11.4 Restrictions . . . 2064
3.11.5 Default . . . 2064
3.12 compute centro/atom command . . . 2064
3.12.1 Syntax . . . 2064
3.12.2 Examples . . . 2064
3.12.3 Description . . . 2065
3.12.4 Output info . . . 2065
3.12.5 Restrictions . . . 2066
3.12.6 Related commands . . . 2066
3.12.7 Default . . . 2066
3.13 compute chunk/atom command . . . 2066
3.13.1 Syntax . . . 2066
3.13.2 Examples . . . 2068
3.13.3 Description . . . 2068
3.13.4 Output info . . . 2074
3.13.5 Dump image info . . . 2074
3.13.6 Restrictions . . . 2075

3.13.7 Related commands . . . 2075
3.13.8 Default . . . 2075
3.14 compute chunk/spread/atom command . . . 2075
3.14.1 Syntax . . . 2075
3.14.2 Examples . . . 2076
3.14.3 Description . . . 2076
3.14.4 Output info . . . 2078
3.14.5 Restrictions . . . 2078
3.14.6 Related commands . . . 2078
3.14.7 Default . . . 2078
3.15 compute cluster/atom command . . . 2078
3.16 compute fragment/atom command . . . 2078
3.17 compute aggregate/atom command . . . 2078
3.17.1 Syntax . . . 2078
3.17.2 Examples . . . 2079
3.17.3 Description . . . 2079
3.17.4 Output info . . . 2080
3.17.5 Restrictions . . . 2080
3.17.6 Related commands . . . 2080
3.17.7 Default . . . 2080
3.18 compute cna/atom command . . . 2080
3.18.1 Syntax . . . 2080
3.18.2 Examples . . . 2080
3.18.3 Description . . . 2080
3.18.4 Output info . . . 2081
3.18.5 Restrictions . . . 2081
3.18.6 Related commands . . . 2081
3.18.7 Default . . . 2082
3.19 compute cnp/atom command . . . 2082
3.19.1 Syntax . . . 2082
3.19.2 Examples . . . 2082
3.19.3 Description . . . 2082
3.19.4 Output info . . . 2083
3.19.5 Restrictions . . . 2083
3.19.6 Related commands . . . 2083
3.19.7 Default . . . 2084
3.20 compute com command . . . 2084
3.20.1 Syntax . . . 2084
3.20.2 Examples . . . 2084
3.20.3 Description . . . 2084
3.20.4 Output info . . . 2084
3.20.5 Restrictions . . . 2084
3.20.6 Related commands . . . 2084
3.20.7 Default . . . 2085
3.21 compute com/chunk command . . . 2085
3.21.1 Syntax . . . 2085
3.21.2 Examples . . . 2085
3.21.3 Description . . . 2085
3.21.4 Output info . . . 2086
3.21.5 Restrictions . . . 2086
3.21.6 Related commands . . . 2086
3.21.7 Default . . . 2086
3.22 compute composition/atom command . . . 2086
3.22.1 Syntax . . . 2086

3.22.2 Examples . . . 2086
3.22.3 Description . . . 2087
3.22.4 Output info . . . 2088
3.22.5 Restrictions . . . 2088
3.22.6 Related commands . . . 2088
3.22.7 Default . . . 2088
3.23 compute contact/atom command . . . 2088
3.23.1 Syntax . . . 2088
3.23.2 Examples . . . 2088
3.23.3 Description . . . 2088
3.23.4 Output info . . . 2089
3.23.5 Restrictions . . . 2089
3.23.6 Related commands . . . 2089
3.23.7 Default . . . 2089
3.24 compute coord/atom command . . . 2089
3.24.1 Syntax . . . 2089
3.24.2 Examples . . . 2089
3.24.3 Description . . . 2090
3.24.4 Output info . . . 2091
3.24.5 Restrictions . . . 2091
3.24.6 Related commands . . . 2091
3.24.7 Default . . . 2091
3.25 compute count/type command . . . 2091
3.25.1 Syntax . . . 2091
3.25.2 Examples . . . 2091
3.25.3 Description . . . 2092
3.25.4 Output info . . . 2093
3.25.5 Restrictions . . . 2093
3.25.6 Related commands . . . 2093
3.25.7 Default . . . 2093
3.26 compute damage/atom command . . . 2093
3.26.1 Syntax . . . 2093
3.26.2 Examples . . . 2093
3.26.3 Description . . . 2093
3.26.4 Output info . . . 2094
3.26.5 Restrictions . . . 2094
3.26.6 Related commands . . . 2094
3.26.7 Default . . . 2094
3.27 compute dihedral command . . . 2094
3.27.1 Syntax . . . 2094
3.27.2 Examples . . . 2094
3.27.3 Description . . . 2094
3.27.4 Output info . . . 2094
3.27.5 Restrictions . . . 2095
3.27.6 Related commands . . . 2095
3.27.7 Default . . . 2095
3.28 compute dihedral/local command . . . 2095
3.28.1 Syntax . . . 2095
3.28.2 Examples . . . 2095
3.28.3 Description . . . 2095
3.28.4 Output info . . . 2096
3.28.5 Restrictions . . . 2096
3.28.6 Related commands . . . 2096
3.28.7 Default . . . 2097

3.29 compute dilatation/atom command . . . 2097
3.29.1 Syntax . . . 2097
3.29.2 Examples . . . 2097
3.29.3 Description . . . 2097
3.29.4 Output info . . . 2097
3.29.5 Restrictions . . . 2097
3.29.6 Related commands . . . 2097
3.29.7 Default . . . 2097
3.30 compute dipole command . . . 2098
3.31 compute dipole/tip4p command . . . 2098
3.31.1 Syntax . . . 2098
3.31.2 Examples . . . 2098
3.31.3 Description . . . 2098
3.31.4 Output info . . . 2098
3.31.5 Restrictions . . . 2098
3.31.6 Related commands . . . 2099
3.31.7 Default . . . 2099
3.32 compute dipole/chunk command . . . 2099
3.33 compute dipole/tip4p/chunk command . . . 2099
3.33.1 Syntax . . . 2099
3.33.2 Examples . . . 2099
3.33.3 Description . . . 2099
3.33.4 Output info . . . 2100
3.33.5 Restrictions . . . 2100
3.33.6 Related commands . . . 2100
3.33.7 Default . . . 2100
3.34 compute displace/atom command . . . 2100
3.34.1 Syntax . . . 2100
3.34.2 Examples . . . 2101
3.34.3 Description . . . 2101
3.34.4 Output info . . . 2102
3.34.5 Restrictions . . . 2102
3.34.6 Related commands . . . 2102
3.34.7 Default . . . 2102
3.35 compute dpd command . . . 2102
3.35.1 Syntax . . . 2102
3.35.2 Examples . . . 2102
3.35.3 Description . . . 2103
3.35.4 Output info . . . 2103
3.35.5 Restrictions . . . 2103
3.35.6 Related commands . . . 2103
3.35.7 Default . . . 2103
3.36 compute dpd/atom command . . . 2104
3.36.1 Syntax . . . 2104
3.36.2 Examples . . . 2104
3.36.3 Description . . . 2104
3.36.4 Output info . . . 2104
3.36.5 Restrictions . . . 2104
3.36.6 Related commands . . . 2104
3.36.7 Default . . . 2104
3.37 compute edpd/temp/atom command . . . 2104
3.37.1 Syntax . . . 2104
3.37.2 Examples . . . 2105
3.37.3 Description . . . 2105

3.37.4 Output info . . . 2105
3.37.5 Restrictions . . . 2105
3.37.6 Related commands . . . 2105
3.37.7 Default . . . 2105
3.38 compute efield/atom command . . . 2105
3.38.1 Syntax . . . 2105
3.38.2 Examples . . . 2106
3.38.3 Description . . . 2106
3.38.4 Output info . . . 2106
3.38.5 Restrictions . . . 2106
3.38.6 Related commands . . . 2106
3.38.7 Default . . . 2106
3.39 compute efield/wolf/atom command . . . 2106
3.39.1 Syntax . . . 2106
3.39.2 Examples . . . 2107
3.39.3 Description . . . 2107
3.39.4 Output info . . . 2107
3.39.5 Restrictions . . . 2108
3.39.6 Related commands . . . 2108
3.39.7 Default . . . 2108
3.40 compute entropy/atom command . . . 2108
3.40.1 Syntax . . . 2108
3.40.2 Examples . . . 2108
3.40.3 Description . . . 2108
3.40.4 Output info . . . 2109
3.40.5 Restrictions . . . 2110
3.40.6 Related commands . . . 2110
3.40.7 Default . . . 2110
3.41 compute erotate/asphere command . . . 2110
3.41.1 Syntax . . . 2110
3.41.2 Examples . . . 2110
3.41.3 Description . . . 2110
3.41.4 Output info . . . 2111
3.41.5 Restrictions . . . 2111
3.41.6 Related commands . . . 2111
3.41.7 Default . . . 2111
3.42 compute erotate/rigid command . . . 2111
3.42.1 Syntax . . . 2111
3.42.2 Examples . . . 2112
3.42.3 Description . . . 2112
3.42.4 Output info . . . 2112
3.42.5 Restrictions . . . 2112
3.42.6 Related commands . . . 2112
3.42.7 Default . . . 2112
3.43 compute erotate/sphere command . . . 2112
3.43.1 Syntax . . . 2112
3.43.2 Examples . . . 2113
3.43.3 Description . . . 2113
3.43.4 Output info . . . 2113
3.43.5 Restrictions . . . 2113
3.43.6 Related commands . . . 2113
3.43.7 Default . . . 2113
3.44 compute erotate/sphere/atom command . . . 2114
3.44.1 Syntax . . . 2114

3.44.2 Examples . . . 2114
3.44.3 Description . . . 2114
3.44.4 Output info . . . 2114
3.44.5 Restrictions . . . 2115
3.44.6 Related commands . . . 2115
3.44.7 Default . . . 2115
3.45 compute event/displace command . . . 2115
3.45.1 Syntax . . . 2115
3.45.2 Examples . . . 2115
3.45.3 Description . . . 2115
3.45.4 Output info . . . 2115
3.45.5 Restrictions . . . 2116
3.45.6 Related commands . . . 2116
3.45.7 Default . . . 2116
3.46 compute fabric command . . . 2116
3.46.1 Syntax . . . 2116
3.46.2 Examples . . . 2116
3.46.3 Description . . . 2116
3.46.4 Output info . . . 2118
3.46.5 Restrictions . . . 2118
3.46.6 Related commands . . . 2118
3.46.7 Default . . . 2118
3.47 compute fep command . . . 2118
3.47.1 Syntax . . . 2118
3.47.2 Examples . . . 2119
3.47.3 Description . . . 2119
3.47.4 Output info . . . 2121
3.47.5 Restrictions . . . 2121
3.47.6 Related commands . . . 2122
3.47.7 Default . . . 2122
3.48 compute fep/ta command . . . 2122
3.48.1 Syntax . . . 2122
3.48.2 Examples . . . 2122
3.48.3 Description . . . 2122
3.48.4 Output info . . . 2123
3.48.5 Restrictions . . . 2123
3.48.6 Related commands . . . 2123
3.48.7 Default . . . 2123
3.49 compute gaussian/grid/local command . . . 2123
3.49.1 Syntax . . . 2123
3.49.2 Examples . . . 2124
3.49.3 Description . . . 2124
3.49.4 Output info . . . 2124
3.49.5 Restrictions . . . 2124
3.49.6 Related commands . . . 2124
3.50 compute global/atom command . . . 2125
3.50.1 Syntax . . . 2125
3.50.2 Examples . . . 2125
3.50.3 Description . . . 2125
3.50.4 Output info . . . 2127
3.50.5 Restrictions . . . 2127
3.50.6 Related commands . . . 2127
3.50.7 Default . . . 2128
3.51 compute group/group command . . . 2128

3.51.1 Syntax . . . 2128
3.51.2 Examples . . . 2128
3.51.3 Description . . . 2128
3.51.4 Output info . . . 2129
3.51.5 Restrictions . . . 2129
3.51.6 Related commands . . . 2130
3.51.7 Default . . . 2130
3.52 compute gyration command . . . 2130
3.52.1 Syntax . . . 2130
3.52.2 Examples . . . 2130
3.52.3 Description . . . 2130
3.52.4 Output info . . . 2131
3.52.5 Restrictions . . . 2131
3.52.6 Related commands . . . 2131
3.52.7 Default . . . 2131
3.53 compute gyration/chunk command . . . 2131
3.53.1 Syntax . . . 2131
3.53.2 Examples . . . 2131
3.53.3 Description . . . 2131
3.53.4 Output info . . . 2132
3.53.5 Restrictions . . . 2132
3.53.6 Related commands . . . 2132
3.53.7 Default . . . 2133
3.54 compute gyration/shape command . . . 2133
3.54.1 Syntax . . . 2133
3.54.2 Examples . . . 2133
3.54.3 Description . . . 2133
3.54.4 Output info . . . 2134
3.54.5 Restrictions . . . 2134
3.54.6 Related commands . . . 2134
3.54.7 Default . . . 2134
3.55 compute gyration/shape/chunk command . . . 2134
3.55.1 Syntax . . . 2134
3.55.2 Examples . . . 2134
3.55.3 Description . . . 2134
3.55.4 Output info . . . 2135
3.55.5 Restrictions . . . 2135
3.55.6 Related commands . . . 2135
3.55.7 Default . . . 2135
3.56 compute hbond/local command . . . 2135
3.56.1 Syntax . . . 2135
3.56.2 Examples . . . 2136
3.56.3 Description . . . 2136
3.56.4 Output info . . . 2138
3.56.5 Dump image info . . . 2139
3.56.6 Restrictions . . . 2140
3.56.7 Related commands . . . 2140
3.56.8 Default . . . 2140
3.57 compute heat/flux command . . . 2140
3.57.1 Syntax . . . 2140
3.57.2 Examples . . . 2140
3.57.3 Description . . . 2140
3.57.4 Output info . . . 2142
3.57.5 Restrictions . . . 2142

3.57.6 Related commands . . . 2142
3.57.7 Default . . . 2142
3.58 compute hexorder/atom command . . . 2144
3.58.1 Syntax . . . 2144
3.58.2 Examples . . . 2144
3.58.3 Description . . . 2144
3.58.4 Output info . . . 2145
3.58.5 Restrictions . . . 2145
3.58.6 Related commands . . . 2145
3.58.7 Default . . . 2145
3.59 compute hma command . . . 2146
3.59.1 Syntax . . . 2146
3.59.2 Examples . . . 2146
3.59.3 Description . . . 2146
3.59.4 Output info . . . 2148
3.59.5 Restrictions . . . 2148
3.59.6 Related commands . . . 2148
3.59.7 Default . . . 2148
3.60 compute improper command . . . 2148
3.60.1 Syntax . . . 2148
3.60.2 Examples . . . 2148
3.60.3 Description . . . 2149
3.60.4 Output info . . . 2149
3.60.5 Restrictions . . . 2149
3.60.6 Related commands . . . 2149
3.60.7 Default . . . 2149
3.61 compute improper/local command . . . 2149
3.61.1 Syntax . . . 2149
3.61.2 Examples . . . 2149
3.61.3 Description . . . 2149
3.61.4 Output info . . . 2150
3.61.5 Restrictions . . . 2150
3.61.6 Related commands . . . 2150
3.61.7 Default . . . 2150
3.62 compute inertia/chunk command . . . 2150
3.62.1 Syntax . . . 2150
3.62.2 Examples . . . 2151
3.62.3 Description . . . 2151
3.62.4 Output info . . . 2151
3.62.5 Restrictions . . . 2151
3.62.6 Related commands . . . 2151
3.62.7 Default . . . 2152
3.63 compute ke command . . . 2152
3.63.1 Syntax . . . 2152
3.63.2 Examples . . . 2152
3.63.3 Description . . . 2152
3.63.4 Output info . . . 2152
3.63.5 Restrictions . . . 2152
3.63.6 Related commands . . . 2152
3.63.7 Default . . . 2152
3.64 compute ke/atom command . . . 2153
3.64.1 Syntax . . . 2153
3.64.2 Examples . . . 2153
3.64.3 Description . . . 2153

3.64.4 Output info . . . 2153
3.64.5 Restrictions . . . 2153
3.64.6 Related commands . . . 2153
3.64.7 Default . . . 2154
3.65 compute ke/atom/eff command . . . 2154
3.65.1 Syntax . . . 2154
3.65.2 Examples . . . 2154
3.65.3 Description . . . 2154
3.65.4 Output info . . . 2154
3.65.5 Restrictions . . . 2155
3.65.6 Related commands . . . 2155
3.65.7 Default . . . 2155
3.66 compute ke/eff command . . . 2155
3.66.1 Syntax . . . 2155
3.66.2 Examples . . . 2155
3.66.3 Description . . . 2155
3.66.4 Output info . . . 2156
3.66.5 Restrictions . . . 2156
3.66.6 Related commands . . . 2156
3.66.7 Default . . . 2156
3.67 compute ke/rigid command . . . 2156
3.67.1 Syntax . . . 2156
3.67.2 Examples . . . 2156
3.67.3 Description . . . 2156
3.67.4 Output info . . . 2156
3.67.5 Restrictions . . . 2157
3.67.6 Related commands . . . 2157
3.67.7 Default . . . 2157
3.68 compute mliap command . . . 2157
3.68.1 Syntax . . . 2157
3.68.2 Examples . . . 2157
3.68.3 Description . . . 2157
3.68.4 Output info . . . 2159
3.68.5 Restrictions . . . 2159
3.68.6 Related commands . . . 2159
3.68.7 Default . . . 2159
3.69 compute momentum command . . . 2159
3.69.1 Syntax . . . 2159
3.69.2 Examples . . . 2159
3.69.3 Description . . . 2159
3.69.4 Output info . . . 2160
3.69.5 Restrictions . . . 2160
3.69.6 Related commands . . . 2160
3.69.7 Default . . . 2160
3.70 compute msd command . . . 2160
3.70.1 Syntax . . . 2160
3.70.2 Examples . . . 2160
3.70.3 Description . . . 2160
3.70.4 Output info . . . 2161
3.70.5 Restrictions . . . 2161
3.70.6 Related commands . . . 2161
3.70.7 Default . . . 2161
3.71 compute msd/chunk command . . . 2162
3.71.1 Syntax . . . 2162

3.71.2 Examples . . . 2162
3.71.3 Description . . . 2162
3.71.4 Output info . . . 2163
3.71.5 Restrictions . . . 2163
3.71.6 Related commands . . . 2163
3.71.7 Default . . . 2163
3.72 compute msd/nongauss command . . . 2163
3.72.1 Syntax . . . 2163
3.72.2 Examples . . . 2164
3.72.3 Description . . . 2164
3.72.4 Output info . . . 2164
3.72.5 Restrictions . . . 2164
3.72.6 Related commands . . . 2164
3.72.7 Default . . . 2164
3.73 compute nbond/atom command . . . 2165
3.73.1 Syntax . . . 2165
3.73.2 Examples . . . 2165
3.73.3 Description . . . 2165
3.73.4 Output info . . . 2165
3.73.5 Restrictions . . . 2165
3.73.6 Related commands . . . 2165
3.73.7 Default . . . 2165
3.74 compute omega/chunk command . . . 2165
3.74.1 Syntax . . . 2165
3.74.2 Examples . . . 2166
3.74.3 Description . . . 2166
3.74.4 Output info . . . 2166
3.74.5 Restrictions . . . 2166
3.74.6 Related commands . . . 2167
3.74.7 Default . . . 2167
3.75 compute orientorder/atom command . . . 2167
3.75.1 Syntax . . . 2167
3.75.2 Examples . . . 2167
3.75.3 Description . . . 2167
3.75.4 Output info . . . 2169
3.75.5 Restrictions . . . 2169
3.75.6 Related commands . . . 2169
3.75.7 Default . . . 2169
3.76 compute pace command . . . 2170
3.76.1 Syntax . . . 2170
3.76.2 Examples . . . 2170
3.76.3 Description . . . 2170
3.76.4 Output info . . . 2172
3.76.5 Restrictions . . . 2172
3.76.6 Related commands . . . 2172
3.76.7 Default . . . 2173
3.77 compute pair command . . . 2173
3.77.1 Syntax . . . 2173
3.77.2 Examples . . . 2173
3.77.3 Description . . . 2173
3.77.4 Output info . . . 2174
3.77.5 Restrictions . . . 2174
3.77.6 Related commands . . . 2174
3.77.7 Default . . . 2174

3.78 compute pair/local command . . . 2174
3.78.1 Syntax . . . 2174
3.78.2 Examples . . . 2175
3.78.3 Description . . . 2175
3.78.4 Output info . . . 2176
3.78.5 Restrictions . . . 2176
3.78.6 Related commands . . . 2176
3.78.7 Default . . . 2176
3.79 compute pe command . . . 2177
3.79.1 Syntax . . . 2177
3.79.2 Examples . . . 2177
3.79.3 Description . . . 2177
3.79.4 Output info . . . 2177
3.79.5 Restrictions . . . 2178
3.79.6 Related commands . . . 2178
3.79.7 Default . . . 2178
3.80 compute pe/atom command . . . 2178
3.80.1 Syntax . . . 2178
3.80.2 Examples . . . 2178
3.80.3 Description . . . 2178
3.80.4 Output info . . . 2179
3.80.5 Restrictions . . . 2179
3.80.6 Related commands . . . 2179
3.80.7 Default . . . 2179
3.81 compute plasticity/atom command . . . 2179
3.81.1 Syntax . . . 2179
3.81.2 Examples . . . 2180
3.81.3 Description . . . 2180
3.81.4 Output info . . . 2180
3.81.5 Restrictions . . . 2180
3.81.6 Related commands . . . 2180
3.81.7 Default . . . 2180
3.82 compute pod/atom command . . . 2180
3.83 compute podd/atom command . . . 2180
3.84 compute pod/local command . . . 2180
3.85 compute pod/global command . . . 2180
3.85.1 Syntax . . . 2180
3.85.2 Examples . . . 2181
3.85.3 Description . . . 2181
3.85.4 Output info . . . 2181
3.85.5 Restrictions . . . 2182
3.85.6 Related commands . . . 2182
3.85.7 Default . . . 2182
3.86 compute pressure command . . . 2182
3.86.1 Syntax . . . 2182
3.86.2 Examples . . . 2182
3.86.3 Description . . . 2182
3.86.4 Output info . . . 2183
3.86.5 Restrictions . . . 2184
3.86.6 Related commands . . . 2184
3.86.7 Default . . . 2184
3.87 compute pressure/alchemy command . . . 2184
3.87.1 Syntax . . . 2184
3.87.2 Examples . . . 2184

3.87.3 Description . . . 2184
3.87.4 Output info . . . 2185
3.87.5 Restrictions . . . 2185
3.87.6 Related commands . . . 2185
3.87.7 Default . . . 2185
3.88 compute pressure/uef command . . . 2185
3.88.1 Syntax . . . 2185
3.88.2 Examples . . . 2185
3.88.3 Description . . . 2185
3.88.4 Restrictions . . . 2186
3.88.5 Related commands . . . 2186
3.88.6 Default . . . 2186
3.89 compute property/atom command . . . 2186
3.89.1 Syntax . . . 2186
3.89.2 Examples . . . 2188
3.89.3 Description . . . 2188
3.89.4 Output info . . . 2190
3.89.5 Restrictions . . . 2190
3.89.6 Related commands . . . 2190
3.89.7 Default . . . 2190
3.90 compute property/chunk command . . . 2190
3.90.1 Syntax . . . 2190
3.90.2 Examples . . . 2190
3.90.3 Description . . . 2191
3.90.4 Output info . . . 2191
3.90.5 Restrictions . . . 2192
3.90.6 Related commands . . . 2192
3.90.7 Default . . . 2192
3.91 compute property/grid command . . . 2192
3.91.1 Syntax . . . 2192
3.91.2 Examples . . . 2192
3.91.3 Description . . . 2192
3.91.4 Output info . . . 2193
3.91.5 Restrictions . . . 2193
3.91.6 Related commands . . . 2193
3.91.7 Default . . . 2193
3.92 compute property/local command . . . 2193
3.92.1 Syntax . . . 2193
3.92.2 Examples . . . 2194
3.92.3 Description . . . 2194
3.92.4 Output info . . . 2195
3.92.5 Restrictions . . . 2196
3.92.6 Related commands . . . 2196
3.92.7 Default . . . 2196
3.93 compute ptm/atom command . . . 2196
3.93.1 Syntax . . . 2196
3.93.2 Examples . . . 2196
3.93.3 Description . . . 2196
3.93.4 Output info . . . 2197
3.93.5 Restrictions . . . 2198
3.93.6 Related commands . . . 2198
3.93.7 Default . . . 2198
3.94 compute rattlers/atom command . . . 2198
3.94.1 Syntax . . . 2198

3.94.2 Examples . . . 2198
3.94.3 Description . . . 2198
3.94.4 Output info . . . 2199
3.94.5 Restrictions . . . 2199
3.94.6 Related commands . . . 2199
3.94.7 Default . . . 2199
3.95 compute rdf command . . . 2199
3.95.1 Syntax . . . 2199
3.95.2 Examples . . . 2199
3.95.3 Description . . . 2200
3.95.4 Output info . . . 2201
3.95.5 Restrictions . . . 2202
3.95.6 Related commands . . . 2202
3.95.7 Default . . . 2202
3.96 compute reaxff/atom command . . . 2202
3.96.1 Syntax . . . 2202
3.96.2 Examples . . . 2203
3.96.3 Description . . . 2203
3.96.4 Output info . . . 2203
3.96.5 Restrictions . . . 2203
3.96.6 Related commands . . . 2204
3.96.7 Default . . . 2204
3.97 compute reduce command . . . 2204
3.98 compute reduce/region command . . . 2204
3.98.1 Syntax . . . 2204
3.98.2 Examples . . . 2204
3.98.3 Description . . . 2205
3.98.4 Output info . . . 2207
3.98.5 Restrictions . . . 2207
3.98.6 Related commands . . . 2207
3.98.7 Default . . . 2207
3.99 compute reduce/chunk command . . . 2207
3.99.1 Syntax . . . 2207
3.99.2 Examples . . . 2208
3.99.3 Description . . . 2208
3.99.4 Output info . . . 2209
3.99.5 Restrictions . . . 2209
3.99.6 Related commands . . . 2210
3.99.7 Default . . . 2210
3.100 compute rheo/property/atom command . . . 2210
3.100.1 Syntax . . . 2210
3.100.2 Examples . . . 2210
3.100.3 Description . . . 2211
3.100.4 Output info . . . 2211
3.100.5 Restrictions . . . 2211
3.100.6 Related commands . . . 2211
3.100.7 Default . . . 2211
3.101 compute rigid/local command . . . 2212
3.101.1 Syntax . . . 2212
3.101.2 Examples . . . 2212
3.101.3 Description . . . 2212
3.101.4 Output info . . . 2214
3.101.5 Restrictions . . . 2214
3.101.6 Related commands . . . 2214

3.101.7 Default . . . 2214
3.102 compute saed command . . . 2214
3.102.1 Syntax . . . 2214
3.102.2 Examples . . . 2215
3.102.3 Description . . . 2215
3.102.4 Output info . . . 2218
3.102.5 Restrictions . . . 2218
3.102.6 Related commands . . . 2218
3.102.7 Default . . . 2218
3.103 compute slcsa/atom command . . . 2218
3.103.1 Syntax . . . 2218
3.103.2 Examples . . . 2219
3.103.3 Description . . . 2219
3.103.4 Output info . . . 2220
3.103.5 Restrictions . . . 2220
3.103.6 Related commands . . . 2220
3.103.7 Default . . . 2220
3.104 compute slice command . . . 2220
3.104.1 Syntax . . . 2220
3.104.2 Examples . . . 2221
3.104.3 Description . . . 2221
3.104.4 Output info . . . 2221
3.104.5 Restrictions . . . 2222
3.104.6 Related commands . . . 2222
3.104.7 Default . . . 2222
3.105 compute smd/contact/radius command . . . 2222
3.105.1 Syntax . . . 2222
3.105.2 Examples . . . 2222
3.105.3 Description . . . 2222
3.105.4 Output info . . . 2222
3.105.5 Restrictions . . . 2222
3.105.6 Related commands . . . 2222
3.105.7 Default . . . 2223
3.106 compute smd/damage command . . . 2223
3.106.1 Syntax . . . 2223
3.106.2 Examples . . . 2223
3.106.3 Description . . . 2223
3.106.4 Restrictions . . . 2223
3.106.5 Related commands . . . 2223
3.106.6 Default . . . 2223
3.107 compute smd/hourglass/error command . . . 2223
3.107.1 Syntax . . . 2223
3.107.2 Examples . . . 2224
3.107.3 Description . . . 2224
3.107.4 Restrictions . . . 2224
3.107.5 Related commands . . . 2224
3.107.6 Default . . . 2224
3.108 compute smd/internal/energy command . . . 2224
3.108.1 Syntax . . . 2224
3.108.2 Examples . . . 2224
3.108.3 Description . . . 2224
3.108.4 Output Info . . . 2225
3.108.5 Restrictions . . . 2225
3.108.6 Related commands . . . 2225

3.108.7 Default . . . 2225
3.109 compute smd/plastic/strain command . . . 2225
3.109.1 Syntax . . . 2225
3.109.2 Examples . . . 2225
3.109.3 Description . . . 2225
3.109.4 Restrictions . . . 2225
3.109.5 Related commands . . . 2226
3.109.6 Default . . . 2226
3.110 compute smd/plastic/strain/rate command . . . 2226
3.110.1 Syntax . . . 2226
3.110.2 Examples . . . 2226
3.110.3 Description . . . 2226
3.110.4 Restrictions . . . 2226
3.110.5 Related commands . . . 2226
3.110.6 Default . . . 2226
3.111 compute smd/rho command . . . 2226
3.111.1 Syntax . . . 2226
3.111.2 Examples . . . 2227
3.111.3 Description . . . 2227
3.111.4 Output info . . . 2227
3.111.5 Restrictions . . . 2227
3.111.6 Related commands . . . 2227
3.111.7 Default . . . 2227
3.112 compute smd/tlsph/defgrad command . . . 2227
3.112.1 Syntax . . . 2227
3.112.2 Examples . . . 2227
3.112.3 Description . . . 2228
3.112.4 Output info . . . 2228
3.112.5 Restrictions . . . 2228
3.112.6 Related commands . . . 2228
3.112.7 Default . . . 2228
3.113 compute smd/tlsph/dt command . . . 2228
3.113.1 Syntax . . . 2228
3.113.2 Examples . . . 2228
3.113.3 Description . . . 2228
3.113.4 Output info . . . 2229
3.113.5 Restrictions . . . 2229
3.113.6 Related commands . . . 2229
3.113.7 Default . . . 2229
3.114 compute smd/tlsph/num/neighs command . . . 2229
3.114.1 Syntax . . . 2229
3.114.2 Examples . . . 2229
3.114.3 Description . . . 2229
3.114.4 Output info . . . 2229
3.114.5 Restrictions . . . 2229
3.114.6 Related commands . . . 2230
3.114.7 Default . . . 2230
3.115 compute smd/tlsph/shape command . . . 2230
3.115.1 Syntax . . . 2230
3.115.2 Examples . . . 2230
3.115.3 Description . . . 2230
3.115.4 Output info . . . 2230
3.115.5 Restrictions . . . 2230
3.115.6 Related commands . . . 2230

3.115.7 Default . . . 2230
3.116 compute smd/tlsph/strain command . . . 2231
3.116.1 Syntax . . . 2231
3.116.2 Examples . . . 2231
3.116.3 Description . . . 2231
3.116.4 Output info . . . 2231
3.116.5 Restrictions . . . 2231
3.116.6 Related commands . . . 2231
3.116.7 Default . . . 2231
3.117 compute smd/tlsph/strain/rate command . . . 2231
3.117.1 Syntax . . . 2231
3.117.2 Examples . . . 2232
3.117.3 Description . . . 2232
3.117.4 Output info . . . 2232
3.117.5 Restrictions . . . 2232
3.117.6 Related commands . . . 2232
3.117.7 Default . . . 2232
3.118 compute smd/tlsph/stress command . . . 2232
3.118.1 Syntax . . . 2232
3.118.2 Examples . . . 2232
3.118.3 Description . . . 2233
3.118.4 Output info . . . 2233
3.118.5 Restrictions . . . 2233
3.118.6 Related commands . . . 2233
3.118.7 Default . . . 2233
3.119 compute smd/triangle/vertices command . . . 2233
3.119.1 Syntax . . . 2233
3.119.2 Examples . . . 2233
3.119.3 Description . . . 2233
3.119.4 Output info . . . 2234
3.119.5 Restrictions . . . 2234
3.119.6 Related commands . . . 2234
3.119.7 Default . . . 2234
3.120 compute smd/ulsph/effm command . . . 2234
3.120.1 Syntax . . . 2234
3.120.2 Examples . . . 2234
3.120.3 Description . . . 2234
3.120.4 Output info . . . 2234
3.120.5 Restrictions . . . 2235
3.120.6 Related commands . . . 2235
3.120.7 Default . . . 2235
3.121 compute smd/ulsph/num/neighs command . . . 2235
3.121.1 Syntax . . . 2235
3.121.2 Examples . . . 2235
3.121.3 Description . . . 2235
3.121.4 Output info . . . 2235
3.121.5 Restrictions . . . 2235
3.121.6 Related commands . . . 2235
3.121.7 Default . . . 2236
3.122 compute smd/ulsph/strain command . . . 2236
3.122.1 Syntax . . . 2236
3.122.2 Examples . . . 2236
3.122.3 Description . . . 2236
3.122.4 Output info . . . 2236

3.122.5 Restrictions . . . 2236
3.122.6 Related commands . . . 2236
3.122.7 Default . . . 2236
3.123 compute smd/ulsph/strain/rate command . . . 2236
3.123.1 Syntax . . . 2236
3.123.2 Examples . . . 2237
3.123.3 Description . . . 2237
3.123.4 Output info . . . 2237
3.123.5 Restrictions . . . 2237
3.123.6 Related commands . . . 2237
3.123.7 Default . . . 2237
3.124 compute smd/ulsph/stress command . . . 2237
3.124.1 Syntax . . . 2237
3.124.2 Examples . . . 2237
3.124.3 Description . . . 2238
3.124.4 Output info . . . 2238
3.124.5 Restrictions . . . 2238
3.124.6 Related commands . . . 2238
3.124.7 Default . . . 2238
3.125 compute smd/vol command . . . 2238
3.125.1 Syntax . . . 2238
3.125.2 Examples . . . 2238
3.125.3 Description . . . 2238
3.125.4 Output info . . . 2239
3.125.5 Restrictions . . . 2239
3.125.6 Related commands . . . 2239
3.125.7 Default . . . 2239
3.126 compute sna/atom command . . . 2239
3.127 compute snad/atom command . . . 2239
3.128 compute snav/atom command . . . 2239
3.129 compute snap command . . . 2239
3.130 compute sna/grid command . . . 2239
3.131 compute sna/grid/local command . . . 2239
3.131.1 Syntax . . . 2239
3.131.2 Examples . . . 2241
3.131.3 Description . . . 2241
3.131.4 Output info . . . 2245
3.131.5 Restrictions . . . 2247
3.131.6 Related commands . . . 2247
3.131.7 Default . . . 2247
3.132 compute sph/e/atom command . . . 2247
3.132.1 Syntax . . . 2247
3.132.2 Examples . . . 2247
3.132.3 Description . . . 2248
3.132.4 Output info . . . 2248
3.132.5 Restrictions . . . 2248
3.132.6 Related commands . . . 2248
3.132.7 Default . . . 2248
3.133 compute sph/rho/atom command . . . 2248
3.133.1 Syntax . . . 2248
3.133.2 Examples . . . 2248
3.133.3 Description . . . 2249
3.133.4 Output info . . . 2249
3.133.5 Restrictions . . . 2249

3.133.6 Related commands . . . 2249
3.133.7 Default . . . 2249
3.134 compute sph/t/atom command . . . 2249
3.134.1 Syntax . . . 2249
3.134.2 Examples . . . 2249
3.134.3 Description . . . 2250
3.134.4 Output info . . . 2250
3.134.5 Restrictions . . . 2250
3.134.6 Related commands . . . 2250
3.134.7 Default . . . 2250
3.135 compute spin command . . . 2250
3.135.1 Syntax . . . 2250
3.135.2 Examples . . . 2250
3.135.3 Description . . . 2251
3.135.4 Output info . . . 2251
3.135.5 Restrictions . . . 2251
3.135.6 Default . . . 2251
3.136 compute stress/atom command . . . 2251
3.137 compute centroid/stress/atom command . . . 2251
3.137.1 Syntax . . . 2251
3.137.2 Examples . . . 2252
3.137.3 Description . . . 2252
3.137.4 Output info . . . 2254
3.137.5 Restrictions . . . 2255
3.137.6 Related commands . . . 2255
3.137.7 Default . . . 2255
3.138 compute stress/cartesian command . . . 2255
3.138.1 Syntax . . . 2255
3.138.2 Examples . . . 2255
3.138.3 Description . . . 2256
3.138.4 Output info . . . 2256
3.138.5 Restrictions . . . 2256
3.138.6 Related commands . . . 2256
3.139 compute stress/cylinder command . . . 2257
3.140 compute stress/spherical command . . . 2257
3.140.1 Syntax . . . 2257
3.140.2 Examples . . . 2257
3.140.3 Description . . . 2257
3.140.4 Output info . . . 2257
3.140.5 Restrictions . . . 2258
3.140.6 Related commands . . . 2258
3.140.7 Default . . . 2258
3.141 compute stress/mop command . . . 2258
3.142 compute stress/mop/profile command . . . 2258
3.142.1 Syntax . . . 2258
3.142.2 Examples . . . 2259
3.142.3 Description . . . 2259
3.142.4 Output info . . . 2259
3.142.5 Restrictions . . . 2260
3.142.6 Related commands . . . 2260
3.142.7 Default . . . 2260
3.143 compute force/tally command . . . 2260
3.144 compute heat/flux/tally command . . . 2260
3.145 compute heat/flux/virial/tally command . . . 2260

3.146 compute pe/tally command . . . 2260
3.147 compute pe/mol/tally command . . . 2260
3.148 compute stress/tally command . . . 2260
3.148.1 Syntax . . . 2260
3.148.2 Examples . . . 2261
3.148.3 Description . . . 2261
3.148.4 Output info . . . 2262
3.148.5 Restrictions . . . 2263
3.148.6 Related commands . . . 2263
3.148.7 Default . . . 2263
3.149 compute tdpd/cc/atom command . . . 2263
3.149.1 Syntax . . . 2263
3.149.2 Examples . . . 2263
3.149.3 Description . . . 2263
3.149.4 Output info . . . 2263
3.149.5 Restrictions . . . 2264
3.149.6 Related commands . . . 2264
3.149.7 Default . . . 2264
3.150 compute temp command . . . 2264
3.150.1 Syntax . . . 2264
3.150.2 Examples . . . 2264
3.150.3 Description . . . 2264
3.150.4 Output info . . . 2265
3.150.5 Restrictions . . . 2265
3.150.6 Related commands . . . 2265
3.150.7 Default . . . 2265
3.151 compute temp/asphere command . . . 2265
3.151.1 Syntax . . . 2265
3.151.2 Examples . . . 2266
3.151.3 Description . . . 2266
3.151.4 Output info . . . 2267
3.151.5 Restrictions . . . 2267
3.151.6 Related commands . . . 2268
3.151.7 Default . . . 2268
3.152 compute temp/body command . . . 2268
3.152.1 Syntax . . . 2268
3.152.2 Examples . . . 2268
3.152.3 Description . . . 2268
3.152.4 Output info . . . 2269
3.152.5 Restrictions . . . 2269
3.152.6 Related commands . . . 2269
3.152.7 Default . . . 2270
3.153 compute temp/chunk command . . . 2270
3.153.1 Syntax . . . 2270
3.153.2 Examples . . . 2270
3.153.3 Description . . . 2270
3.153.4 Output info . . . 2273
3.153.5 Restrictions . . . 2273
3.153.6 Related commands . . . 2273
3.153.7 Default . . . 2273
3.154 compute temp/com command . . . 2273
3.154.1 Syntax . . . 2273
3.154.2 Examples . . . 2273
3.154.3 Description . . . 2273

3.154.4 Output info . . . 2274
3.154.5 Restrictions . . . 2274
3.154.6 Related commands . . . 2274
3.154.7 Default . . . 2275
3.155 compute temp/cs command . . . 2275
3.155.1 Syntax . . . 2275
3.155.2 Examples . . . 2275
3.155.3 Description . . . 2275
3.155.4 Output info . . . 2276
3.155.5 Restrictions . . . 2276
3.155.6 Related commands . . . 2276
3.155.7 Default . . . 2276
3.156 compute temp/deform command . . . 2276
3.156.1 Syntax . . . 2276
3.156.2 Examples . . . 2277
3.156.3 Description . . . 2277
3.156.4 Output info . . . 2279
3.156.5 Restrictions . . . 2279
3.156.6 Related commands . . . 2279
3.156.7 Default . . . 2279
3.157 compute temp/deform/eff command . . . 2279
3.157.1 Syntax . . . 2279
3.157.2 Examples . . . 2279
3.157.3 Description . . . 2280
3.157.4 Output info . . . 2280
3.157.5 Restrictions . . . 2280
3.157.6 Related commands . . . 2280
3.157.7 Default . . . 2280
3.158 compute temp/drude command . . . 2280
3.158.1 Syntax . . . 2280
3.158.2 Examples . . . 2281
3.158.3 Description . . . 2281
3.158.4 Output info . . . 2281
3.158.5 Restrictions . . . 2281
3.158.6 Related commands . . . 2281
3.158.7 Default . . . 2282
3.159 compute temp/eff command . . . 2282
3.159.1 Syntax . . . 2282
3.159.2 Examples . . . 2282
3.159.3 Description . . . 2282
3.159.4 Output info . . . 2283
3.159.5 Restrictions . . . 2283
3.159.6 Related commands . . . 2283
3.159.7 Default . . . 2283
3.160 compute temp/partial command . . . 2283
3.160.1 Syntax . . . 2283
3.160.2 Examples . . . 2283
3.160.3 Description . . . 2283
3.160.4 Output info . . . 2284
3.160.5 Restrictions . . . 2284
3.160.6 Related commands . . . 2284
3.160.7 Default . . . 2284
3.161 compute temp/profile command . . . 2284
3.161.1 Syntax . . . 2284

3.161.2 Examples . . . 2285
3.161.3 Description . . . 2285
3.161.4 Output info . . . 2286
3.161.5 Restrictions . . . 2287
3.161.6 Related commands . . . 2287
3.161.7 Default . . . 2287
3.162 compute temp/ramp command . . . 2287
3.162.1 Syntax . . . 2287
3.162.2 Examples . . . 2287
3.162.3 Description . . . 2287
3.162.4 Output info . . . 2288
3.162.5 Restrictions . . . 2288
3.162.6 Related commands . . . 2288
3.162.7 Default . . . 2288
3.163 compute temp/region command . . . 2289
3.163.1 Syntax . . . 2289
3.163.2 Examples . . . 2289
3.163.3 Description . . . 2289
3.163.4 Output info . . . 2290
3.163.5 Restrictions . . . 2290
3.163.6 Related commands . . . 2290
3.163.7 Default . . . 2290
3.164 compute temp/region/eff command . . . 2290
3.164.1 Syntax . . . 2290
3.164.2 Examples . . . 2290
3.164.3 Description . . . 2290
3.164.4 Output info . . . 2290
3.164.5 Restrictions . . . 2291
3.164.6 Related commands . . . 2291
3.164.7 Default . . . 2291
3.165 compute temp/rotate command . . . 2291
3.165.1 Syntax . . . 2291
3.165.2 Examples . . . 2291
3.165.3 Description . . . 2291
3.165.4 Output info . . . 2292
3.165.5 Restrictions . . . 2292
3.165.6 Related commands . . . 2292
3.165.7 Default . . . 2292
3.166 compute temp/sphere command . . . 2292
3.166.1 Syntax . . . 2292
3.166.2 Examples . . . 2293
3.166.3 Description . . . 2293
3.166.4 Output info . . . 2294
3.166.5 Restrictions . . . 2294
3.166.6 Related commands . . . 2294
3.166.7 Default . . . 2294
3.167 compute temp/uef command . . . 2294
3.167.1 Syntax . . . 2294
3.167.2 Examples . . . 2295
3.167.3 Description . . . 2295
3.167.4 Restrictions . . . 2295
3.167.5 Related commands . . . 2295
3.167.6 Default . . . 2295
3.168 compute ti command . . . 2295

3.168.1 Syntax . . . 2295
3.168.2 Examples . . . 2296
3.168.3 Description . . . 2296
3.168.4 Output info . . . 2296
3.168.5 Restrictions . . . 2297
3.168.6 Related commands . . . 2297
3.168.7 Default . . . 2297
3.169 compute torque/chunk command . . . 2297
3.169.1 Syntax . . . 2297
3.169.2 Examples . . . 2297
3.169.3 Description . . . 2297
3.169.4 Output info . . . 2298
3.169.5 Restrictions . . . 2298
3.169.6 Related commands . . . 2298
3.169.7 Default . . . 2298
3.170 compute vacf command . . . 2298
3.170.1 Syntax . . . 2298
3.170.2 Examples . . . 2298
3.170.3 Description . . . 2299
3.170.4 Output info . . . 2299
3.170.5 Restrictions . . . 2299
3.170.6 Related commands . . . 2299
3.170.7 Default . . . 2299
3.171 compute vacf/chunk command . . . 2299
3.171.1 Syntax . . . 2299
3.171.2 Examples . . . 2300
3.171.3 Description . . . 2300
3.171.4 Output info . . . 2301
3.171.5 Restrictions . . . 2301
3.171.6 Related commands . . . 2301
3.171.7 Default . . . 2301
3.172 compute vcm/chunk command . . . 2301
3.172.1 Syntax . . . 2301
3.172.2 Examples . . . 2301
3.172.3 Description . . . 2301
3.172.4 Output info . . . 2302
3.172.5 Restrictions . . . 2302
3.172.6 Related commands . . . 2302
3.172.7 Default . . . 2302
3.173 compute viscosity/cos command . . . 2302
3.173.1 Syntax . . . 2302
3.173.2 Examples . . . 2302
3.173.3 Description . . . 2303
3.173.4 Output info . . . 2304
3.173.5 Restrictions . . . 2304
3.173.6 Related commands . . . 2304
3.173.7 Default . . . 2304
3.174 compute voronoi/atom command . . . 2304
3.174.1 Syntax . . . 2304
3.174.2 Examples . . . 2305
3.174.3 Description . . . 2305
3.174.4 Output info . . . 2306
3.174.5 Restrictions . . . 2307
3.174.6 Related commands . . . 2307

3.174.7 Default . . . 2307
3.175 compute xrd command . . . 2307
3.175.1 Syntax . . . 2307
3.175.2 Examples . . . 2308
3.175.3 Description . . . 2308
3.175.4 Output info . . . 2310
3.175.5 Restrictions . . . 2310
3.175.6 Related commands . . . 2310
3.175.7 Default . . . 2310

**4 Pair Styles 2311**
4.1 pair_style adp command . . . 2311
4.1.1 Syntax . . . 2311
4.1.2 Examples . . . 2311
4.1.3 Description . . . 2311
4.1.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2313
4.1.5 Restrictions . . . 2313
4.1.6 Related commands . . . 2313
4.1.7 Default . . . 2313
4.2 pair_style agni command . . . 2313
4.2.1 Syntax . . . 2313
4.2.2 Examples . . . 2314
4.2.3 Description . . . 2314
4.2.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2315
4.2.5 Restrictions . . . 2315
4.2.6 Related commands . . . 2315
4.2.7 Default . . . 2315
4.3 pair_style aip/water/2dm command . . . 2315
4.3.1 Syntax . . . 2315
4.3.2 Examples . . . 2315
4.3.3 Description . . . 2316
4.3.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2317
4.3.5 Restrictions . . . 2317
4.3.6 Related commands . . . 2318
4.3.7 Default . . . 2318
4.4 pair_style airebo command . . . 2318
4.5 pair_style airebo/morse command . . . 2318
4.6 pair_style airebo/bc command . . . 2318
4.7 pair_style rebo command . . . 2318
4.7.1 Syntax . . . 2318
4.7.2 Examples . . . 2318
4.7.3 Description . . . 2319
4.7.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2321
4.7.5 Restrictions . . . 2321
4.7.6 Related commands . . . 2322
4.7.7 Default . . . 2322
4.8 pair_style amoeba command . . . 2322
4.9 pair_style hippo command . . . 2322
4.9.1 Syntax . . . 2322
4.9.2 Examples . . . 2322
4.9.3 Additional info . . . 2322
4.9.4 Description . . . 2323
4.9.5 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2324
4.9.6 Restrictions . . . 2325

4.9.7 Related commands . . . 2326
4.9.8 Default . . . 2326
4.10 pair_style atm command . . . 2326
4.10.1 Syntax . . . 2326
4.10.2 Examples . . . 2326
4.10.3 Description . . . 2326
4.10.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2328
4.10.5 Restrictions . . . 2328
4.10.6 Related commands . . . 2328
4.10.7 Default . . . 2329
4.11 pair_style beck command . . . 2329
4.11.1 Syntax . . . 2329
4.11.2 Examples . . . 2329
4.11.3 Description . . . 2329
4.11.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2330
4.11.5 Restrictions . . . 2330
4.11.6 Related commands . . . 2330
4.11.7 Default . . . 2330
4.12 pair_style body/nparticle command . . . 2330
4.12.1 Syntax . . . 2330
4.12.2 Examples . . . 2330
4.12.3 Description . . . 2331
4.12.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2332
4.12.5 Restrictions . . . 2332
4.12.6 Related commands . . . 2332
4.12.7 Default . . . 2332
4.13 pair_style body/rounded/polygon command . . . 2332
4.13.1 Syntax . . . 2332
4.13.2 Examples . . . 2332
4.13.3 Description . . . 2333
4.13.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2335
4.13.5 Restrictions . . . 2335
4.13.6 Related commands . . . 2335
4.13.7 Default . . . 2335
4.14 pair_style body/rounded/polyhedron command . . . 2335
4.14.1 Syntax . . . 2335
4.14.2 Examples . . . 2336
4.14.3 Description . . . 2336
4.14.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2338
4.14.5 Restrictions . . . 2338
4.14.6 Related commands . . . 2338
4.14.7 Default . . . 2338
4.15 pair_style bondval command . . . 2338
4.16 pair_style bondval/vec command . . . 2338
4.16.1 Syntax . . . 2338
4.16.2 Examples . . . 2339
4.16.3 Description . . . 2339
4.16.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2341
4.16.5 Restrictions . . . 2341
4.16.6 Related commands . . . 2341
4.16.7 Default . . . 2341
4.17 pair_style bop command . . . 2342
4.17.1 Syntax . . . 2342
4.17.2 Examples . . . 2342

4.17.3 Description . . . 2342
4.17.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2347
4.17.5 Restrictions . . . 2347
4.17.6 Related commands . . . 2347
4.17.7 Default . . . 2347
4.18 pair_style born command . . . 2347
4.19 pair_style born/coul/long command . . . 2347
4.20 pair_style born/coul/msm command . . . 2348
4.21 pair_style born/coul/wolf command . . . 2348
4.22 pair_style born/coul/dsf command . . . 2348
4.22.1 Syntax . . . 2348
4.22.2 Examples . . . 2348
4.22.3 Description . . . 2349
4.22.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2350
4.22.5 Restrictions . . . 2350
4.22.6 Related commands . . . 2350
4.22.7 Default . . . 2350
4.23 pair_style born/gauss command . . . 2350
4.23.1 Syntax . . . 2351
4.23.2 Examples . . . 2351
4.23.3 Description . . . 2351
4.23.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2352
4.23.5 Restrictions . . . 2352
4.23.6 Related commands . . . 2352
4.23.7 Default . . . 2352
4.24 pair_style bpm/spring command . . . 2352
4.24.1 Syntax . . . 2352
4.24.2 Examples . . . 2352
4.24.3 Description . . . 2353
4.24.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2353
4.24.5 Restrictions . . . 2354
4.24.6 Related commands . . . 2354
4.24.7 Default . . . 2354
4.25 pair_style brownian command . . . 2354
4.26 pair_style brownian/poly command . . . 2354
4.26.1 Syntax . . . 2354
4.26.2 Examples . . . 2354
4.26.3 Description . . . 2355
4.26.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2355
4.26.5 Restrictions . . . 2356
4.26.6 Related commands . . . 2356
4.26.7 Default . . . 2356
4.27 pair_style buck command . . . 2356
4.28 pair_style buck/coul/cut command . . . 2356
4.29 pair_style buck/coul/long command . . . 2356
4.30 pair_style buck/coul/msm command . . . 2356
4.30.1 Syntax . . . 2356
4.30.2 Examples . . . 2357
4.30.3 Description . . . 2357
4.30.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2358
4.30.5 Restrictions . . . 2359
4.30.6 Related commands . . . 2359
4.30.7 Default . . . 2359
4.31 pair_style buck6d/coul/gauss/dsf command . . . 2359

4.32 pair_style buck6d/coul/gauss/long command . . . 2359
4.32.1 Syntax . . . 2359
4.32.2 Examples . . . 2359
4.32.3 Description . . . 2359
4.32.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2361
4.32.5 Restrictions . . . 2361
4.32.6 Related commands . . . 2361
4.32.7 Default . . . 2361
4.33 pair_style buck/long/coul/long command . . . 2361
4.33.1 Syntax . . . 2361
4.33.2 Examples . . . 2361
4.33.3 Description . . . 2362
4.33.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2363
4.33.5 Restrictions . . . 2363
4.33.6 Related commands . . . 2363
4.33.7 Default . . . 2363
4.34 pair_style lj/charmm/coul/charmm command . . . 2363
4.35 pair_style lj/charmm/coul/charmm/implicit command . . . 2363
4.36 pair_style lj/charmm/coul/long command . . . 2364
4.37 pair_style lj/charmm/coul/msm command . . . 2364
4.38 pair_style lj/charmmfsw/coul/charmmfsh command . . . 2364
4.39 pair_style lj/charmmfsw/coul/long command . . . 2364
4.39.1 Syntax . . . 2364
4.39.2 Examples . . . 2365
4.39.3 Description . . . 2365
4.39.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2367
4.39.5 Restrictions . . . 2367
4.39.6 Related commands . . . 2367
4.39.7 Default . . . 2367
4.40 pair_style lj/class2 command . . . 2367
4.41 pair_style lj/class2/coul/cut command . . . 2367
4.42 pair_style lj/class2/coul/long command . . . 2368
4.42.1 Syntax . . . 2368
4.42.2 Examples . . . 2368
4.42.3 Description . . . 2368
4.42.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2369
4.42.5 Restrictions . . . 2370
4.42.6 Related commands . . . 2370
4.42.7 Default . . . 2370
4.43 pair_style colloid command . . . 2370
4.43.1 Syntax . . . 2370
4.43.2 Examples . . . 2370
4.43.3 Description . . . 2371
4.43.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2372
4.43.5 Restrictions . . . 2373
4.43.6 Related commands . . . 2373
4.43.7 Default . . . 2373
4.44 pair_style comb command . . . 2373
4.45 pair_style comb3 command . . . 2373
4.45.1 Syntax . . . 2373
4.45.2 Examples . . . 2373
4.45.3 Description . . . 2374
4.45.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2375
4.45.5 Restrictions . . . 2376

4.45.6 Related commands . . . 2376
4.45.7 Default . . . 2376
4.46 pair_style cosine/squared command . . . 2376
4.46.1 Syntax . . . 2376
4.46.2 Examples . . . 2377
4.46.3 Description . . . 2377
4.46.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2378
4.46.5 Restrictions . . . 2378
4.46.6 Related commands . . . 2378
4.46.7 Default . . . 2378
4.47 pair_style coul/cut command . . . 2378
4.48 pair_style coul/cut/global command . . . 2378
4.49 pair_style coul/ctip command . . . 2379
4.50 pair_style coul/debye command . . . 2379
4.51 pair_style coul/dsf command . . . 2379
4.52 pair_style coul/exclude command . . . 2379
4.53 pair_style coul/long command . . . 2379
4.54 pair_style coul/msm command . . . 2379
4.55 pair_style coul/streitz command . . . 2379
4.56 pair_style coul/wolf command . . . 2379
4.57 pair_style tip4p/cut command . . . 2379
4.58 pair_style tip4p/long command . . . 2379
4.58.1 Syntax . . . 2379
4.58.2 Examples . . . 2380
4.58.3 Description . . . 2381
4.58.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2384
4.58.5 Restrictions . . . 2385
4.58.6 Related commands . . . 2385
4.58.7 Default . . . 2385
4.59 pair_style coul/diel command . . . 2385
4.59.1 Syntax . . . 2385
4.59.2 Examples . . . 2385
4.59.3 Description . . . 2385
4.59.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2386
4.59.5 Restrictions . . . 2386
4.59.6 Related commands . . . 2386
4.59.7 Default . . . 2387
4.60 pair_style coul/shield command . . . 2387
4.60.1 Syntax . . . 2387
4.60.2 Examples . . . 2387
4.60.3 Description . . . 2387
4.60.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2388
4.60.5 Restrictions . . . 2388
4.60.6 Related commands . . . 2388
4.60.7 Default . . . 2388
4.61 pair_style coul/slater command . . . 2388
4.62 pair_style coul/slater/cut command . . . 2388
4.63 pair_style coul/slater/long command . . . 2389
4.63.1 Syntax . . . 2389
4.63.2 Examples . . . 2389
4.63.3 Description . . . 2389
4.63.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2390
4.63.5 Restrictions . . . 2390
4.63.6 Related commands . . . 2390

4.63.7 Default . . . 2390
4.64 pair_style coul/tt command . . . 2390
4.64.1 Syntax . . . 2390
4.64.2 Examples . . . 2391
4.64.3 Description . . . 2391
4.64.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2391
4.64.5 Restrictions . . . 2392
4.64.6 Related commands . . . 2392
4.64.7 Default . . . 2392
4.65 pair_style born/coul/dsf/cs command . . . 2392
4.66 pair_style born/coul/long/cs command . . . 2392
4.67 pair_style born/coul/wolf/cs command . . . 2392
4.68 pair_style buck/coul/long/cs command . . . 2392
4.69 pair_style coul/long/cs command . . . 2392
4.70 pair_style coul/wolf/cs command . . . 2392
4.71 pair_style lj/cut/coul/long/cs command . . . 2392
4.72 pair_style lj/class2/coul/long/cs command . . . 2393
4.72.1 Syntax . . . 2393
4.72.2 Examples . . . 2393
4.72.3 Description . . . 2394
4.72.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2395
4.72.5 Restrictions . . . 2395
4.72.6 Related commands . . . 2395
4.72.7 Default . . . 2395
4.73 pair_style coul/cut/dielectric command . . . 2395
4.74 pair_style coul/long/dielectric command . . . 2395
4.75 pair_style lj/cut/coul/cut/dielectric command . . . 2395
4.76 pair_style lj/cut/coul/debye/dielectric command . . . 2395
4.77 pair_style lj/cut/coul/long/dielectric command . . . 2395
4.78 pair_style lj/cut/coul/msm/dielectric command . . . 2396
4.79 pair_style lj/long/coul/long/dielectric command . . . 2396
4.79.1 Syntax . . . 2396
4.79.2 Examples . . . 2396
4.79.3 Description . . . 2396
4.79.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2397
4.79.5 Restrictions . . . 2397
4.79.6 Related commands . . . 2397
4.79.7 Default . . . 2397
4.80 pair_style lj/cut/dipole/cut command . . . 2397
4.81 pair_style lj/sf/dipole/sf command . . . 2397
4.82 pair_style lj/cut/dipole/long command . . . 2397
4.83 pair_style lj/long/dipole/long command . . . 2397
4.83.1 Syntax . . . 2397
4.83.2 Examples . . . 2398
4.83.3 Description . . . 2398
4.83.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2402
4.83.5 Restrictions . . . 2402
4.83.6 Related commands . . . 2403
4.83.7 Default . . . 2403
4.84 pair_style dispersion/d3 command . . . 2403
4.84.1 Syntax . . . 2403
4.84.2 Examples . . . 2403
4.84.3 Description . . . 2403
4.84.4 Coefficients . . . 2404

4.84.5 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2404
4.84.6 Restrictions . . . 2405
4.84.7 Related commands . . . 2405
4.84.8 Default . . . 2405
4.85 pair_style dpd command . . . 2405
4.86 pair_style dpd/tstat command . . . 2405
4.86.1 Syntax . . . 2405
4.86.2 Examples . . . 2405
4.86.3 Description . . . 2406
4.86.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2407
4.86.5 Restrictions . . . 2408
4.86.6 Related commands . . . 2408
4.86.7 Default . . . 2408
4.87 pair_style dpd/coul/slater/long command . . . 2408
4.87.1 Syntax . . . 2408
4.87.2 Examples . . . 2408
4.87.3 Description . . . 2409
4.87.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2410
4.87.5 Restrictions . . . 2410
4.87.6 Related commands . . . 2411
4.87.7 Default . . . 2411
4.88 pair_style dpd/ext command . . . 2411
4.89 pair_style dpd/ext/tstat command . . . 2411
4.89.1 Syntax . . . 2411
4.89.2 Examples . . . 2411
4.89.3 Description . . . 2411
4.89.4 Restrictions . . . 2414
4.89.5 Related commands . . . 2414
4.90 pair_style dpd/fdt command . . . 2414
4.91 pair_style dpd/fdt/energy command . . . 2414
4.91.1 Syntax . . . 2414
4.91.2 Examples . . . 2414
4.91.3 Description . . . 2415
4.91.4 Restrictions . . . 2416
4.91.5 Related commands . . . 2416
4.91.6 Default . . . 2416
4.92 pair_style drip command . . . 2417
4.92.1 Syntax . . . 2417
4.92.2 Examples . . . 2417
4.92.3 Description . . . 2417
4.92.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2418
4.92.5 Restrictions . . . 2418
4.92.6 Related commands . . . 2419
4.93 pair_style dsmc command . . . 2419
4.93.1 Syntax . . . 2419
4.93.2 Examples . . . 2419
4.93.3 Description . . . 2419
4.93.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2420
4.93.5 Restrictions . . . 2420
4.93.6 Related commands . . . 2420
4.93.7 Default . . . 2421
4.94 pair_style e3b command . . . 2421
4.94.1 Syntax . . . 2421
4.94.2 Examples . . . 2421

4.94.3 Description . . . 2422
4.94.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2423
4.94.5 Restrictions . . . 2423
4.94.6 Related commands . . . 2423
4.94.7 Default . . . 2424
4.95 pair_style eam command . . . 2424
4.96 pair_style eam/alloy command . . . 2424
4.97 pair_style eam/cd command . . . 2424
4.98 pair_style eam/cd/old command . . . 2424
4.99 pair_style eam/fs command . . . 2424
4.100 pair_style eam/he command . . . 2424
4.100.1 Syntax . . . 2424
4.100.2 Examples . . . 2424
4.100.3 Description . . . 2425
4.100.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2430
4.100.5 Restrictions . . . 2430
4.100.6 Related commands . . . 2430
4.100.7 Default . . . 2430
4.101 pair_style eam/apip command . . . 2431
4.102 pair_style eam/fs/apip command . . . 2431
4.102.1 Syntax . . . 2431
4.102.2 Examples . . . 2431
4.102.3 Description . . . 2431
4.102.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2431
4.102.5 Restrictions . . . 2432
4.102.6 Related commands . . . 2432
4.102.7 Default . . . 2432
4.103 pair_style edip command . . . 2432
4.104 pair_style edip/multi command . . . 2432
4.104.1 Syntax . . . 2432
4.104.2 Examples . . . 2432
4.104.3 Description . . . 2432
4.104.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2434
4.104.5 Restrictions . . . 2435
4.104.6 Related commands . . . 2435
4.104.7 Default . . . 2435
4.105 pair_style eff/cut command . . . 2435
4.105.1 Syntax . . . 2435
4.105.2 Examples . . . 2435
4.105.3 Description . . . 2435
4.105.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2438
4.105.5 Restrictions . . . 2438
4.105.6 Related commands . . . 2439
4.105.7 Default . . . 2439
4.106 pair_style eim command . . . 2439
4.106.1 Syntax . . . 2439
4.106.2 Examples . . . 2439
4.106.3 Description . . . 2439
4.106.4 Restrictions . . . 2441
4.106.5 Related commands . . . 2441
4.106.6 Default . . . 2441
4.107 pair_style exp6/rx command . . . 2441
4.107.1 Syntax . . . 2442
4.107.2 Examples . . . 2442

4.107.3 Description . . . 2442
4.107.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2444
4.107.5 Restrictions . . . 2444
4.107.6 Related commands . . . 2444
4.107.7 Default . . . 2444
4.108 pair_style extep command . . . 2444
4.108.1 Syntax . . . 2444
4.108.2 Examples . . . 2444
4.108.3 Description . . . 2445
4.108.4 Restrictions . . . 2445
4.108.5 Related commands . . . 2445
4.108.6 Default . . . 2445
4.109 pair_style lj/cut/soft command . . . 2445
4.110 pair_style lj/cut/soft/gapsys command . . . 2445
4.111 pair_style lj/cut/coul/cut/soft command . . . 2445
4.112 pair_style lj/cut/coul/long/soft command . . . 2445
4.113 pair_style lj/cut/tip4p/long/soft command . . . 2445
4.114 pair_style lj/charmm/coul/long/soft command . . . 2446
4.115 pair_style lj/class2/soft command . . . 2446
4.116 pair_style lj/class2/coul/cut/soft command . . . 2446
4.117 pair_style lj/class2/coul/long/soft command . . . 2446
4.118 pair_style coul/cut/soft command . . . 2446
4.119 pair_style coul/cut/soft/gapsys command . . . 2446
4.120 pair_style coul/long/soft command . . . 2446
4.121 pair_style tip4p/long/soft command . . . 2446
4.122 pair_style morse/soft command . . . 2446
4.122.1 Syntax . . . 2446
4.122.2 Examples . . . 2448
4.122.3 Description . . . 2449
4.122.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2453
4.122.5 Restrictions . . . 2454
4.122.6 Related commands . . . 2454
4.122.7 Default . . . 2454
4.123 pair_style gauss command . . . 2454
4.124 pair_style gauss/cut command . . . 2454
4.124.1 Syntax . . . 2454
4.124.2 Examples . . . 2454
4.124.3 Description . . . 2455
4.124.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2456
4.124.5 Restrictions . . . 2456
4.124.6 Related commands . . . 2457
4.124.7 Default . . . 2457
4.125 pair_style gayberne command . . . 2457
4.125.1 Syntax . . . 2457
4.125.2 Examples . . . 2457
4.125.3 Description . . . 2457
4.125.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2459
4.125.5 Restrictions . . . 2459
4.125.6 Related commands . . . 2459
4.125.7 Default . . . 2460
4.126 pair_style gran/hooke command . . . 2460
4.127 pair_style gran/hooke/history command . . . 2460
4.128 pair_style gran/hertz/history command . . . 2460
4.128.1 Syntax . . . 2460

4.128.2 Examples . . . 2461
4.128.3 Description . . . 2461
4.128.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2463
4.128.5 Restrictions . . . 2463
4.128.6 Related commands . . . 2464
4.128.7 Default . . . 2464
4.129 pair_style granular command . . . 2464
4.129.1 Syntax . . . 2464
4.129.2 Examples . . . 2464
4.129.3 Description . . . 2465
4.129.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2475
4.129.5 Restrictions . . . 2476
4.129.6 Related commands . . . 2476
4.129.7 Default . . . 2477
4.129.8 References . . . 2477
4.130 pair_style granular/superellipsoid command . . . 2478
4.130.1 Syntax . . . 2478
4.130.2 Examples . . . 2478
4.130.3 Description . . . 2478
4.130.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2483
4.130.5 Restrictions . . . 2483
4.130.6 Related commands . . . 2484
4.130.7 Default . . . 2484
4.130.8 References . . . 2484
4.131 pair_style lj/gromacs command . . . 2484
4.132 pair_style lj/gromacs/coul/gromacs command . . . 2484
4.132.1 Syntax . . . 2484
4.132.2 Examples . . . 2485
4.132.3 Description . . . 2485
4.132.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2486
4.132.5 Restrictions . . . 2486
4.132.6 Related commands . . . 2486
4.132.7 Default . . . 2486
4.133 pair_style gw command . . . 2487
4.134 pair_style gw/zbl command . . . 2487
4.134.1 Syntax . . . 2487
4.134.2 Examples . . . 2487
4.134.3 Description . . . 2487
4.134.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2488
4.134.5 Restrictions . . . 2488
4.134.6 Related commands . . . 2488
4.134.7 Default . . . 2488
4.135 pair_style harmonic/cut command . . . 2488
4.135.1 Syntax . . . 2489
4.135.2 Examples . . . 2489
4.135.3 Description . . . 2489
4.135.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2489
4.135.5 Restrictions . . . 2490
4.135.6 Related commands . . . 2490
4.135.7 Default . . . 2490
4.136 pair_style hbond/dreiding/lj command . . . 2490
4.137 pair_style hbond/dreiding/lj/angleoffset command . . . 2490
4.138 pair_style hbond/dreiding/morse command . . . 2490
4.139 pair_style hbond/dreiding/morse/angleoffset command . . . 2490

4.139.1 Syntax . . . 2490
4.139.2 Examples . . . 2490
4.139.3 Description . . . 2491
4.139.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2494
4.139.5 Restrictions . . . 2494
4.139.6 Related commands . . . 2494
4.139.7 Default . . . 2494
4.140 pair_style hdnnp command . . . 2495
4.140.1 Syntax . . . 2495
4.140.2 Examples . . . 2495
4.140.3 Description . . . 2495
4.140.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2497
4.140.5 Restrictions . . . 2497
4.141 pair_style hybrid command . . . 2498
4.142 pair_style hybrid/molecular command . . . 2498
4.143 pair_style hybrid/overlay command . . . 2498
4.144 pair_style hybrid/scaled command . . . 2498
4.144.1 Syntax . . . 2498
4.144.2 Examples . . . 2498
4.144.3 Description . . . 2499
4.144.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2504
4.144.5 Restrictions . . . 2505
4.144.6 Related commands . . . 2505
4.144.7 Default . . . 2505
4.145 pair_style ilp/graphene/hbn command . . . 2505
4.145.1 Syntax . . . 2505
4.145.2 Examples . . . 2505
4.145.3 Description . . . 2506
4.145.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2507
4.145.5 Restrictions . . . 2507
4.145.6 Related commands . . . 2507
4.145.7 Default . . . 2508
4.146 pair_style ilp/tmd command . . . 2508
4.146.1 Syntax . . . 2508
4.146.2 Examples . . . 2508
4.146.3 Description . . . 2508
4.146.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2510
4.146.5 Restrictions . . . 2510
4.146.6 Related commands . . . 2510
4.146.7 Default . . . 2510
4.147 pair_style kim command . . . 2511
4.147.1 Syntax . . . 2511
4.147.2 Examples . . . 2511
4.147.3 Description . . . 2511
4.147.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2512
4.147.5 Restrictions . . . 2512
4.147.6 Related commands . . . 2512
4.147.7 Default . . . 2512
4.148 pair_style kolmogorov/crespi/full command . . . 2512
4.148.1 Syntax . . . 2512
4.148.2 Examples . . . 2512
4.148.3 Description . . . 2513
4.148.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2514
4.148.5 Restrictions . . . 2514

4.148.6 Related commands . . . 2514
4.148.7 Default . . . 2514
4.149 pair_style kolmogorov/crespi/z command . . . 2514
4.149.1 Syntax . . . 2514
4.149.2 Examples . . . 2514
4.149.3 Description . . . 2515
4.149.4 Restrictions . . . 2515
4.149.5 Related commands . . . 2515
4.149.6 Default . . . 2515
4.150 pair_style lambda/input/apip command . . . 2516
4.151 pair_style lambda/input/csp/apip command . . . 2516
4.151.1 Syntax . . . 2516
4.152 pair_style lambda/input/csp/apip command . . . 2516
4.152.1 Syntax . . . 2516
4.152.2 Examples . . . 2516
4.152.3 Description . . . 2516
4.152.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2517
4.152.5 Restrictions . . . 2517
4.152.6 Related commands . . . 2517
4.152.7 Default . . . 2517
4.153 pair_style lambda/zone/apip command . . . 2517
4.153.1 Syntax . . . 2517
4.153.2 Examples . . . 2518
4.153.3 Description . . . 2518
4.153.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2518
4.153.5 Restrictions . . . 2519
4.153.6 Related commands . . . 2519
4.153.7 Default . . . 2519
4.154 pair_style lcbop command . . . 2519
4.154.1 Syntax . . . 2519
4.154.2 Examples . . . 2519
4.154.3 Description . . . 2519
4.154.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2520
4.154.5 Restrictions . . . 2520
4.154.6 Related commands . . . 2520
4.154.7 Default . . . 2520
4.155 pair_style ldd command . . . 2520
4.155.1 Syntax . . . 2520
4.155.2 Examples . . . 2521
4.155.3 Description . . . 2521
4.155.4 Theory . . . 2521
4.155.5 Indicator functions . . . 2522
4.155.6 Potential and gradient functions . . . 2522
4.155.7 Accessing the local-density data . . . 2523
4.155.8 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2523
4.155.9 Restrictions . . . 2523
4.155.10 Related commands . . . 2524
4.156 pair_style lebedeva/z command . . . 2524
4.156.1 Syntax . . . 2524
4.156.2 Examples . . . 2524
4.156.3 Description . . . 2524
4.156.4 Restrictions . . . 2525
4.156.5 Related commands . . . 2525
4.156.6 Default . . . 2525

4.157 pair_style lepton command . . . 2526
4.158 pair_style lepton/coul command . . . 2526
4.159 pair_style lepton/sphere command . . . 2526
4.159.1 Syntax . . . 2526
4.159.2 Examples . . . 2526
4.159.3 Description . . . 2527
4.159.4 Lepton expression syntax and features . . . 2528
4.159.5 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2529
4.159.6 Restrictions . . . 2529
4.159.7 Related commands . . . 2529
4.159.8 Default . . . 2529
4.160 pair_style line/lj command . . . 2529
4.160.1 Syntax . . . 2529
4.160.2 Examples . . . 2530
4.160.3 Description . . . 2530
4.160.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2531
4.160.5 Restrictions . . . 2531
4.160.6 Related commands . . . 2531
4.160.7 Default . . . 2531
4.161 pair_style list command . . . 2532
4.161.1 Syntax . . . 2532
4.161.2 Examples . . . 2532
4.161.3 Description . . . 2532
4.161.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2533
4.161.5 Restrictions . . . 2534
4.161.6 Related commands . . . 2534
4.161.7 Default . . . 2534
4.162 pair_style lj/cut command . . . 2534
4.162.1 Syntax . . . 2534
4.162.2 Examples . . . 2534
4.162.3 Description . . . 2534
4.162.4 Coefficients . . . 2535
4.162.5 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2535
4.162.6 Related commands . . . 2536
4.162.7 Default . . . 2536
4.163 pair_style lj96/cut command . . . 2536
4.163.1 Syntax . . . 2536
4.163.2 Examples . . . 2536
4.163.3 Description . . . 2536
4.163.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2537
4.163.5 Restrictions . . . 2537
4.163.6 Related commands . . . 2537
4.163.7 Default . . . 2537
4.164 pair_style lj/cubic command . . . 2537
4.164.1 Syntax . . . 2538
4.164.2 Examples . . . 2538
4.164.3 Description . . . 2538
4.164.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2539
4.164.5 Restrictions . . . 2539
4.164.6 Related commands . . . 2539
4.164.7 Default . . . 2539
4.165 pair_style lj/cut/coul/cut command . . . 2539
4.166 pair_style lj/cut/coul/debye command . . . 2539
4.167 pair_style lj/cut/coul/dsf command . . . 2539

4.168 pair_style lj/cut/coul/long command . . . 2539
4.169 pair_style lj/cut/coul/msm command . . . 2540
4.170 pair_style lj/cut/coul/wolf command . . . 2540
4.170.1 Syntax . . . 2540
4.170.2 Examples . . . 2540
4.170.3 Description . . . 2541
4.170.4 Coefficients . . . 2542
4.170.5 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2543
4.170.6 Restrictions . . . 2543
4.170.7 Related commands . . . 2543
4.170.8 Default . . . 2543
4.171 pair_style lj/cut/sphere command . . . 2543
4.171.1 Syntax . . . 2544
4.171.2 Examples . . . 2544
4.171.3 Description . . . 2544
4.171.4 Coefficients . . . 2545
4.171.5 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2546
4.171.6 Restrictions . . . 2546
4.171.7 Related commands . . . 2546
4.171.8 Default . . . 2546
4.172 pair_style lj/cut/tip4p/cut command . . . 2546
4.173 pair_style lj/cut/tip4p/long command . . . 2546
4.173.1 Syntax . . . 2546
4.173.2 Examples . . . 2547
4.173.3 Description . . . 2547
4.173.4 Coefficients . . . 2548
4.173.5 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2549
4.173.6 Restrictions . . . 2549
4.173.7 Related commands . . . 2549
4.173.8 Default . . . 2550
4.174 pair_style lj/expand command . . . 2550
4.175 pair_style lj/expand/coul/long command . . . 2550
4.175.1 Syntax . . . 2550
4.175.2 Examples . . . 2550
4.175.3 Description . . . 2550
4.175.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2551
4.175.5 Restrictions . . . 2551
4.175.6 Related commands . . . 2551
4.175.7 Default . . . 2551
4.176 pair_style lj/expand/sphere command . . . 2551
4.176.1 Syntax . . . 2552
4.176.2 Examples . . . 2552
4.176.3 Description . . . 2552
4.176.4 Coefficients . . . 2553
4.176.5 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2554
4.176.6 Restrictions . . . 2554
4.176.7 Related commands . . . 2554
4.176.8 Default . . . 2554
4.177 pair_style lj/long/coul/long command . . . 2554
4.178 pair_style lj/long/tip4p/long command . . . 2554
4.178.1 Syntax . . . 2554
4.178.2 Examples . . . 2555
4.178.3 Description . . . 2555
4.178.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2557

4.178.5 Restrictions . . . 2557
4.178.6 Related commands . . . 2557
4.178.7 Default . . . 2558
4.179 pair_style lj/pirani command . . . 2558
4.179.1 Syntax . . . 2558
4.179.2 Examples . . . 2558
4.179.3 Description . . . 2558
4.179.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2559
4.179.5 Restrictions . . . 2560
4.179.6 Related commands . . . 2560
4.179.7 Default . . . 2560
4.180 pair_style lj/relres command . . . 2560
4.180.1 Syntax . . . 2560
4.180.2 Examples . . . 2560
4.180.3 Description . . . 2560
4.180.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2563
4.180.5 Restrictions . . . 2563
4.180.6 Related commands . . . 2564
4.180.7 Default . . . 2564
4.181 pair_style lj/smooth command . . . 2564
4.181.1 Syntax . . . 2564
4.181.2 Examples . . . 2564
4.181.3 Description . . . 2564
4.181.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2565
4.181.5 Restrictions . . . 2565
4.181.6 Related commands . . . 2565
4.181.7 Default . . . 2566
4.182 pair_style lj/smooth/linear command . . . 2566
4.182.1 Syntax . . . 2566
4.182.2 Examples . . . 2566
4.182.3 Description . . . 2566
4.182.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2567
4.182.5 Restrictions . . . 2567
4.182.6 Related commands . . . 2567
4.182.7 Default . . . 2567
4.183 pair_style lj/switch3/coulgauss/long command . . . 2567
4.184 pair_style mm3/switch3/coulgauss/long command . . . 2567
4.184.1 Syntax . . . 2568
4.184.2 Examples . . . 2568
4.184.3 Description . . . 2568
4.184.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2569
4.184.5 Restrictions . . . 2569
4.184.6 Related commands . . . 2570
4.184.7 Default . . . 2570
4.185 pair_style local/density command . . . 2570
4.185.1 Syntax . . . 2570
4.185.2 Examples . . . 2570
4.185.3 Description . . . 2570
4.185.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2573
4.185.5 Restrictions . . . 2573
4.185.6 Related commands . . . 2573
4.185.7 Default . . . 2573
4.186 pair_style lubricate command . . . 2573
4.187 pair_style lubricate/poly command . . . 2573

4.187.1 Syntax . . . 2573
4.187.2 Examples . . . 2574
4.187.3 Description . . . 2574
4.187.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2576
4.187.5 Restrictions . . . 2576
4.187.6 Related commands . . . 2576
4.187.7 Default . . . 2576
4.188 pair_style lubricateU command . . . 2577
4.189 pair_style lubricateU/poly command . . . 2577
4.189.1 Syntax . . . 2577
4.189.2 Examples . . . 2577
4.189.3 Description . . . 2577
4.189.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2579
4.189.5 Restrictions . . . 2579
4.189.6 Related commands . . . 2579
4.189.7 Default . . . 2579
4.190 pair_style mbx command . . . 2580
4.190.1 Syntax . . . 2580
4.190.2 Examples . . . 2580
4.190.3 Description . . . 2580
4.190.4 Pair coeff syntax . . . 2581
4.190.5 Restrictions . . . 2582
4.190.6 Related commands . . . 2582
4.191 pair_style lj/mdf command . . . 2582
4.192 pair_style buck/mdf command . . . 2583
4.193 pair_style lennard/mdf command . . . 2583
4.193.1 Syntax . . . 2583
4.193.2 Examples . . . 2583
4.193.3 Description . . . 2583
4.193.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2585
4.193.5 Restrictions . . . 2585
4.193.6 Related commands . . . 2585
4.193.7 Default . . . 2585
4.194 pair_style meam command . . . 2585
4.195 pair_style meam/ms command . . . 2585
4.195.1 Syntax . . . 2586
4.195.2 Examples . . . 2586
4.195.3 Description . . . 2586
4.195.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2591
4.195.5 Restrictions . . . 2591
4.195.6 Related commands . . . 2591
4.195.7 Default . . . 2592
4.196 pair_style meam/spline command . . . 2592
4.196.1 Syntax . . . 2592
4.196.2 Examples . . . 2592
4.196.3 Description . . . 2592
4.196.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2594
4.196.5 Restrictions . . . 2594
4.196.6 Related commands . . . 2594
4.196.7 Default . . . 2594
4.197 pair_style meam/sw/spline command . . . 2594
4.197.1 Syntax . . . 2594
4.197.2 Examples . . . 2594
4.197.3 Description . . . 2594

4.197.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2596
4.197.5 Restrictions . . . 2596
4.197.6 Related commands . . . 2596
4.197.7 Default . . . 2596
4.198 pair_style mesocnt command . . . 2596
4.199 pair_style mesocnt/viscous command . . . 2596
4.199.1 Syntax . . . 2596
4.199.2 Examples . . . 2597
4.199.3 Description . . . 2597
4.199.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2599
4.199.5 Restrictions . . . 2599
4.199.6 Related commands . . . 2599
4.199.7 Default . . . 2599
4.200 pair_style edpd command . . . 2599
4.201 pair_style mdpd command . . . 2599
4.202 pair_style mdpd/rhosum command . . . 2600
4.203 pair_style tdpd command . . . 2600
4.203.1 Syntax . . . 2600
4.203.2 Examples . . . 2600
4.203.3 Description . . . 2600
4.203.4 Example scripts . . . 2603
4.203.5 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2605
4.203.6 Restrictions . . . 2605
4.203.7 Related commands . . . 2605
4.203.8 Default . . . 2605
4.204 pair_style mgpt command . . . 2606
4.204.1 Syntax . . . 2606
4.204.2 Examples . . . 2606
4.204.3 Description . . . 2606
4.204.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2607
4.204.5 Restrictions . . . 2608
4.204.6 Related commands . . . 2608
4.204.7 Default . . . 2608
4.205 pair_style mie/cut command . . . 2608
4.205.1 Syntax . . . 2608
4.205.2 Examples . . . 2608
4.205.3 Description . . . 2609
4.205.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2609
4.205.5 Restrictions . . . 2610
4.205.6 Related commands . . . 2610
4.205.7 Default . . . 2610
4.206 pair_style mliap command . . . 2610
4.206.1 Syntax . . . 2610
4.206.2 Examples . . . 2610
4.206.3 Description . . . 2611
4.206.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2613
4.206.5 Restrictions . . . 2614
4.206.6 Related commands . . . 2614
4.206.7 Default . . . 2614
4.207 pair_style momb command . . . 2614
4.207.1 Syntax . . . 2614
4.207.2 Examples . . . 2614
4.207.3 Description . . . 2615
4.207.4 Restrictions . . . 2615

4.207.5 Related commands . . . 2615
4.207.6 Default . . . 2615
4.208 pair_style morse command . . . 2616
4.209 pair_style morse/smooth/linear command . . . 2616
4.209.1 Syntax . . . 2616
4.209.2 Examples . . . 2616
4.209.3 Description . . . 2616
4.209.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2617
4.209.5 Restrictions . . . 2617
4.209.6 Related commands . . . 2617
4.209.7 Default . . . 2617
4.210 pair_style multi/lucy command . . . 2618
4.210.1 Syntax . . . 2618
4.210.2 Examples . . . 2618
4.210.3 Description . . . 2618
4.210.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2620
4.210.5 Restrictions . . . 2620
4.210.6 Related commands . . . 2620
4.210.7 Default . . . 2620
4.211 pair_style multi/lucy/rx command . . . 2620
4.211.1 Syntax . . . 2620
4.211.2 Examples . . . 2621
4.211.3 Description . . . 2621
4.211.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2623
4.211.5 Restrictions . . . 2623
4.211.6 Related commands . . . 2623
4.211.7 Default . . . 2623
4.212 pair_style nb3b/harmonic command . . . 2624
4.213 pair_style nb3b/screened command . . . 2624
4.213.1 Syntax . . . 2624
4.213.2 Examples . . . 2624
4.213.3 Description . . . 2624
4.213.4 Restrictions . . . 2625
4.213.5 Related commands . . . 2625
4.213.6 Default . . . 2625
4.214 pair_style nm/cut command . . . 2625
4.215 pair_style nm/cut/split command . . . 2626
4.216 pair_style nm/cut/coul/cut command . . . 2626
4.217 pair_style nm/cut/coul/long command . . . 2626
4.217.1 Syntax . . . 2626
4.217.2 Examples . . . 2626
4.217.3 Description . . . 2627
4.217.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2627
4.217.5 Restrictions . . . 2628
4.217.6 Related commands . . . 2628
4.217.7 Default . . . 2628
4.218 pair_style none command . . . 2628
4.218.1 Syntax . . . 2628
4.218.2 Examples . . . 2628
4.218.3 Description . . . 2629
4.218.4 Restrictions . . . 2629
4.218.5 Related commands . . . 2629
4.218.6 Default . . . 2629
4.219 pair_style oxdna/excv command . . . 2629

4.220 pair_style oxdna/stk command . . . 2629
4.221 pair_style oxdna/hbond command . . . 2629
4.222 pair_style oxdna/xstk command . . . 2629
4.223 pair_style oxdna/coaxstk command . . . 2629
4.223.1 Syntax . . . 2629
4.223.2 Examples . . . 2630
4.223.3 Description . . . 2631
4.223.4 Potential file reading . . . 2632
4.223.5 Unique base pairing . . . 2633
4.223.6 Restrictions . . . 2634
4.223.7 Related commands . . . 2634
4.223.8 Default . . . 2634
4.224 pair_style oxdna2/excv command . . . 2634
4.225 pair_style oxdna2/stk command . . . 2634
4.226 pair_style oxdna2/hbond command . . . 2634
4.227 pair_style oxdna2/xstk command . . . 2634
4.228 pair_style oxdna2/coaxstk command . . . 2634
4.229 pair_style oxdna2/dh command . . . 2634
4.229.1 Syntax . . . 2634
4.229.2 Examples . . . 2635
4.229.3 Description . . . 2636
4.229.4 Potential file reading . . . 2637
4.229.5 Unique base pairing . . . 2638
4.229.6 Restrictions . . . 2639
4.229.7 Related commands . . . 2639
4.229.8 Default . . . 2639
4.230 pair_style oxdna3/excv command . . . 2640
4.231 pair_style oxdna3/stk command . . . 2640
4.232 pair_style oxdna3/hbond command . . . 2640
4.233 pair_style oxdna3/xstk command . . . 2640
4.234 pair_style oxdna3/coaxstk command . . . 2640
4.235 pair_style oxdna3/dh command . . . 2640
4.235.1 Syntax . . . 2640
4.235.2 Examples . . . 2640
4.235.3 Description . . . 2641
4.235.4 Unique base pairing . . . 2642
4.235.5 Restrictions . . . 2643
4.235.6 Related commands . . . 2643
4.235.7 Default . . . 2643
4.236 pair_style oxrna2/excv command . . . 2644
4.237 pair_style oxrna2/stk command . . . 2644
4.238 pair_style oxrna2/hbond command . . . 2644
4.239 pair_style oxrna2/xstk command . . . 2644
4.240 pair_style oxrna2/coaxstk command . . . 2644
4.241 pair_style oxrna2/dh command . . . 2644
4.241.1 Syntax . . . 2644
4.241.2 Examples . . . 2644
4.241.3 Description . . . 2646
4.241.4 Potential file reading . . . 2647
4.241.5 Unique base pairing . . . 2648
4.241.6 Restrictions . . . 2649
4.241.7 Related commands . . . 2649
4.241.8 Default . . . 2649
4.242 pair_style pace command . . . 2649

4.243 pair_style pace/extrapolation command . . . 2649
4.243.1 Syntax . . . 2649
4.243.2 Examples . . . 2649
4.243.3 Description . . . 2650
4.243.4 Extrapolation grade . . . 2650
4.243.5 Core repulsion . . . 2651
4.243.6 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2651
4.243.7 Restrictions . . . 2652
4.243.8 Related commands . . . 2652
4.243.9 Default . . . 2652
4.244 pair_style pace/apip command . . . 2652
4.245 pair_style pace/fast/apip command . . . 2652
4.246 pair_style pace/precise/apip command . . . 2652
4.246.1 Syntax . . . 2652
4.246.2 Examples . . . 2652
4.246.3 Description . . . 2653
4.246.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2653
4.246.5 Restrictions . . . 2653
4.246.6 Related commands . . . 2653
4.246.7 Default . . . 2654
4.247 pair_style pedone command . . . 2654
4.247.1 Syntax . . . 2654
4.247.2 Examples . . . 2654
4.247.3 Description . . . 2654
4.247.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2655
4.247.5 Restrictions . . . 2655
4.247.6 Related commands . . . 2655
4.247.7 Default . . . 2655
4.248 pair_style peri/pmb command . . . 2656
4.249 pair_style peri/lps command . . . 2656
4.250 pair_style peri/ves command . . . 2656
4.251 pair_style peri/eps command . . . 2656
4.251.1 Syntax . . . 2656
4.251.2 Examples . . . 2656
4.251.3 Description . . . 2656
4.251.4 Bond breaking criterion . . . 2658
4.251.5 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2659
4.251.6 Restrictions . . . 2659
4.251.7 Related commands . . . 2659
4.251.8 Default . . . 2659
4.252 pair_style pod command . . . 2659
4.252.1 Syntax . . . 2659
4.252.2 Examples . . . 2659
4.252.3 Description . . . 2660
4.252.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2660
4.252.5 Restrictions . . . 2661
4.252.6 Related commands . . . 2661
4.252.7 Default . . . 2661
4.253 pair_style polymorphic command . . . 2661
4.253.1 Syntax . . . 2661
4.253.2 Examples . . . 2661
4.253.3 Description . . . 2662
4.253.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2665
4.253.5 Restrictions . . . 2666

4.253.6 Related commands . . . 2666
4.254 pair_style python command . . . 2666
4.254.1 Syntax . . . 2666
4.254.2 Examples . . . 2666
4.254.3 Description . . . 2666
4.254.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2670
4.254.5 Restrictions . . . 2670
4.254.6 Related commands . . . 2670
4.254.7 Default . . . 2670
4.255 pair_style quip command . . . 2670
4.255.1 Syntax . . . 2670
4.255.2 Examples . . . 2670
4.255.3 Description . . . 2671
4.255.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2671
4.255.5 Restrictions . . . 2671
4.255.6 Related commands . . . 2671
4.256 pair_style rann command . . . 2672
4.256.1 Syntax . . . 2672
4.256.2 Examples . . . 2672
4.256.3 Description . . . 2672
4.256.4 Potential file syntax . . . 2672
4.256.5 Formulation . . . 2675
4.256.6 Restrictions . . . 2677
4.256.7 Defaults . . . 2677
4.257 pair_style reaxff command . . . 2677
4.257.1 Syntax . . . 2677
4.257.2 Examples . . . 2678
4.257.3 Description . . . 2678
4.257.4 Control file . . . 2681
4.257.5 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2682
4.257.6 Restrictions . . . 2683
4.257.7 Related commands . . . 2683
4.257.8 Default . . . 2683
4.258 pair_style rebomos command . . . 2683
4.258.1 Syntax . . . 2683
4.258.2 Examples . . . 2683
4.258.3 Description . . . 2683
4.258.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2684
4.258.5 Restrictions . . . 2684
4.258.6 Related commands . . . 2685
4.258.7 Default . . . 2685
4.259 pair_style resquared command . . . 2685
4.259.1 Syntax . . . 2685
4.259.2 Examples . . . 2685
4.259.3 Description . . . 2685
4.259.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2687
4.259.5 Restrictions . . . 2687
4.259.6 Related commands . . . 2688
4.259.7 Default . . . 2688
4.260 pair_style rheo command . . . 2688
4.260.1 Syntax . . . 2688
4.260.2 Examples . . . 2688
4.260.3 Description . . . 2688
4.260.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2689

4.260.5 Restrictions . . . 2689
4.260.6 Related commands . . . 2689
4.260.7 Default . . . 2689
4.261 pair_style rheo/solid command . . . 2689
4.261.1 Syntax . . . 2689
4.261.2 Examples . . . 2689
4.261.3 Description . . . 2689
4.261.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2690
4.261.5 Restrictions . . . 2690
4.261.6 Related commands . . . 2690
4.261.7 Default . . . 2690
4.262 pair_style runner command . . . 2690
4.262.1 Syntax . . . 2690
4.262.2 Examples . . . 2691
4.262.3 Description . . . 2691
4.262.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2694
4.262.5 Restrictions . . . 2694
4.262.6 Related commands . . . 2694
4.262.7 Default . . . 2694
4.262.8 References . . . 2695
4.263 pair_style saip/metal command . . . 2695
4.264 pair_style saip/metal/tmd command . . . 2695
4.264.1 Syntax . . . 2695
4.264.2 Examples . . . 2696
4.264.3 Description . . . 2696
4.264.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2697
4.264.5 Restrictions . . . 2697
4.264.6 Related commands . . . 2698
4.264.7 Default . . . 2698
4.265 pair_style sdpd/taitwater/isothermal command . . . 2698
4.265.1 Syntax . . . 2698
4.265.2 Examples . . . 2698
4.265.3 Description . . . 2698
4.265.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2699
4.265.5 Restrictions . . . 2699
4.265.6 Related commands . . . 2699
4.265.7 Default . . . 2699
4.266 pair_style smatb command . . . 2700
4.267 pair_style smatb/single command . . . 2700
4.267.1 Syntax . . . 2700
4.267.2 Examples . . . 2700
4.267.3 Description . . . 2700
4.267.4 Coefficients . . . 2700
4.267.5 Mixing info . . . 2701
4.267.6 Restrictions . . . 2701
4.267.7 Related commands . . . 2701
4.267.8 Default . . . 2701
4.268 pair_style smd/hertz command . . . 2701
4.268.1 Syntax . . . 2701
4.268.2 Examples . . . 2701
4.268.3 Description . . . 2702
4.268.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2702
4.268.5 Restrictions . . . 2702
4.268.6 Related commands . . . 2702

4.268.7 Default . . . 2702
4.269 pair_style smd/tlsph command . . . 2702
4.269.1 Syntax . . . 2702
4.269.2 Examples . . . 2702
4.269.3 Description . . . 2702
4.269.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2703
4.269.5 Restrictions . . . 2703
4.269.6 Related commands . . . 2703
4.269.7 Default . . . 2703
4.270 pair_style smd/tri_surface command . . . 2703
4.270.1 Syntax . . . 2703
4.270.2 Examples . . . 2703
4.270.3 Description . . . 2703
4.270.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2704
4.270.5 Restrictions . . . 2704
4.270.6 Related commands . . . 2704
4.270.7 Default . . . 2704
4.271 pair_style smd/ulsph command . . . 2704
4.271.1 Syntax . . . 2704
4.271.2 Examples . . . 2704
4.271.3 Description . . . 2704
4.271.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2705
4.271.5 Restrictions . . . 2705
4.271.6 Related commands . . . 2705
4.271.7 Default . . . 2705
4.272 pair_style smtbq command . . . 2705
4.272.1 Syntax . . . 2705
4.272.2 Examples . . . 2705
4.272.3 Description . . . 2705
4.272.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2709
4.272.5 Restrictions . . . 2709
4.272.6 Citing this work . . . 2709
4.273 pair_style snap command . . . 2710
4.273.1 Syntax . . . 2710
4.273.2 Examples . . . 2710
4.273.3 Description . . . 2710
4.273.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2712
4.273.5 Restrictions . . . 2713
4.273.6 Related commands . . . 2713
4.273.7 Default . . . 2713
4.274 pair_style soft command . . . 2713
4.274.1 Syntax . . . 2714
4.274.2 Examples . . . 2714
4.274.3 Description . . . 2714
4.274.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2715
4.274.5 Restrictions . . . 2715
4.274.6 Related commands . . . 2715
4.274.7 Default . . . 2715
4.275 pair_style sph/heatconduction command . . . 2715
4.275.1 Syntax . . . 2715
4.275.2 Examples . . . 2716
4.275.3 Description . . . 2716
4.275.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2716
4.275.5 Restrictions . . . 2716

4.275.6 Related commands . . . 2717
4.275.7 Default . . . 2717
4.276 pair_style sph/idealgas command . . . 2717
4.276.1 Syntax . . . 2717
4.276.2 Examples . . . 2717
4.276.3 Description . . . 2717
4.276.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2717
4.276.5 Restrictions . . . 2718
4.276.6 Related commands . . . 2718
4.276.7 Default . . . 2718
4.277 pair_style sph/lj command . . . 2718
4.277.1 Syntax . . . 2718
4.277.2 Examples . . . 2718
4.277.3 Description . . . 2718
4.277.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2719
4.277.5 Restrictions . . . 2719
4.277.6 Related commands . . . 2719
4.277.7 Default . . . 2719
4.278 pair_style sph/rhosum command . . . 2719
4.278.1 Syntax . . . 2719
4.278.2 Examples . . . 2719
4.278.3 Description . . . 2720
4.278.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2720
4.278.5 Restrictions . . . 2720
4.278.6 Related commands . . . 2720
4.278.7 Default . . . 2720
4.279 pair_style sph/taitwater command . . . 2720
4.279.1 Syntax . . . 2720
4.279.2 Examples . . . 2721
4.279.3 Description . . . 2721
4.279.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2721
4.279.5 Restrictions . . . 2722
4.279.6 Related commands . . . 2722
4.279.7 Default . . . 2722
4.280 pair_style sph/taitwater/morris command . . . 2722
4.280.1 Syntax . . . 2722
4.280.2 Examples . . . 2722
4.280.3 Description . . . 2722
4.280.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2723
4.280.5 Restrictions . . . 2723
4.280.6 Related commands . . . 2723
4.280.7 Default . . . 2723
4.281 pair_style lj/spica command . . . 2723
4.282 pair_style lj/spica/coul/long command . . . 2723
4.283 pair_style lj/spica/coul/msm command . . . 2723
4.283.1 Syntax . . . 2723
4.283.2 Examples . . . 2724
4.283.3 Description . . . 2724
4.283.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2725
4.283.5 Restrictions . . . 2725
4.283.6 Related commands . . . 2725
4.283.7 Default . . . 2725
4.284 pair_style spin/dipole/cut command . . . 2726
4.285 pair_style spin/dipole/long command . . . 2726

4.285.1 Syntax . . . 2726
4.285.2 Examples . . . 2726
4.285.3 Description . . . 2726
4.285.4 Restrictions . . . 2727
4.285.5 Related commands . . . 2727
4.285.6 Default . . . 2727
4.286 pair_style spin/dmi command . . . 2727
4.286.1 Syntax . . . 2727
4.286.2 Examples . . . 2727
4.286.3 Description . . . 2727
4.286.4 Restrictions . . . 2728
4.286.5 Related commands . . . 2728
4.286.6 Default . . . 2728
4.287 pair_style spin/exchange command . . . 2728
4.288 pair_style spin/exchange/biquadratic command . . . 2728
4.288.1 Syntax . . . 2728
4.288.2 Examples . . . 2728
4.288.3 Description . . . 2729
4.288.4 Restrictions . . . 2730
4.288.5 Related commands . . . 2731
4.288.6 Default . . . 2731
4.289 pair_style spin/magelec command . . . 2731
4.289.1 Syntax . . . 2731
4.289.2 Examples . . . 2731
4.289.3 Description . . . 2731
4.289.4 Restrictions . . . 2732
4.289.5 Related commands . . . 2732
4.289.6 Default . . . 2732
4.290 pair_style spin/neel command . . . 2732
4.290.1 Syntax . . . 2732
4.290.2 Examples . . . 2732
4.290.3 Description . . . 2732
4.290.4 Restrictions . . . 2733
4.290.5 Related commands . . . 2733
4.290.6 Default . . . 2733
4.291 pair_style srp command . . . 2733
4.292 pair_style srp/react command . . . 2733
4.292.1 Syntax . . . 2733
4.292.2 Examples . . . 2734
4.292.3 Description . . . 2734
4.292.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2735
4.292.5 Restrictions . . . 2736
4.292.6 Related commands . . . 2736
4.292.7 Default . . . 2736
4.293 pair_style surf/granular command . . . 2736
4.293.1 Syntax . . . 2736
4.293.2 Examples . . . 2736
4.293.3 Description . . . 2736
4.293.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2737
4.293.5 Restrictions . . . 2737
4.293.6 Related commands . . . 2738
4.293.7 Default . . . 2738
4.294 pair_style sw command . . . 2738
4.295 pair_style sw/mod command . . . 2738

4.295.1 Syntax . . . 2738
4.295.2 Examples . . . 2738
4.295.3 Description . . . 2739
4.295.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2742
4.295.5 Restrictions . . . 2742
4.295.6 Related commands . . . 2742
4.295.7 Default . . . 2742
4.296 pair_style sw/angle/table command . . . 2742
4.296.1 Syntax . . . 2742
4.296.2 Examples . . . 2743
4.296.3 Description . . . 2743
4.296.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2746
4.296.5 Restrictions . . . 2746
4.296.6 Related commands . . . 2746
4.297 pair_style table command . . . 2746
4.297.1 Syntax . . . 2746
4.297.2 Examples . . . 2746
4.297.3 Description . . . 2747
4.297.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2749
4.297.5 Restrictions . . . 2750
4.297.6 Related commands . . . 2750
4.297.7 Default . . . 2750
4.298 pair_style table/rx command . . . 2750
4.298.1 Syntax . . . 2750
4.298.2 Examples . . . 2750
4.298.3 Description . . . 2750
4.298.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2752
4.298.5 Restrictions . . . 2753
4.298.6 Related commands . . . 2753
4.298.7 Default . . . 2753
4.299 pair_style tersoff command . . . 2753
4.300 pair_style tersoff/table command . . . 2753
4.300.1 Syntax . . . 2753
4.300.2 Examples . . . 2754
4.300.3 Description . . . 2754
4.300.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2757
4.300.5 Restrictions . . . 2757
4.300.6 Related commands . . . 2757
4.300.7 Default . . . 2758
4.301 pair_style tersoff/mod command . . . 2758
4.302 pair_style tersoff/mod/c command . . . 2758
4.302.1 Syntax . . . 2758
4.302.2 Examples . . . 2758
4.302.3 Description . . . 2758
4.302.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2761
4.302.5 Restrictions . . . 2761
4.302.6 Related commands . . . 2761
4.302.7 Default . . . 2761
4.303 pair_style tersoff/zbl command . . . 2761
4.303.1 Syntax . . . 2762
4.303.2 Examples . . . 2762
4.303.3 Description . . . 2762
4.303.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2765
4.303.5 Restrictions . . . 2765

4.303.6 Related commands . . . 2766
4.303.7 Default . . . 2766
4.304 pair_style thole command . . . 2766
4.305 pair_style lj/cut/thole/long command . . . 2766
4.305.1 Syntax . . . 2766
4.305.2 Examples . . . 2766
4.305.3 Description . . . 2766
4.305.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2768
4.305.5 Restrictions . . . 2768
4.305.6 Related commands . . . 2768
4.305.7 Default . . . 2768
4.306 pair_style threebody/table command . . . 2768
4.306.1 Syntax . . . 2768
4.306.2 Examples . . . 2769
4.306.3 Description . . . 2769
4.306.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2771
4.306.5 Restrictions . . . 2772
4.306.6 Related commands . . . 2772
4.307 pair_style tracker command . . . 2772
4.307.1 Syntax . . . 2772
4.307.2 Examples . . . 2773
4.307.3 Description . . . 2773
4.307.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2774
4.307.5 Restrictions . . . 2774
4.307.6 Related commands . . . 2774
4.307.7 Default . . . 2774
4.308 pair_style tri/lj command . . . 2774
4.308.1 Syntax . . . 2774
4.308.2 Examples . . . 2774
4.308.3 Description . . . 2775
4.308.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2775
4.308.5 Restrictions . . . 2776
4.308.6 Related commands . . . 2776
4.308.7 Default . . . 2776
4.309 pair_style uf3 command . . . 2776
4.309.1 Syntax . . . 2776
4.309.2 Examples . . . 2776
4.309.3 Description . . . 2776
4.309.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2779
4.309.5 Restrictions . . . 2779
4.309.6 Related commands . . . 2779
4.309.7 Default . . . 2779
4.310 pair_style ufm command . . . 2779
4.310.1 Syntax . . . 2779
4.310.2 Examples . . . 2779
4.310.3 Description . . . 2780
4.310.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2781
4.310.5 Restrictions . . . 2781
4.310.6 Related commands . . . 2781
4.310.7 Default . . . 2781
4.311 pair_style vashishta command . . . 2781
4.312 pair_style vashishta/table command . . . 2781
4.312.1 Syntax . . . 2781
4.312.2 Examples . . . 2782

4.312.3 Description . . . 2782
4.312.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2784
4.312.5 Restrictions . . . 2784
4.312.6 Related commands . . . 2784
4.312.7 Default . . . 2784
4.313 pair_style wf/cut command . . . 2785
4.313.1 Syntax . . . 2785
4.313.2 Examples . . . 2785
4.313.3 Description . . . 2785
4.313.4 Restrictions . . . 2786
4.313.5 Related commands . . . 2786
4.314 pair_style ylz command . . . 2787
4.314.1 Syntax . . . 2787
4.314.2 Examples . . . 2787
4.314.3 Description . . . 2787
4.314.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2788
4.314.5 Restrictions . . . 2788
4.314.6 Related commands . . . 2789
4.314.7 Default . . . 2789
4.315 pair_style yukawa command . . . 2789
4.315.1 Syntax . . . 2789
4.315.2 Examples . . . 2789
4.315.3 Description . . . 2789
4.315.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2790
4.315.5 Restrictions . . . 2790
4.315.6 Related commands . . . 2790
4.315.7 Default . . . 2790
4.316 pair_style yukawa/colloid command . . . 2790
4.316.1 Syntax . . . 2790
4.316.2 Examples . . . 2791
4.316.3 Description . . . 2791
4.316.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2792
4.316.5 Restrictions . . . 2792
4.316.6 Related commands . . . 2792
4.316.7 Default . . . 2792
4.317 pair_style zbl command . . . 2793
4.317.1 Syntax . . . 2793
4.317.2 Examples . . . 2793
4.317.3 Description . . . 2793
4.317.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2794
4.317.5 Restrictions . . . 2794
4.317.6 Related commands . . . 2794
4.317.7 Default . . . 2794
4.318 pair_style zero command . . . 2795
4.318.1 Syntax . . . 2795
4.318.2 Examples . . . 2795
4.318.3 Description . . . 2795
4.318.4 Mixing, shift, table, tail correction, restart, rRESPA info . . . 2795
4.318.5 Restrictions . . . 2796
4.318.6 Related commands . . . 2796
4.318.7 Default . . . 2796

**5 Bond Styles** **2797**
5.1 bond_style bpm/rotational command . . . 2797

5.1.1 Syntax . . . 2797
5.1.2 Examples . . . 2798
5.1.3 Description . . . 2798
5.1.4 Restart and other info . . . 2800
5.1.5 Restrictions . . . 2800
5.1.6 Related commands . . . 2801
5.1.7 Default . . . 2801
5.2 bond_style bpm/spring command . . . 2801
5.2.1 Syntax . . . 2801
5.2.2 Examples . . . 2802
5.2.3 Description . . . 2802
5.2.4 Restart and other info . . . 2804
5.2.5 Restrictions . . . 2804
5.2.6 Related commands . . . 2804
5.2.7 Default . . . 2804
5.3 bond_style bpm/spring/plastic command . . . 2805
5.3.1 Syntax . . . 2805
5.3.2 Examples . . . 2805
5.3.3 Description . . . 2805
5.3.4 Restart and other info . . . 2806
5.3.5 Restrictions . . . 2806
5.3.6 Related commands . . . 2807
5.3.7 Default . . . 2807
5.4 bond_style class2 command . . . 2807
5.4.1 Syntax . . . 2807
5.4.2 Examples . . . 2807
5.4.3 Description . . . 2807
5.4.4 Restrictions . . . 2808
5.4.5 Related commands . . . 2808
5.4.6 Default . . . 2808
5.5 bond_style fene command . . . 2808
5.6 bond_style fene/nm command . . . 2808
5.6.1 Syntax . . . 2808
5.6.2 Examples . . . 2808
5.6.3 Description . . . 2809
5.6.4 Restrictions . . . 2810
5.6.5 Related commands . . . 2810
5.6.6 Default . . . 2810
5.7 bond_style fene/expand command . . . 2810
5.7.1 Syntax . . . 2810
5.7.2 Examples . . . 2810
5.7.3 Description . . . 2810
5.7.4 Restrictions . . . 2811
5.7.5 Related commands . . . 2811
5.7.6 Default . . . 2811
5.8 bond_style gaussian command . . . 2811
5.8.1 Syntax . . . 2811
5.8.2 Examples . . . 2811
5.8.3 Description . . . 2812
5.8.4 Restrictions . . . 2812
5.8.5 Related commands . . . 2812
5.8.6 Default . . . 2812
5.9 bond_style gromos command . . . 2813
5.9.1 Syntax . . . 2813

5.9.2 Examples . . . 2813
5.9.3 Description . . . 2813
5.9.4 Restrictions . . . 2813
5.9.5 Related commands . . . 2813
5.9.6 Default . . . 2813
5.10 bond_style harmonic command . . . 2814
5.10.1 Syntax . . . 2814
5.10.2 Examples . . . 2814
5.10.3 Description . . . 2814
5.10.4 Restrictions . . . 2814
5.10.5 Related commands . . . 2814
5.10.6 Default . . . 2814
5.11 bond_style harmonic/restrain command . . . 2815
5.11.1 Syntax . . . 2815
5.11.2 Examples . . . 2815
5.11.3 Description . . . 2815
5.11.4 Restart info . . . 2815
5.11.5 Restrictions . . . 2816
5.11.6 Related commands . . . 2816
5.11.7 Default . . . 2816
5.12 bond_style harmonic/shift command . . . 2816
5.12.1 Syntax . . . 2816
5.12.2 Examples . . . 2816
5.12.3 Description . . . 2816
5.12.4 Restrictions . . . 2817
5.12.5 Related commands . . . 2817
5.12.6 Default . . . 2817
5.13 bond_style harmonic/shift/cut command . . . 2817
5.13.1 Syntax . . . 2817
5.13.2 Examples . . . 2817
5.13.3 Description . . . 2817
5.13.4 Restrictions . . . 2818
5.13.5 Related commands . . . 2818
5.13.6 Default . . . 2818
5.14 bond_style hybrid command . . . 2818
5.14.1 Syntax . . . 2818
5.14.2 Examples . . . 2818
5.14.3 Description . . . 2818
5.14.4 Restrictions . . . 2819
5.14.5 Related commands . . . 2819
5.14.6 Default . . . 2819
5.15 bond_style lepton command . . . 2819
5.15.1 Syntax . . . 2820
5.15.2 Examples . . . 2820
5.15.3 Description . . . 2820
5.15.4 Lepton expression syntax and features . . . 2821
5.15.5 Restrictions . . . 2822
5.15.6 Related commands . . . 2822
5.15.7 Default . . . 2822
5.16 bond_style mesocnt command . . . 2822
5.16.1 Syntax . . . 2822
5.16.2 Examples . . . 2822
5.16.3 Description . . . 2822
5.16.4 Restrictions . . . 2823

5.16.5 Related commands . . . 2823
5.16.6 Default . . . 2823
5.17 bond_style mm3 command . . . 2823
5.17.1 Syntax . . . 2823
5.17.2 Examples . . . 2823
5.17.3 Description . . . 2823
5.17.4 Restrictions . . . 2824
5.17.5 Related commands . . . 2824
5.17.6 Default . . . 2824
5.18 bond_style morse command . . . 2824
5.18.1 Syntax . . . 2824
5.18.2 Examples . . . 2824
5.18.3 Description . . . 2825
5.18.4 Restrictions . . . 2825
5.18.5 Related commands . . . 2825
5.18.6 Default . . . 2825
5.19 bond_style none command . . . 2825
5.19.1 Syntax . . . 2825
5.19.2 Examples . . . 2825
5.19.3 Description . . . 2826
5.19.4 Restrictions . . . 2826
5.19.5 Related commands . . . 2826
5.19.6 Default . . . 2826
5.20 bond_style nonlinear command . . . 2826
5.20.1 Syntax . . . 2826
5.20.2 Examples . . . 2826
5.20.3 Description . . . 2826
5.20.4 Restrictions . . . 2827
5.20.5 Related commands . . . 2827
5.20.6 Default . . . 2827
5.21 bond_style oxdna/fene command . . . 2827
5.22 bond_style oxdna2/fene command . . . 2827
5.23 bond_style oxdna3/fene command . . . 2827
5.24 bond_style oxrna2/fene command . . . 2827
5.24.1 Syntax . . . 2827
5.24.2 Examples . . . 2827
5.24.3 Description . . . 2828
5.24.4 Potential file reading . . . 2829
5.24.5 Restrictions . . . 2830
5.24.6 Related commands . . . 2830
5.24.7 Default . . . 2830
5.25 bond_style quartic command . . . 2830
5.25.1 Syntax . . . 2830
5.25.2 Examples . . . 2830
5.25.3 Description . . . 2830
5.25.4 Restrictions . . . 2832
5.25.5 Related commands . . . 2832
5.25.6 Default . . . 2832
5.26 bond_style quartic/exp command . . . 2832
5.26.1 Syntax . . . 2832
5.26.2 Examples . . . 2832
5.26.3 Description . . . 2832
5.26.4 Restrictions . . . 2833
5.26.5 Related commands . . . 2833

5.26.6 Default . . . 2833
5.27 bond_style rheo/shell command . . . 2833
5.27.1 Syntax . . . 2833
5.27.2 Examples . . . 2834
5.27.3 Description . . . 2834
5.27.4 Restart and other info . . . 2835
5.27.5 Restrictions . . . 2835
5.27.6 Related commands . . . 2835
5.27.7 Default . . . 2835
5.28 bond_style special command . . . 2835
5.28.1 Syntax . . . 2835
5.28.2 Examples . . . 2836
5.28.3 Description . . . 2836
5.28.4 Restrictions . . . 2836
5.28.5 Related commands . . . 2837
5.28.6 Default . . . 2837
5.29 bond_style table command . . . 2837
5.29.1 Syntax . . . 2837
5.29.2 Examples . . . 2837
5.29.3 Description . . . 2837
5.29.4 Restart info . . . 2839
5.29.5 Restrictions . . . 2839
5.29.6 Related commands . . . 2839
5.29.7 Default . . . 2839
5.30 bond_style zero command . . . 2839
5.30.1 Syntax . . . 2839
5.30.2 Examples . . . 2839
5.30.3 Description . . . 2839
5.30.4 Restrictions . . . 2840
5.30.5 Related commands . . . 2840
5.30.6 Default . . . 2840

**6 Angle Styles 2841**
6.1 angle_style amoeba command . . . 2841
6.1.1 Syntax . . . 2841
6.1.2 Examples . . . 2841
6.1.3 Description . . . 2841
6.1.4 Restrictions . . . 2842
6.1.5 Related commands . . . 2842
6.1.6 Default . . . 2843
6.2 angle_style charmm command . . . 2843
6.2.1 Syntax . . . 2843
6.2.2 Examples . . . 2843
6.2.3 Description . . . 2843
6.2.4 Restrictions . . . 2844
6.2.5 Related commands . . . 2844
6.2.6 Default . . . 2844
6.3 angle_style class2 command . . . 2844
6.4 angle_style class2xe command . . . 2844
6.5 angle_style class2/p6 command . . . 2844
6.5.1 Syntax . . . 2844
6.5.2 Examples . . . 2844
6.5.3 Description . . . 2844
6.5.4 Restrictions . . . 2847

6.5.5 Related commands . . . 2847
6.5.6 Default . . . 2847
6.6 angle_style cosine command . . . 2848
6.6.1 Syntax . . . 2848
6.6.2 Examples . . . 2848
6.6.3 Description . . . 2848
6.6.4 Restrictions . . . 2848
6.6.5 Related commands . . . 2848
6.6.6 Default . . . 2848
6.7 angle_style cosine/buck6d command . . . 2849
6.7.1 Syntax . . . 2849
6.7.2 Examples . . . 2849
6.7.3 Description . . . 2849
6.7.4 Restrictions . . . 2849
6.7.5 Related commands . . . 2850
6.7.6 Default . . . 2850
6.8 angle_style cosine/delta command . . . 2850
6.8.1 Syntax . . . 2850
6.8.2 Examples . . . 2850
6.8.3 Description . . . 2850
6.8.4 Restrictions . . . 2851
6.8.5 Related commands . . . 2851
6.8.6 Default . . . 2851
6.9 angle_style cosine/periodic command . . . 2851
6.9.1 Syntax . . . 2851
6.9.2 Examples . . . 2851
6.9.3 Description . . . 2851
6.9.4 Restrictions . . . 2852
6.9.5 Related commands . . . 2852
6.9.6 Default . . . 2852
6.10 angle_style cosine/shift command . . . 2852
6.10.1 Syntax . . . 2852
6.10.2 Examples . . . 2852
6.10.3 Description . . . 2852
6.10.4 Restrictions . . . 2853
6.10.5 Related commands . . . 2853
6.10.6 Default . . . 2853
6.11 angle_style cosine/shift/exp command . . . 2853
6.11.1 Syntax . . . 2853
6.11.2 Examples . . . 2853
6.11.3 Description . . . 2853
6.11.4 Restrictions . . . 2854
6.11.5 Related commands . . . 2854
6.11.6 Default . . . 2854
6.12 angle_style cosine/squared command . . . 2854
6.12.1 Syntax . . . 2854
6.12.2 Examples . . . 2854
6.12.3 Description . . . 2854
6.12.4 Restrictions . . . 2855
6.12.5 Related commands . . . 2855
6.12.6 Default . . . 2855
6.13 angle_style cosine/squared/restricted command . . . 2855
6.13.1 Syntax . . . 2855
6.13.2 Examples . . . 2855

6.13.3 Description . . . 2856
6.13.4 Restrictions . . . 2856
6.13.5 Related commands . . . 2856
6.13.6 Default . . . 2856
6.14 angle_style cross command . . . 2856
6.14.1 Syntax . . . 2857
6.14.2 Examples . . . 2857
6.14.3 Description . . . 2857
6.14.4 Restrictions . . . 2857
6.14.5 Related commands . . . 2858
6.14.6 Default . . . 2858
6.15 angle_style dipole command . . . 2858
6.15.1 Syntax . . . 2858
6.15.2 Examples . . . 2858
6.15.3 Description . . . 2858
6.15.4 Restrictions . . . 2859
6.15.5 Related commands . . . 2859
6.15.6 Default . . . 2859
6.16 angle_style fourier command . . . 2860
6.16.1 Syntax . . . 2860
6.16.2 Examples . . . 2860
6.16.3 Description . . . 2860
6.16.4 Restrictions . . . 2860
6.16.5 Related commands . . . 2860
6.16.6 Default . . . 2861
6.17 angle_style fourier/simple command . . . 2861
6.17.1 Syntax . . . 2861
6.17.2 Examples . . . 2861
6.17.3 Description . . . 2861
6.17.4 Restrictions . . . 2861
6.17.5 Related commands . . . 2862
6.17.6 Default . . . 2862
6.18 angle_style gaussian command . . . 2862
6.18.1 Syntax . . . 2862
6.18.2 Examples . . . 2862
6.18.3 Description . . . 2862
6.18.4 Restrictions . . . 2863
6.18.5 Related commands . . . 2863
6.18.6 Default . . . 2863
6.19 angle_style harmonic command . . . 2863
6.19.1 Syntax . . . 2863
6.19.2 Examples . . . 2863
6.19.3 Description . . . 2863
6.19.4 Restrictions . . . 2864
6.19.5 Related commands . . . 2864
6.19.6 Default . . . 2864
6.20 angle_style hybrid command . . . 2864
6.20.1 Syntax . . . 2864
6.20.2 Examples . . . 2864
6.20.3 Description . . . 2864
6.20.4 Restrictions . . . 2865
6.20.5 Related commands . . . 2866
6.20.6 Default . . . 2866
6.21 angle_style lepton command . . . 2866

6.21.1 Syntax . . . 2866
6.21.2 Examples . . . 2866
6.21.3 Description . . . 2866
6.21.4 Lepton expression syntax and features . . . 2867
6.21.5 Restrictions . . . 2868
6.21.6 Related commands . . . 2868
6.21.7 Default . . . 2868
6.22 angle_style mesocnt command . . . 2868
6.22.1 Syntax . . . 2868
6.22.2 Examples . . . 2868
6.22.3 Description . . . 2869
6.22.4 Restrictions . . . 2870
6.22.5 Related commands . . . 2870
6.22.6 Default . . . 2870
6.23 angle_style mm3 command . . . 2870
6.23.1 Syntax . . . 2870
6.23.2 Examples . . . 2870
6.23.3 Description . . . 2870
6.23.4 Restrictions . . . 2871
6.23.5 Related commands . . . 2871
6.23.6 Default . . . 2871
6.24 angle_style mwlc command . . . 2871
6.24.1 Syntax . . . 2871
6.24.2 Examples . . . 2871
6.24.3 Description . . . 2872
6.24.4 Restrictions . . . 2872
6.24.5 Related commands . . . 2872
6.24.6 Default . . . 2873
6.25 angle_style none command . . . 2873
6.25.1 Syntax . . . 2873
6.25.2 Examples . . . 2873
6.25.3 Description . . . 2873
6.25.4 Restrictions . . . 2873
6.25.5 Related commands . . . 2873
6.25.6 Default . . . 2873
6.26 angle_style quartic command . . . 2873
6.26.1 Syntax . . . 2873
6.26.2 Examples . . . 2873
6.26.3 Description . . . 2874
6.26.4 Restrictions . . . 2874
6.26.5 Related commands . . . 2874
6.26.6 Default . . . 2874
6.27 angle_style spica command . . . 2874
6.27.1 Syntax . . . 2874
6.27.2 Examples . . . 2875
6.27.3 Description . . . 2875
6.27.4 Restrictions . . . 2875
6.27.5 Related commands . . . 2875
6.27.6 Default . . . 2876
6.28 angle_style table command . . . 2876
6.28.1 Syntax . . . 2876
6.28.2 Examples . . . 2876
6.28.3 Description . . . 2876
6.28.4 Restart, fix_modify, output, run start/stop, minimize info . . . 2878

6.28.5 Restrictions . . . 2878
6.28.6 Related commands . . . 2878
6.28.7 Default . . . 2878
6.29 angle_style zero command . . . 2878
6.29.1 Syntax . . . 2878
6.29.2 Examples . . . 2878
6.29.3 Description . . . 2878
6.29.4 Restrictions . . . 2879
6.29.5 Related commands . . . 2879
6.29.6 Default . . . 2879

**7 Dihedral Styles 2881**
7.1 dihedral_style charmm command . . . 2881
7.2 dihedral_style charmmfsw command . . . 2881
7.2.1 Syntax . . . 2881
7.2.2 Examples . . . 2881
7.2.3 Description . . . 2881
7.2.4 Restrictions . . . 2883
7.2.5 Related commands . . . 2883
7.2.6 Default . . . 2883
7.3 dihedral_style class2 command . . . 2883
7.4 dihedral_style class2xe command . . . 2883
7.4.1 Syntax . . . 2883
7.4.2 Examples . . . 2883
7.4.3 Description . . . 2884
7.4.4 Restrictions . . . 2887
7.4.5 Related commands . . . 2887
7.4.6 Default . . . 2887
7.5 dihedral_style cosine/shift/exp command . . . 2888
7.5.1 Syntax . . . 2888
7.5.2 Examples . . . 2888
7.5.3 Description . . . 2888
7.5.4 Restrictions . . . 2888
7.5.5 Related commands . . . 2889
7.5.6 Default . . . 2889
7.6 dihedral_style cosine/squared/restricted command . . . 2889
7.6.1 Syntax . . . 2889
7.6.2 Examples . . . 2889
7.6.3 Description . . . 2889
7.6.4 Restrictions . . . 2890
7.6.5 Related commands . . . 2890
7.6.6 Default . . . 2890
7.7 dihedral_style fourier command . . . 2890
7.7.1 Syntax . . . 2890
7.7.2 Examples . . . 2890
7.7.3 Description . . . 2890
7.7.4 Restrictions . . . 2891
7.7.5 Related commands . . . 2891
7.7.6 Default . . . 2891
7.8 dihedral_style harmonic command . . . 2891
7.8.1 Syntax . . . 2891
7.8.2 Examples . . . 2891
7.8.3 Description . . . 2891
7.8.4 Restrictions . . . 2892

7.8.5 Related commands . . . 2892
7.8.6 Default . . . 2892
7.9 dihedral_style helix command . . . 2892
7.9.1 Syntax . . . 2892
7.9.2 Examples . . . 2892
7.9.3 Description . . . 2893
7.9.4 Restrictions . . . 2893
7.9.5 Related commands . . . 2893
7.9.6 Default . . . 2893
7.10 dihedral_style hybrid command . . . 2893
7.10.1 Syntax . . . 2894
7.10.2 Examples . . . 2894
7.10.3 Description . . . 2894
7.10.4 Restrictions . . . 2895
7.10.5 Related commands . . . 2895
7.10.6 Default . . . 2895
7.11 dihedral_style lepton command . . . 2895
7.11.1 Syntax . . . 2895
7.11.2 Examples . . . 2895
7.11.3 Description . . . 2895
7.11.4 Lepton expression syntax and features . . . 2896
7.11.5 Restrictions . . . 2897
7.11.6 Related commands . . . 2897
7.11.7 Default . . . 2897
7.12 dihedral_style multi/harmonic command . . . 2897
7.12.1 Syntax . . . 2897
7.12.2 Examples . . . 2897
7.12.3 Description . . . 2898
7.12.4 Restrictions . . . 2898
7.12.5 Related commands . . . 2898
7.12.6 Default . . . 2898
7.13 dihedral_style nharmonic command . . . 2898
7.13.1 Syntax . . . 2898
7.13.2 Examples . . . 2899
7.13.3 Description . . . 2899
7.13.4 Restrictions . . . 2899
7.13.5 Related commands . . . 2899
7.13.6 Default . . . 2899
7.14 dihedral_style none command . . . 2899
7.14.1 Syntax . . . 2899
7.14.2 Examples . . . 2900
7.14.3 Description . . . 2900
7.14.4 Restrictions . . . 2900
7.14.5 Related commands . . . 2900
7.14.6 Default . . . 2900
7.15 dihedral_style opls command . . . 2900
7.15.1 Syntax . . . 2900
7.15.2 Examples . . . 2900
7.15.3 Description . . . 2900
7.15.4 Restrictions . . . 2901
7.15.5 Related commands . . . 2901
7.15.6 Default . . . 2901
7.16 dihedral_style quadratic command . . . 2901
7.16.1 Syntax . . . 2901

7.16.2 Examples . . . 2901
7.16.3 Description . . . 2902
7.16.4 Restrictions . . . 2902
7.16.5 Related commands . . . 2902
7.16.6 Default . . . 2902
7.17 dihedral_style spherical command . . . 2902
7.17.1 Syntax . . . 2902
7.17.2 Examples . . . 2902
7.17.3 Description . . . 2903
7.17.4 Restrictions . . . 2904
7.17.5 Related commands . . . 2904
7.17.6 Default . . . 2904
7.18 dihedral_style table command . . . 2904
7.19 dihedral_style table/cut command . . . 2904
7.19.1 Syntax . . . 2905
7.19.2 Examples . . . 2905
7.19.3 Description . . . 2905
7.19.4 Restart, fix_modify, output, run start/stop, minimize info . . . 2908
7.19.5 Restrictions . . . 2908
7.19.6 Related commands . . . 2908
7.19.7 Default . . . 2908
7.20 dihedral_style zero command . . . 2908
7.20.1 Syntax . . . 2908
7.20.2 Examples . . . 2908
7.20.3 Description . . . 2908
7.20.4 Restrictions . . . 2909
7.20.5 Related commands . . . 2909
7.20.6 Default . . . 2909

**8 Improper Styles 2911**
8.1 improper_style amoeba command . . . 2911
8.1.1 Syntax . . . 2911
8.1.2 Examples . . . 2911
8.1.3 Description . . . 2911
8.1.4 Symmetry convention . . . 2911
8.1.5 Restrictions . . . 2912
8.1.6 Related commands . . . 2912
8.1.7 Default . . . 2912
8.2 improper_style class2 command . . . 2912
8.2.1 Syntax . . . 2912
8.2.2 Examples . . . 2912
8.2.3 Description . . . 2912
8.2.4 Symmetry convention . . . 2913
8.2.5 Restrictions . . . 2913
8.2.6 Related commands . . . 2914
8.2.7 Default . . . 2914
8.3 improper_style cossq command . . . 2914
8.3.1 Syntax . . . 2914
8.3.2 Examples . . . 2914
8.3.3 Description . . . 2914
8.3.4 Symmetry convention . . . 2915
8.3.5 Restrictions . . . 2915
8.3.6 Related commands . . . 2915
8.3.7 Default . . . 2915

8.4 improper_style cvff command . . . 2915
8.4.1 Syntax . . . 2915
8.4.2 Examples . . . 2915
8.4.3 Description . . . 2915
8.4.4 Symmetry convention . . . 2916
8.4.5 Restrictions . . . 2916
8.4.6 Related commands . . . 2916
8.4.7 Default . . . 2916
8.5 improper_style distance command . . . 2916
8.5.1 Syntax . . . 2916
8.5.2 Examples . . . 2916
8.5.3 Description . . . 2917
8.5.4 Symmetry convention . . . 2917
8.5.5 Restrictions . . . 2918
8.5.6 Related commands . . . 2918
8.5.7 Default . . . 2918
8.6 improper_style distharm command . . . 2918
8.6.1 Syntax . . . 2918
8.6.2 Examples . . . 2918
8.6.3 Description . . . 2918
8.6.4 Symmetry convention . . . 2919
8.6.5 Restrictions . . . 2919
8.6.6 Related commands . . . 2919
8.6.7 Default . . . 2919
8.7 improper_style fourier command . . . 2919
8.7.1 Syntax . . . 2919
8.7.2 Examples . . . 2919
8.7.3 Description . . . 2919
8.7.4 Symmetry convention . . . 2920
8.7.5 Restrictions . . . 2921
8.7.6 Related commands . . . 2921
8.7.7 Default . . . 2921
8.8 improper_style harmonic command . . . 2921
8.8.1 Syntax . . . 2921
8.8.2 Examples . . . 2921
8.8.3 Description . . . 2921
8.8.4 Symmetry convention . . . 2922
8.8.5 Restrictions . . . 2922
8.8.6 Related commands . . . 2922
8.8.7 Default . . . 2922
8.9 improper_style hybrid command . . . 2922
8.9.1 Syntax . . . 2922
8.9.2 Examples . . . 2922
8.9.3 Description . . . 2922
8.9.4 Restrictions . . . 2923
8.9.5 Related commands . . . 2924
8.9.6 Default . . . 2924
8.10 improper_style inversion/harmonic command . . . 2924
8.10.1 Syntax . . . 2924
8.10.2 Examples . . . 2924
8.10.3 Description . . . 2924
8.10.4 Symmetry convention . . . 2925
8.10.5 Restrictions . . . 2925
8.10.6 Related commands . . . 2925

8.10.7 Default . . . 2925
8.11 improper_style none command . . . 2925
8.11.1 Syntax . . . 2925
8.11.2 Examples . . . 2925
8.11.3 Description . . . 2926
8.11.4 Restrictions . . . 2926
8.11.5 Related commands . . . 2926
8.11.6 Default . . . 2926
8.12 improper_style ring command . . . 2926
8.12.1 Syntax . . . 2926
8.12.2 Examples . . . 2926
8.12.3 Description . . . 2926
8.12.4 Symmetry convention . . . 2927
8.12.5 Restrictions . . . 2927
8.12.6 Related commands . . . 2927
8.13 improper_style sqdistharm command . . . 2927
8.13.1 Syntax . . . 2927
8.13.2 Examples . . . 2927
8.13.3 Description . . . 2928
8.13.4 Symmetry convention . . . 2928
8.13.5 Restrictions . . . 2928
8.13.6 Related commands . . . 2928
8.13.7 Default . . . 2928
8.14 improper_style umbrella command . . . 2928
8.14.1 Syntax . . . 2929
8.14.2 Examples . . . 2929
8.14.3 Description . . . 2929
8.14.4 Symmetry convention . . . 2930
8.14.5 Restrictions . . . 2930
8.14.6 Related commands . . . 2930
8.14.7 Default . . . 2930
8.15 improper_style zero command . . . 2930
8.15.1 Syntax . . . 2930
8.15.2 Examples . . . 2930
8.15.3 Description . . . 2930
8.15.4 Restrictions . . . 2931
8.15.5 Related commands . . . 2931
8.15.6 Default . . . 2931

**9 Dump Styles 2933**
9.1 dump command . . . 2933
9.2 *dump vtk* command . . . 2933
9.3 *dump h5md* command . . . 2933
9.4 *dump molfile* command . . . 2933
9.5 *dump netcdf* command . . . 2933
9.6 *dump image* command . . . 2933
9.7 *dump movie* command . . . 2933
9.8 *dump atom/adios* command . . . 2933
9.9 *dump custom/adios* command . . . 2933
9.10 *dump cfg/uef* command . . . 2933
9.10.1 Syntax . . . 2933
9.10.2 Examples . . . 2935
9.10.3 Description . . . 2936
9.10.4 Restrictions . . . 2946

9.10.5 Related commands . . . 2946
9.10.6 Default . . . 2946
9.11 dump atom/adios command . . . 2946
9.12 dump custom/adios command . . . 2946
9.12.1 Syntax . . . 2946
9.12.2 Examples . . . 2946
9.12.3 Description . . . 2947
9.12.4 Restrictions . . . 2947
9.12.5 Related commands . . . 2947
9.13 dump cfg/uef command . . . 2947
9.13.1 Syntax . . . 2947
9.13.2 Examples . . . 2947
9.13.3 Description . . . 2948
9.13.4 Restrictions . . . 2948
9.13.5 Related commands . . . 2948
9.13.6 Default . . . 2948
9.14 dump h5md command . . . 2948
9.14.1 Syntax . . . 2948
9.14.2 Examples . . . 2949
9.14.3 Description . . . 2949
9.14.4 Restrictions . . . 2949
9.14.5 Related commands . . . 2949
9.15 dump image command . . . 2950
9.16 dump movie command . . . 2950
9.16.1 Syntax . . . 2950
9.17 dump_modify options for dump image/movie . . . 2952
9.17.1 Syntax . . . 2952
9.17.2 Examples . . . 2954
9.17.3 Description . . . 2954
9.17.4 Dump_modify keywords for dump image and dump movie . . . 2961
9.17.5 Restrictions . . . 2966
9.17.6 Related commands . . . 2967
9.17.7 Default . . . 2967
9.18 dump_modify command . . . 2970
9.19 *dump_modify* command for image/movie options . . . 2970
9.19.1 Syntax . . . 2970
9.19.2 Examples . . . 2972
9.19.3 Description . . . 2972
9.19.4 Restrictions . . . 2982
9.19.5 Related commands . . . 2982
9.19.6 Default . . . 2982
9.20 dump molfile command . . . 2983
9.20.1 Syntax . . . 2983
9.20.2 Examples . . . 2984
9.20.3 Description . . . 2984
9.20.4 Restrictions . . . 2984
9.20.5 Related commands . . . 2985
9.20.6 Default . . . 2985
9.21 dump netcdf command . . . 2985
9.22 dump netcdf/mpiio command . . . 2985
9.22.1 Syntax . . . 2985
9.22.2 Examples . . . 2985
9.22.3 Description . . . 2986
9.22.4 Restrictions . . . 2986

9.22.5 Related commands . . . 2986
9.23 dump vtk command . . . 2986
9.23.1 Syntax . . . 2986
9.23.2 Examples . . . 2986
9.23.3 Description . . . 2987
9.23.4 Restrictions . . . 2988
9.23.5 Related commands . . . 2988
9.23.6 Default . . . 2988

**10 Bibliography** **2989**

## IV Indices and tables 3019

**Index** **3021**

# Part

# About LAMMPS and this manual

LAMMPS stands for **L**arge-scale **A**tomic/**M**olecular **M**assively **P**arallel **S**imulator.

LAMMPS is a classical molecular dynamics simulation code focusing on materials modeling. It was designed to run efficiently on parallel computers and to be easy to extend and modify. Originally developed at Sandia National Laboratories, a US Department of Energy facility, LAMMPS now includes contributions from many research groups and individuals from many institutions. Most of the funding for LAMMPS has come from the US Department of Energy (DOE). LAMMPS is open-source software distributed under the terms of the GNU Public License Version 2 (GPLv2).

The LAMMPS website has a variety of information about the code. It includes links to an online version of this manual, an online forum where users can post questions and discuss LAMMPS, and a GitHub site where all LAMMPS development is coordinated.

---

The content for this manual is part of the LAMMPS distribution in its doc directory.

- The version of the manual on the LAMMPS website corresponds to the latest LAMMPS feature release. It is available at: https://docs.lammps.org/.
- A version of the manual corresponding to the latest LAMMPS stable release (state of the *stable* branch on GitHub) is available online at: https://docs.lammps.org/stable/
- A version of the manual with the features most recently added to LAMMPS (state of the *develop* branch on GitHub) is available at: https://docs.lammps.org/latest/

If needed, you can build a copy on your local machine of the manual (HTML pages or PDF file) for the version of LAMMPS you have downloaded. Follow the steps on the *Build the LAMMPS documentation* page.

---

The manual is organized into three parts:

1. The *User Guide* with information about how to obtain, configure, compile, install, and use LAMMPS,
2. the *Programmer Guide* with information about how to use the LAMMPS library interface from different programming languages, how to modify and extend LAMMPS, the program design, internal programming interfaces, and code design conventions,
3. the *Command Reference* with detailed descriptions of all input script commands available in LAMMPS.

---

---

# Part I

# User Guide

CHAPTER

ONE

# INTRODUCTION

These pages provide a brief introduction to LAMMPS.

## 1.1 Overview of LAMMPS

LAMMPS is a classical molecular dynamics (MD) code that models ensembles of particles in a liquid, solid, or gaseous state. It can model atomic, polymeric, biological, solid-state (metals, ceramics, oxides), granular, coarse-grained, or macroscopic systems using a variety of interatomic potentials (force fields) and boundary conditions. It can model 2d or 3d systems with sizes ranging from only a few particles up to billions.

LAMMPS can be built and run on single laptop or desktop machines, but is designed for parallel computers. It will run in serial and on any parallel machine that supports the MPI message-passing library. This includes shared-memory multicore, multi-CPU servers and distributed-memory clusters and supercomputers. Parts of LAMMPS also support OpenMP multi-threading, vectorization, and GPU acceleration.

LAMMPS is written in C++ and currently requires a compiler that is at least compatible with the C++-11 standard. Earlier versions were written in F77, F90, and C++-98. See the History page of the website for details. All versions can be downloaded as source code from the LAMMPS website.

Through a *C language API* LAMMPS functionality can be accessed and managed from other programming languages rather than running the LAMMPS executable. Ready to use modules for *Python* and *Fortran* exist, and an example *SWIG interface file* as well as example C files for dynamically loading LAMMPS as a shared library into other executables are provided.

LAMMPS is designed to be easy to modify or extend with new capabilities, such as new force fields, atom types, boundary conditions, or diagnostics. See the *Modifying & extending LAMMPS* section for more details.

In the most general sense, LAMMPS integrates Newton's equations of motion for a collection of interacting particles. A single particle can be an atom or molecule or electron, a coarse-grained cluster of atoms, or a mesoscopic or macroscopic clump of material. The interaction models that LAMMPS includes are mostly short-ranged in nature; some long-range models are included as well.

LAMMPS uses neighbor lists to keep track of nearby particles. The lists are optimized for systems with particles that are repulsive at short distances, so that the local density of particles never becomes too large. This is in contrast to methods used for modeling plasma or gravitational bodies (like galaxy formation).

On parallel machines, LAMMPS uses spatial-decomposition techniques with MPI parallelization to partition the simulation domain into subdomains of equal computational cost, one of which is assigned to each processor. Processors communicate and store "ghost" atom information for atoms that border their subdomain. Multi-threading parallelization and GPU acceleration with particle-decomposition can be used in addition.

## 1.2 What does a LAMMPS version mean

The LAMMPS "version" is the date when it was released, such as 1 May 2014. LAMMPS is updated continuously, and with the help of extensive automated testing (mostly applied *before* changes are included) we aim to keep it working correctly and reliably at all times, but there also are regular *feature releases* with new and expanded functionality, and there are designated *stable releases* that receive updates with bug fixes back-ported from the development branch.

In addition to automated testing, several variants of static code analysis are run regularly to maintain or improve the overall code quality, consistency, and compliance with programming standards, best practices and style conventions. You can follow its development in a public git repository on GitHub.

Each version of LAMMPS contains all the documented *features* up to and including its version date. For recently added features, we add markers to the documentation at which specific LAMMPS version a feature or keyword was added or significantly changed.

### 1.2.1 Identifying the Version

The version date is printed to the screen and log file every time you run LAMMPS. There also is an indication, if a LAMMPS binary was compiled from version with modifications **after** a release, either from the development version or the maintenance version of the last stable release. It is also visible in the file src/version.h and in the LAMMPS directory name created when you unpack a downloaded tarball. And it is on the first page of the *manual*.

- If you browse the HTML pages of the online version of the LAMMPS manual, they will by default describe the most current feature release version of LAMMPS. In the navigation bar on the bottom left, there is the option to view instead the documentation for the most recent *stable* version or the documentation corresponding to the state of the development branch *develop*.
- If you browse the HTML pages included in your downloaded tarball, they describe the version you have, which may be older than the online version.

### 1.2.2 LAMMPS releases, branches, and tags

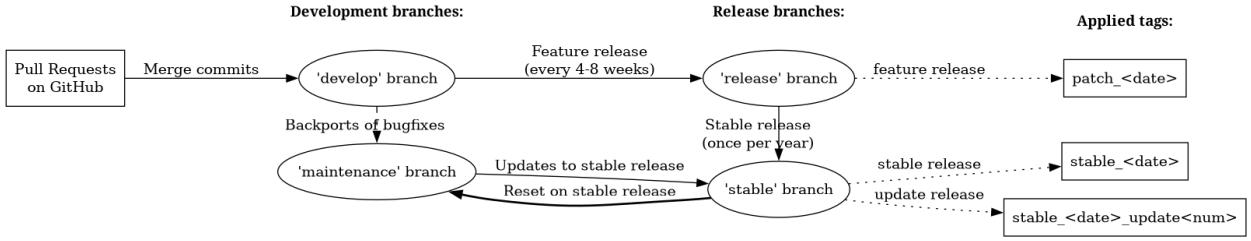



Fig. 1: Relations between releases, main branches, and tags in the LAMMPS git repository

#### Development

Modifications of the LAMMPS source code (like bug fixes, code refactoring, updates to existing features, or addition of new features) are organized into pull requests. Pull requests will be merged into the *develop* branch of the LAMMPS git repository on GitHub after they pass automated testing and code review by *core LAMMPS developers*.

#### Feature Releases

When a sufficient number of new features and updates have accumulated *and* the LAMMPS version on the *develop* branch passes an extended set of automated tests, we release it as a *feature release*, which are currently made every 4 to 8 weeks. The *release* branch of the git repository is updated with every such *feature release* and a tag in the format `patch_1May2014` is added. A summary of the most important changes of these releases for the current year are posted on this website page. More detailed release notes are available on GitHub.

#### Stable Releases

About once a year, we release a *stable release* version of LAMMPS. This is done after a "stabilization period" where we apply only bug fixes and small, non-intrusive changes to the *develop* branch but no new features to the core code. At the same time, the code is subjected to more detailed and thorough manual testing that goes beyond the automated testing done by default. After such a *stable release*, both the *release* and the *stable* branches are updated and two tags are applied, a `patch_1May2014` format and a `stable_1May2014` format tag.

#### Stable Release Updates

Between *stable releases*, we collect bug fixes and updates that are back-ported from the *develop* branch in a branch called *maintenance*. From the *maintenance* branch we make occasional *stable update releases* and update the *stable* branch accordingly. The first update to the `stable_1May2014` release would be tagged as `stable_1May2014_update1`. These updates contain no new features and are intended to be fully backward compatible.

## 1.3 LAMMPS features

LAMMPS is a classical molecular dynamics (MD) code with these general classes of functionality:

1. *General features*
2. *Particle and model types*
3. *Interatomic potentials (force fields)*
4. *Atom creation*
5. *Ensembles, constraints, and boundary conditions*
6. *Integrators*
7. *Diagnostics*
8. *Output*
9. *Multi-replica models*
10. *Pre- and post-processing*
11. *Specialized features (beyond MD itself)*

---

### 1.3.1 General features

- runs on a single processor or in parallel
- distributed memory message-passing parallelism (MPI)
- shared memory multi-threading parallelism (OpenMP)
- spatial decomposition of simulation domain for MPI parallelism
- particle decomposition inside spatial decomposition for OpenMP and GPU parallelism
- GPLv2 licensed open-source distribution
- highly portable C++-17 (optional packages may require C++20)
- modular code with most functionality in optional packages
- only depends on MPI library for basic parallel functionality, MPI stub for serial compilation
- other libraries are optional and only required for specific packages

- GPU (CUDA, OpenCL, HIP, SYCL) and OpenMP support for many code features
- easy to extend with new features and functionality
- runs from an input script
- syntax for defining and using variables and formulas
- syntax for looping over runs and breaking out of loops
- run one or multiple simulations simultaneously (in parallel) from one script
- build as library, invoke LAMMPS through library interface (from C, C++, Fortran) or provided Python wrapper or SWIG based wrappers
- couple with other codes: LAMMPS calls other code, other code calls LAMMPS, umbrella code calls both, MDI coupling interface
- call out to Python for computing forces, time integration, or other tasks
- plugin interface for loading external features at runtime
- large integrated collection of tests

### 1.3.2 Particle and model types

(See *atom style* command)

- atoms
- coarse-grained particles (e.g. bead-spring polymers)
- united-atom polymers or organic molecules
- all-atom polymers, organic molecules, proteins, DNA
- metals
- metal oxides
- granular materials
- coarse-grained mesoscale models
- finite-size spherical and ellipsoidal particles
- finite-size line segment (2d) and triangle (3d) particles
- finite-size rounded polygons (2d) and polyhedra (3d) particles
- point dipole particles
- particles with magnetic spin
- rigid collections of n particles
- hybrid combinations of these

### 1.3.3 Interatomic potentials (force fields)

(See *pair style*, *bond style*, *angle style*, *dihedral style*, *improper style*, *kspace style* commands)

- pairwise potentials: Lennard-Jones, Buckingham, Morse, Born-Mayer-Huggins, Yukawa, soft, Class II (COMPASS), hydrogen bond, harmonic, gaussian, tabulated, scripted
- charged pairwise potentials: Coulombic, point-dipole

- many-body potentials: EAM, Finnis/Sinclair, MEAM, MEAM+SW, EIM, EDIP, ADP, Stillinger-Weber, Tersoff, REBO, AIREBO, ReaxFF, COMB, Streitz-Mintmire, 3-body polymorphic, BOP, Vashishta
- machine learning potentials: ACE, AGNI, GAP, Behler-Parrinello (N2P2), POD, RANN, SNAP
- interfaces to ML potentials distributed by external groups: ANI, ChIMES, DeepPot, HIPNN, MTP
- long-range interactions for charge, point-dipoles, and LJ dispersion: Ewald, Wolf, PPPM (similar to particle-mesh Ewald), MSM, ScaFaCoS
- polarization models: *QEq*, *core/shell model*, *Drude dipole model*
- charge equilibration (QEq via dynamic, point, shielded, Slater methods)
- coarse-grained potentials: DPD, GayBerne, REsquared, colloidal, DLVO, oxDNA / oxRNA, SPICA
- mesoscopic potentials: granular, Peridynamics, SPH, mesoscopic tubular potential (MESONT)
- semi-empirical potentials: multi-ion generalized pseudopotential theory (MGPT), second moment tight binding + QEq (SMTB-Q)
- electron force field (eFF)
- bond potentials: harmonic, FENE, Morse, nonlinear, Class II (COMPASS), quartic (breakable), tabulated, scripted
- angle potentials: harmonic, CHARMM, cosine, cosine/squared, cosine/periodic, Class II (COMPASS), tabulated, scripted
- dihedral potentials: harmonic, CHARMM, multi-harmonic, helix, Class II (COMPASS), OPLS, tabulated, scripted
- improper potentials: harmonic, cvff, umbrella, Class II (COMPASS), tabulated
- polymer potentials: all-atom, united-atom, bead-spring, breakable
- water potentials: TIP3P, TIP4P, SPC, SPC/E and variants
- interlayer potentials for graphene and analogues, hetero-junctions
- metal-organic framework potentials (QuickFF, MO-FF)
- implicit solvent potentials: hydrodynamic lubrication, Debye
- force-field compatibility with CHARMM, AMBER, DREIDING, OPLS, GROMACS, Class II (COMPASS), UFF, ClayFF, DREIDING, AMOEBA, INTERFACE
- access to the OpenKIM Repository of potentials via the *kim command*
- hybrid potentials: multiple pair, bond, angle, dihedral, improper potentials can be used in one simulation
- overlaid potentials: superposition of multiple pair potentials (including many-body) with optional scale factor

### 1.3.4 Atom creation

(See *read_data*, *lattice*, *create_atoms*, *delete_atoms*, *displace_atoms*, *replicate* commands)

- read in atom coordinates from files
- create atoms on one or more lattices (e.g. grain boundaries)
- delete geometric or logical groups of atoms (e.g. voids)
- replicate existing atoms multiple times
- displace atoms

### 1.3.5 Ensembles, constraints, and boundary conditions

(See *fix* command)

- 2d or 3d systems
- orthogonal or non-orthogonal (triclinic symmetry) simulation domains
- constant NVE, NVT, NPT, NPH, Parrinello/Rahman integrators
- thermostatting options for groups and geometric regions of atoms
- pressure control via Nose/Hoover or Berendsen barostatting in 1 to 3 dimensions
- simulation box deformation (tensile and shear)
- harmonic (umbrella) constraint forces
- rigid body constraints
- SHAKE / RATTLE bond and angle constraints
- motion constraints to manifold surfaces
- Monte Carlo bond breaking, formation, swapping, template based reaction modeling
- atom/molecule insertion and deletion
- walls of various kinds, static and moving
- non-equilibrium molecular dynamics (NEMD)
- variety of additional boundary conditions and constraints

### 1.3.6 Integrators

(See *run*, *run_style*, *minimize* commands)

- velocity-Verlet integrator
- Brownian dynamics
- rigid body integration
- energy minimization via conjugate gradient, steepest descent relaxation, or damped dynamics (FIRE, Quickmin)
- rRESPA hierarchical timestepping
- fixed or adaptive time step
- rerun command for post-processing of dump files

### 1.3.7 Diagnostics

- see various flavors of the *fix* and *compute* commands
- introspection command for system, simulation, and compile time settings and configurations

### 1.3.8 Output

(*dump*, *restart* commands)

- log file of thermodynamic info
- text dump files of atom coordinates, velocities, other per-atom quantities
- dump output on fixed and variable intervals, based timestep or simulated time

- binary restart files
- parallel I/O of dump and restart files
- per-atom quantities (energy, stress, centro-symmetry parameter, CNA, etc.)
- user-defined system-wide (log file) or per-atom (dump file) calculations
- custom partitioning (chunks) for binning, and static or dynamic grouping of atoms for analysis
- spatial, time, and per-chunk averaging of per-atom quantities
- time averaging and histogramming of system-wide quantities
- atom snapshots in native, XYZ, XTC, DCD, CFG, NetCDF, HDF5, ADIOS2, YAML formats
- on-the-fly compression of output and decompression of read in files

### 1.3.9 Multi-replica models

- *nudged elastic band*
- *hyperdynamics*
- *parallel replica dynamics*
- *temperature accelerated dynamics*
- *parallel tempering*
- path-integral MD: *first variant*, *second variant*
- multi-walker collective variables with *Colvars* and *Plumed*

### 1.3.10 Pre- and post-processing

- A handful of pre- and post-processing tools are packaged with LAMMPS, some of which can convert input and output files to/from formats used by other codes; see the *Tools* page.
- Our group has also written and released a separate toolkit called Pizza.py which provides tools for doing setup, analysis, plotting, and visualization for LAMMPS simulations. Pizza.py is written in Python and is available for download from the Pizza.py WWW site.

### 1.3.11 Specialized features

LAMMPS can be built with optional packages which implement a variety of additional capabilities. See the *Optional Packages* page for details.

These are LAMMPS capabilities which you may not think of as typical classical MD options:

- *static* and *dynamic load-balancing*, optional with recursive bisectioning decomposition
- *generalized aspherical particles*
- *stochastic rotation dynamics (SRD)*
- *real-time visualization and interactive MD*, *built-in renderer for images and movies*
- calculate *virtual diffraction patterns*
- calculate *finite temperature phonon dispersion* and the *dynamical matrix of minimized structures*
- *QM/MM coupling*
- Monte Carlo via *GCMC* and *tfMC* and *atom swapping*
- *path-integral molecular dynamics (PIMD)* and *this as well*

- *Direct Simulation Monte Carlo* for low-density fluids
- *Peridynamics modeling*
- *Lattice Boltzmann fluid*
- *targeted* and *steered* molecular dynamics
- *two-temperature electron model*

## 1.4 LAMMPS non-features

LAMMPS is designed to be a fast, parallel engine for molecular dynamics (MD) simulations. It provides only a modest amount of functionality for setting up simulations and analyzing their output.

Originally, LAMMPS was not conceived and designed for:

- being run through a GUI
- building molecular systems, or building molecular topologies
- assign force-field coefficients automagically
- perform sophisticated analysis of your MD simulation
- visualize your MD simulation interactively
- plot your output data

Over the years many of these limitations have been reduced or removed. In part through features added to LAMMPS and in part through external tools that either closely interface with LAMMPS or extend LAMMPS.

Here are suggestions on how to perform these tasks:

- **GUI:** LAMMPS can be built as a library and a Python module that wraps the library interface is provided. Thus, GUI interfaces can be written in Python or C/C++ that run LAMMPS and visualize or plot its output. Examples of this are provided in the python directory and described on the *Python* doc page. Since version 2 August 2023 the LAMMPS-GUI application is available and can be compiled together with LAMMPS and linked to the LAMMPS library for running and visualizing LAMMPS simulation inputs. As of August 2025, LAMMPS-GUI is maintained in its own repository on GitHub. Also, there are several external wrappers or GUI front ends that are mentioned on the Pre-/post-processing tools page of the LAMMPS homepage.
- **Builder:** Several pre-processing tools are packaged with LAMMPS. Some of them convert input files in formats produced by other MD codes such as CHARMM, AMBER, or Insight into LAMMPS input formats. Some of them are simple programs that will build simple molecular systems, such as linear bead-spring polymer chains. The moltemplate program is a true molecular builder that will generate complex molecular models. See the *Tools* page for details on tools packaged with LAMMPS. The Pre-/post-processing tools page of the LAMMPS homepage describes a variety of third party tools for this task. Furthermore, some internal LAMMPS commands allow reconstructing, or selectively adding topology information, as well as provide the option to insert molecule templates instead of atoms for building bulk molecular systems.
- **Force-field assignment:** The conversion tools described in the previous bullet for CHARMM, AMBER, and Insight will also assign force field coefficients in the LAMMPS format, assuming you provide CHARMM, AMBER, or BIOVIA (formerly Accelrys) force field files. The tools ParmEd and InterMol are particularly powerful and flexible in converting force field and topology data between various MD simulation programs.
- **Simulation analysis:** If you want to perform analysis on-the-fly as your simulation runs, see the *compute* and *fix* doc pages, which list commands that can be used in a LAMMPS input script. Also see the *Modify* page for info on how to add your own analysis code or algorithms to LAMMPS. For post-processing, LAMMPS output such as *dump file snapshots* can be converted into formats used by other MD or post-processing codes. To some degree, that conversion can be done directly inside LAMMPS by interfacing to the VMD molfile plugins. The *rerun* command also allows post-processing of existing trajectories, and through being able to read a variety of

file formats, this can also be used for analyzing trajectories from other MD codes. Some post-processing tools packaged with LAMMPS will do these conversions. Scripts provided in the tools/python directory can extract and massage data in dump files to make it easier to import into other programs. See the *Tools* page for details on these various options.

The Pre-/post-processing page on the LAMMPS homepage lists some external packages for analysis of MD simulation data, including data produced by LAMMPS.

- **Visualization:** LAMMPS can produce NETPBM, TGA, JPG, or PNG format snapshot images on-the-fly via its *dump image* command and pass them to an external program, FFmpeg, to generate movies from them. The *LAMMPS-GUI tool* has a *Snapshot Image Viewer* which uses *dump image* and allows to modify the visualization settings interactively. It also has a *Slide Show* feature where images created by *dump image* are collected during a simulation and can be animated interactively or exported to a movie with FFmpeg or ImageMagick.

  For high-quality, interactive visualization, there are many excellent and free tools available. See the Visualization Tools page of the LAMMPS website for visualization packages that can process LAMMPS output data.

- **Plotting:** See the next bullet about Pizza.py as well as the *Python* page for examples of plotting LAMMPS output. Scripts provided with the *python* tool in the `tools` directory will extract and process data in log and dump files to make it easier to analyze and plot. See the *Tools* doc page for more discussion of the various tools.

  The *LAMMPS-GUI tool* has a *Chart Viewer* where *thermodynamic data* computed by LAMMPS is collected during the simulation and plotted immediately.

- **Pizza.py:** Our group has also written a separate toolkit called Pizza.py which can do certain kinds of setup, analysis, plotting, and visualization (via OpenGL) for LAMMPS simulations. It thus provides some functionality for several of the above bullets. Pizza.py is written in Python and is available for download from https://lammps.github.io/pizza/.

## 1.5 LAMMPS portability and compatibility

The primary form of distributing LAMMPS is through highly portable source code. But also several ways of obtaining LAMMPS as *precompiled packages or through automated build mechanisms* exist. Most of LAMMPS is written in C++, some support tools are written in Fortran or Python or MATLAB.

### 1.5.1 Programming language standards

Changed in version 10Sep2025.

The C++ code in LAMMPS currently requires a compiler that is compatible with the C++17 standard. The Kokkos library used for the KOKKOS package currently also requires at least C++17. If your compilers are not compatible *and* you cannot upgrade to a compatible version, please use LAMMPS version 22 July 2025, which requires only C++11 as the minimum C++ standard.

Most of the Python code in LAMMPS is written to be compatible with Python 3.6 and later.

Deprecated since version 2Apr2025.

Python 2.x is no longer supported and trying to use it, e.g. for the LAMMPS Python module should result in an error. If you come across some part of the LAMMPS distribution that is not (yet) compatible with Python 3, please notify *the LAMMPS developers*.

### 1.5.2 Build systems

LAMMPS can be compiled from source code using the cross-platform CMake system. CMake must be at least version 3.20. Alternatively, using a (traditional) build system based on shell scripts, a few shell utilities (grep, sed, cat, tr) and the GNU make program. This requires running within a Bourne shell (`/bin/sh` or `/bin/bash`).

Changed in version 10Sep2025.

The traditional GNU make based build system no longer supports all packages. Details can be found in the *package specific build instructions*.

### 1.5.3 Operating systems

The primary development platform for LAMMPS is Linux. Thus, the chances for LAMMPS to compile without problems are the best on Linux machines. Also, compilation and correct execution on macOS and Windows (using Microsoft Visual C++) is checked automatically for the largest part of the source code. Some (optional) features are not compatible with all operating systems, either through limitations of the corresponding LAMMPS source code or through incompatibilities or build system limitations of required external libraries or packages.

Executables for Windows may be created either natively using Cygwin, MinGW, Intel, Clang, or Microsoft Visual C++ compilers, or with a Linux to Windows MinGW cross-compiler. Native compilation is supported using Microsoft Visual Studio or a terminal window (using the CMake build system).

Executables for macOS may be created either using Xcode or GNU compilers installed with Homebrew. In the latter case, building of LAMMPS through Homebrew instead of a manual compile is also possible.

Additionally, FreeBSD and Solaris have been tested successfully to run LAMMPS and produce results consistent with those on Linux.

### 1.5.4 Compilers

The most commonly used compilers are the GNU compilers, but also Clang and the Intel compilers have been successfully used on Linux, macOS, and Windows. Also, the Nvidia HPC SDK (formerly PGI compilers) will compile LAMMPS (tested on Linux).

Changed in version 10Sep2025.

The GNU compilers *before* version 9.3 have known problems with supporting C++17 and thus are **not** recommended to build LAMMPS.

### 1.5.5 CPU architectures

The primary CPU architecture for running LAMMPS is 64-bit x86, but also 64-bit ARM is currently regularly tested. Further architectures are tested by Linux distributions that bundle LAMMPS.

### 1.5.6 Portability compliance

Only a subset of the LAMMPS source code is *fully* compliant to *all* of the above mentioned standards. There is also the case that the source code bundled with LAMMPS *is* compliant and portable, but an external library it depends on is not.

This is rather typical for projects like LAMMPS that largely depend on contributions from the user community. Not all contributors are trained as programmers and not all of them have access to multiple platforms for testing or are familiar with requirement of different C++ standards. As part of the continuous integration process, however, all contributions are automatically tested to compile, link, and pass *some* runtime tests on a selection of Linux flavors, macOS, and Windows, and on Linux with different compilers. Thus portability issues are often found *before* a pull request is merged or a release is made. Other platforms may be checked occasionally or when portability bugs are reported.

### 1.5.7 Code review and static code analysis

In addition to using automated tests, code contributed to LAMMPS is subject to a code review by core LAMMPS developers (that includes contributions by the core LAMMPS developers themselves).

We make use of a number of static code analysis tools for maintaining and improving the quality of the LAMMPS source code. Tools used include:

- Coverity SCAN

- CodeQL
- Clang Static Analyzer
- Clang-Tidy
- Enabling and looking at compiler warnings

CodeQL alerts for the `develop` branch of LAMMPS can be seen at https://github.com/lammps/lammps/security/code-scanning. Static code analysis reports for the `develop` branch from the clang tools are available at https://download.lammps.org/analysis/

We currently also request code reviews from GitHub Copilot to lower the code review workload for the LAMMPS developers. The LAMMPS git repository includes specific instructions for Copilot to particularly look for compliance of submitted pull requests with LAMMPS coding and documentation conventions.

A discussion of software engineering methods applied to LAMMPS over time can be found in the paper *LAMMPS: A Case Study For Applying Modern Software Engineering to an Established Research Software Package, <https://doi.org/10.5281/zenodo.17117558>* in USRSE'25 conference proceedings.

## 1.6 LAMMPS open-source license

### 1.6.1 GPL version of LAMMPS

LAMMPS is an open-source code, available free-of-charge, and distributed under the terms of the GNU Public License Version 2 (GPLv2), which means you can use or modify the code however you wish for your own purposes, but have to adhere to certain rules when redistributing it - specifically in binary form - or are distributing software derived from it or that includes parts of it.

LAMMPS comes with no warranty of any kind.

As each source file states in its header, it is a copyrighted code, and thus not in the public domain. For more information about open-source software and open-source distribution, see www.gnu.org or www.opensource.org. The legal text of the GPL as it applies to LAMMPS is in the LICENSE file included in the LAMMPS distribution.

Here is a more specific summary of what the GPL means for LAMMPS users:

(1) Anyone is free to use, copy, modify, or extend LAMMPS in any way they choose, including for commercial purposes.

(2) If you **distribute** a modified version of LAMMPS, it must remain open-source, meaning you are required to distribute **all** of it under the terms of the GPLv2. You should **clearly** annotate such a modified code as a derivative version of LAMMPS. This is best done by changing the name (example: LIGGGHTS is such a modified and extended version of LAMMPS).

(3) If you release any code that includes or uses LAMMPS source code, then it must also be open-sourced, meaning you distribute it under the terms of the GPLv2. You may write code that interfaces LAMMPS to a differently licensed library. In that case the code that provides the interface must be licensed GPLv2, but not necessarily that library unless you are distributing binaries that require the library to run.

(4) If you give LAMMPS files to someone else, the GPLv2 LICENSE file and source file headers (including the copyright and GPLv2 notices) should remain part of the code.

### 1.6.2 LGPL version of LAMMPS

We occasionally make stable LAMMPS releases available under the GNU Lesser Public License v2.1. This is on request only and with non-LGPL compliant files removed. This allows uses linking non-GPL compatible software with the (otherwise unmodified) LAMMPS library or loading it dynamically at runtime. Any **modifications** to the LAMMPS code however, even with the LGPL licensed version, must still be made available under the same open source terms as LAMMPS itself.

## 1.7 Authors of LAMMPS

The current core LAMMPS developers are listed here (grouped by seniority and sorted alphabetically by last name). You can email an individual developer with code related questions for their area of expertise, or send an email to all of them at this address: "developers at lammps.org". General questions about LAMMPS should be posted in the LAMMPS forum on MatSci.

| Name | Affiliation | Email | Areas of expertise |
|---|---|---|---|
| Axel Kohlmeyer | Temple U | akohlmey at gmail.com | OpenMP, library interfaces, LAMMPS-GUI, GitHub, MatSci forum, code maintenance, testing, releases |
| Steve Plimpton | SNL (retired) | sjplimp at gmail.com | original author, MD kernels, parallel algorithms & scalability, code structure and design |
| Aidan Thompson | SNL | athomps at sandia.gov | manybody potentials, machine learned potentials, materials science, statistical mechanics |
| | | | |
| Richard Berger | LANL | richard.berger at outlook.com | Python, HPC, DevOps |
| Germain Clavier | U Caen | germain.clavier at unicaen.fr | organic molecules, polymers, mechanical properties, surfaces, integrators, coarse-graining |
| Joel Clemmer | SNL | jtclemm at sandia.gov | granular systems fluid/solid mechanics |
| Jacob R. Gissinger | Stevens Institute of Technology | jgissing at stevens.edu | reactive molecular dynamics, macro-molecular systems, type labels |
| James Goff | SNL | jmgoff at sandia.gov | machine learned potentials, QEq solvers, Python |
| Meg McCarthy | SNL | megmcca at sandia.gov | alloys, micro-structure, machine learned potentials |
| Stan Moore | SNL | stamoor at sandia.gov | Kokkos, KSpace solvers, ReaxFF |
| Trung Nguyen | U Chicago | ndactrung at gmail.com | soft matter, GPU package, DIELECTRIC package, regression testing |

Past core developers include Paul Crozier and Mark Stevens, both at SNL, and Ray Shan while at SNL and later at Materials Design, now at Thermo Fisher Scientific.

---

The Authors page of the LAMMPS website has a comprehensive list of all the individuals who have contributed code for a new feature or command or tool to LAMMPS.

---

The following folks deserve special recognition. Many of the packages they have written are unique for an MD code and LAMMPS would not be as general-purpose as it is without their expertise and efforts.

- Metin Aktulga (MSU), REAXFF package for C/C++ version of ReaxFF
- Mike Brown (Intel), GPU and INTEL packages
- Colin Denniston (U Western Ontario), LATBOLTZ package
- Georg Ganzenmuller (EMI), MACHDYN and SPH packages
- Andres Jaramillo-Botero (Caltech), EFF package for electron force field
- Reese Jones (Sandia) and colleagues, ATC package for atom/continuum coupling
- Christoph Kloss (DCS Computing), LIGGGHTS code for granular materials, built on top of LAMMPS
- Rudra Mukherjee (JPL), POEMS package for articulated rigid body motion
- Trung Nguyen (U Chicago), GPU, RIGID, BODY, and DIELECTRIC packages
- Mike Parks (Sandia), PERI package for Peridynamics
- Roy Pollock (LLNL), Ewald and PPPM solvers

- Julien Tranchida (CEA Cadarache), SPIN package
- Christian Trott (Sandia), CUDA and KOKKOS packages
- Ilya Valuev (JIHT), AWPMD package for wave packet MD
- Greg Wagner (Northwestern U), MEAM package for MEAM potential

---

As discussed on the History page of the website, LAMMPS originated as a cooperative project between DOE labs and industrial partners. Folks involved in the design and testing of the original version of LAMMPS were the following:

- John Carpenter (Mayo Clinic, formerly at Cray Research)
- Terry Stouch (Lexicon Pharmaceuticals, formerly at Bristol Myers Squibb)
- Steve Lustig (Dupont)
- Jim Belak and Roy Pollock (LLNL)

## 1.8 Citing LAMMPS

### 1.8.1 Core Algorithms

The paper mentioned below is the best overview of LAMMPS, but there are also publications describing particular models or algorithms implemented in LAMMPS or complementary software that it has interfaces to. Please see below for how to cite contributions to LAMMPS.

The latest canonical publication that describes the basic features, the source code design, the program structure, the spatial decomposition approach, the neighbor finding, basic communications algorithms, and how users and developers have contributed to LAMMPS is:

> LAMMPS - A flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales, Comp. Phys. Comm. 271, 108171 (2022)

So a project using LAMMPS or a derivative application that uses LAMMPS as a simulation engine should cite this paper. The paper is expected to be published in its final form under the same DOI in the first half of 2022. Please also give the URL of the LAMMPS website in your paper, namely https://www.lammps.org.

The original publication describing the parallel algorithms used in the initial versions of LAMMPS is:

> S. Plimpton, Fast Parallel Algorithms for Short-Range Molecular Dynamics, J Comp Phys, 117, 1-19 (1995).

### 1.8.2 DOI for the LAMMPS source code

The LAMMPS developers use the Zenodo service at CERN to create digital object identifiers (DOI) for stable releases of the LAMMPS source code. There are two types of DOIs for the LAMMPS source code.

The canonical DOI for **all** versions of LAMMPS, which will always point to the **latest** stable release version, is:

- DOI: 10.5281/zenodo.3726416

In addition there are DOIs generated for individual stable releases:

- 3 March 2020 version: DOI:10.5281/zenodo.3726417
- 29 October 2020 version: DOI:10.5281/zenodo.4157471
- 29 September 2021 version: DOI:10.5281/zenodo.6386596
- 23 June 2022 version: DOI:10.5281/zenodo.10806836
- 2 August 2023 version: DOI:10.5281/zenodo.10806852

### 1.8.3 Home page

The LAMMPS website at https://www.lammps.org/ is the canonical location for information about LAMMPS and its features.

### 1.8.4 Citing contributions

LAMMPS has many features that use either previously published methods and algorithms or novel features. It also includes potential parameter files for specific models. Where available, a reminder about references for optional features used in a specific run is printed to the screen and log file. Style and output location can be selected with the *-cite command-line switch*. Additional references are given in the documentation of the *corresponding commands* or in the *Howto tutorials*. Please make certain, that you provide the proper acknowledgments and citations in any published works using LAMMPS.

## 1.9 Additional website links

The LAMMPS website has a variety of additional info about LAMMPS, beyond what is in this manual. Some other useful resources available online are listed below.

- LAMMPS source code repository on GitHub
- LAMMPS plugins source code repository on GitHub
- LAMMPS forum on matsci.org
- Recent bug fixes and new features
- Download info
- Glossary of terms relevant to LAMMPS
- LAMMPS highlights with images
- LAMMPS highlights with movies
- Workshops
- Tutorials
- Pre- and post-processing tools for LAMMPS
- Other software usable with LAMMPS
- Viz tools usable with LAMMPS
- Benchmark performance
- Publications that have cited LAMMPS
- Authors of LAMMPS
- History of LAMMPS development
- Funding for LAMMPS

CHAPTER

TWO

# INSTALL LAMMPS

You can download LAMMPS as an executable or as source code.

When downloading the LAMMPS source code, you also have to *build LAMMPS*. But you have more flexibility as to what features to include or exclude in the build. When you download and install pre-compiled LAMMPS executables, you are limited to install which version of LAMMPS is available and which features are included of these builds. If you plan to *modify or extend LAMMPS*, then you **must** build LAMMPS from the source code.

**Note**

If you have questions about the pre-compiled LAMMPS executables, you need to contact the people preparing those executables. The LAMMPS developers have no control over their choices of how they configure and build their packages and when they update them.

## 2.1 Download an executable for Linux

Binaries are available for different versions of Linux:

- *Pre-built static Linux x86_64 executables*
- *Pre-built Ubuntu and Debian Linux executables*
- *Pre-built Fedora Linux executables*
- *Pre-built EPEL Linux executables (RHEL, CentOS)*
- *Pre-built OpenSuse Linux executables*
- *Gentoo Linux executable*
- *Arch Linux build-script*

**Note**

If you have questions about these pre-compiled LAMMPS executables, you need to contact the people preparing those packages. The LAMMPS developers have no control over how they configure and build their packages and when they update them. They may only provide packages for stable release versions and not always update the packages in a timely fashion after a new LAMMPS release is made.

### 2.1.1 Pre-built static Linux x86_64 executables

Pre-built LAMMPS executables for Linux, that are statically linked and compiled for 64-bit x86 CPUs (x86_64 or AMD64) are available for download at https://download.lammps.org/static/. Because of that static linkage (and unlike the Linux distribution specific packages listed below), they do not depend on any installed software and thus should run on *any* 64-bit x86 machine with *any* Linux version.

These executable include most of the available packages and multi-thread parallelization (via INTEL, KOKKOS, or OPENMP package). They are **not** compatible with MPI. Several of the LAMMPS tools executables (e.g. `msi2lmp`) are included as well. Because of the static linkage, there is no `liblammps.so` library file and thus also the LAMMPS python module, which depends on it, is not included.

The compressed tar archives available for download have names following the pattern `lammps-linux-x86_64-<version>.tar.gz` and will all unpack into a `lammps-static` folder. The executables are then in the `lammps-static/bin/` folder. Since they do not depend on any other software, they may be freely moved or copied around.

---

### 2.1.2 Pre-built Ubuntu and Debian Linux executables

A pre-built LAMMPS executable, suitable for running on the latest Ubuntu and Debian Linux versions, can be downloaded as a Debian package. This allows you to install LAMMPS with a single command, and stay (mostly) up-to-date with the current stable version of LAMMPS by simply updating your operating system.

To install LAMMPS do the following once:

```
sudo apt-get install lammps
```

This downloads an executable named `lmp` to your box and multiple packages with supporting data, examples and libraries as well as any missing dependencies. For example, the LAMMPS binary in this package is built with the *KIM package* enabled, which results in the above command also installing the `kim-api` binaries when LAMMPS is installed, unless they were installed already. In order to use potentials from openkim.org, you can also install the `openkim-models` package:

```
sudo apt-get install openkim-models
```

Or use the KIM-API commands to download and install individual models.

This LAMMPS executable can then be used in the usual way to run input scripts:

```
lmp -in in.lj
```

To update LAMMPS to the latest packaged version, do the following:

```
sudo apt-get update
```

This will also update other packages on your system.

To uninstall LAMMPS, do the following:

```
sudo apt-get remove lammps
```

Please use `lmp -help` to see which compilation options, packages, and styles are included in the binary.

Thanks to Anton Gladky (gladky.anton at gmail.com) for setting up this Ubuntu package capability.

---

### 2.1.3 Pre-built Fedora Linux executables

Pre-built LAMMPS packages for stable releases are available in the Fedora Linux distribution since Fedora version 28. The packages can be installed via the dnf package manager. There are 3 basic varieties (lammps = no MPI, lammps-mpich = MPICH MPI library, lammps-openmpi = OpenMPI MPI library) and for each support for linking to the C library interface (lammps-devel, lammps-mpich-devel, lammps-openmpi-devel), the header for compiling programs using the C library interface (lammps-headers), and the LAMMPS python module for Python 3. All packages can be installed at the same time and the name of the LAMMPS executable is `lmp` and `lmp_openmpi` or `lmp_mpich` respectively. By default, `lmp` will refer to the serial executable, unless one of the MPI environment modules is loaded (`module load mpi/mpich-x86_64` or `module load mpi/openmpi-x86_64`). Then the corresponding parallel LAMMPS executable can be used. The same mechanism applies when loading the LAMMPS python module.

To install LAMMPS with OpenMPI and run an input `in.lj` with 2 CPUs do:

```
dnf install lammps-openmpi
module load mpi/openmpi-x86_64
mpirun -np 2 lmp -in in.lj
```

The `dnf install` command is needed only once. In case of a new LAMMPS stable release, `dnf update` will automatically update to the newer version as soon as the RPM files are built and uploaded to the download mirrors. The `module load` command is needed once per (shell) session or shell terminal instance, unless it is automatically loaded from the shell profile.

The LAMMPS binary is built with the *KIM package* which results in the above command also installing the *kim-api* binaries when LAMMPS is installed. In order to use potentials from openkim.org, you can install the *openkim-models* package

```
dnf install openkim-models
```

Please use `lmp -help` to see which compilation options, packages, and styles are included in the binary.

Thanks to Christoph Junghans (LANL) for making LAMMPS available in Fedora.

---

### 2.1.4 Pre-built EPEL Linux executable

Pre-built LAMMPS (and KIM) packages for stable releases are available in the Extra Packages for Enterprise Linux (EPEL) repository for use with Red Hat Enterprise Linux (RHEL) or CentOS version 7.x and compatible Linux distributions. Names of packages, executable, and content are the same as described above for Fedora Linux. But RHEL/CentOS 7.x uses the `yum` package manager instead of `dnf` in Fedora 28.

Please use `lmp -help` to see which compilation options, packages, and styles are included in the binary.

Thanks to Christoph Junghans (LANL) for making LAMMPS available in EPEL.

---

### 2.1.5 Pre-built OpenSuse Linux executable

A pre-built LAMMPS package for stable releases is available in OpenSuse as of Leap 15.0. You can install the package with:

```
zypper install lammps
```

This includes support for OpenMPI. The name of the LAMMPS executable is `lmp`. To run an input in parallel on 2 CPUs you would do:

```
mpirun -np 2 lmp -in in.lj
```

Please use `lmp -help` to see which compilation options, packages, and styles are included in the binary.

The LAMMPS binary is built with the *KIM package* which results in the above command also installing the *kim-api* binaries when LAMMPS is installed. In order to use potentials from openkim.org, you can install the *openkim-models* package

```
zypper install openkim-models
```

Thanks to Christoph Junghans (LANL) for making LAMMPS available in OpenSuse.

---

### 2.1.6 Gentoo Linux executable

LAMMPS is part of Gentoo's main package tree and can be installed by typing:

```
emerge --ask lammps
```

Note that in Gentoo the LAMMPS source code is downloaded and the package is then compiled and installed on your machine.

Certain LAMMPS packages can be enabled via USE flags, type

```
equery uses lammps
```

for details.

Thanks to Nicolas Bock and Christoph Junghans (LANL) for setting up this Gentoo capability.

---

### 2.1.7 Archlinux build-script

LAMMPS is available via Arch's unofficial Arch User repository (AUR). There are three scripts available, named lammps, lammps-beta and lammps-git. They respectively package the stable, feature, and git releases.

To install, you will need to have the git package installed. You may use any of the above names in-place of lammps.

```
git clone https://aur.archlinux.org/lammps.git
cd lammps
makepkg -s
makepkg -i
```

To update LAMMPS, you may repeat the above, or change into the cloned directory, and execute the following, after which, if there are any changes, you may use makepkg as above.

```
git pull
```

Alternatively, you may use an AUR helper to install these packages.

Note that the AUR provides build-scripts that download the source code and then build and install the package on your machine.

## 2.2 Download an executable for macOS

LAMMPS can be downloaded, built, and configured for macOS with Homebrew. (Alternatively, see the installation instructions for *downloading an executable via Conda*.) The following LAMMPS packages are unavailable at this time because of additional requirements not yet met: GPU, KOKKOS.

After installing Homebrew, you can install LAMMPS on your system with the following commands:

```
brew install lammps
```

This will install the executables "lammps_serial" and "lammps_mpi", as well as the LAMMPS "doc", "potentials", "tools", "bench", and "examples" directories.

Once LAMMPS is installed, you can test the installation with the Lennard-Jones benchmark file:

```
brew test lammps -v
```

The LAMMPS binary is built with the *KIM package*, which results in Homebrew also installing the *kim-api* binaries when LAMMPS is installed. In order to use potentials from openkim.org, you can install the *openkim-models* package

```
brew install openkim-models
```

If you have problems with the installation, you can post issues to this link.

Thanks to Derek Thomas (derekt at cello.t.u-tokyo.ac.jp) for setting up the Homebrew capability.

## 2.3 Download an executable for Windows

Pre-compiled Windows installers which install LAMMPS executables on a Windows system can be downloaded from this site:

`https://packages.lammps.org/windows.html`

Note that each installer package has a date in its name, which corresponds to the LAMMPS version of the same date. Installers for current and older versions of LAMMPS are available. 32-bit and 64-bit installers are available, and each installer contains both a serial and parallel executable. The installer website also explains how to install the Windows MPI package (MPICH2 from Argonne National Labs), needed to run in parallel with MPI.

The LAMMPS binaries contain *all optional packages* included in the source distribution except: ADIOS, H5MD, KIM, ML-PACE, ML-QUIP, MSCG, NETCDF, QMMM, SCAFACOS, and VTK. The serial version also does not include the LATBOLTZ package. The PYTHON package is only available in the Python installers that bundle a Python runtime. The GPU package is compiled for OpenCL with mixed precision kernels.

The LAMMPS library is compiled as a shared library and the *LAMMPS Python module* is installed, so that it is possible to load LAMMPS into a Python interpreter.

The installer site also has instructions on how to run LAMMPS under Windows, once it is installed, in both serial and parallel.

When you download the installer package, you run it on your Windows machine. It will then prompt you with a dialog, where you can choose the installation directory, unpack and copy several executables, potential files, documentation PDFs, selected example files, etc. It will then update a few system settings (e.g. `PATH`, `LAMMPS_POTENTIALS`) and add an entry into the Start Menu (with references to the documentation, LAMMPS homepage and more). From that menu, there is also a link to an uninstaller that removes the files and undoes the environment manipulations.

Note that to update to a newer version of LAMMPS, you should typically uninstall the version you currently have, download a new installer, and go through the installation procedure described above. I.e. the same procedure for installing/updating most Windows programs. You can install multiple versions of LAMMPS (in different directories),

but only the executable for the last-installed package will be found automatically, so this should only be done for debugging purposes.

## 2.4 Download an executable for Linux or macOS via Conda

Pre-compiled LAMMPS binaries are available for macOS and Linux via the Conda package management system.

First, one must set up the Conda package manager on your system. Follow the instructions to install Miniconda, then create a conda environment (named *my-lammps-env* or whatever you prefer) for your LAMMPS install:

```
conda config --add channels conda-forge
conda create -n my-lammps-env
```

Then, you can install LAMMPS on your system with the following command:

```
conda activate my-lammps-env
conda install lammps
```

The LAMMPS binary is built with the *KIM package*, which results in Conda also installing the *kim-api* binaries when LAMMPS is installed. In order to use potentials from openkim.org, you can install the *openkim-models* package

```
conda install openkim-models
```

If you have problems with the installation, you can post issues to this link. Thanks to Jan Janssen (Max-Planck-Institut fuer Eisenforschung) for setting up the Conda capability.

**Note**

If you have questions about these pre-compiled LAMMPS executables, you need to contact the people preparing those packages. The LAMMPS developers have no control over their choices of how they configure and build their packages and when they update them.

## 2.5 Download source and documentation as a tarball

You can download a current LAMMPS tarball from the download page of the LAMMPS website or from GitHub (see below).

You have two choices of tarballs, either the most recent stable release or the most recent feature release. Stable releases occur a few times per year, and undergo more testing before release. Also, between stable releases bug fixes from the feature releases are back-ported and the tarball occasionally updated. Feature releases occur every 4 to 8 weeks. The new contents in all feature releases are listed on the bug and feature page of the LAMMPS homepage.

Tarballs of older LAMMPS versions can also be downloaded from this page.

Tarballs downloaded from the LAMMPS homepage include the pre-translated LAMMPS documentation (HTML and PDF files) corresponding to that version.

Once you have a tarball, uncompress and untar it with the following command:

```
tar -xzvf lammps*.tar.gz
```

This will create a LAMMPS directory with the version date in its name, e.g. `lammps-28Mar23`.

You can also download a compressed tar or zip archives from the "Assets" sections of the LAMMPS GitHub releases site. The file name will be lammps-<version>.zip which can be unzipped with the following command, to create a lammps-<version> directory:

```
unzip lammps*.zip
```

This version corresponds to the selected LAMMPS feature or stable release (as indicated by the matching git tag) and will only contain the source code and no pre-built documentation.

## 2.6 Download the LAMMPS source with git

LAMMPS development is coordinated through the "LAMMPS GitHub site". If you clone the LAMMPS repository onto your local machine, it has several advantages:

- You can stay current with changes to LAMMPS with a single git command.
- You can create your own development branches to add code to LAMMPS.
- You can submit your new features back to GitHub for inclusion in LAMMPS. For that, you should first create your own *fork on GitHub*, though.

You must have git installed on your system to use the commands explained below to communicate with the git servers on GitHub.

You can follow the LAMMPS development on 4 different git branches:

- **develop** : this branch follows the ongoing development and is updated with every merge commit of a pull request
- **release** : this branch is updated with every "feature release" and updates are always "fast-forward" merges from *develop*
- **maintenance** : this branch collects back-ported bug fixes from the *develop* branch to the *stable* branch. It is used to update the *stable* branch for "stable update releases".
- **stable** : this branch is updated from the *release* branch with every "stable release" version and also has selected bug fixes with every "update release" when the *maintenance* branch is merged into it

To access the git repository on your box, use the clone command to create a local copy of the LAMMPS repository with a command like:

```
git clone -b release https://github.com/lammps/lammps.git mylammps
```

where "mylammps" is the name of the directory you wish to create on your machine and "release" is one of the 3 branches listed above. (Note that you actually download all 3 branches; you can switch between them at any time using "git checkout <branch name>".)

**Saving time and disk space when using `git clone`**

The complete git history of the LAMMPS project is quite large because it contains the entire commit history of the project since fall 2006, which includes the time when LAMMPS was managed with subversion. This includes a few commits that have added and removed some large files (mostly by accident). If you do not need access to the entire commit history (most people don't), you can speed up the "cloning" process and reduce local disk space requirements by using the `--depth` git command-line flag. That will create a "shallow clone" of the repository, which contains only a subset of the git history. Using a depth of 1000 is usually sufficient to include the head commits of the *develop*, the *release*, and the *maintenance* branches. To include the head commit of the *stable* branch you may need a depth of up to 10000. If you later need more of the git history, you can always convert the shallow clone into a "full clone".

Once the command completes, your directory will contain the same files as if you unpacked a current LAMMPS tarball, with the exception, that the HTML documentation files are not included. They can be generated from the content provided in `doc/src` by typing `make html` from the `doc` directory.

After initial cloning, as bug fixes and new features are added to LAMMPS you can stay up-to-date by typing the following git commands from within the "mylammps" directory:

```
git checkout release      # not needed if you always stay in this branch
git checkout stable       # use one of these 4 checkout commands
git checkout develop      # to choose the branch to follow
git checkout maintenance
git pull
```

Doing a "pull" will not change any files you have added to the LAMMPS directory structure. It will also not change any existing LAMMPS files you have edited, unless those files have changed in the repository. In that case, git will attempt to merge the changes from the repository file with your version of the file and tell you if there are any conflicts. See the git documentation for details.

If you want to access a particular previous release version of LAMMPS, you can instead "check out" any version with a published tag. See the output of `git tag -l` for the list of tags. The git command to do this is as follows.

```
git checkout tagID
```

Stable versions and what tagID to use for a particular stable version are discussed on this page. Note that this command will print some warnings, because in order to get back to the latest revision and to be able to update with `git pull` again, you will need to do `git checkout release` (or check out any other desired branch) first.

Once you have updated your local files with a `git pull` (or `git checkout`), you still need to re-build LAMMPS if any source files have changed. How to do this depends on the build system you are using.

**CMake build**

Change to your build folder and type:

```
cmake --build .
```

CMake should auto-detect whether it needs to re-run the CMake configuration step and otherwise redo the build for all files that have been changed or files that depend on changed files. In case some build options have been changed or renamed, you may have to update those by running:

```
cmake .
```

and then rebuild.

**Traditional make**

Switch to the src directory and type:

```
make purge              # remove any deprecated src files
make package-update     # sync package files with src files
make foo                # re-build for your machine (mpi, serial, etc)
```

to enforce consistency of the source between the src folder and package directories. This is OK to do even if you don't use any packages. The `make purge` command removes any deprecated src files if they were removed by the update from a package subdirectory.

**Warning**

If you wish to edit/change a src file that is from a package, you should edit the version of the file inside the package subdirectory with src, then re-install the package. The version in the source directory is merely a copy and will be wiped out if you type "make package-update".

**Git protocols**

The servers at github.com support the "https" access protocol for anonymous, read-only access. If you have a suitably configured GitHub account, you may also use SSH protocol with the URL `git@github.com:lammps/lammps.git`.

The LAMMPS GitHub project is currently overseen by Axel Kohlmeyer (Temple U, akohlmey at gmail.com), contact him if you have any questions or concerns.

---

You can find additional LAMMPS features for dynamically loading with the *plugin command* in the LAMMPS plugins source code repository on GitHub

---

These are the files and subdirectories in the LAMMPS distribution:

| | |
|---|---|
| `README` | Short description of the LAMMPS package |
| `LICENSE` | GNU General Public License (GPL) |
| `SECURITY.md` | Security policy for the LAMMPS package |
| `bench` | benchmark inputs |
| `cmake` | CMake build files |
| `doc` | documentation and tools to build the manual |
| `examples` | example input files |
| `fortran` | Fortran module for LAMMPS library interface |
| `lib` | additional provided or external libraries |
| `potentials` | selected interatomic potential files |
| `python` | Python module for LAMMPS library interface |
| `src` | LAMMPS source files |
| `tools` | pre- and post-processing tools |
| `unittest` | source code and inputs for testing LAMMPS |

You will have all of these if you downloaded the LAMMPS source code. You will have only some of them if you downloaded executables, as explained on the pages listed above.

CHAPTER

THREE

# BUILD LAMMPS

LAMMPS is built as a library and an executable from source code using a build environment generated by CMake (Unix Makefiles, Ninja, Xcode, Visual Studio, KDevelop, CodeBlocks and more depending on the platform). Using CMake is the preferred way to build LAMMPS.

In addition, LAMMPS can be compiled using the legacy build system based on traditional makefiles for use with GNU make (which may require manual editing).

**Warning**

The LAMMPS developers plan to retire the legacy build system. Please switch to *using CMake* instead. Contact the the LAMMPS developers at developers@lammps.org if you have any concerns or questions about this.

As an alternative, you can download a package with pre-built executables or automated build trees, as described in the *Install* section of the manual.

## 3.1 Prerequisites

Which software you need to compile and use LAMMPS strongly depends on which *features and settings* and which *optional packages* you are trying to include. Common to all is that you need a C++ and C compiler, where the C++ compiler has to support at least the C++17 standard (note that some compilers require a command-line flag to activate C++17 support).

### 3.1.1 CMake build system

If you are building with CMake, you need at least CMake version 3.20 and a compatible build tool (e.g. GNU make or ninja-build on Linux). The CMake scripting includes tests for required software and will auto-detect and auto-enable available tools and libraries for optional features (of course those can be disabled, if desired). If required software is missing, CMake will try to download the compile the missing parts automatically, if possible or stop with an error.

### 3.1.2 GNU build system

If you are building LAMMPS with the legacy GNU make based build system you need GNU make (other make variants are not going to work since the build system uses features unique to GNU make) and a Unix-like build environment with a Bourne shell, and shell tools like “sed”, “grep”, “touch”, “test”, “tr”, “cp”, “mv”, “rm”, “ln”, “diff” and so on. Parts of LAMMPS interface with or use Python version 3.6 or later.

### 3.1.3 Compiler and OS compatibility

The LAMMPS developers aim to keep LAMMPS very portable and usable - at least in parts - on most operating systems commonly used for running MD simulations. Please see the *section on portability* for more details.

**Warning: LLVM based Intel Compilers**

In recent years, Intel switched their compilers from a proprietary code base to compilers based on LLVM (which is also the foundation of the Clang compilers). It has since stopped updating their legacy compilers. Some versions of those LLVM compilers creates incorrect binary code that results in incorrect behavior of LAMMPS; this is particularly common when using high optimization levels and vectorization. There have been several cases where people reported bugs in LAMMPS which turned out to be caused by bugs in the Intel LLVM compilers.

Unfortunately there is no simple way to detect whether a binary is working correctly outside of running the unit and regression tests, but those do not cover all of LAMMPS and would be reliable only for no or moderate optimization anyway. For most of LAMMPS there is not much of a benefit (if any) to use the Intel compilers over the GCC or Clang compilers, except for the INTEL package (which *can* be compiled with other compilers, but most vectorization directives are inactive for those) or KOKKOS with SYCL.

Our recommendation is thus to compile with GCC or Clang unless you are using features that *require* the Intel compilers. In that case, it is recommended to have a backup GCC/Clang compiled binary available to validate correctness of the simulation. If you have access to multiple Intel compiler versions, please try the latest version first and avoid "20##.0" versions which seem to have problems most often.

## 3.2 Build LAMMPS with CMake

This page describes how to use CMake in general to build LAMMPS. Details for specific compile time settings and options to enable and configure add-on packages are discussed with those packages. Links to those pages on the *Build overview* page.

The following text assumes some familiarity with CMake and focuses on using the command-line tool `cmake` and what settings are supported for building LAMMPS. A more detailed tutorial on how to use CMake itself, the text mode or graphical user interface, to change the generated output files for different build tools and development environments is on a *separate page*.

**Note**

LAMMPS currently requires CMake version 3.20 or later.

**Warning**

You **must not** mix the *traditional make based* LAMMPS build procedure with using CMake. No packages may be installed or a build been previously attempted in the LAMMPS source directory by using `make <machine>`. CMake will detect if this is the case and generate an error. To remove conflicting files from the `src` you can use the command `make no-all purge` which will uninstall all packages and delete all auto-generated files.

### 3.2.1 Advantages of using CMake

CMake is the preferred way of compiling LAMMPS in contrast to the legacy build system based on GNU make and through *(manually customized) makefiles*. Using CMake has multiple advantages that are specifically helpful for people with limited experience in compiling software or for people that want to modify or extend LAMMPS.

- CMake can detect available hardware, tools, features, and libraries and adapt the LAMMPS default build configuration accordingly.
- CMake can generate files for different build tools and integrated development environments (IDE).
- CMake supports customization of settings with a command-line, text mode, or graphical user interface. No manual editing of files, knowledge of file formats or complex command-line syntax is required.
- All enabled components are compiled in a single build operation.
- Automated dependency tracking for all files and configuration options.
- Support for true out-of-source compilation. Multiple configurations and settings with different choices of LAMMPS packages, settings, or compilers can be configured and built concurrently from the same source tree.
- Simplified packaging of LAMMPS for Linux distributions, environment modules, or automated build tools like Spack or Homebrew.
- Integration of automated unit and regression testing.

### 3.2.2 Getting started

Building LAMMPS with CMake is a two-step process. In the first step, you use CMake to generate a build environment in a new directory. For that purpose you can use either the command-line utility `cmake` (or `cmake3`), the text-mode UI utility `ccmake` (or `ccmake3`) or the graphical utility `cmake-gui`, or use them interchangeably. The second step is then the compilation and linking of all objects, libraries, and executables using the selected build tool. Here is a minimal example using the command-line version of CMake to build LAMMPS with no add-on packages enabled and no customization:

```
cd lammps                 # change to the LAMMPS source distribution directory
cmake -S cmake -B build   # configure the "build" folder with CMake scripts from "cmake"
cmake --build build       # compilation (or type "make -C build")
```

This will create a folder called `build`, then run the configuration step to generate build files for the default build command and then launch that build command to compile LAMMPS. During the configuration step CMake will try to detect whether support for MPI, OpenMP, FFTW, gzip, JPEG, PNG, and ffmpeg are available and enable the corresponding configuration settings. The progress of this configuration can be followed on the screen and a summary of selected options and settings will be printed at the end. The `cmake --build build` command will launch the compilation, which, if successful, will ultimately produce a library `liblammps.a` and the LAMMPS executable `lmp` inside the `build` folder.

Compilation can take a long time, since LAMMPS is a large project with many features. If your machine has multiple CPU cores (most do these days), you can speed this up by compiling sources in parallel with adding `--parallel N` to the `cmake` command line (with *N* being the maximum number of concurrently executed tasks). Installation of the ccache (= Compiler Cache) software may speed up repeated compilation even more, e.g. during code development, especially when repeatedly switching between branches.

After the initial build, whenever you edit LAMMPS source files, enable or disable packages, change compiler flags or build options, you must re-compile and relink the LAMMPS executable with `cmake --build build` (or `make -C build`). If the compilation fails for some reason, try running `cmake build` and then compile again. The included dependency tracking should make certain that only the necessary subset of files is re-compiled. You can also delete compiled objects, libraries, and executables with `cmake --build build --target clean` (or `make -C build clean`).

After compilation, you may optionally install the LAMMPS executable into your system with:

```
cmake --install build    # optional, copy compiled files into installation location
```

This will install the LAMMPS executable and library, some tools (if configured) and additional files like LAMMPS API headers, manpages, potential and force field files. The location of the installation tree defaults to `${HOME}/.local`.

**Note**

If you have set *-D CMAKE_INSTALL_PREFIX* to install LAMMPS into a system location on a Linux machine, you also have to run (as root) the *ldconfig* program to update the cache file for fast lookup of system shared libraries.

**Using the installed library**

The CMake installation functionality is an experimental work in progress and thus not without problems, especially when writing your own program that is trying to use the LAMMPS C++ classes directly.

While there is a well-defined *C-language interface* with the `library.h` header file, there is no equivalent for the C++ interface yet. When installing LAMMPS, only the core header files are copied into the installation folder and thus only high-level access to C++ features is available.

The following is a minimal CMake example file for using the installed LAMMPS package which represents the current state of development:

```
cmake_minimum_required(VERSION 3.20)
project(simpleCC CXX)
# set this to the LAMMPS installation location
if(NOT CMAKE_PREFIX_PATH)
  set(CMAKE_PREFIX_PATH $ENV{HOME}/.local)
endif()
find_package(LAMMPS REQUIRED)
add_executable(simpleCC simple.cpp)
target_link_libraries(simpleCC PRIVATE LAMMPS::LAMMPS)
```

The `CMAKE_PREFIX_PATH` setting tells CMake where to find the generated CMake configuration files for the find_package() CMake command. You can also specify a required minimal version or version range. For that a numeric representation in the "YYYY.MM.DD" format has to be used: the 10 September 2025 release thus becomes version 2025.09.10. The include statements in the `simple.cpp` source file have to be prefixed with `lammps/` as follows:

```
#include <lammps/lammps.h>
#include <lammps/input.h>
#include <lammps/atom.h>
#include <lammps/library.h>
```

Using the `LAMMPS::LAMMPS` target imported from the installed LAMMPS CMake configuration files should set up the include and linker flags and folders automatically. Below is the output for an example session (note how it checks for and includes MPI and OpenMP support since that specific LAMMPS library was set up this way):

```
$ cmake -S . -B build -D CMAKE_PREFIX_PATH=$HOME/Downloads/test-install
-- The CXX compiler identification is GNU 15.2.1
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/lib64/ccache/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found MPI_CXX: /usr/lib64/mpich/lib/libmpicxx.so (found version "4.1")
-- Found MPI: TRUE (found version "4.1") found components: CXX
-- Found OpenMP_CXX: -fopenmp (found version "4.5")
-- Found OpenMP: TRUE (found version "4.5") found components: CXX
-- Found LAMMPS: /home/akohlmey/Downloads/test-install/lib64/liblammps.so.0
-- Configuring done (0.9s)
-- Generating done (0.0s)
-- Build files have been written to: /home/akohlmey/Downloads/test-simple/build
$ cmake --build build
[ 50%] Building CXX object CMakeFiles/simpleCC.dir/simple.cpp.o
[100%] Linking CXX executable simpleCC
[100%] Built target simpleCC
```

### 3.2.3 Configuration and build options

The CMake commands have one mandatory argument: a folder containing a file called `CMakeLists.txt` (for LAMMPS it is located in the `cmake` folder, in that case the current working directory becomes the build folder) or a build folder containing a file called `CMakeCache.txt`, which is generated at the end of the CMake configuration step. The cache file contains all current CMake settings. This is a "legacy mode" of running CMake and thus often found when searching the web. We recommend to use the `-S` and `-B` folders to explicitly set the path to the folder containing the `CMakeLists.txt` file and the build folder, respectively.

To modify settings, enable or disable features, you need to set *variables* with either the `-D` command-line flag (`-D VARIABLE1_NAME=value`) or change them in the text mode of the graphical user interface. The `-D` flag can be used several times in one command.

For your convenience, we provide *CMake presets* that combine multiple settings to enable optional LAMMPS packages or use a different compiler tool chain. Those are loaded with the `-C` flag (`-C ../cmake/presets/basic.cmake`). This step would only be needed once, as the settings from the preset files are stored in the `CMakeCache.txt` file. It is also possible to customize the build by adding one or more `-D` flags to the CMake command.

Generating files for alternate build tools (e.g. Ninja) and project files for IDEs like Eclipse, CodeBlocks, or Kate can be selected using the `-G` command-line flag. A list of available generator settings for your specific CMake version is given when running `cmake --help`.

### 3.2.4 Build targets

The LAMMPS CMake files are set up to have multiple "targets", one for each component that is set up to be built. If you do not provide an explicit target when building, the target "all" is assumed. This should build all configured targets and include re-running CMake to update the build files when files have changed or new files were added which could require updates in the build process. The most important individual targets are "lammps" to build the LAMMPS library and "lmp" to build the LAMMPS executable. If you want to selectively want to update only one of them you can use the command:

```
cmake --build build --target lmp
```

Especially, when programming LAMMPS and you are making only local changes, or only want to make certain that the code still compiles and links, you may want to skip many of the extra steps that are run by default and then append "/fast" to the target. Example:

```
cmake --build build --target lammps/fast
```

Note that this speeds up the build by avoiding to re-run CMake and skipping a lot of checks for possible dependencies, so you may occasionally need to compile without the "/fast" suffix or use the "all" (= default) target to update everything. This would be *required* when adding new files, additional dependencies, or making changes to the CMake scripts.

Some custom targets, e.g. "install-python" are explicitly excluded from the "all" target and must be built individually.

### 3.2.5 Multi-configuration build systems

Throughout this manual, it is mostly assumed that LAMMPS is being built on a Unix-like operating system with "make" as the underlying "builder", since this is the most common case. In this case the build "configuration" is chosen using `-D CMAKE_BUILD_TYPE=<configuration>` with `<configuration>` being one of "Release", "Debug", "RelWithDebInfo", or "MinSizeRel". Some build tools, however, can also use or even require having a so-called multi-configuration build system setup. For a multi-configuration build, the built type (or configuration) is selected at compile time using the same build files. E.g. with:

```
cmake --build build-multi --config Release
```

In that case the resulting binaries are not in the build folder directly but in subdirectories corresponding to the build type (i.e. Release in the example from above). Similarly, for running unit tests the configuration is selected with the `-C` flag:

```
ctest -C Debug
```

The CMake scripts in LAMMPS have basic support for being compiled using a multi-config build system, but not all of it has been ported. This is in particular applicable to compiling packages that require additional libraries that would be downloaded and compiled by CMake. The `windows.cmake` preset file tries to keep track of which packages can be compiled natively with the MSVC compilers out-of-the box. Not all of the external libraries are portable to Windows, either.

### 3.2.6 Installing CMake

Check if your machine already has CMake installed:

```
which cmake             # do you have it?
which cmake3            # version 3 may have this name
cmake --version         # what specific version you have
```

On clusters or supercomputers which use environment modules to manage software packages, do this:

```
module list            # is a module for cmake already loaded?
module avail           # is a module for cmake available?
module load cmake      # load cmake module with appropriate name
```

Most Linux distributions offer pre-compiled cmake packages through their package management system. If you do not have CMake or a recent enough version (Note: for CentOS 7.x you need to enable the EPEL repository), you can download the latest version from https://cmake.org/download/. Links to more details on CMake can be found on this page.

## 3.3 Build LAMMPS with make

**Warning**

The LAMMPS developers plan to retire the legacy build system. Please switch to *using CMake* instead. There is also a detailed discussion of *how to use CMake with LAMMPS* and the CMake online documentation. For the impatient, there is also a quick transition guide below. Contact the the LAMMPS developers at developers@lammps.org if you have any concerns or questions about this.

### 3.3.1 Make to CMake quick-start guide

| Action | Traditional `make` command in (`src/`) | CMake equivalent in (main folder) |
|---|---|---|
| Initial Setup | None or Edit `Makefile.<machine>` | `cmake -S cmake -B build` |
| Build | `make <machine>` | `cmake --build build` |
| Parallel build | `make -j 8 <machine>` | `cmake --build build --parallel 8` |
| Enable package | `make yes-<package name>` | `cmake build -D PKG_<PACKAGE_NAME>=yes` |
| Disable package | `make no-<package name>` | `cmake build -D PKG_<PACKAGE_NAME>=no` |
| Load a preset configuration | Choose a `Makefile.<machine>` from `MAKE` folders | `cmake build -C cmake/presets/<preset>.cmake` |
| List enabled packages | `make ps` (or `make package-status`) | Check CMake output or use `ccmake build` |
| Change configuration settings | Copy `Makefile.<machine>` to MAKE/MINE and edit | Rerun CMake with changed options |
| Clean build files | `make clean-<machine>` | `cmake --build build --target clean` |
| Full reset | `make no-all clean-all purge` | `rm -rf build` |
| Enable or disable MPI | Use `make mpi` or `make serial` | Auto-detected, override with `-D BUILD_MPI=no` |
| Enable or disable OpenMP | Edit `Makefile.<machine>` | Auto-detected, override with `-D BUILD_OMP=no` |
| Build LAMMPS shared library | `make mode=shared <machine>` | Use `-D BUILD_SHARED_LIBS=yes` |

Some general CMake tips:

- You can have multiple independent build folders with different configurations (e.g. MPI on/off or different package selections). The convention is that the name starts with "build" (e.g. `build-serial`)
- To build a LAMMPS executable with defaults and most packages that do not need manual configurations: `cmake -S cmake -B build -C cmake/presets/most.cmake; cmake --build build`
- To make multiple configuration changes interactively use either `ccmake build` or edit `build/CMakeCache.txt` in a text editor and run `cmake build`

### 3.3.2 Basics

Building LAMMPS with traditional makefiles requires that you have a `Makefile.<machine>` file appropriate for your system in either the `src/MAKE`, `src/MAKE/MACHINES`, `src/MAKE/OPTIONS`, or `src/MAKE/MINE` directory (see below). It can include various options for customizing your LAMMPS build with a number of global compilation options and features.

This build system is slowly being phased out and does no longer support all optional features and packages in LAMMPS. It is recommended to switch to the *CMake based build system*.

### 3.3.3 Requirements

Those makefiles are written for and tested with GNU make and may not be compatible with other make programs. In most cases, if the "make" program is not GNU make, then there will be a GNU make program available under the name "gmake". If GNU make or a compatible make is not available, you may have to first install it or switch to building with *CMake*. The makefiles of the traditional make based build process and the scripts they are calling expect a few additional tools to be available and functioning.

- A working C/C++ compiler tool chain supporting the C++17 standard; on Linux, these are often the GNU compilers. Some older compiler versions require adding flags like `-std=c++17` to enable C++17 mode.
- A Bourne shell compatible "Unix" shell program (frequently this is `bash`)
- A few shell utilities: `ls`, `mv`, `ln`, `rm`, `grep`, `sed`, `tr`, `cat`, `touch`, `diff`, `dirname`
- Python (optional, required for `make lib-<pkg>` in the `src` folder). Python scripts are currently tested with 3.6 to 3.11. The procedure for *building the documentation* *requires* Python 3.8 or later.

### 3.3.4 Getting started

To include LAMMPS packages (i.e. optional commands and styles) you must enable (or "install") them first, as discussed on the *Build package* page. If a package requires (provided or external) libraries, you must configure and build those libraries **before** building LAMMPS itself and especially **before** enabling such a package with `make yes-<package>`. *Building LAMMPS with CMake* can automate much of this for many types of machines, especially workstations, desktops, and laptops, so we suggest you try it first when building LAMMPS in those cases.

The commands below perform a default LAMMPS build, producing the LAMMPS executable `lmp_serial` and `lmp_mpi` in `lammps/src`:

```
cd lammps/src   # change to main LAMMPS source folder
make serial     # build a serial LAMMPS executable using GNU g++
make mpi        # build a parallel LAMMPS executable with MPI
make            # see a variety of make options
```

Compilation can take a long time, since LAMMPS is a large project with many features. If your machine has multiple CPU cores (most do these days), you can speed this up by compiling sources in parallel with `make -j N` (with N being the maximum number of concurrently executed tasks). Installation of the ccache (= Compiler Cache) software may speed up repeated compilation even more, e.g. during code development, especially when repeatedly switching between branches.

After the initial build, whenever you edit LAMMPS source files, or add or remove new files to the source directory (e.g. by installing or uninstalling packages), you must re-compile and relink the LAMMPS executable with the same `make <machine>` command. The makefile's dependency tracking should ensure that only the necessary subset of files is re-compiled. If you change settings in the makefile, you have to recompile *everything*. To delete all objects, you can use `make clean-<machine>`.

> **Note**
>
> Before the actual compilation starts, LAMMPS will perform several steps to collect information from the configuration and setup that is then embedded into the executable. When you build LAMMPS for the first time, it will also compile a tool to quickly determine a list of dependencies. Those are required for the make program to correctly detect, which files need to be recompiled or relinked after changes were made to the sources.

### 3.3.5 Customized builds and alternate makefiles

The `src/MAKE` directory tree contains the `Makefile.<machine>` files included in the LAMMPS distribution. Typing `make example` uses `Makefile.example` from one of those folders, if available. The `make serial` and `make mpi` lines above, for example, use `src/MAKE/Makefile.serial` and `src/MAKE/Makefile.mpi`, respectively. Other makefiles are in these directories:

```
OPTIONS      # Makefiles which enable specific options
MACHINES     # Makefiles for specific machines
MINE         # customized Makefiles you create (you may need to create this folder)
```

Simply typing `make` lists all the available `Makefile.<machine>` files with a single line description toward the end of the output. A file with the same name can appear in multiple folders (not a good idea). The order the directories are searched is as follows: `src/MAKE/MINE`, `src/MAKE`, `src/MAKE/OPTIONS`, `src/MAKE/MACHINES`. This gives preference to a customized file you put in `src/MAKE/MINE`. If you create your own custom makefile under a new name, please edit the first line with the description and machine name, so you will not confuse yourself, when looking at the machine summary.

Makefiles you may wish to try out, include those listed below (some require a package first be installed). Many of these include specific compiler flags for optimized performance. Please note, however, that some of these customized machine Makefile are contributed by users, and thus may have modifications specific to the systems of those users. Since compilers, OS configurations, and LAMMPS itself keep changing, their settings may become outdated, too:

```
make mac              # build serial LAMMPS on macOS
make mac_mpi          # build parallel LAMMPS on macOS
make intel_cpu        # build with the INTEL package optimized for CPUs
make opt              # build with the OPT package optimized for CPUs
make omp              # build with the OPENMP package optimized for OpenMP
```

## 3.4 Link LAMMPS as a library to another code

LAMMPS is designed as a library of C++ objects that can be integrated into other applications, including Python scripts. The files `src/library.cpp` and `src/library.h` define a C-style API for using LAMMPS as a library. See the *Library interface to LAMMPS* page for a description of the interface and how to use it for your needs.

The *Basic build options* page explains how to build LAMMPS as either a shared or static library. This results in a file in the compilation folder called `liblammps.a` or `liblammps_<name>.a` in case of building a static library. In case of a shared library, the name is the same only that the suffix is going to be either `.so` or `.dylib` or `.dll` instead of `.a` depending on the OS. In some cases, the `.so` file may be a symbolic link to a file with the suffix `.so.0` (or some other number).

> **Note**
>
> Care should be taken to use the same MPI library for the calling code and the LAMMPS library, unless LAMMPS is to be compiled without (real) MPI support using the included STUBS MPI library.

### 3.4.1 Link with LAMMPS as a static library

The calling application can link to LAMMPS as a static library with compilation and link commands, as in the examples shown below. These are examples for a code written in C in the file `caller.c`. The benefit of linking to a static library is, that the resulting executable is independent of that library since all required executable code from the library is copied into the calling executable.

**CMake build**

This assumes that LAMMPS has been configured without setting a `LAMMPS_MACHINE` name, installed with `make install`, and the `PKG_CONFIG_PATH` environment variable has been updated to include the `liblammps.pc` file installed into the configured destination folder. The commands to compile and link a coupled executable are then:

```
mpicc -c -O $(pkg-config --cflags liblammps) caller.c
mpicxx -o caller caller.o -$(pkg-config --libs liblammps)
```

**Traditional make**

This assumes that LAMMPS has been compiled in the folder `${HOME}/lammps/src` with "make mpi". The commands to compile and link a coupled executable are then:

```
mpicc -c -O -I${HOME}/lammps/src caller.c
mpicxx -o caller caller.o -L${HOME}/lammps/src -llammps_mpi
```

The `-I` argument is the path to the location of the `library.h` header file containing the interface to the LAMMPS C-style library interface. The `-L` argument is the path to where the `liblammps_mpi.a` file is located. The `-llammps_mpi` argument is shorthand for telling the compiler to link the file `liblammps_mpi.a`. If LAMMPS has been built as a shared library, then the linker will use `liblammps_mpi.so` instead. If both files are available, the linker will usually prefer the shared library. In case of a shared library, you may need to update the `LD_LIBRARY_PATH` environment variable or running the `caller` executable will fail since it cannot find the shared library at runtime.

However, it is only as simple as shown above for the case of a plain LAMMPS library without any optional packages that depend on libraries (bundled or external) or when using a shared library. Otherwise, you need to include all flags, libraries, and paths for the coupled executable, that are also required to link the LAMMPS executable.

**CMake build**

When using CMake, additional libraries with sources in the lib folder are built, but not included in `liblammps.a` and (currently) not installed with `make install` and not included in the `pkgconfig` configuration file. They can be found in the top level build folder, but you have to determine the necessary link flags manually. It is therefore recommended to either use the traditional make procedure to build and link with a static library or build and link with a shared library instead.

**Traditional make**

Changed in version 10Sep2025.

The traditional make build process no longer supports building packages that require extra build steps in the `lib` folder.

### 3.4.2 Link with LAMMPS as a shared library

When linking to LAMMPS built as a shared library, the situation becomes much simpler, as all dependent libraries and objects are either included in the shared library or registered as a dependent library in the shared library file. Thus, those libraries need not be specified when linking the calling executable. Only the `-I` flags are needed. So the example case from above of the serial version static LAMMPS library with the POEMS package installed becomes:

**CMake build**

The commands with a shared LAMMPS library compiled with the CMake build process are the same as for the static library.

```
mpicc -c -O $(pkg-config --cflags liblammps) caller.c
mpicxx -o caller caller.o -$(pkg-config --libs liblammps)
```

**Traditional make**

The commands with a shared LAMMPS library compiled with the traditional make build using `make mode=shared serial` becomes:

```
gcc -c -O -I${HOME}/lammps/src caller.c
g++ -o caller caller.o -L${HOME}/lammps/src -llammps_serial
```

#### Locating liblammps.so at runtime

Unlike with a static link, now the `liblammps.so` file is required at runtime and needs to be in a folder, where the shared linker program of the operating system can find it. This would be either a folder like `/usr/local/lib64` or `${HOME}/.local/lib64` or a folder pointed to by the `LD_LIBRARY_PATH` environment variable. You can type

```
printenv LD_LIBRARY_PATH
```

to see what directories are in that list.

Or you can add the LAMMPS src directory or the directory you performed a CMake style build in to your `LD_LIBRARY_PATH` environment variable, so that the current version of the shared library is always available to programs that use it.

For the Bourne or Korn shells (/bin/sh, /bin/ksh, /bin/bash etc.), you would add something like this to your `${HOME}/.profile` file:

```
LD_LIBRARY_PATH=${LD_LIBRARY_PATH-/usr/lib64}:${HOME}/lammps/src
export LD_LIBRARY_PATH
```

For the csh or tcsh shells, you would equivalently add something like this to your `${HOME}/.cshrc` file:

```
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:${HOME}/lammps/src
```

You can verify whether all required shared libraries are found with the `ldd` tool. Example:

```
LD_LIBRARY_PATH=/home/user/lammps/src ldd caller
     linux-vdso.so.1 (0x00007ffe729e0000)
     liblammps.so => /home/user/lammps/src/liblammps.so (0x00007fc91bb9e000)
     libstdc++.so.6 => /lib64/libstdc++.so.6 (0x00007fc91b984000)
     libm.so.6 => /lib64/libm.so.6 (0x00007fc91b83e000)
```

(continues on next page)

(continued from previous page)

```
    libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007fc91b824000)
    libc.so.6 => /lib64/libc.so.6 (0x00007fc91b65b000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fc91c094000)
```

If a required library is missing, you would get a 'not found' entry:

```
ldd caller
    linux-vdso.so.1 (0x00007ffd672fe000)
    liblammps.so => not found
    libstdc++.so.6 => /usr/lib64/libstdc++.so.6 (0x00007fb7c7e86000)
    libm.so.6 => /usr/lib64/libm.so.6 (0x00007fb7c7d40000)
    libgcc_s.so.1 => /usr/lib64/libgcc_s.so.1 (0x00007fb7c7d26000)
    libc.so.6 => /usr/lib64/libc.so.6 (0x00007fb7c7b5d000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fb7c80a2000)
```

## 3.5 Basic build options

The following topics are covered on this page, for building with both CMake and make:

- *Serial vs parallel build*
- *Choice of compiler and compile/link options*
- *Build the LAMMPS executable and library*
- *Including and removing debug support*
- *Install LAMMPS after a build*

---

### 3.5.1 Serial vs parallel build

LAMMPS is written to use the ubiquitous MPI (Message Passing Interface) library API for distributed memory parallel computation. You need to have such a library installed for building and running LAMMPS in parallel using a domain decomposition parallelization. It is compatible with the MPI standard version 2.x and later. LAMMPS can also be built into a "serial" executable for use with a single processor using the bundled MPI STUBS library.

Independent of the distributed memory MPI parallelization, parts of LAMMPS are also written with support for shared memory parallelization using the OpenMP threading standard. A more detailed discussion of that is below.

**CMake build**

```
-D BUILD_MPI=value          # yes or no, default is yes if CMake finds MPI
-D BUILD_OMP=value          # yes or no, default is yes if a compatible
                            # compiler is detected
-D LAMMPS_MACHINE=name      # name = mpi, serial, mybox, titan, laptop, etc
                            # no default value
```

The executable created by CMake (after running make) is named `lmp` unless the `LAMMPS_MACHINE` option is set. When setting `LAMMPS_MACHINE=name`, the executable will be called `lmp_name`. Using `BUILD_MPI=no` will enforce building a serial executable using the MPI STUBS library.

**Traditional make**

The build with traditional makefiles has to be done inside the source folder `src`.

```
make mpi      # parallel build, produces lmp_mpi using Makefile.mpi
make serial   # serial build, produces lmp_serial using Makefile/serial
make mybox    # uses Makefile.mybox to produce lmp_mybox
```

Any `make machine` command will look up the make settings from a file `Makefile.machine` in the folder `src/MAKE` or one of its subdirectories `MINE`, `MACHINES`, or `OPTIONS`, create a folder `Obj_machine` with all objects and generated files and an executable called `lmp_machine`. The standard parallel build with `make mpi` assumes a standard MPI installation with MPI compiler wrappers where all necessary compiler and linker flags to get access and link with the suitable MPI headers and libraries are set by the wrapper programs. For other cases or the serial build, you have to adjust the make file variables `MPI_INC`, `MPI_PATH`, `MPI_LIB` as well as `CC` and `LINK`. To enable OpenMP threading usually a compiler specific flag needs to be added to the compile and link commands. For the GNU compilers, this is `-fopenmp`, which can be added to the `CC` and `LINK` makefile variables.

For the serial build the following make variables are set (see `src/MAKE/Makefile.serial`):

```
CC =        g++
LINK =      g++
MPI_INC =   -I../STUBS
MPI_PATH =  -L../STUBS
MPI_LIB =   -lmpi_stubs
```

You also need to build the STUBS library for your platform before making LAMMPS itself. A `make serial` build does this for you automatically, otherwise, type `make mpi-stubs` from the src directory, or `make` from the `src/STUBS` dir. If the build fails, you may need to edit the `STUBS/Makefile` for your platform. The stubs library does not provide MPI/IO functions required by some LAMMPS packages, e.g. `LATBOLTZ`, and thus is not compatible with those packages.

**Note**

The file `src/STUBS/mpi.cpp` provides a CPU timer function called `MPI_Wtime()` that calls `gettimeofday()`. If your operating system does not support `gettimeofday()`, you will need to insert code to call another timer. Note that the ANSI-standard function `clock()` rolls over after an hour or so, and is therefore insufficient for timing long LAMMPS simulations.

### MPI and OpenMP support in LAMMPS

If you are installing MPI yourself to build a parallel LAMMPS executable, we recommend either MPICH or OpenMPI, which are regularly used and tested with LAMMPS by the LAMMPS developers. MPICH can be downloaded from the MPICH home page, and OpenMPI can be downloaded correspondingly from the OpenMPI home page. Other MPI packages should also work. No specific vendor provided and standard compliant MPI library is currently known to be incompatible with LAMMPS. If you are running on a large parallel machine, your system admins or the vendor should have already installed a version of MPI, which is likely to be faster than a self-installed MPICH or OpenMPI, so you should study the provided documentation to find out how to build and link with it.

The majority of OpenMP (threading) support in LAMMPS is provided by the `OPENMP` package; see the *OPENMP package* page for details. The `INTEL` package also includes OpenMP threading (it is compatible with `OPENMP` and will usually fall back on styles from that package, if a `INTEL` does not exist) and adds vectorization support when compiled with compatible compilers, in particular the Intel compilers on top of OpenMP. Also, the `KOKKOS` package can be compiled to include OpenMP threading.

In addition, there are a few commands in LAMMPS that have native OpenMP support included as well. These are

commands in the ML-SNAP, DIFFRACTION, and DPD-REACT packages. Furthermore, some packages support OpenMP threading indirectly through the libraries they interface to: e.g. KSPACE, and COLVARS. See the *Packages details* page for more info on these packages, and the pages for their respective commands for OpenMP threading info.

For CMake, if you use BUILD_OMP=yes, you can use these packages and turn on their native OpenMP support at run time by setting the OMP_NUM_THREADS environment variable before you launch LAMMPS.

When building LAMMPS with conventional make, the CCFLAGS and LINKFLAGS variables in Makefile.machine need to include the compiler flag that enables OpenMP. For the GNU compilers or Clang, it is -fopenmp. For (recent) Intel compilers, it is -qopenmp. If you are using a different compiler, please refer to its documentation.

#### OpenMP Compiler compatibility

Some compilers do not fully support the default(none) directive and others (e.g. GCC version 9 and beyond, Clang version 10 and later) may implement strict OpenMP 4.0 and later semantics, which are incompatible with the OpenMP 3.1 semantics used in LAMMPS for maximal compatibility with compiler versions in use. If compilation with OpenMP enabled fails because of your compiler requiring strict OpenMP 4.0 semantics, you can change the behavior by adding -D LAMMPS_OMP_COMPAT=4 to the LMP_INC variable in your makefile, or add it to the command-line flags while configuring with CMake. LAMMPS will auto-detect a suitable setting for most GNU, Clang, and Intel compilers.

### 3.5.2 Choice of compiler and compile/link options

The choice of compiler and compiler flags can be important for maximum performance. Vendor provided compilers for a specific hardware can produce faster code than open-source compilers like the GNU compilers. On the most common x86 hardware, the most popular C++ compilers are quite similar in their ability to optimize regular C/C++ source code at high optimization levels. When using the INTEL package, there is a distinct advantage in using the Intel C++ compiler due to much improved vectorization through SSE and AVX instructions on compatible hardware. The source code in that package conditionally includes compiler specific directives to enable these high degrees of vectorization. This may change over time as equivalent vectorization directives are included into the OpenMP standard and other compilers adopt them.

On parallel clusters or supercomputers which use "environment modules" for their compile/link environments, you can often access different compilers by simply loading the appropriate module before building LAMMPS.

**CMake build**

By default CMake will use the compiler it finds according to its internal preferences, and it will add optimization flags appropriate to that compiler and any *accelerator packages* you have included in the build. CMake will check if the detected or selected compiler is compatible with the C++ support requirements of LAMMPS and stop with an error, if this is not the case. A C++17 compatible compiler is required.

You can tell CMake to look for a specific compiler with setting CMake variables (listed below) during configuration. For a few common choices, there are also presets in the cmake/presets folder. You may also specify the corresponding CMAKE_*_FLAGS variables individually, if you want to experiment with alternate optimization flags. You should specify all 3 compilers, so that the (few) LAMMPS source files written in C or Fortran are built with a compiler consistent with the one used for the C++ files:

```
-D CMAKE_CXX_COMPILER=name            # name of C++ compiler
-D CMAKE_C_COMPILER=name              # name of C compiler
-D CMAKE_Fortran_COMPILER=name        # name of Fortran compiler

-D CMAKE_CXX_STANDARD=17              # put compiler in C++17 mode
-D CMAKE_CXX_FLAGS=string             # flags to use with C++ compiler
-D CMAKE_C_FLAGS=string               # flags to use with C compiler
-D CMAKE_Fortran_FLAGS=string         # flags to use with Fortran compiler
```

A few example command lines are:

```
# Building with GNU Compilers:
cmake -DCMAKE_C_COMPILER=gcc -DCMAKE_CXX_COMPILER=g++ \
      -DCMAKE_Fortran_COMPILER=gfortran ../cmake
# Building with Intel Classic Compilers:
cmake -DCMAKE_C_COMPILER=icc -DCMAKE_CXX_COMPILER=icpc \
      -DCMAKE_Fortran_COMPILER=ifort ../cmake
# Building with Intel oneAPI Compilers:
cmake -DCMAKE_C_COMPILER=icx -DCMAKE_CXX_COMPILER=icpx \
      -DCMAKE_Fortran_COMPILER=ifx ../cmake
# Building with LLVM/Clang Compilers:
cmake -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++ \
      -DCMAKE_Fortran_COMPILER=flang ../cmake
# Building with PGI/Nvidia Compilers:
cmake -DCMAKE_C_COMPILER=pgcc -DCMAKE_CXX_COMPILER=pgc++ \
      -DCMAKE_Fortran_COMPILER=pgfortran ../cmake
# Building with the NVHPC Compilers:
cmake -DCMAKE_C_COMPILER=nvc -DCMAKE_CXX_COMPILER=nvc++ \
      -DCMAKE_Fortran_COMPILER=nvfortran ../cmake
```

For compiling with the Clang/LLVM compilers a CMake preset is provided that can be loaded with `-C ../cmake/presets/clang.cmake`. Similarly, `-C ../cmake/presets/intel.cmake` should switch the compiler tool chain to the legacy Intel compilers, `-C ../cmake/presets/oneapi.cmake` will switch to the LLVM based oneAPI Intel compilers, `-C ../cmake/presets/pgi.cmake` will switch the compiler to the PGI compilers, and `-C ../cmake/presets/nvhpc.cmake` will switch to the NVHPC compilers.

**Note**

When the cmake command completes, it prints a summary to the screen which compilers it is using and what flags and settings will be used for the compilation. Note that if the top-level compiler is `mpicxx`, it is simply a wrapper on a real compiler. The underlying compiler info is what CMake will try to determine and report. You should check to confirm you are using the compiler and optimization flags you want.

**Makefile.machine settings for traditional make**

The "compiler/linker settings" section of a Makefile.machine lists compiler and linker settings for your C++ compiler, including optimization flags. For a parallel build it is recommended to use `mpicxx` or `mpiCC`, since these compiler wrappers will include a variety of settings appropriate for your MPI installation and thus avoiding the guesswork of finding the right flags.

Parallel build (see `src/MAKE/Makefile.mpi`):

```
CC =        mpicxx
CCFLAGS =   -g -O3
LINK =      mpicxx
LINKFLAGS = -g -O
```

Serial build with GNU gcc (see `src/MAKE/Makefile.serial`):

```
CC =         g++
CCFLAGS =    -g -O3
LINK =       g++
LINKFLAGS = -g -O
```

**Note**

If compilation stops with a message like the following:

```
g++ -g -O3  -DLAMMPS_GZIP -DLAMMPS_MEMALIGN=64    -I../STUBS     -c ../
↪main.cpp
In file included from ../pointers.h:24:0,
                 from ../input.h:17,
                 from ../main.cpp:16:
../lmptype.h:34:2: error: #error LAMMPS requires a C++17 (or later)␣
↪compliant compiler. Enable C++17 compatibility or upgrade the compiler.
```

then you have either an unsupported (old) compiler or you have to turn on C++17 mode. For those compilers, you need to add the `-std=c++17` flag. If there is no compiler that supports this flag (or equivalent), you would have to install a newer compiler that supports C++17; either as a binary package or through compiling from source.

If you build LAMMPS with any *Accelerator packages* included, there may be specific compiler or linker flags that are either required or recommended to enable required features and to achieve optimal performance. You need to include these in the `CCFLAGS` and `LINKFLAGS` settings above. For details, see the documentation for the individual packages listed on the *Accelerator packages* page. Or examine these files in the `src/MAKE/OPTIONS` directory. They correspond to each of the 5 accelerator packages and their hardware variants:

```
Makefile.opt                    # OPT package
Makefile.omp                    # OPENMP package
Makefile.intel_cpu              # INTEL package for CPUs
Makefile.gpu                    # GPU package
```

### 3.5.3 Build the LAMMPS executable and library

LAMMPS is always built as a library of C++ classes plus an executable. The executable is a simple `main()` function that sets up MPI and then creates a LAMMPS class instance from the LAMMPS library, which will then process commands provided via a file or from the console input. The LAMMPS library can also be called from another application or a scripting language. See the *Howto couple* doc page for more info on coupling LAMMPS to other codes. See the *Python* page for more info on wrapping and running LAMMPS from Python via its library interface.

**CMake build**

For CMake builds, you can select through setting CMake variables between building a shared or a static LAMMPS library and what kind of suffix is added to them (in case you want to concurrently install multiple variants of binaries with different settings). If none are set, defaults are applied.

```
-D BUILD_SHARED_LIBS=value   # yes or no (default)
-D LAMMPS_MACHINE=name       # name = mpi, serial, mybox, titan, laptop, etc
                             # no default value
```

The compilation will always produce a LAMMPS library and an executable linked to it. By default, this will be a static library named `liblammps.a` and an executable named `lmp` Setting `BUILD_SHARED_LIBS=yes` will instead produce a shared library called `liblammps.so` (or `liblammps.dylib` or `liblammps.dll` depending on the platform) If `LAMMPS_MACHINE=name` is set in addition, the name of the generated libraries will be changed to either `liblammps_name.a` or `liblammps_name.so`, respectively and the executable will be called `lmp_name`.

**Traditional make**

With the traditional makefile based build process, the choice of the generated executable or library depends on the "mode" setting. Several options are available and `mode=static` is the default.

```
make machine                # build LAMMPS executable lmp_machine
make mode=static machine    # same as "make machine"
make mode=shared machine    # build LAMMPS shared lib liblammps_machine.so
                            # instead
```

The "static" build will generate a static library called `liblammps_machine.a` and an executable named `lmp_machine`, while the "shared" build will generate a shared library `liblammps_machine.so` instead and `lmp_machine` will be linked to it. The build step will also create generic soft links, named `liblammps.a` and `liblammps.so`, which point to the specific `liblammps_machine.a/so` files.

### Additional information

Note that for creating a shared library, all the libraries it depends on must be compiled to be compatible with shared libraries. This should be the case for libraries included with LAMMPS, such as the dummy MPI library in `src/STUBS` or any package libraries in the `lib` directory, since they are always built in a shared library compatible way using the `-fPIC` compiler switch. However, if an auxiliary library (like MPI or FFTW) does not exist as a compatible format, the shared library linking step may generate an error. This means you will need to install a compatible version of the auxiliary library. The build instructions for that library should tell you how to do this.

As an example, here is how to build and install the MPICH library, a popular open-source version of MPI, as a shared library in the default /usr/local/lib location:

```
./configure --enable-shared
make
make install
```

You may need to use `sudo make install` in place of the last line if you do not have write privileges for `/usr/local/lib` or use the `--prefix` configuration option to select an installation folder, where you do have write access. The end result should be the file `/usr/local/lib/libmpich.so`. On many Linux installations, the folder `${HOME}/.local` is an alternative to using `/usr/local` and does not require superuser or sudo access. In that case the configuration step becomes:

```
./configure --enable-shared --prefix=${HOME}/.local
```

Avoiding the use of "sudo" for custom software installation (i.e. from source and not through a package manager tool provided by the OS) is generally recommended to ensure the integrity of the system software installation.

### 3.5.4 Including or removing debug support

By default the compilation settings will include the `-g` flag which instructs the compiler to include debug information (e.g. which line of source code a particular instruction correspond to). This can be extremely useful in case LAMMPS crashes and can help to provide crucial information in *tracking down the origin of a crash* and help the LAMMPS developers fix bugs in the source code. However, this increases the storage requirements for object files, libraries, and the executable 3-5 fold.

If this is a concern, you can change the compilation settings or remove the debug information from the LAMMPS executable:

- **Traditional make**: edit your `Makefile.<machine>` to remove the `-g` flag from the `CCFLAGS` and `LINKFLAGS` definitions
- **CMake**: use `-D CMAKE_BUILD_TYPE=Release` or explicitly reset the applicable compiler flags (best done using the text mode or graphical user interface).
- **Remove debug info**: If you are only concerned about the executable being too large, you can use the `strip` tool (e.g. `strip lmp_serial`) to remove the debug information from the executable file. Do not strip libraries or object files, as that will render them unusable.

---

### 3.5.5 Build LAMMPS tools

Some tools described in *Auxiliary tools* can be built directly using CMake or Make.

**CMake build**

```
-D BUILD_TOOLS=value          # yes or no (default). Build binary2txt,
                              # chain.x, micelle2d.x, msi2lmp, phana,
                              # stl_bin2txt
-D BUILD_LAMMPS_GUI=value     # yes or no (default). Build LAMMPS-GUI
-D BUILD_WHAM=value           # yes (default). Download and build WHAM;
                              # only available for BUILD_LAMMPS_GUI=yes
```

The generated binaries will also become part of the LAMMPS installation (see below).

**Traditional make**

```
cd lammps/tools
make all               # build all binaries of tools
make binary2txt        # build only binary2txt tool
make chain             # build only chain tool
make micelle2d         # build only micelle2d tool
```

**Note**

Building LAMMPS-GUI *requires* building LAMMPS with CMake.

---

### 3.5.6 Install LAMMPS after a build

After building LAMMPS, you may wish to copy the LAMMPS executable or library, along with other LAMMPS files (library header, doc files), to a globally visible place on your system, for others to access. Note that you may need super-user privileges (e.g. sudo) if the directory you want to copy files to is protected.

**CMake build**

```
cmake -D CMAKE_INSTALL_PREFIX=path [options ...] ../cmake
make                          # perform make after CMake command
make install                  # perform the installation into prefix
```

During the installation process CMake will by default remove any runtime path settings for loading shared libraries. Because of this you may have to set or modify the `LD_LIBRARY_PATH` (or `DYLD_LIBRARY_PATH`) environment variable, if you are installing LAMMPS into a non-system location and/or are linking to libraries in a non-system location that depend on such runtime path settings. As an alternative, you may set the CMake variable `LAMMPS_INSTALL_RPATH` to `on` and then the runtime paths for any linked shared libraries and the library installation folder for the LAMMPS library will be embedded and thus the requirement to set environment variables is avoided. The `off` setting is usually preferred for packaged binaries or when setting up environment modules, the `on` setting is more convenient for installing software into a non-system or personal folder.

**Traditional make**

There is no "install" option in the `src/Makefile` for LAMMPS. If you wish to do this you will need to first build LAMMPS, then manually copy the desired LAMMPS files to the appropriate system directories.

## 3.6 Optional build settings

LAMMPS can be built with several optional settings. Each subsection explains how to do this for building both with CMake and make.

- *C++17 standard compliance* when building all of LAMMPS
- *FFT library* for use with the *kspace_style pppm* command
- *Size of LAMMPS integer types and size limits*
- *Read or write compressed files*
- *Support for downloading files from the input*
- *Prevent download of large potential files*
- *Memory allocation alignment*
- *Workaround for long long integers*
- *Exception handling when using LAMMPS as a library* to capture errors

---

### 3.6.1 C++17 standard compliance

Changed in version 10Sep2025.

A C++17 standard compatible compiler is currently the minimum requirement for compiling LAMMPS. LAMMPS version 22 July 2025 is the last version compatible with the C++11 standard for the core code and most packages. Most currently used C++ compilers are compatible with C++17, but some older ones may need extra flags to enable C++17 compliance.

```
CCFLAGS = -g -O3 -std=c++17
```

Individual packages may require compliance with a later C++ standard like C++20. These requirements will be documented with the *individual packages*.

---

### 3.6.2 FFT library

When the KSPACE package is included in a LAMMPS build, the *kspace_style pppm* command performs 3d FFTs which require use of an FFT library to compute 1d FFTs. The KISS FFT library is included with LAMMPS, but other libraries can be faster. LAMMPS can use them if they are available on your system.

Added in version 7Feb2024.

Alternatively, LAMMPS can use the heFFTe library for the MPI communication algorithms, which comes with many optimizations for special cases, e.g. leveraging available 2D and 3D FFTs in the back end libraries and better pipelining for packing and communication.

**CMake build**

```
-D FFT=value              # FFTW3 or MKL or NVPL or KISS,
                          # default is FFTW3 if found, else KISS
-D FFT_KOKKOS=value       # FFTW3 or MKL or NVPL or KISS or CUFFT
                          # or HIPFFT or MKL_GPU, default is KISS
-D FFT_SINGLE=value       # yes or no (default), no = double precision
```

```
-D FFT_PACK=value          # array (default) or pointer or memcpy
-D FFT_USE_HEFFTE=value    # yes or no (default), yes links to heFFTe
```

**Note**

When the Kokkos variant of a package is compiled and selected at run time, the FFT library selected by the FFT_KOKKOS variable applies. Otherwise, the FFT library selected by the FFT variable applies. The same FFT settings apply to both. FFT_KOKKOS must be compatible with the Kokkos back end - for example, when using the CUDA back end of Kokkos, you must use either CUFFT or KISS.

Usually these settings are all that is needed. If FFTW3 is selected, then CMake will try to detect, if threaded FFTW libraries are available and enable them by default. This setting is independent of whether OpenMP threads are enabled and a package like KOKKOS or OPENMP is used. If CMake cannot detect the FFT library, you can set these variables to assist:

```
-D FFTW3_INCLUDE_DIR=path    # path to FFTW3 include files
-D FFTW3_LIBRARY=path        # path to FFTW3 libraries
-D FFTW3_OMP_LIBRARY=path    # path to FFTW3 OpenMP wrapper libraries
-D FFT_FFTW_THREADS=on       # enable using OpenMP threaded FFTW3 libraries
-D MKL_INCLUDE_DIR=path      # ditto for Intel MKL library
-D FFT_MKL_THREADS=on        # enable using threaded FFTs with MKL libraries
-D MKL_LIBRARY=path          # path to MKL libraries
-D FFT_HEFFTE_BACKEND=value # FFTW or MKL or empty/undefined for the stock
                            # heFFTe back end
-D Heffte_ROOT=path          # path to an existing heFFTe installation
-D nvpl_fft_INCLUDE_DIR=path # path to NVPL FFT include files
-D nvpl_fft_LIBRARY_DIR=path # path to NVPL FFT libraries
```

**Note**

heFFTe comes with a builtin (= stock) back end for FFTs, i.e. a default internal FFT implementation; however, this stock back end is intended for testing purposes only and is not optimized for production runs.

**Traditional make**

To change the FFT library to be used and its options, you have to edit your machine Makefile. Below are examples how the makefile variables could be changed.

```
FFT_INC = -DFFT_<NAME>         # where <NAME> is KISS (default), FFTW3,
                               # FFTW (same as FFTW3), NVPL, or MKL
FFT_INC = -DFFT_KOKKOS_<NAME> # where <NAME> is KISS (default), FFTW3,
                               # FFTW (same as FFTW3), NVPL, MKL, CUFFT,
                               # HIPFFT, or MKL_GPU
FFT_INC = -DFFT_SINGLE        # do not specify for double precision
FFT_INC = -DFFT_FFTW_THREADS # enable using threaded FFTW3 libraries
FFT_INC = -DFFT_MKL_THREADS  # enable using threaded FFTs with MKL libraries
FFT_INC = -DFFT_PACK_ARRAY    # or -DFFT_PACK_POINTER or -DFFT_PACK_MEMCPY
                               # default is FFT_PACK_ARRAY if not specified
```

```
FFT_INC =  -I/usr/local/include
```

```
FFT_PATH = -L/usr/local/lib

# hipFFT either precision
FFT_LIB =  -lhipfft

# cuFFT either precision
FFT_LIB =  -lcufft

# MKL_GPU either precision
FFT_LIB = -lmkl_sycl_dft -lmkl_intel_ilp64 -lmkl_tbb_thread -lmkl_core -ltbb

# FFTW3 double precision
FFT_LIB =  -lfftw3

# FFTW3 double precision with threads (needs -DFFT_FFTW_THREADS)
FFT_LIB =  -lfftw3 -lfftw3_omp

# FFTW3 single precision
FFT_LIB =  -lfftw3 -lfftw3f

# serial MKL with Intel compiler
FFT_LIB =  -lmkl_intel_lp64 -lmkl_sequential -lmkl_core

# serial MKL with GNU compiler
FFT_LIB =  -lmkl_gf_lp64 -lmkl_sequential -lmkl_core

# threaded MKL with Intel compiler
FFT_LIB =  -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core

# threaded MKL with GNU compiler
FFT_LIB =  -lmkl_gf_lp64 -lmkl_gnu_thread -lmkl_core

# MKL with automatic runtime selection of interface libs
FFT_LIB =  -lmkl_rt

# threaded NVPL FFT
FFT_LIB =  -lnvpl_fftw
```

As with CMake, you do not need to set paths in `FFT_INC` or `FFT_PATH`, if the compiler can find the FFT header and library files in its default search path. You must specify `FFT_LIB` with the appropriate FFT libraries to include in the link.

Traditional make can also link to heFFTe using an existing installation

```
include <path-to-heffte-installation>/share/heffte/HeffteMakefile.in
FFT_INC = -DFFT_HEFFTE -DFFT_HEFFTE_FFTW $(heffte_include)
FFT_PATH =
FFT_LIB = $(heffte_link) $(heffte_libs)
```

The heFFTe install path will contain `HeffteMakefile.in`. which will define the `heffte_` include variables needed to link to heFFTe from an external project using traditional make. The `-DFFT_HEFFTE` is required to switch to using heFFTe, while the optional `-DFFT_HEFFTE_FFTW` selects the desired heFFTe back end, e.g., `-DFFT_HEFFTE_FFTW` or `-DFFT_HEFFTE_MKL`, omitting the variable will default to the *stock* back end. The heFFTe *stock* back end is intended to be used for testing and debugging,

but is not performance optimized for large scale production runs.

The KISS FFT library is included in the LAMMPS distribution. It is portable across all platforms. Depending on the size of the FFTs and the number of processors used, the other libraries listed here can be faster.

However, note that long-range Coulombics are only a portion of the per-timestep CPU cost, FFTs are only a portion of long-range Coulombics, and 1d FFTs are only a portion of the FFT cost (parallel communication can be costly). A breakdown of these timings is printed to the screen at the end of a run when using the *kspace_style pppm* command. The *Screen and logfile output* page gives more details. A more detailed (and time consuming) report of the FFT performance is generated with the *kspace_modify fftbench yes* command.

FFTW is a fast, portable FFT library that should also work on any platform and can be faster than the KISS FFT library. You can download it from www.fftw.org. LAMMPS requires version 3.X; the legacy version 2.1.X is no longer supported.

Building FFTW for your box should be as simple as `./configure; make; make install`. The install command typically requires root privileges (e.g. invoke it via sudo), unless you specify a local directory with the `--prefix` option of configure. Type `./configure --help` to see various options.

The Intel MKL math library is part of the Intel compiler suite. It can be used with the Intel or GNU compiler (see the `FFT_LIB` setting above).

The NVIDIA Performance Libraries (NVPL) FFT library is optimized for NVIDIA Grace Armv9.0 architecture. You can download it from https://docs.nvidia.com/nvpl/

The cuFFT and hipFFT FFT libraries are packaged with NVIDIA's CUDA and AMD's HIP installations, respectively. These FFT libraries require the Kokkos acceleration package to be enabled and the Kokkos back end to be GPU-resident (i.e., HIP or CUDA). Similarly, GPU offload of FFTs on Intel GPUs with oneMKL currently requires the Kokkos acceleration package to be enabled with the SYCL back end.

Performing 3d FFTs in parallel can be time-consuming due to data access and required communication. This cost can be reduced by performing single-precision FFTs instead of double precision. Single precision means the real and imaginary parts of a complex datum are 4-byte floats. Double precision means they are 8-byte doubles. Note that Fourier transform and related PPPM operations are somewhat less sensitive to floating point truncation errors, and thus the resulting error is generally less than the difference in precision. Using the `-DFFT_SINGLE` setting trades off a little accuracy for reduced memory use and parallel communication costs for transposing 3d FFT data.

When using `-DFFT_SINGLE` with FFTW3, you may need to ensure that the FFTW3 installation includes support for single-precision.

When compiling FFTW3 from source, you can do the following, which should produce the additional libraries `libfftw3f.a` and `libfftw3f.so`.

```
make clean
./configure --enable-single; make; make install
```

Performing 3d FFTs requires communication to transpose the 3d FFT grid. The data packing/unpacking for this can be done in one of 3 modes (ARRAY, POINTER, MEMCPY) as set by the `FFT_PACK` syntax above. Depending on the machine, the size of the FFT grid, the number of processors used, one option may be slightly faster. The default is ARRAY mode.

When using `-DFFT_HEFFTE` CMake will first look for an existing install with hints provided by `-DHeffte_ROOT`, as recommended by the CMake standard and note that the name is case sensitive. If CMake cannot find a heFFTe installation with the correct back end (e.g., FFTW or MKL), it will attempt to download and build the library automatically. In this case, LAMMPS CMake will also accept all heFFTe specific variables listed in the heFFTe documentation and those variables will be passed into the heFFTe build.

### 3.6.3 Size of LAMMPS integer types and size limits

LAMMPS uses a few custom integer data types, which can be defined as either 4-byte (= 32-bit) or 8-byte (= 64-bit) integers at compile time. This has an impact on the size of a system that can be simulated, or how large counters can become before "rolling over". The default setting of "smallbig" is almost always adequate.

**CMake build**

With CMake the choice of integer types is made via setting a variable during configuration.

```
-D LAMMPS_SIZES=value    # smallbig (default) or bigbig
```

If the variable is not set explicitly, "smallbig" is used.

**Traditional build**

If you want a setting different from the default, you need to edit the `LMP_INC` variable setting your machine Makefile.

```
LMP_INC = -DLAMMPS_SMALLBIG     # or -DLAMMPS_BIGBIG
```

The default setting is `-DLAMMPS_SMALLBIG` if nothing is specified

#### LAMMPS system size restrictions

| | smallbig | bigbig |
|---|---|---|
| Total atom count | $2^{63}$ atoms ($= 9.223 \cdot 10^{18}$) | $2^{63}$ atoms ($= 9.223 \cdot 10^{18}$) |
| Total timesteps | $2^{63}$ steps ($= 9.223 \cdot 10^{18}$) | $2^{63}$ steps ($= 9.223 \cdot 10^{18}$) |
| Atom ID values | $1 \leq i \leq 2^{31} (= 2.147 \cdot 10^{9})$ | $1 \leq i \leq 2^{63} (= 9.223 \cdot 10^{18})$ |
| Image flag values | $-512 \leq i \leq 511$ | $-1\,048\,576 \leq i \leq 1\,048\,575$ |

The "bigbig" setting increases the size of image flags and atom IDs over the default "smallbig" setting.

These are limits for the core of the LAMMPS code, specific features or some styles may impose additional limits. Also, there are limitations when using the library interface where some functions with known issues have been replaced by dummy calls printing a corresponding error message rather than crashing randomly or corrupting data.

Atom IDs are not required for atomic systems which do not store bond topology information, though IDs are enabled by default. The *atom_modify id no* command will turn them off. Atom IDs are required for molecular systems with bond topology (bonds, angles, dihedrals, etc). Similarly, some force or compute or fix styles require atom IDs. Thus, if you model a molecular system or use one of those styles with more than 2 billion atoms, you need the "bigbig" setting.

Regardless of the total system size limits, the maximum number of atoms per MPI rank (local + ghost atoms) is limited to 2 billion for atomic systems and 500 million for systems with bonds (the additional restriction is due to using the 2 upper bits of the local atom index in neighbor lists for storing special bonds info).

Image flags store 3 values per atom in a single integer, which count the number of times an atom has moved through the periodic box in each dimension. See the *dump* manual page for a discussion. If an atom moves through the periodic box more than this limit, the value will "roll over", e.g. from 511 to -512, which can cause diagnostics like the mean-squared displacement, as calculated by the *compute msd* command, to be faulty.

Also note that the GPU package requires its lib/gpu library to be compiled with the same size setting, or the link will fail. A CMake build does this automatically. When building with make, the setting in whichever `lib/gpu/Makefile` is used must be the same as above.

### 3.6.4 Read or write compressed files

Changed in version 11Feb2026: Added support for `brotli` and `7-zip`

If this option is enabled, large files can be read or written with compression by `gzip` or similar tools by several LAMMPS commands, including *read_data*, *write_data*, *rerun*, *dump*, and *write_dump*. Supported compression tools and algorithms are currently `gzip`, `bzip2`, `zstd`, `xz`, `lz4`, `lzma` (via xz), `brotli`, and `7-zip (via 7z)`. LAMMPS checks at runtime, which compression commands are available and adjusts the check for supported suffixes accordingly. The list of available compression formats and suffixes is shown when running LAMMPS with the *-help or -h command_line flag*.

**CMake build**

```
-D WITH_GZIP=value  # yes or no
                    # default is yes if CMake can find the gzip program
```

**Traditional make**

```
LMP_INC = -DLAMMPS_GZIP   <other LMP_INC settings>
```

This option requires that your operating system fully supports the "popen()" function in the standard runtime library and that a `gzip` or other executable can be found by LAMMPS in the standard search path during a run.

**Note**

On clusters with high-speed networks, using the "fork()" library call (required by "popen()") can interfere with the fast communication library and lead to simulations using compressed output or input to hang or crash. For selected operations, compressed file I/O is also available using a compression library instead, which is what the *COMPRESS package* provides.

---

### 3.6.5 Support for downloading files from the input

Added in version 29Aug2024.

The *geturl command* command uses the the libcurl library to download files. This requires that LAMMPS is compiled accordingly which needs the following settings:

**CMake build**

```
-D WITH_CURL=value      # yes or no
                        # default = yes if CMake finds CURL development␣
↪files, else no
```

Usually these settings are all that is needed. If CMake cannot find the graphics header, library, executable files, you can set these variables:

```
-D CURL_INCLUDE_DIR=path    # path to folder which contains curl.h header␣
↪file
-D CURL_LIBRARY=path        # path to libcurls.a (.so) file
```

**Traditional make**

```
LMP_INC = -DLAMMPS_CURL  <other LMP_INC settings>

CURL_INC = -I/usr/local/include   # path to curl folder with curl.h
CURL_PATH = -L/usr/lib            # paths to libcurl.a(.so) if make cannot␣
↪find it
CURL_LIB = -lcurl                 # library names
```

As with CMake, you do not need to set `CURL_INC` or `CURL_PATH`, if make can find the libcurl header and library files in their default system locations. You must specify `CURL_LIB` with a paths or linker flags to link to libcurl.

### 3.6.6 Prevent download of large potential files

Added in version 8Feb2023.

LAMMPS bundles a selection of potential files in the `potentials` folder as examples of how those kinds of potential files look like and for use with the provided input examples in the `examples` tree. To keep the size of the distributed LAMMPS source package small, very large potential files (> 5 MBytes) are not bundled, but only downloaded on demand when the *corresponding package* is installed. This automatic download can be prevented when *building LAMMPS with CMake* by adding the setting *-D DOWNLOAD_POTENTIALS=off* when configuring.

### 3.6.7 Memory allocation alignment

This setting enables the use of the `posix_memalign()` call instead of `malloc()` when LAMMPS allocates large chunks of memory. Vector instructions on CPUs may become more efficient, if dynamically allocated memory is aligned on larger-than-default byte boundaries. On most current operating systems, the `malloc()` implementation returns pointers that are aligned to 16-byte boundaries. Using SSE vector instructions efficiently, however, requires memory blocks being aligned on 64-byte boundaries.

**CMake build**

```
-D LAMMPS_MEMALIGN=value            # 0, 8, 16, 32, 64 (default)
```

Use a `LAMMPS_MEMALIGN` value of 0 to disable using `posix_memalign()` and revert to using the `malloc()` C-library function instead. When compiling LAMMPS for Windows systems, `malloc()` will always be used and this setting is ignored.

**Traditional make**

```
LMP_INC = -DLAMMPS_MEMALIGN=value    # 8, 16, 32, 64
```

Do not set `-DLAMMPS_MEMALIGN`, if you want to have memory allocated with the `malloc()` function call instead. `-DLAMMPS_MEMALIGN` **cannot** be used on Windows, as Windows different function calls with different semantics for allocating aligned memory, that are not compatible with how LAMMPS manages its dynamical memory.

### 3.6.8 Workaround for long long integers

If your system or MPI version does not recognize "long long" data types, the following setting will be needed. It converts "long long" to a "long" data type, which should be the desired 8-byte integer on those systems:

**CMake build**

```
-D LAMMPS_LONGLONG_TO_LONG=value      # yes or no (default)
```

**Traditional make**

```
LMP_INC = -DLAMMPS_LONGLONG_TO_LONG  <other LMP_INC settings>
```

---

### 3.6.9 Exception handling when using LAMMPS as a library

LAMMPS errors do not kill the calling code, but throw an exception. In the C-library interface, the call stack is unwound and control returns to the caller, e.g. to Python or a code that is coupled to LAMMPS. The error status can then be queried. When using C++ directly, the calling code has to be set up to *catch* exceptions thrown from within LAMMPS.

**Note**

When LAMMPS is running in parallel, it is not always possible to cleanly recover from an exception since not all parallel ranks may throw an exception and thus other MPI ranks may get stuck waiting for messages from the ones with errors.

## 3.7 Include packages in build

In LAMMPS, a package is a group of files that enable a specific set of features. For example, force fields for molecular systems or rigid-body constraints are in packages. In the src directory, each package is a subdirectory with the package name in capital letters.

An overview of packages is given on the *Packages* doc page. Brief overviews of each package are on the *Packages details* page.

When building LAMMPS, you can choose to include or exclude each package. Generally, there is no need to include a package if you never plan to use its features.

If you get a run-time error that a LAMMPS command or style is "unknown", it is often because the command is contained in a package, and your build did not include that package. If the command or style *is* available in a package included in the LAMMPS distribution, the error message will indicate which package would be needed. Running LAMMPS with the *-h command-line switch* will print *all* optional commands and packages that were enabled when building that executable.

For the majority of packages, if you follow the single step below to include it, you can then build LAMMPS exactly the same as you would without any packages installed. A few packages may require additional steps, as explained on the *Build extras* page.

These links take you to the extra instructions for those select packages:

| | | | | | |
|---|---|---|---|---|---|
| *ADIOS* | *APIP* | *COLVARS* | *COMPRESS* | *ELECTRODE* | *GPU* |
| *H5MD* | *INTEL* | *KIM* | *KOKKOS* | *LEPTON* | *MACHDYN* |
| *MDI* | *MISC* | *ML-HDNNP* | *ML-IAP* | *ML-PACE* | *ML-POD* |
| *ML-QUIP* | *ML-RUNNER* | *MOLFILE* | *NETCDF* | *OPENMP* | *OPT* |
| *PLUMED* | *PYTHON* | *QMMM* | *RHEO* | *SCAFACOS* | *VORONOI* |
| *VTK* | | | | | |

The mechanism for including packages is simple but different for the CMake build system in comparison to the traditional make build.

**CMake build**

```
-D PKG_NAME=value          # yes or no (default)
```

Examples:

```
-D PKG_MANYBODY=yes
-D PKG_INTEL=yes
```

All packages are included the same way. See the shortcut section below for how to install many packages at once with CMake.

**Note**

If you switch between building with CMake and make builds, no packages in the src directory can be installed when you invoke `cmake`. CMake will give an error if that is not the case, indicating how you can uninstall all packages in the src dir.

**Traditional make**

```
cd lammps/src
make ps         # check which packages are currently installed
make yes-name   # install a package with name
make no-name    # uninstall a package with name
make mpi        # build LAMMPS with whatever packages are now installed
```

Examples:

```
make no-rigid
make yes-intel
```

All packages are included the same way. See the shortcut section below for how to install many packages at once with make.

**Note**

You must always re-build LAMMPS (via make) after installing or uninstalling a package, for the action to take effect. The included dependency tracking will make certain only files that are required to be rebuilt are recompiled.

> **Note**
>
> You cannot install or uninstall packages and build LAMMPS in a single make command with multiple targets, e.g. `make yes-colloid mpi`. This is because the make procedure creates a list of source files that will be out-of-date for the build if the package configuration changes within the same command. You can include or exclude multiple packages in a single make command, e.g. `make yes-colloid no-manybody`.

### 3.7.1 Information for both build systems

Almost all packages can be included or excluded in a LAMMPS build, independent of the other packages. However, some packages include files derived from files in other packages. LAMMPS checks for this and does the right thing. Individual files are only included if their dependencies are already included. Likewise, if a package is excluded, other files dependent on that package are also excluded.

> **Note**
>
> By default **no** packages are installed. Prior to August 2018, however, if you downloaded a tarball, 3 packages (KSPACE, MANYBODY, MOLECULE) were pre-installed via the traditional make procedure in the `src` directory. That is no longer the case, so that CMake will build as-is without needing to first uninstall those packages. You can quickly include those packages (plus RIGID and GRAPHICS) by using the "basic" preset with CMake or `make yes-basic` with traditional make as discussed below.

---

### 3.7.2 CMake presets for installing many packages

Instead of specifying all the CMake options via the command-line, CMake allows initializing its settings cache using script files. These are regular CMake files which can manipulate and set CMake variables (which represent selected options), and can also contain control flow constructs for more complex operations.

LAMMPS includes several of these files to define configuration "presets", similar to the options that exist for the Make based system. Using these files, you can enable/disable portions of the available packages in LAMMPS. If you need a custom preset, you can make a copy of one of them and modify it to suit your needs.

```
# enable just a few core packages (MOLECULE, KSPACE, MANYBODY, RIGID, GRAPHICS)
cmake -C ../cmake/presets/basic.cmake    [OPTIONS] ../cmake

# enable most packages
cmake -C ../cmake/presets/most.cmake     [OPTIONS] ../cmake

# enable packages which download sources or potential files
cmake -C ../cmake/presets/download.cmake [OPTIONS] ../cmake

# disable packages that do require extra libraries or tools
cmake -C ../cmake/presets/nolib.cmake    [OPTIONS] ../cmake

# change settings to use the Clang compilers by default
cmake -C ../cmake/presets/clang.cmake    [OPTIONS] ../cmake

# change settings to use the GNU compilers by default
```

(continues on next page)

(continued from previous page)

```
cmake -C ../cmake/presets/gcc.cmake      [OPTIONS] ../cmake

# change settings to use the Intel compilers by default
cmake -C ../cmake/presets/intel.cmake    [OPTIONS] ../cmake

# change settings to use the PGI compilers by default
cmake -C ../cmake/presets/pgi.cmake      [OPTIONS] ../cmake

# enable all packages
cmake -C ../cmake/presets/all_on.cmake   [OPTIONS] ../cmake

# disable all packages
cmake -C ../cmake/presets/all_off.cmake  [OPTIONS] ../cmake

#  compile with MinGW cross-compilers
mingw64-cmake -C ../cmake/presets/mingw-cross.cmake [OPTIONS] ../cmake

# compile serial multi-arch binaries on macOS
cmake -C ../cmake/presets/macos-multiarch.cmake [OPTIONS] ../cmake
```

Presets that have names starting with "windows" are specifically for compiling LAMMPS *natively on Windows* and presets that have names starting with "kokkos" are specifically for selecting configurations for compiling LAMMPS with *KOKKOS*.

**Note**

Running cmake this way manipulates the CMake settings cache in your current build directory. You can combine multiple presets and options in a single cmake run, or change settings incrementally by running cmake with new flags. If you use a present for selecting a set of compilers, it will reset all settings from previous CMake runs.

### Example

```
# build LAMMPS with most commonly used packages, but then remove
# those requiring additional library or tools, but still enable
# GPU package and configure it for using CUDA. You can run.
mkdir build
cd build
cmake -C ../cmake/presets/most.cmake -C ../cmake/presets/nolib.cmake \
      -D PKG_GPU=on -D GPU_API=cuda ../cmake

# to add another package, say BODY to the previous configuration you can run:
cmake -D PKG_BODY=on .

# to reset the package selection from above to the default of no packages
# but leaving all other settings untouched. You can run:
cmake -C ../cmake/presets/all_off.cmake .
```

### 3.7.3 Make shortcuts for installing many packages

The following commands are useful for managing package source files and their installation when building LAMMPS via traditional make. Just type `make` in lammps/src to see a one-line summary.

These commands install/uninstall sets of packages:

```
make yes-all                    # install all packages
make no-all                     # check for changes and uninstall all packages
make no-installed               # only check and uninstall installed packages
make yes-basic                  # install a few commonly used packages'
make no-basic                   # remove a few commonly used packages'
make yes-most                   # install most packages w/o libs'
make no-most                    # remove most packages w/o libs'
```

which install/uninstall various sets of packages. Typing `make package` will list all the these commands.

**Note**

Installing or uninstalling a package for the make based build process works by simply copying files back and forth between the main source directory src and the subdirectories with the package name (e.g. src/KSPACE, src/MANYBODY), so that the files are included or excluded when LAMMPS is built. Only source files in the src folder will be compiled.

The following make commands help manage files that exist in both the src directory and in package subdirectories. You do not normally need to use these commands unless you are editing LAMMPS files or are updating LAMMPS via git.

Type `make package-status` or `make ps` to show which packages are currently installed. For those that are installed, it will list any files that are different in the src directory and package subdirectory.

Type `make package-installed` or `make pi` to show which packages are currently installed, without listing the status of packages that are not installed.

Type `make package-update` or `make pu` to overwrite src files with files from the package subdirectories if the package is installed. It should be used after the checkout has been *updated or changed with git*, this will only update the files in the package subdirectories, but not the copies in the src folder.

Type `make package-overwrite` to overwrite files in the package subdirectories with src files.

Type `make package-diff` to list all differences between pairs of files in both the source directory and the package directory.

## 3.8 Packages with extra build options

When building with some packages, additional steps may be required, in addition to

| CMake build | Traditional make |
|---|---|
| `cmake -D PKG_NAME=yes` | `make yes-name` |

as described on the *Build_package* page.

For a CMake build there may be additional optional or required variables to set.

Changed in version 10Sep2025.

The traditional build system with GNU make no longer supports packages that require extra steps in the `lammps/lib` directory.

This is the list of packages that may require additional steps.

| | | | | | |
|---|---|---|---|---|---|
| *ADIOS* | *APIP* | *COLVARS* | *COMPRESS* | *ELECTRODE* | *GRAPHICS* |
| *GPU* | *H5MD* | *INTEL* | *KIM* | *KOKKOS* | *LEPTON* |
| *MACHDYN* | *MBX* | *MDI* | *MISC* | *ML-HDNNP* | *ML-IAP* |
| *ML-PACE* | *ML-POD* | *ML-QUIP* | *ML-RUNNER* | *MOLFILE* | *NETCDF* |
| *OPENMP* | *OPT* | *PLUMED* | *PYTHON* | *QMMM* | *RHEO* |
| *SCAFACOS* | *VORONOI* | *VTK* | | | |

### 3.8.1 COMPRESS package

To build with this package you must have the zlib compression library available on your system to build dump styles with a `/gz` suffix. There are also styles using the Zstandard library which have a '/zstd' suffix. The zstd library version must be at least 1.4. Older versions use an incompatible API and thus LAMMPS will fail to compile.

**CMake build**

If CMake cannot find the zlib library or include files, you can set these variables:

```
-D ZLIB_INCLUDE_DIR=path     # path to zlib.h header file
-D ZLIB_LIBRARY=path         # path to libz.a (.so) file
```

Support for Zstandard compression is auto-detected and for that CMake depends on the pkg-config tool to identify the necessary flags to compile with this library, so the corresponding `libzstandard.pc` file must be in a folder where `pkg-config` can find it, which may require adding it to the `PKG_CONFIG_PATH` environment variable.

**Traditional make**

Changed in version 10Sep2025.

The COMPRESS package no longer supports the traditional make build. You need to build LAMMPS with CMake.

### 3.8.2 GRAPHICS package

Added in version 11Feb2026: *dump image*, *dump_movie* and supporting classes were moved to form the new GRAPHICS package together with several *fix graphics/...* styles.

The *dump image* command has options to output JPEG or PNG image files in addition to the default PPM format, and the *fix graphics/labels* can read images in JPEG or PNG format in addition to PPM format files. Likewise, the *dump movie* command outputs movie files in a variety of movie formats. Using these additional options requires the following settings:

**CMake build**

```
-D WITH_JPEG=value     # yes or no
                       # default = yes if CMake finds JPEG development files,␣
↪else no
-D WITH_PNG=value      # yes or no
                       # default = yes if CMake finds PNG and ZLIB␣
↪development files,
                       # else no
-D WITH_FFMPEG=value   # yes or no
                       # default = yes if CMake can find ffmpeg, else no
```

Usually these settings are all that is needed. If CMake cannot find the graphics header, library, executable files, you can set these variables:

```
-D JPEG_INCLUDE_DIR=path     # path to jpeglib.h header file
-D JPEG_LIBRARY=path         # path to libjpeg.a (.so) file
-D PNG_INCLUDE_DIR=path      # path to png.h header file
-D PNG_LIBRARY=path          # path to libpng.a (.so) file
-D ZLIB_INCLUDE_DIR=path     # path to zlib.h header file
-D ZLIB_LIBRARY=path         # path to libz.a (.so) file
-D FFMPEG_EXECUTABLE=path    # path to ffmpeg executable
```

**Traditional make**

```
LMP_INC = -DLAMMPS_JPEG -DLAMMPS_PNG -DLAMMPS_FFMPEG  <other LMP_INC␣
↪settings>

JPG_INC = -I/usr/local/include   # path to jpeglib.h, png.h, zlib.h headers
                                 # if make cannot find them
JPG_PATH = -L/usr/lib            # paths to libjpeg.a, libpng.a, libz.a (.so)
                                 # files if make cannot find them
JPG_LIB = -ljpeg -lpng -lz       # library names
```

As with CMake, you do not need to set `JPG_INC` or `JPG_PATH`, if make can find the graphics header and library files in their default system locations. You must specify `JPG_LIB` with a list of graphics libraries to include in the link. You must make certain that the ffmpeg executable (or ffmpeg.exe on Windows) is in a directory where LAMMPS can find it at runtime; that is usually a directory list in your `PATH` environment variable.

Using `ffmpeg` to output movie files requires that your machine supports the "popen" function in the standard runtime library.

**Note**

On some clusters with high-speed networks, using the fork() library call (required by popen()) can interfere with the fast communication library and lead to simulations using ffmpeg to hang or crash.

### 3.8.3 GPU package

To build with this package, you must choose options for precision and which GPU hardware to build for. The GPU package currently supports three different types of back ends: OpenCL, CUDA and HIP.

#### CMake build

```
-D GPU_API=value              # value = opencl (default) or cuda or hip
-D GPU_PREC=value             # precision setting
                              # value = double or mixed (default) or single
-D GPU_ARCH=value             # primary GPU hardware choice for all GPU_API back ends
                              # value = sm_XX for cuda and hip/nvcc (see below),
                              # gfx<XXX> for hip/amd, or spirv for hip/spirv
                              # defaults: sm_75 (cuda, hip/nvcc), gfx906 (hip/amd),
                              # spirv (hip/spirv)
-D GPU_DEBUG=value            # enable debug code in the GPU package library,
                              # mostly useful for developers
                              # value = yes or no (default)
-D HIP_PATH=value             # value = path to HIP installation. Must be set if
                              # GPU_API=HIP
-D HIP_ARCH=value             # deprecated, use GPU_ARCH instead (still accepted,
                              # but prints a deprecation warning)
-D HIP_USE_DEVICE_SORT=value # enables GPU sorting
                              # value = yes (default) or no
-D CUDPP_OPT=value            # use GPU binning with CUDA (should be off for modern GPUs)
                              # enables CUDA Performance Primitives, must be "no" for
                              # CUDA_MPS_SUPPORT=yes
                              # value = yes or no (default)
-D CUDA_MPS_SUPPORT=value     # enables some tweaks required to run with active
                              # nvidia-cuda-mps daemon
                              # value = yes or no (default)
-D CUDA_BUILD_MULTIARCH=value  # enables building CUDA kernels for all supported GPU
                               # architectures
                               # value = yes (default) or no
-D USE_STATIC_OPENCL_LOADER=value  # downloads/includes OpenCL ICD loader library,
                                   # no local OpenCL headers/libs needed
                                   # value = yes (default) or no
```

The GPU package supports 3 precision modes: single, double, and mixed, with the latter being the default. In the double precision mode, atom positions, forces and energies are stored, computed and accumulated in double precision. In the mixed precision mode, forces and energies are accumulated in double precision while atom coordinates are stored and arithmetic operations are performed in single precision. In the single precision mode, all are stored, executed and accumulated in single precision.

To specify the precision mode (output to the screen before LAMMPS runs for verification), set `GPU_PREC` to one of `single`, `double`, or `mixed`.

Some accelerators or OpenCL implementations only support single precision. This mode should be used with care and appropriate validation as the errors can scale with system size in this implementation. This can be useful for accelerating test runs when setting up a simulation for production runs on another machine. In the case where only single precision is supported, either LAMMPS must be compiled with `-DFFT_SINGLE` to use PPPM with GPU acceleration or GPU acceleration should be disabled for PPPM (e.g. suffix off or `pair/only` as described in the LAMMPS documentation).

Changed in version 4Jul2026.

`GPU_ARCH` is the canonical architecture setting for all `GPU_API` back ends. The back end specific `HIP_ARCH` (for

GPU_API=hip) variable is still accepted for backward compatibility, but its use is deprecated and prints a warning.

For GPU_API=cuda and GPU_API=hip with HIP_PLATFORM=nvcc, the GPU_ARCH settings for different GPU hardware are as follows:

- sm_30 for Kepler (supported since CUDA 5 and until CUDA 10.x)
- sm_35 or sm_37 for Kepler (supported since CUDA 5 and until CUDA 11.x)
- sm_50 or sm_52 for Maxwell (supported since CUDA 6)
- sm_60 or sm_61 for Pascal (supported since CUDA 8)
- sm_70 for Volta (supported since CUDA 9)
- sm_75 for Turing (supported since CUDA 10)
- sm_80 or sm_86 for Ampere (supported since CUDA 11, sm_86 since CUDA 11.1)
- sm_89 for Lovelace (supported since CUDA 11.8)
- sm_90 or sm_90a for Hopper (supported since CUDA 12.0)
- sm_100 or sm_103 for Blackwell B100/B200/B300 (supported since CUDA 12.8)
- sm_120 for Blackwell B20x/B40 (supported since CUDA 12.8)
- sm_121 for Blackwell (supported since CUDA 12.9)

A more detailed list can be found, for example, at Wikipedia's CUDA article

CMake can detect which version of the CUDA toolkit is used and thus will try to include support for **all** major GPU architectures supported by this toolkit. Thus the GPU_ARCH setting is merely an optimization, to have code for the preferred GPU architecture directly included rather than having to wait for the JIT compiler of the CUDA driver to translate it. This behavior can be turned off (e.g. to speed up compilation) by setting CUDA_ENABLE_MULTIARCH to no.

When compiling for CUDA or HIP with CUDA, version 8.0 or later of the CUDA toolkit is required and a GPU architecture of Kepler or later, which must *also* be supported by the CUDA toolkit in use **and** the CUDA driver in use. When compiling for OpenCL, OpenCL version 1.2 or later is required and the GPU must be supported by the GPU driver and OpenCL runtime bundled with the driver.

Please note that the GPU library accesses the CUDA driver library directly, so it needs to be linked with the CUDA driver library (libcuda.so) that ships with the Nvidia driver. If you are compiling LAMMPS on the head node of a GPU cluster, this library may not be installed, so you may need to copy it over from one of the compute nodes (best into this directory). Recent versions of the CUDA toolkit starting from CUDA 9 provide a dummy libcuda.so library (typically under $(CUDA_HOME)/lib64/stubs), that can be used for linking. If you are compiling LAMMPS with the -D BUILD_SHARED_LIBS=ON setting, you may get to see ld: warning: libcuda.so.1, needed by liblammps.so.0, not found. This may be worked around by also setting: -DCMAKE_EXE_LINKER_FLAGS=-Wl, --unresolved-symbols=ignore-in-shared-libs.

To support the CUDA multi-process server (MPS) you can set the define -DCUDA_MPS_SUPPORT. Please note that in this case you must **not** use the CUDA performance primitives and thus set the variable CUDPP_OPT to empty.

If you are compiling for OpenCL, the default setting is to download, build, and link with a static OpenCL ICD loader library and standard OpenCL headers. This way no local OpenCL development headers or library needs to be present and only OpenCL compatible drivers need to be installed to use OpenCL. If this is not desired, you can set USE_STATIC_OPENCL_LOADER to no.

If GERYON_NUMA_FISSION is defined at build time (-DGPU_DEBUG=no), LAMMPS will consider separate NUMA nodes on GPUs or accelerators as separate devices. For example, a 2-socket CPU would appear as two separate devices for OpenCL (and LAMMPS would require two MPI processes to use both sockets with the GPU library - each with its own device ID as output by ocl_get_devices). OpenCL version 1.2 or later is required.

If you are compiling with HIP, note that before running CMake you will have to set appropriate environment variables. Some variables such as `HCC_AMDGPU_TARGET` (for ROCm <= 4.0) or `CUDA_PATH` are necessary for `hipcc` and the linker to work correctly.

When compiling for HIP ROCm, GPU sorting with `-D HIP_USE_DEVICE_SORT=on` requires installing the `hipcub` library (https://github.com/ROCmSoftwarePlatform/hipCUB). Setting `-DDOWNLOAD_CUB=yes` will download and compile CUB.

The GPU library has some multi-thread support using OpenMP. If LAMMPS is built with `-D BUILD_OMP=on` this will also be enabled.

**Note**

Some Clang-based tool chains - in particular `hipcc` from ROCm - do not ship the `omp.h` header in the compiler's own resource directory. When building with `-D BUILD_OMP=on` and such a compiler, host code that includes `<omp.h>` would fail to compile even though the `-fopenmp` flag is accepted. CMake detects this case and adds the `omp.h` from a matching version of the system Clang installation as a fallback include path (using `-idirafter` so it does not shadow other headers). If no matching `omp.h` can be found automatically, you may need to add the directory containing it yourself, for example with `-D CMAKE_CXX_FLAGS=-idirafter/usr/lib/clang/<version>/include`.

For a debug build, set `GPU_DEBUG` to be `yes`.

Added in version 3Aug2022.

Using the CHIP-SPV implementation of HIP is supported. It allows one to run HIP code on Intel GPUs via the OpenCL or Level Zero back ends. To use CHIP-SPV, you must set `-DHIP_USE_DEVICE_SORT=OFF` in your CMake command-line as CHIP-SPV does not yet support hipCUB. As of Summer 2022, the use of HIP for Intel GPUs is experimental. You should only use this option in preparations to run on Aurora system at Argonne.

```
# AMDGPU target (ROCm <= 4.0)
export HIP_PLATFORM=hcc
export HIP_PATH=/path/to/HIP/install
export HCC_AMDGPU_TARGET=gfx906
cmake -D PKG_GPU=on -D GPU_API=HIP -D HIP_ARCH=gfx906 -D CMAKE_CXX_COMPILER=hipcc ..
make -j 4
```

```
# AMDGPU target (ROCm >= 4.1)
export HIP_PLATFORM=amd
export HIP_PATH=/path/to/HIP/install
cmake -D PKG_GPU=on -D GPU_API=HIP -D HIP_ARCH=gfx906 -D CMAKE_CXX_COMPILER=hipcc ..
make -j 4
```

```
# CUDA target (not recommended, use GPU_API=cuda)
# !!! DO NOT set CMAKE_CXX_COMPILER !!!
export HIP_PLATFORM=nvcc
export HIP_PATH=/path/to/HIP/install
export CUDA_PATH=/usr/local/cuda
cmake -D PKG_GPU=on -D GPU_API=HIP -D HIP_ARCH=sm_70 ..
make -j 4
```

```
# SPIR-V target (Intel GPUs)
export HIP_PLATFORM=spirv
export HIP_PATH=/path/to/HIP/install
```

(continues on next page)

(continued from previous page)

```
export CMAKE_CXX_COMPILER=<hipcc/clang++>
cmake -D PKG_GPU=on -D GPU_API=HIP ..
make -j 4
```

### 3.8.4 KIM package

To build with this package, the KIM API v2.0+ must be downloaded and built on your system. See Obtaining KIM Models to learn how to install the KIM API, as well as how to install any models you wish to use afterward. See the list of all KIM models here: https://openkim.org/browse/models

If you would like to use the *kim query* command, you also need to have libcurl installed with the matching development headers and the curl-config tool.

If you would like to use the *kim property* command, you need to build LAMMPS with the PYTHON package installed and linked to Python 3.6 or later. See the *PYTHON package build info* for more details on this. After successfully building LAMMPS with Python, you also need to install the `kim-property` Python package, which can be easily done using *pip* as `pip install kim-property`, or from the *conda-forge* channel as `conda install kim-property` if LAMMPS is built in Conda. More detailed information is available at: kim-property installation.

**CMake build**

```
-D DOWNLOAD_KIM=value           # download OpenKIM API v2 for build
                                # value = no (default) or yes
-D LMP_DEBUG_CURL=value         # set libcurl verbose mode on/off
                                # value = off (default) or on
-D LMP_NO_SSL_CHECK=value       # tell libcurl to not verify the peer
                                # value = no (default) or yes
-D KIM_EXTRA_UNITTESTS=value    # enables extra unit tests
                                # value = no (default) or yes
```

If `DOWNLOAD_KIM` is set to `yes` (or `on`), the KIM API library will be downloaded and built inside the CMake build directory. Note that in most cases it is recommended that you do not use this option, and instead provide a KIM API installation yourself before building LAMMPS. If the KIM library is already installed on your system (in a location where CMake cannot find it), you may need to set the `PKG_CONFIG_PATH` environment variable so that libkim-api can be found, or run the command `source kim-api-activate`. If CMake cannot find the KIM API when configuring for the first time (or after clearing the CMake cache), the default value of the `DOWNLOAD_KIM` option will be `yes`.

Extra unit tests can only be available if they are explicitly requested (`KIM_EXTRA_UNITTESTS` is set to `yes` (or `on`)) and the prerequisites are met. See *KIM Extra unit tests* for more details on this.

**Traditional make**

Changed in version 10Sep2025.

The KIM package no longer supports the traditional make build. You need to build LAMMPS with CMake.

#### Debugging OpenKIM web queries in LAMMPS

If `LMP_DEBUG_CURL` is set, the libcurl verbose mode will be turned on, and any libcurl calls within the KIM web query display a lot of information about libcurl operations. You hardly ever want this set in production use, you will almost always want this when you debug or report problems.

The libcurl library performs peer SSL certificate verification by default. This verification is done using a CA certificate store that the SSL library can use to make sure the peer's server certificate is valid. If SSL reports an error ("certificate verify failed") during the handshake and thus refuses further communicate with that server, you can set `LMP_NO_SSL_CHECK` to override that behavior. When LAMMPS is compiled with `LMP_NO_SSL_CHECK` set, libcurl does not verify the peer and connection attempts will succeed regardless of the names in the certificate. This option is insecure. As an alternative, you can specify your own CA cert path by setting the environment variable `CURL_CA_BUNDLE` to the path of your choice. A call to the KIM web query would get this value from the environment variable.

#### KIM Extra unit tests (CMake only)

During development, testing, or debugging, if *unit testing* is enabled in LAMMPS, one can also enable extra tests on *KIM commands* by setting the `KIM_EXTRA_UNITTESTS` to `yes` (or `on`).

Enabling the extra unit tests have some requirements,

- It requires to have internet access.
- It requires to have libcurl installed with the matching development headers and the curl-config tool.
- It requires to build LAMMPS with the PYTHON package installed and linked to Python 3.6 or later. See the *PYTHON package build info* for more details on this.
- It requires to have `kim-property` Python package installed, which can be easily done using *pip* as `pip install kim-property`, or from the *conda-forge* channel as `conda install kim-property` if LAMMPS is built in Conda. More detailed information is available at: kim-property installation.
- It is also necessary to install the following KIM models:
  - `EAM_Dynamo_MendelevAckland_2007v3_Zr__MO_004835508849_001`
  - `EAM_Dynamo_ErcolessiAdams_1994_Al__MO_123629422045_006`
  - `LennardJones612_UniversalShifted__MO_959249795837_003` (this model is an example model automatically built with the API unless explicitly disabled)

  See Obtaining KIM Models to learn how to install a pre-built binary of the OpenKIM Repository of Models or see Installing KIM Models to learn how to install the specific KIM models.

---

### 3.8.5 KOKKOS package

Using the KOKKOS package requires choosing several settings. You have to select whether you want to compile with parallelization on the host and whether you want to include offloading of calculations to a device (e.g. a GPU). The default setting is to have no host parallelization and no device offloading. In addition, you can select the hardware architecture to select the instruction set. Since most hardware is backward compatible, you may choose settings for an older architecture to have an executable that will run on this and newer architectures.

**Note**

If you run Kokkos on a different GPU architecture than what LAMMPS was compiled with, there will be a delay during device initialization while the just-in-time compiler is recompiling all GPU kernels for the new hardware. This is, however, only supported for GPUs of the **same** major hardware version and different minor hardware versions, e.g. 5.0 and 5.2 but not 5.2 and 6.0. LAMMPS will abort with an error message indicating a mismatch, if the major version differs.

The settings discussed below have been tested with LAMMPS and are confirmed to work. Kokkos is an active project with ongoing improvements and projects working on including support for additional architectures. More information on Kokkos can be found on the Kokkos GitHub project.

### Available Architecture settings

These are the possible choices for the Kokkos architecture ID. They must be specified in uppercase.

| Arch-ID | HOST or GPU | Description |
|---|---|---|
| NATIVE | HOST | Local machine |
| AMDAVX | HOST | AMD chip |
| ARMV80 | HOST | ARMv8.0 Compatible CPU |
| ARMV81 | HOST | ARMv8.1 Compatible CPU |
| ARMV84 | HOST | ARMv8.4 Compatible CPU |
| ARMV84_SVE | HOST | Generic ARMv8.4 with SVE support (-march=armv8.4-a+sve) |
| ARMV8_THUNDERX | HOST | ARMv8 Cavium ThunderX CPU |
| ARMV8_THUNDERX2 | HOST | ARMv8 Cavium ThunderX2 CPU |
| A64FX | HOST | ARMv8.2 with SVE Support |
| ARMV9_GRACE | HOST | ARMv9 NVIDIA Grace CPU |
| SNB | HOST | Intel Sandy/Ivy Bridge CPUs |
| HSW | HOST | Intel Haswell CPUs |
| BDW | HOST | Intel Broadwell Xeon E-class CPUs |
| ICL | HOST | Intel Ice Lake Client CPUs (AVX512) |
| ICX | HOST | Intel Ice Lake Xeon Server CPUs (AVX512) |
| SKL | HOST | Intel Skylake Client CPUs |
| SKX | HOST | Intel Skylake Xeon Server CPUs (AVX512) |
| KNC | HOST | Intel Knights Corner Xeon Phi |
| KNL | HOST | Intel Knights Landing Xeon Phi |
| SPR | HOST | Intel Sapphire Rapids Xeon Server CPUs (AVX512) |
| POWER8 | HOST | IBM POWER8 CPUs |
| POWER9 | HOST | IBM POWER9 CPUs |
| ZEN | HOST | AMD Zen architecture |
| ZEN2 | HOST | AMD Zen2 architecture |
| ZEN3 | HOST | AMD Zen3 architecture |
| ZEN4 | HOST | AMD Zen4 architecture |
| ZEN5 | HOST | AMD Zen5 architecture |
| RISCV_SG2042 | HOST | SG2042 (RISC-V) CPUs |
| RISCV_RVA22V | HOST | RVA22V (RISC-V) CPUs |
| RISCV_U74MC | HOST | U74MC (RISC-V) CPUs |

continues on next page

Table 1 – continued from previous page

| | | |
|---|---|---|
| KEPLER30 | GPU | NVIDIA Kepler generation CC 3.0 |
| KEPLER32 | GPU | NVIDIA Kepler generation CC 3.2 |
| KEPLER35 | GPU | NVIDIA Kepler generation CC 3.5 |
| KEPLER37 | GPU | NVIDIA Kepler generation CC 3.7 |
| MAXWELL50 | GPU | NVIDIA Maxwell generation CC 5.0 |
| MAXWELL52 | GPU | NVIDIA Maxwell generation CC 5.2 |
| MAXWELL53 | GPU | NVIDIA Maxwell generation CC 5.3 |
| PASCAL60 | GPU | NVIDIA Pascal generation CC 6.0 |
| PASCAL61 | GPU | NVIDIA Pascal generation CC 6.1 |
| VOLTA70 | GPU | NVIDIA Volta generation CC 7.0 |
| VOLTA72 | GPU | NVIDIA Volta generation CC 7.2 |
| TURING75 | GPU | NVIDIA Turing generation CC 7.5 |
| AMPERE80 | GPU | NVIDIA Ampere generation CC 8.0 |
| AMPERE86 | GPU | NVIDIA Ampere generation CC 8.6 |
| AMPERE87 | GPU | NVIDIA Ampere generation CC 8.7 |
| ADA89 | GPU | NVIDIA Ada generation CC 8.9 |
| HOPPER90 | GPU | NVIDIA Hopper generation CC 9.0 |
| BLACKWELL100 | GPU | NVIDIA Blackwell generation CC 10.0 |
| BLACKWELL103 | GPU | NVIDIA Blackwell generation CC 10.3 |
| BLACKWELL120 | GPU | NVIDIA Blackwell generation CC 12.0 |
| BLACKWELL121 | GPU | NVIDIA Blackwell generation CC 12.1 |
| AMD_GFX906 | GPU | AMD GPU MI50/60 |
| AMD_GFX908 | GPU | AMD GPU MI100 |
| AMD_GFX90A | GPU | AMD GPU MI200 |
| AMD_GFX940 | GPU | AMD GPU MI300 |
| AMD_GFX942 | GPU | AMD GPU MI300 |
| AMD_GFX942_APU | GPU | AMD APU MI300A |
| AMD_GFX950 | GPU | AMD GPU MI350 |
| AMD_GFX1030 | GPU | AMD GPU V620/W6800 |
| AMD_GFX1100 | GPU | AMD GPU RX7900XTX |
| AMD_GFX1103 | GPU | AMD APU Phoenix |
| INTEL_GEN | GPU | SPIR64-based devices, e.g. Intel GPUs, using JIT |
| INTEL_DG1 | GPU | Intel Iris XeMAX GPU |
| INTEL_GEN9 | GPU | Intel GPU Gen9 |
| INTEL_GEN11 | GPU | Intel GPU Gen11 |
| INTEL_GEN12LP | GPU | Intel GPU Gen12LP |
| INTEL_XEHP | GPU | Intel GPU Xe-HP |
| INTEL_PVC | GPU | Intel GPU Ponte Vecchio |
| INTEL_DG2 | GPU | Intel GPU DG2 |

This list was last updated for version 5.1.0 of the Kokkos library.

**Basic CMake build settings**

For multicore CPUs using OpenMP, set these 2 variables.

```
-D Kokkos_ARCH_HOSTARCH=yes  # HOSTARCH = HOST from list above
-D Kokkos_ENABLE_OPENMP=yes
-D BUILD_OMP=yes
```

Please note that enabling OpenMP for KOKKOS requires that OpenMP is also *enabled for the rest of LAMMPS*.

For Intel KNLs using OpenMP, set these variables:

```
-D Kokkos_ARCH_KNL=yes
-D Kokkos_ENABLE_OPENMP=yes
```

For NVIDIA GPUs using CUDA, set these variables:

```
-D Kokkos_ARCH_HOSTARCH=yes   # HOSTARCH = HOST from list above
-D Kokkos_ARCH_GPUARCH=yes    # GPUARCH = GPU from list above
-D Kokkos_ENABLE_CUDA=yes
-D Kokkos_ENABLE_OPENMP=yes
```

This will also enable executing FFTs on the GPU, either via the internal KISSFFT library, or - by preference - with the cuFFT library bundled with the CUDA toolkit, depending on whether CMake can identify its location.

For AMD or NVIDIA GPUs using HIP, set these variables:

```
-D Kokkos_ARCH_HOSTARCH=yes   # HOSTARCH = HOST from list above
-D Kokkos_ARCH_GPUARCH=yes    # GPUARCH = GPU from list above
-D Kokkos_ENABLE_HIP=yes
-D Kokkos_ENABLE_OPENMP=yes
```

This will enable FFTs on the GPU, either by the internal KISSFFT library or with the hipFFT wrapper library, which will call out to the platform-appropriate vendor library: rocFFT on AMD GPUs or cuFFT on NVIDIA GPUs.

For Intel GPUs using SYCL, set these variables:

```
-D Kokkos_ARCH_HOSTARCH=yes   # HOSTARCH = HOST from list above
-D Kokkos_ARCH_GPUARCH=yes    # GPUARCH = GPU from list above
-D Kokkos_ENABLE_SYCL=yes
-D Kokkos_ENABLE_OPENMP=yes
-D FFT_KOKKOS=MKL_GPU
```

This will enable FFTs on the GPU using the oneMKL library.

To simplify compilation, seven preset files are included in the `cmake/presets` folder, `kokkos-serial.cmake`, `kokkos-openmp.cmake`, `kokkos-cuda.cmake`, `kokkos-cuda-nowrapper.cmake`, `kokkos-hip.cmake`, `kokkos-sycl-nvidia.cmake`, and `kokkos-sycl-intel.cmake`. They will enable the KOKKOS package and enable some hardware choices. For GPU support those preset files may need to be customized to match the hardware used. For some platforms, e.g. CUDA, the Kokkos library will try to auto-detect a suitable configuration. So to compile with CUDA device parallelization with some common packages enabled, you can do the following:

```
mkdir build-kokkos-cuda
cd build-kokkos-cuda
cmake -C ../cmake/presets/basic.cmake \
      -C ../cmake/presets/kokkos-cuda-nowrapper.cmake ../cmake
cmake --build .
```

The `kokkos-openmp.cmake` preset can be combined with any of the others, but it is not possible to combine multiple GPU acceleration settings (CUDA, HIP, SYCL) into a single executable.

**Basic traditional make settings**

Changed in version 11Feb2026.

The KOKKOS package no longer supports the traditional make build. You need to build LAMMPS with CMake.

### Advanced KOKKOS compilation settings

There are other allowed options when building with the KOKKOS package that can improve performance or assist in debugging or profiling. Below are some examples that may be useful in combination with LAMMPS. For the full list (which keeps changing as the Kokkos package itself evolves), please consult the Kokkos library documentation.

As alternative to using multi-threading via OpenMP (`-DKokkos_ENABLE_OPENMP=on`) it is also possible to use Posix threads directly (`-DKokkos_ENABLE_PTHREAD=on`). While binding of threads to individual or groups of CPU cores is managed in OpenMP with environment variables, you need assistance from either the "hwloc" or "libnuma" library for the Pthread thread parallelization option. To enable use with CMake: `-DKokkos_ENABLE_HWLOC=on` or `-DKokkos_ENABLE_LIBNUMA=on`.

The CMake option `-DKokkos_ENABLE_LIBRT=on` enables the use of a more accurate timer mechanism on many Unix-like platforms for internal profiling.

The CMake option `-DKokkos_ENABLE_DEBUG=on` enables printing of run-time debugging information that can be useful. It also enables runtime bounds checking on Kokkos data structures. As to be expected, enabling this option will negatively impact the performance and thus is only recommended when developing a Kokkos-enabled style in LAMMPS.

The CMake option `-DKokkos_ENABLE_CUDA_UVM=on` enables the use of CUDA "Unified Virtual Memory" (UVM) in Kokkos. UVM allows to transparently use RAM on the host to supplement the memory used on the GPU (with some performance penalty) and thus enables running larger problems that would otherwise not fit into the RAM on the GPU.

Added in version 10Sep2025.

The CMake option `-D KOKKOS_PREC=value` sets the floating point precision of the calculations, where `value` can be one of: `double` (FP64, default) or `mixed` (FP64 for accumulation of forces, energy, and virial, FP32 otherwise) or `single` (FP32). When using reduced precision (single or mixed), the simulation and its results should be carefully checked to ensure it is stable and that, for example, energy is sufficiently well conserved. Using a lower floating point precision works best when simulating homogeneous bulk systems because those have the best error cancellation. Using `mixed` precision provides most of the performance advantages of using single precision while performing the steps most relevant for accuracy in double precision.

Added in version 10Sep2025.

The CMake option `-D KOKKOS_LAYOUT=value` sets the array layout of Kokkos views (e.g. forces, velocities, etc.) on GPUs, where `value` can be one of: `legacy` (mostly LayoutRight, default) or `default` (mostly LayoutLeft). Using the default layout (LayoutLeft) can give speedup on GPUs for some models, but a slowdown for others. LayoutRight is always used for positions on GPUs since it has been found to be faster, and when compiling exclusively for CPUs.

### 3.8.6 LEPTON package

To build with this package, you must build the Lepton library which is included in the LAMMPS source distribution in the `lib/lepton` folder.

**CMake build**

This is the recommended build procedure for using Lepton in LAMMPS. No additional settings are normally needed besides `-D PKG_LEPTON=yes`.

On x86 hardware the Lepton library will also include a just-in-time compiler for faster execution. This is auto detected but can be explicitly disabled by setting `-D LEPTON_ENABLE_JIT=no` (or enabled by setting it to yes).

**Traditional make**

Changed in version 10Sep2025.

The LEPTON package no longer supports the traditional make build. You need to build LAMMPS with CMake.

---

### 3.8.7 MACHDYN package

To build with this package, you must download the Eigen3 library. Eigen3 is a template library, so you do not need to build it.

**CMake build**

```
-D DOWNLOAD_EIGEN3              # download Eigen3, value = no (default) or yes
-D EIGEN3_INCLUDE_DIR=path      # path to Eigen library (only needed if a
                                # custom location)
```

If `DOWNLOAD_EIGEN3` is set, the Eigen3 library will be downloaded and inside the CMake build directory. If the Eigen3 library is already on your system (in a location where CMake cannot find it), set `EIGEN3_INCLUDE_DIR` to the directory the `Eigen3` include file is in.

**Traditional make**

Changed in version 10Sep2025.

The MACHDYN package no longer supports the traditional make build. You need to build LAMMPS with CMake.

---

### 3.8.8 ML-IAP package

Building the ML-IAP package requires including the *ML-SNAP* package. There will be an error message if this requirement is not satisfied. Using the *mliappy* model also requires enabling Python support, which in turn requires to include the *PYTHON* package **and** requires to have the cython software installed and with it a working `cythonize` command. This feature requires compiling LAMMPS with Python version 3.6 or later.

**CMake build**

```
-D MLIAP_ENABLE_PYTHON=value    # enable mliappy model (default is autodetect)
```

Without this setting, CMake will check whether it can find a suitable Python version and the `cythonize` command and choose the default accordingly. During the build procedure the provided .pyx file(s) will be automatically translated to C++ code and compiled. Please do **not** run `cythonize` manually in the `src/ML-IAP` folder, as that can lead to compilation errors if Python support is not enabled. If you did it by accident, please remove the generated .cpp and .h files.

**Traditional make**

The build uses the `lib/python/Makefile.mliap_python` file in the compile/link process to add a rule to update the files generated by the `cythonize` command in case the corresponding .pyx file(s) were modified. You may need to modify `lib/python/Makefile.lammps` if the LAMMPS build fails.

To enable building the ML-IAP package with Python support enabled, you need to add `-DMLIAP_PYTHON` to the `LMP_INC` variable in your machine makefile. You may have to manually run the `cythonize` command on .pyx file(s) in the `src` folder, if this is not automatically done during installing the ML-IAP package. Please do **not** run `cythonize` in the `src/ML-IAP` folder, as that can lead to compilation errors if Python support is not enabled. If you did this by accident, please remove the generated .cpp and .h files.

### 3.8.9 OPT package

**CMake build**

No additional settings are needed besides `-D PKG_OPT=yes`

**Traditional make**

The compiler flag `-restrict` must be used to build LAMMPS with the OPT package when using Intel compilers. It should be added to the `CCFLAGS` line of your `Makefile.machine`. See `src/MAKE/OPTIONS/Makefile.opt` for an example.

### 3.8.10 PYTHON package

Building with the PYTHON package requires you have a the Python development headers and library available on your system, which needs to be Python version 3.6 or later. See `lib/python/README` for additional details.

**CMake build**

```
-D Python_EXECUTABLE=path    # path to Python executable to use
```

Without this setting, CMake will guess the default Python version on your system. To use a different Python version, you can either create a virtualenv, activate it and then run cmake. Or you can set the Python_EXECUTABLE variable to specify which Python interpreter should be used. Note note that you will also need to have the development headers installed for this version, e.g. python3-devel.

**Traditional make**

Changed in version 10Sep2025.

The PYTHON package no longer supports the traditional make build. You need to build LAMMPS with CMake.

### 3.8.11 VORONOI package

To build with this package, you must download and build the Voro++ library or install a binary package provided by your operating system.

**CMake build**

```
-D DOWNLOAD_VORO=value     # download Voro++ for build
                           # value = no (default) or yes
-D VORO_LIBRARY=path       # Voro++ library file
                           # (only needed if at custom location)
-D VORO_INCLUDE_DIR=path   # Voro++ include directory
                           # (only needed if at custom location)
```

If `DOWNLOAD_VORO` is set, the Voro++ library will be downloaded and built inside the CMake build directory. If the Voro++ library is already on your system (in a location CMake cannot find it), `VORO_LIBRARY` is the filename (plus path) of the Voro++ library file, not the directory the library file is in. `VORO_INCLUDE_DIR` is the directory the Voro++ include file is in.

**Traditional make**

Changed in version 10Sep2025.

The VORONOI package no longer supports the traditional make build. You need to build LAMMPS with CMake.

### 3.8.12 ADIOS package

The ADIOS package requires the ADIOS I/O library, version 2.3.1 or newer. Make sure that you have ADIOS built either with or without MPI to match if you build LAMMPS with or without MPI. ADIOS compilation settings for LAMMPS are automatically detected, if the PATH and LD_LIBRARY_PATH environment variables have been updated for the local ADIOS installation and the instructions below are followed for the respective build systems.

**CMake build**

```
-D ADIOS2_DIR=path        # path is where ADIOS 2.x is installed
-D PKG_ADIOS=yes
```

**Traditional make**

Turn on the ADIOS package before building LAMMPS. If the ADIOS 2.x software is installed in PATH, there is nothing else to do:

```
make yes-adios
```

otherwise, set ADIOS2_DIR environment variable when turning on the package:

```
ADIOS2_DIR=path make yes-adios    # path is where ADIOS 2.x is installed
```

### 3.8.13 APIP package

The APIP package depends on the library of the *ML-PACE* package. The code for the library can be found at: https://github.com/ICAMS/lammps-user-pace/

**CMake build**

No additional settings are needed besides `-D PKG_APIP=yes` and `-D PKG_ML-PACE=yes`. One can use a local version of the ML-PACE library instead of automatically downloading the library as described *here*.

**Traditional make**

Changed in version 10Sep2025.

The APIP package no longer supports the traditional make build. You need to build LAMMPS with CMake.

### 3.8.14 COLVARS package

This package enables the use of the Colvars module included in the LAMMPS source distribution.

**CMake build**

This is the recommended build procedure for using Colvars in LAMMPS. No additional settings are normally needed besides `-D PKG_COLVARS=yes`. The following CMake variables are available.

```
-D PKG_COLVARS=yes          # enable the package itself
-D COLVARS_LEPTON=yes       # use the Lepton library for custom expression␣
↪(on by defaul)
```

```
-D COLVARS_DEBUG=no         # eneable debugging message (verbose, off by␣
↪default)
```

**Traditional make**

Changed in version 10Sep2025.

The COLVARS package no longer supports the traditional make build. You need to build LAMMPS with CMake.

---

### 3.8.15 ELECTRODE package

This package depends on the KSPACE package.

**CMake build**

```
-D PKG_ELECTRODE=yes          # enable the package itself
-D PKG_KSPACE=yes             # the ELECTRODE package requires KSPACE
-D USE_INTERNAL_LINALG=value  #
```

Features in the ELECTRODE package are dependent on code in the KSPACE package so the latter one *must* be enabled.

The ELECTRODE package also requires LAPACK (and BLAS) and CMake can identify their locations and pass that info to the ELECTRODE build script. But on some systems this may cause problems when linking or the dependency is not desired. Try enabling `USE_INTERNAL_LINALG` in those cases to use the bundled linear algebra library and work around the limitation.

**Traditional make**

The ELECTRODE package no longer supports the traditional make build. You need to build LAMMPS with CMake.

---

### 3.8.16 MBX package

Added in version 11Feb2026.

This package requires the MBX library that can be downloaded and built either before LAMMPS is built or as part of the LAMMPS compilation. The code for the library can be found at: https://github.com/paesanilab/MBX/

Instead of including the MBX package directly into LAMMPS, it is also possible to skip this step and build the MBX package as a plugin using the CMake script files in the `examples/PACKAGE/mbx/plugin` folder and then load this plugin at runtime with the *plugin command*.

**CMake build**

Changed in version 30Mar2026: Replaced MD5 checksums with SHA-256

By default the MBX library will be downloaded from the git repository and built automatically when the MBX package is enabled with `-D PKG_MBX=yes`. The location for the sources may be customized by setting the variable `MBXLIB_URL` when configuring with CMake (e.g. to use a local archive on machines without internet access). Since CMake checks the validity of the archive using a SHA-256 checksum you may also need to set the `MBXLIB_SHA256` variable to the corresponding checksum (e.g. computed with `sha256sum`) if you provide a different library version than what is downloaded automatically.

**Traditional make**

The MBX package does not support the traditional make build. You need to build LAMMPS with CMake to use it.

### 3.8.17 ML-PACE package

This package requires a library that can be downloaded and built in lib/pace or somewhere else, which must be done before building LAMMPS with this package. The code for the library can be found at: https://github.com/ICAMS/lammps-user-pace/

Instead of including the ML-PACE package directly into LAMMPS, it is also possible to skip this step and build the ML-PACE package as a plugin using the CMake script files in the `examples/PACKAGE/pace/plugin` folder and then load this plugin at runtime with the *plugin command*.

**CMake build**

Changed in version 30Mar2026: Replaced MD5 checksums with SHA-256

By default the library will be downloaded from the git repository and built automatically when the ML-PACE package is enabled with `-D PKG_ML-PACE=yes`. The location for the sources may be customized by setting the variable `PACELIB_URL` when configuring with CMake (e.g. to use a local archive on machines without internet access). Since CMake checks the validity of the archive using a SHA-256 checksum you may also need to set the `PACELIB_SHA256` variable to the corresponding checksum (e.g. computed with `sha256sum`) if you provide a different library version than what is downloaded automatically.

**Traditional make**

Changed in version 10Sep2025.

The ML-PACE package no longer supports the traditional make build. You need to build LAMMPS with CMake.

### 3.8.18 ML-POD package

**CMake build**

No additional settings are needed besides `-D PKG_ML-POD=yes`.

**Traditional make**

Changed in version 10Sep2025.

The ML-POD package no longer supports the traditional make build. You need to build LAMMPS with CMake.

---

### 3.8.19 ML-QUIP package

To build with this package, you must download and build the QUIP library. It can be obtained from GitHub. For support of GAP potentials, additional files with specific licensing conditions need to be downloaded and configured. The automatic download will from within CMake will download the non-commercial use version.

**CMake build**

```
-D DOWNLOAD_QUIP=value        # download QUIP library for build
                              # value = no (default) or yes
-D QUIP_LIBRARY=path          # path to libquip.a
                              # (only needed if a custom location)
-D USE_INTERNAL_LINALG=value # Use the internal linear algebra library
                              # instead of LAPACK
                              # value = no (default) or yes
```

CMake will try to download and build the QUIP library from GitHub, if it is not found on the local machine. This requires to have git installed. It will use the same compilers and flags as used for compiling LAMMPS. Currently this is only supported for the GNU and the Intel compilers. Set the `QUIP_LIBRARY` variable if you want to use a previously compiled and installed QUIP library and CMake cannot find it.

The QUIP library requires LAPACK (and BLAS) and CMake can identify their locations and pass that info to the QUIP build script. But on some systems this triggers a (current) limitation of CMake and the configuration will fail. Try enabling `USE_INTERNAL_LINALG` in those cases to use the bundled linear algebra library and work around the limitation.

**Traditional make**

Changed in version 10Sep2025.

The ML-QUIP package no longer supports the traditional make build. You need to build LAMMPS with CMake.

---

### 3.8.20 PLUMED package

Before building LAMMPS with this package, you must first build PLUMED. PLUMED can be built as part of the LAMMPS build or installed separately from LAMMPS using the generic PLUMED installation instructions. The PLUMED package has been tested to work with Plumed versions 2.4.x, to 2.9.x and will error out, when trying to run calculations with a different version of the Plumed kernel.

PLUMED can be linked into MD codes in three different modes: static, shared, and runtime. With the "static" mode, all the code that PLUMED requires is linked statically into LAMMPS. LAMMPS is then fully independent from the PLUMED installation, but you have to rebuild/relink it in order to update the PLUMED code inside it. With the "shared" linkage mode, LAMMPS is linked to a shared library that contains the PLUMED code. This library should preferably be installed in a globally accessible location. When PLUMED is linked in this way the same library can be used by multiple MD packages. Furthermore, the PLUMED library LAMMPS uses can be updated without the need for a recompile of LAMMPS for as long as the shared PLUMED library is ABI-compatible.

The third linkage mode is "runtime" which allows the user to specify which PLUMED kernel should be used at runtime by using the PLUMED_KERNEL environment variable. This variable should point to the location of the libplumedKernel.so dynamical shared object, which is then loaded at runtime. This mode of linking is particularly convenient for doing PLUMED development and comparing multiple PLUMED versions as these sorts of comparisons can be done without recompiling the hosting MD code. All three linkage modes are supported by LAMMPS on selected operating systems (e.g. Linux) and using either CMake or traditional make build. The "static" mode should be the most portable, while the "runtime" mode support in LAMMPS makes the most assumptions about operating system and compiler environment. If one mode does not work, try a different one, switch to a different build system, consider a global PLUMED installation or consider downloading PLUMED during the LAMMPS build.

Instead of including the PLUMED package directly into LAMMPS, it is also possible to skip this step and build the PLUMED package as a plugin using the CMake script files in the `examples/PACKAGE/plumed/plugin` folder and then load this plugin at runtime with the *plugin command*.

**CMake build**

When the `-D PKG_PLUMED=yes` flag is included in the cmake command you must ensure that *the GNU Scientific Library (GSL) <https://www.gnu.org/software/gsl/>* is installed in locations that are accessible in your environment. There are then two additional variables that control the manner in which PLUMED is obtained and linked into LAMMPS.

```
-D DOWNLOAD_PLUMED=value   # download PLUMED for build
                           # value = no (default) or yes
-D PLUMED_MODE=value       # Linkage mode for PLUMED
                           # value = static (default), shared,
                           #         or runtime
```

If `DOWNLOAD_PLUMED` is set to `yes`, the PLUMED library will be downloaded (the version of PLUMED that will be downloaded is hard-coded to a vetted version of PLUMED, usually a recent stable release version) and built inside the CMake build directory. If `DOWNLOAD_PLUMED` is set to "no" (the default), CMake will try to detect and link to an installed version of PLUMED. For this to work, the PLUMED library has to be installed into a location where the `pkg-config` tool can find it or the `PKG_CONFIG_PATH` environment variable has to be set up accordingly. PLUMED should be installed in such a location if you compile it using the default make; make install commands.

The `PLUMED_MODE` setting determines the linkage mode for the PLUMED library. The allowed values for this flag are "static" (default), "shared", or "runtime". If you want to switch the linkage mode, just re-run CMake with a different setting. For a discussion of PLUMED linkage modes, please see above. When `DOWNLOAD_PLUMED` is enabled the static linkage mode is recommended.

**Traditional make**

Changed in version 10Sep2025.

The PLUMED package no longer supports the traditional make build. You need to build LAMMPS with CMake.

### 3.8.21 H5MD package

To build with this package you must have the HDF5 software package installed on your system, which should include the h5cc compiler and the HDF5 library.

**CMake build**

No additional settings are needed besides `-D PKG_H5MD=yes`.

This should auto-detect the H5MD library on your system. Several advanced CMake H5MD options exist if you need to specify where it is installed. Use the ccmake (terminal window) or cmake-gui (graphical) tools to see these options and set them interactively from their user interfaces.

**Traditional make**

Changed in version 10Sep2025.

The H5MD package no longer supports the traditional make build. You need to build LAMMPS with CMake.

### 3.8.22 ML-HDNNP package

To build with the ML-HDNNP package it is required to download and build the external n2p2 library `v2.1.4` (or higher). The LAMMPS build process offers an automatic download and compilation of *n2p2* or allows you to choose the installation directory of *n2p2* manually. Please see the boxes below for the CMake and traditional build system for detailed information.

In case of a manual installation of *n2p2* you only need to build the *n2p2* core library `libnnp` and interface library `libnnpif`. When using GCC it should suffice to execute `make libnnpif` in the *n2p2* `src` directory. For more details please see `lib/hdnnp/README` and the n2p2 build documentation.

**CMake build**

```
-D DOWNLOAD_N2P2=value    # download n2p2 for build
                          # value = no (default) or yes
-D N2P2_DIR=path          # n2p2 base directory
                          # (only needed if a custom location)
```

If `DOWNLOAD_N2P2` is set, the *n2p2* library will be downloaded and built inside the CMake build directory. If the *n2p2* library is already on your system (in a location CMake cannot find it), set the `N2P2_DIR` to path where *n2p2* is located. If *n2p2* is located directly in `lib/hdnnp/n2p2` it will be automatically found by CMake.

**Failure to build n2p2 due to git branch names**

Some script code inside the *n2p2* library build processes the current branch name used by git and that will fail for LAMMPS repository branch names containing the forward slash '/' character, for example: `user/update-n2p2`. The workaround is to change the (local) branch name, e.g. for the given example with: `git branch -m user/update-n2p2 user_update-n2p2`

**Traditional make**

Changed in version 10Sep2025.

The ML-HDNNP package no longer supports the traditional make build. You need to build LAMMPS with CMake.

### 3.8.23 ML-RUNNER package

The ML-RUNNER package provides an interface to the RuNNer (Ruhr University Neural Network Energy Representation) library for high-dimensional neural network potentials (HDNNP).

**Prerequisites**

- **Fortran Compiler:** Since the RuNNer library is written in Fortran, a working Fortran compiler must be available on your system and detectable by CMake.
- **BLAS/LAPACK:** RuNNer requires BLAS and LAPACK libraries for linear algebra operations.
- **FFT Library:** RuNNer uses an FFT library for electrostatic calculations (3G/4G). It can use either MKL or FFTW3 and the choice is imported from the KSPACE package configuration. When using FFTW3 also FFTW3 threading (-DFFT_FFTW_THREADING=ON) needs to be enabled which is usually auto-detected. These restrictions are needed so that LAMMPS and RuNNer use the same FFT library settings and link to the same library. In both cases the Fortran 03 wrapper file `fftw3.f03` must be in the Fortran compiler include path.

**Building RuNNer**

By default, the LAMMPS build process automatically downloads and compiles the RuNNer library as a static library. Alternatively, you can point LAMMPS to a pre-compiled version already present on your system.

**CMake build**

**Basic Options:**

```
-D PKG_ML-RUNNER=yes       # yes (default): Download and build RuNNer␣
↪automatically.
-D DOWNLOAD_RUNNER=yes     # yes (default): clone the stable version of the␣
↪official RuNNer repo.
                           # no: Use a pre-compiled RuNNer library.
-D RUNNER_SHARED_LIB=yes    # no: (default): Look for static library (.a).
                            # yes: Look for shared library (.so).
```

**Manual Library Configuration (if DOWNLOAD_RUNNER=no):**

```
-D RUNNER_LIB_DIR=path      # Directory containing the RuNNer library.
                            # (default: $HOME/.local/lib)
-D RUNNER_LIB_NAME=name     # Filename of the library (without extension).
                            # (default: libRuNNer_mpi)
```

**FFT Library Selection:**

The build system uses the FFT selection from the KSPACE package. Only MKL and FFTW3 are currently supported

```
-D FFT=value                # FFTW3 or MKL
-D FFT_MKL_THREADS=yes      # required with MKL (default)
-D FFT_FFTW_THREADS=yes     # required with FFTW3 (default)
```

**Traditional make**

The ML-RUNNER package does not support the traditional make build system. You must build LAMMPS with CMake.

**Detailed Option Table**

| Option | Description | Default |
|---|---|---|
| DOWNLOAD_RUNNER | Download and build RuNNer from source | yes |
| RUNNER_LIB_DIR | Path to a pre-installed RuNNer library | $HOME/.local/lib |
| RUNNER_LIB_NAME | Name of the RuNNer library file without extension | libRuNNer_mpi |
| RUNNER_SHARED_LIB | Link against a shared RuNNer library | yes |
| FFT | FFT library to use (FFTW3 or MKL) | auto-detected |
| FFT_MKL_THREADS | Use multi-threaded MKL FFT | yes |
| FFT_FFTW_THREADS | Use multi-threaded FFTW | yes |

### 3.8.24 INTEL package

This package optimizes styles for x86 CPUs. You should also typically *install the OPENMP package*, as it can be used in tandem with the INTEL package to good effect, as explained on the *INTEL package* page.

When using Intel compilers version 16.0 or later is required. You can also use the GNU or Clang compilers and they will provide performance improvements over regular styles and OPENMP styles, but less so than with the Intel compilers. Please also note, that some compilers have been found to apply memory alignment constraints incompletely or incorrectly and thus can cause segmentation faults in otherwise correct code when using features from the INTEL package.

**CMake build**

```
-D INTEL_LRT_MODE=value # value = threads, none, or c++17
```

**Traditional make**

Choose compiler flags in Makefile.machine via the following settings. See `src/MAKE/OPTIONS/Makefile.intel_cpu*` files for examples and `src/INTEL/README` for additional information.

```
OPTFLAGS =  -xHost -O2 -fp-model fast=2 -no-prec-div -qoverride-limits -qopt-
↪zmm-usage=high
```

```
CCFLAGS =   -g -qopenmp -DLAMMPS_MEMALIGN=64 -fno-alias -ansi-alias -
↪restrict $(OPTFLAGS)
LINKFLAGS = -g -qopenmp $(OPTFLAGS)
LIB =       -ltbbmalloc
```

In Long-range thread mode (LRT) a modified verlet style is used, that operates the Kspace calculation in a separate thread concurrently to other calculations. This has to be enabled in the *package intel* command at runtime. With the setting "threads" it used the pthreads library, while "c++17" will use the built-in thread support of C++17 compilers. The option "none" skips compilation of this feature. The default is to use "threads" if pthreads is available and otherwise "none".

Best performance is achieved with Intel hardware, Intel compilers, as well as the Intel TBB and MKL libraries. However, the code also compiles, links, and runs with other compilers / hardware and without TBB and MKL.

---

### 3.8.25 MDI package

**CMake build**

```
-D DOWNLOAD_MDI=value    # download MDI Library for build
                         # value = no (default) or yes
```

**Traditional make**

Changed in version 10Sep2025.

The MDI package no longer supports the traditional make build. You need to build LAMMPS with CMake.

---

### 3.8.26 MISC package

The *fix imd* style in this package can be run either synchronously (communication with IMD clients is done in the main process) or asynchronously (the fix spawns a separate thread that can communicate with IMD clients concurrently to the LAMMPS execution).

**CMake build**

```
-D LAMMPS_ASYNC_IMD=value  # Run IMD server asynchronously
                           # value = no (default) or yes
```

**Traditional make**

To enable asynchronous mode the `-DLAMMPS_ASYNC_IMD` define needs to be added to the `LMP_INC` variable in the `Makefile.machine` you are using. For example:

```
LMP_INC = -DLAMMPS_ASYNC_IMD -DLAMMPS_MEMALIGN=64
```

### 3.8.27 MOLFILE package

**CMake build**

```
-D MOLFILE_INCLUDE_DIR=path    # (optional) path where VMD molfile
                               # plugin headers are installed
-D PKG_MOLFILE=yes
```

Using `-D PKG_MOLFILE=yes` enables the package, and setting `-D MOLFILE_INCLUDE_DIR` allows to provide a custom location for the molfile plugin header files. These should match the ABI of the plugin files used, and thus one typically sets them to include folder of the local VMD installation in use. LAMMPS ships with a couple of default header files that correspond to a popular VMD version, usually the latest release.

**Traditional make**

Changed in version 10Sep2025.

The MOLFILE package no longer supports the traditional make build. You need to build LAMMPS with CMake.

### 3.8.28 NETCDF package

To build with this package you must have the NetCDF library installed on your system.

**CMake build**

No additional settings are needed besides `-D PKG_NETCDF=yes`.

This should auto-detect the NETCDF library if it is installed on your system at standard locations. Several advanced CMake NETCDF options exist if you need to specify where it was installed. Use the `ccmake` (terminal window) or `cmake-gui` (graphical) tools to see these options and set them interactively from their user interfaces.

**Traditional make**

Changed in version 10Sep2025.

The NETCDF package no longer supports the traditional make build. You need to build LAMMPS with CMake.

### 3.8.29 OPENMP package

**CMake build**

No additional settings are required besides `-D PKG_OPENMP=yes`. If CMake detects OpenMP compiler support, the OPENMP code will be compiled with multi-threading support enabled, otherwise as optimized serial code.

**Traditional make**

To enable multi-threading support in the OPENMP package (and other styles supporting OpenMP) the following compile and link flags must be added to your Makefile.machine file. See `src/MAKE/OPTIONS/Makefile.omp` for an example.

```
CCFLAGS: -fopenmp               # for GNU and Clang Compilers
CCFLAGS: -qopenmp -restrict     # for Intel compilers on Linux
LINKFLAGS: -fopenmp             # for GNU and Clang Compilers
LINKFLAGS: -qopenmp             # for Intel compilers on Linux
```

For other platforms and compilers, please consult the documentation about OpenMP support for your compiler.

**Adding OpenMP support on macOS**

Apple offers the Xcode package and IDE for compiling software on macOS, so you have likely installed it to compile LAMMPS. Their compiler is based on Clang, but while it is capable of processing OpenMP directives, the necessary header files and OpenMP runtime library are missing. The R developers have figured out a way to build those in a compatible fashion. One can download them from https://mac.r-project.org/openmp/. Simply adding those files as instructed enables the Xcode C++ compiler to compile LAMMPS with `-D BUILD_OMP=yes`.

### 3.8.30 QMMM package

For using LAMMPS to do QM/MM simulations via the QMMM package you need to build LAMMPS as a library. A LAMMPS executable with *fix qmmm* included can be built, but will not be able to do a QM/MM simulation on as such. You must also build a QM code - currently only Quantum ESPRESSO (QE) is supported - and create a new executable which links LAMMPS and the QM code together. Details are given in the `lib/qmmm/README` file. It is also recommended to read the instructions for *linking with LAMMPS as a library* for background information. This requires compatible Quantum Espresso and LAMMPS versions. The current interface and makefiles have last been verified to work in February 2020 with Quantum Espresso versions 6.3 to 6.5.

**CMake build**

When using CMake, building a LAMMPS library is required and it is recommended to build a shared library, since any libraries built from the sources in the *lib* folder (including the essential libqmmm.a) are not included in the static LAMMPS library and are (currently) not installed, while their code is included in the shared LAMMPS library. Thus a typical command to configure building LAMMPS for QMMM would be:

```
cmake -C ../cmake/presets/basic.cmake -D PKG_QMMM=yes \
    -D BUILD_LIB=yes -DBUILD_SHARED_LIBS=yes ../cmake
```

After completing the LAMMPS build and also configuring and compiling Quantum ESPRESSO with external library support (via "make couple"), go back to the `lib/qmmm` folder and follow the instructions on the README file to build the combined LAMMPS/QE QM/MM executable (pwqmmm.x) in the `lib/qmmm` folder.

**Traditional make**

Changed in version 10Sep2025.

The QMMM package no longer supports the traditional make build. You need to build LAMMPS with CMake.

### 3.8.31 RHEO package

This package depends on the BPM package.

**CMake build**

```
-D PKG_RHEO=yes                # enable the package itself
-D PKG_BPM=yes                 # the RHEO package requires BPM
-D USE_INTERNAL_LINALG=value   # prefer internal LAPACK if true
```

Some features in the RHEO package are dependent on code in the BPM package so the latter one *must* be enabled as well.

The RHEO package also requires LAPACK (and BLAS) and CMake can identify their locations and pass that info to the RHEO build script. But on some systems this may cause problems when linking or the dependency is not desired. By using the setting `-D USE_INTERNAL_LINALG=yes` when running the CMake configuration, you will select compiling and linking the bundled linear algebra library and work around the limitations.

**Traditional make**

Changed in version 10Sep2025.

The RHEO package no longer supports the traditional make build. You need to build LAMMPS with CMake.

### 3.8.32 SCAFACOS package

To build with this package, you must download and build the ScaFaCoS Coulomb solver library

**CMake build**

```
-D DOWNLOAD_SCAFACOS=value     # download ScaFaCoS for build, value = no␣
↪(default) or yes
-D SCAFACOS_LIBRARY=path       # ScaFaCos library file (only needed if at␣
↪custom location)
-D SCAFACOS_INCLUDE_DIR=path   # ScaFaCoS include directory (only needed if␣
↪at custom location)
```

If `DOWNLOAD_SCAFACOS` is set, the ScaFaCoS library will be downloaded and built inside the CMake build directory. If the ScaFaCoS library is already on your system (in a location CMake cannot find it), `SCAFACOS_LIBRARY` is the filename (plus path) of the ScaFaCoS library file, not the directory the library file is in. `SCAFACOS_INCLUDE_DIR` is the directory the ScaFaCoS include file is in.

**Traditional make**

Changed in version 10Sep2025.

The SCAFACOS package no longer supports the traditional make build. You need to build LAMMPS with CMake.

---

### 3.8.33 VTK package

To build with this package you must have the VTK library installed on your system.

**CMake build**

No additional settings are needed besides `-D PKG_VTK=yes`.

This should auto-detect the VTK library if it is installed on your system at standard locations. Several advanced VTK options exist if you need to specify where it was installed. Use the `ccmake` (terminal window) or `cmake-gui` (graphical) tools to see these options and set them interactively from their user interfaces.

**Traditional make**

Changed in version 10Sep2025.

The VTK package no longer supports the traditional make build. You need to build LAMMPS with CMake.

## 3.9 Build the LAMMPS documentation

Depending on how you obtained LAMMPS and whether you have built the manual yourself, this directory has a number of subdirectories and files. Here is a list with descriptions:

```
README            # brief info about the documentation
src               # content files for LAMMPS documentation
html              # HTML version of the LAMMPS manual (see html/Manual.html)
utils             # tools and settings for building the documentation
lammps.1          # man page for the lammps command
msi2lmp.1         # man page for the msi2lmp command
Manual.pdf        # large PDF version of entire manual
LAMMPS.epub       # Manual in ePUB e-book format
LAMMPS.mobi       # Manual in MOBI e-book format
docenv            # virtualenv folder for processing the manual sources
doctrees          # temporary data from processing the manual
doxygen           # doxygen configuration and output
.gitignore        # list of files and folders to be ignored by git
doxygen-warn.log # logfile with warnings from running doxygen
github-development-workflow.md   # notes on the LAMMPS development workflow
```

If you downloaded LAMMPS as a tarball from the LAMMPS website, the html folder and the PDF files should be included.

If you downloaded LAMMPS from the public git repository, then the HTML and PDF files are not included. You can build the HTML or PDF files yourself, by typing `make html` or `make pdf` in the `doc` folder. This requires various tools and files. Some of them have to be installed (see below). For the rest the build process will attempt to download and install them into a python virtual environment and local folders.

A current version of the manual (latest feature release, that is the state of the *release* branch) is available online at: https://docs.lammps.org/. A version of the manual corresponding to the ongoing development (that is the state of the *develop* branch) is available online at: https://docs.lammps.org/latest/ A version of the manual corresponding to the latest stable LAMMPS release (that is the state of the *stable* branch) is available online at: https://docs.lammps.org/stable/

### 3.9.1 Build using GNU make

The LAMMPS manual is written in reStructuredText format which can be translated to different output format using the Sphinx document generator tool. It also incorporates programmer documentation extracted from the LAMMPS C++ sources through the Doxygen program. Currently the translation to HTML, PDF (via LaTeX), ePUB (for many e-book readers) and MOBI (for Amazon Kindle readers) are supported. For that to work a Python interpreter version 3.8 or later, the `doxygen` tools and internet access to download additional files and tools are required. This download is usually only required once or after the documentation folder is returned to a pristine state with `make clean-all`. You can also upgrade those packages to their latest available versions with `make upgrade`.

For the documentation build a python virtual environment is set up in the folder `doc/docenv` and various python packages are installed into that virtual environment via the `pip` tool. For rendering embedded LaTeX code also the MathJax JavaScript engine needs to be downloaded. If you need to pass additional options to the pip commands to work

(e.g. to use a web proxy or to point to additional SSL certificates) you can set them via the PIP_OPTIONS environment variable or uncomment and edit the PIP_OPTIONS setting at beginning of the makefile.

The actual translation is then done via make commands in the doc folder. The following make commands are available:

```
make html          # generate HTML in html dir using Sphinx
make pdf           # generate PDF  as Manual.pdf using Sphinx and PDFLaTeX
make epub          # generate LAMMPS.epub in ePUB format using Sphinx
make mobi          # generate LAMMPS.mobi in MOBI format using ebook-convert

make fasthtml      # generate approximate HTML in fasthtml dir using pandoc

make upgrade       # upgrade sphinx, extensions, and dependencies to latest supported␣
↪versions
make clean         # remove intermediate RST files created by HTML build
make clean-all     # remove entire build folder and any cached data
make upgrade       # upgrade the python packages in the virtual environment

make check         # run all checks listed in this block
make anchor_check  # check for duplicate anchor labels
make style_check   # check for complete and consistent style lists
make package_check # check for complete and consistent package lists
make char_check    # check for non-ASCII characters
make role_check    # check for misformatted role keywords

make link_check    # check for broken external URLs
make spelling      # spell-check the manual
```

### 3.9.2 Build using CMake

It is also possible to create the HTML version (and **only** the HTML version) of the manual within the *CMake build directory*. The reason for this option is to include the installation of the HTML manual pages into the "install" step when installing LAMMPS after the CMake build via `cmake --build . --target install`. The documentation build is included in the default build target, but can also be requested independently with `cmake --build . --target doc`. If you need to pass additional options to the pip commands to work (e.g. to use a web proxy or to point to additional SSL certificates) you can set them via the PIP_OPTIONS environment variable.

```
-D BUILD_DOC=value       # yes or no (default)
```

### 3.9.3 Prerequisites for HTML

To run the HTML documentation build tool chain, Python 3.8 or later, git, doxygen, and virtualenv have to be installed locally. Here are instructions for common setups:

**Ubuntu**

```
sudo apt-get install git doxygen
```

**Fedora or RHEL/AlmaLinux/RockyLinux (8.x or later)**

```
sudo dnf install git doxygen
```

**macOS**

*Python 3*

If Python 3 is not available on your macOS system, you can download the latest Python 3 macOS package from https://www.python.org and install it. This will install both Python 3 and pip3.

### 3.9.4 Prerequisites for PDF

In addition to the tools needed for building the HTML format manual, a working LaTeX installation with support for PDFLaTeX and a selection of LaTeX styles/packages are required. Apart from LaTeX packages that are usually installed by default, the following packages are required:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| amsmath | anysize | babel | capt-of | cmap | dvipng | ellipse | fncychap | fontawesome | framed | geometry |
| gyre | hyperref | hyphcap | needspace | pict2e | times | tabulary | titlesec | upquote | wrapfig | xindy |

To run the PDFLaTeX translation the `latexmk` script needs to be installed as well.

### 3.9.5 Prerequisites for ePUB and MOBI

In addition to the tools needed for building the HTML format manual, a working LaTeX installation with a few addon LaTeX packages as well as the `dvipng` tool are required to convert embedded math expressions transparently into embedded images.

For converting the generated ePUB file to a MOBI format file (for e-book readers, like Kindle, that cannot read ePUB), you also need to have the `ebook-convert` tool from the "calibre" software installed. https://calibre-ebook.com/ Typing `make mobi` will first create the ePUB file and then convert it. On the Kindle readers in particular, you also have support for PDF files, so you could download and view the PDF version as an alternative.

### 3.9.6 Instructions for Developers

When adding new styles or options to the LAMMPS code, corresponding documentation is required and either existing files in the `src` folder need to be updated or new files added. These files are written in reStructuredText markup for translation with the Sphinx tool.

#### Testing your contribution

Before contributing any documentation, please check that both the HTML and the PDF format documentation can translate without errors and that there are no spelling issues. This is done with `make html`, `make pdf`, and `make spelling`, respectively.

### Fast and approximate translation to HTML

Translating the full manual to HTML or PDF can take a long time. Thus there is a fast and approximate way to translate the reStructuredText to HTML as a quick-n-dirty way of checking your manual page.

This translation uses Pandoc instead of Sphinx and thus all special Sphinx features (cross-references, advanced tables, embedding of Python docstrings or doxygen documentation, and so on) will not render correctly. Most embedded math should render correctly. This is a **very fast** way to check the syntax and layout of a documentation file translated to HTML while writing or updating it.

To translate **all** manual pages, you can type `make fasthtml` at the command line. The translated HTML files are then in the `fasthtml` folder. All subsequent `make fasthtml` commands will only translate `.rst` files that have been changed. The `make fasthtml` command can be parallelized with make using the *-j* flag. You can also directly translate only individual pages: e.g. to translate only the `doc/src/pair_lj.rst` page type `make fasthtml/pair_lj.html`

After writing the documentation is completed, you will still need to verify with `make html` and `make pdf` that it translates correctly in both formats.

### Tests for consistency, completeness, and other known issues

Please also check the output to the console for any warnings or problems. There will be multiple tests run automatically:

- A test for correctness of all anchor labels and their references
- A test that all LAMMPS packages (= folders with sources in `lammps/src`) are documented and listed. A typical warning shows the name of the folder with the suspected new package code and the documentation files where they need to be listed:

```
Found 88 packages
Package NEWPACKAGE missing in Packages_list.rst
Package NEWPACKAGE missing in Packages_details.rst
```

- A test that only standard, printable ASCII text characters are used. This runs the command `env LC_ALL=C grep -n '[^ -~]' src/*.rst` and thus prints all offending lines with filename and line number prepended to the screen. Special characters like Greek letters ($\alpha$ $\sigma$ $\varepsilon$), super- or subscripts ($x^2$ $U_{LJ}$), mathematical expressions ($\frac{1}{2}N$ $x \to \infty$), or the Angstrom symbol (Å) should be typeset with embedded LaTeX (like this `:math:`\alpha \sigma \epsilon``, `:math:`x^2 \mathrm{E}_{LJ}``, `:math:`\frac{1}{2}\mathrm{N} x\to\infty``, or `:math:`\AA``).
- Embedded LaTeX is rendered in HTML output with MathJax and in PDF output by passing the embedded text to LaTeX. Some care has to be taken, though, since there are limitations which macros and features can be used in either mode, so it is recommended to always check whether any new or changed documentation does translate and render correctly with either output.
- A test whether all styles are documented and listed in their respective overview pages. A typical output with warnings looks like this:

```
Parsed style names w/o suffixes from C++ tree in ../src:
   Angle styles:      21    Atom styles:       24
   Body styles:        3    Bond styles:       17
   Command styles:    41    Compute styles:   143
   Dihedral styles:   16    Dump styles:       26
   Fix styles:       223    Improper styles:   13
   Integrate styles:   4    Kspace styles:     15
   Minimize styles:    9    Pair styles:      234
   Reader styles:      4    Region styles:      8
Compute style entry newcomp is missing or incomplete in Commands_compute.rst
Compute style entry newcomp is missing or incomplete in compute.rst
```

(continues on next page)

(continued from previous page)

```
Fix style entry newfix is missing or incomplete in Commands_fix.rst
Fix style entry newfix is missing or incomplete in fix.rst
Pair style entry new is missing or incomplete in Commands_pair.rst
Pair style entry new is missing or incomplete in pair_style.rst
Found 6 issue(s) with style lists
```

In addition, there is the option to run a spellcheck on the entire manual with `make spelling`. This requires a library called enchant. To avoid printing out *false positives* (e.g. keywords, names, abbreviations) those can be added to the file `lammps/doc/utils/sphinx-config/false_positives.txt`.

## 3.10 Notes for building LAMMPS on Windows


- *General remarks*
- *Running Linux on Windows*
- *Using GNU GCC ported to Windows*
- *Using Visual Studio*
- *Using Intel oneAPI compilers and libraries*
- *Using a cross-compiler*


### 3.10.1 General remarks

LAMMPS is developed and tested primarily on Linux machines. The vast majority of HPC clusters and supercomputers today run on Linux as well. While portability to other platforms is desired, it is not always achieved. That is sometimes due to non-portable code in LAMMPS itself, but more often due to portability limitations of external libraries and tools required to build a specific feature or package. The LAMMPS developers are dependent on LAMMPS users giving feedback and providing assistance in resolving portability issues. This is particularly true for compiling LAMMPS on Windows, since this platform has significant differences in some low-level functionality. As of LAMMPS version 14 December 2021, large parts of LAMMPS can be compiled natively with the Microsoft Visual C++ Compilers. As of LAMMPS version 31 May 2022, also the Intel oneAPI compilers can compile large parts of LAMMPS natively on Windows. This is mostly facilitated by using the *Platform abstraction functions* in the `platform` namespace and CMake.

Before trying to build LAMMPS on Windows yourself, please consider the pre-compiled Windows installer packages and see if they are sufficient for your needs.

### 3.10.2 Running Linux on Windows

If it is necessary for you to compile LAMMPS on a Windows machine (e.g. because it is your main desktop), please also consider using a virtual machine software and compile and run LAMMPS in a Linux virtual machine, or - if you have a sufficiently up-to-date Windows 10 or Windows 11 installation - consider using the Windows subsystem for Linux. This optional Windows feature allows you to run the bash shell of a Linux system (Ubuntu by default) from within Windows and from there on, you can pretty much use that shell like you are running on a regular Ubuntu Linux machine (e.g. installing software via apt-get and more). For more details on that, please see *this tutorial*.

### 3.10.3 Using a GNU GCC ported to Windows

One option for compiling LAMMPS on Windows natively is to install a Bash shell, Unix shell utilities, Perl, Python, GNU make, and a GNU compiler ported to Windows. The Cygwin package provides a unix/linux interface to low-level Windows functions, so LAMMPS can be compiled on Windows. The necessary (minor) modifications to LAMMPS are included, but may not always up-to-date for recently added functionality and the corresponding new code. A machine makefile for using cygwin for the old build system is provided. Using CMake for this mode of compilation is untested and not likely to work.

When compiling for Windows do **not** set the `-DLAMMPS_MEMALIGN` define in the LMP_INC makefile variable and add `-lwsock32 -lpsapi` to the linker flags in LIB makefile variable. Try adding `-static-libgcc` or `-static` or both to the linker flags when your resulting LAMMPS Windows executable complains about missing .dll files. The CMake configuration should set this up automatically, but is untested.

In case of problems, you are recommended to contact somebody with experience in using Cygwin. If you do come across portability problems requiring changes to the LAMMPS source code, or figure out corrections yourself, please report them on the LAMMPS forum at MatSci, or file them as an issue or pull request on the LAMMPS GitHub project.

### 3.10.4 Using Microsoft Visual Studio

Following the integration of the *platform namespace* into the LAMMPS code base, portability of LAMMPS for native compilation on Windows using Visual Studio has been significantly improved. This has been tested with Visual Studio 2019 (aka version 16) and Visual Studio 2022 (aka version 17). We strongly recommend using Visual Studio 2022 version 17.1 or later. Not all features and packages in LAMMPS are currently supported out of the box, but a preset `cmake/presets/windows.cmake` is provided that contains the packages that have been compiled successfully so far. You **must** use the CMake based build procedure, since there is no support for GNU make or the Unix shell utilities required for the GNU make build procedure.

It is possible to use both the integrated CMake support of the Visual Studio IDE or use an external CMake installation (e.g. downloaded from cmake.org) to create build files and compile LAMMPS from the command-line.

Compilation via command-line and unit tests are checked automatically for the LAMMPS development branch through GitHub Actions.

> **Note**
>
> Versions of Visual Studio before version 17.1 may scan the entire LAMMPS source tree and likely miss the correct master `CMakeLists.txt` and get confused since there are multiple files of that name in different folders but none in top level folder.

Please note, that for either approach CMake will create a so-called *"multi-configuration" build environment*, and the commands for building and testing LAMMPS must be adjusted accordingly.

The LAMMPS cmake folder contains a `CMakeSettings.json` file with build configurations for MSVC compilers and the MS provided Clang compiler package in Debug and Release mode.

To support running in parallel you can compile with OpenMP enabled using the OPENMP package or install Microsoft MPI (including the SDK) and compile LAMMPS with MPI enabled.

> **Note**
>
> This is work in progress and you should contact the LAMMPS developers via GitHub or the LAMMPS forum at MatSci, if you have questions or LAMMPS specific problems.

### 3.10.5 Using Intel oneAPI Compilers and Libraries

Added in version 31May2022.

After installing the Intel oneAPI base toolkit and the HPC toolkit, it is also possible to compile large parts of LAMMPS natively on Windows using Intel compilers. The HPC toolkit provides two sets of C/C++ and Fortran compilers: the so-called "classic" compilers (`icl.exe` and `ifort.exe`) and newer, LLVM based compilers (`icx.exe` and `ifx.exe`). In addition to the compilers and their dependent modules, also the thread building blocks (TBB) and the math kernel library (MKL) need to be installed. Two presets (`cmake/presets/windows-intel-llvm.cmake` and `cmake/presets/windows-intel-classic.cmake`) are provided for selecting the LLVM based or classic compilers, respectively. The preset `cmake/presets/windows.cmake` enables compatible packages that are not dependent on additional features or libraries. You **must** use the CMake based build procedure and use Ninja as build tool. For compiling from the command prompt, thus both CMake and Ninja-build binaries must be installed. It is also possible to use Visual Studio, if it is started (`devenv.exe`) from a command prompt that has the Intel oneAPI compilers enabled. The Visual Studio settings file in the `cmake` folder contains configurations for both compiler variants in debug and release settings. Those will use the CMake and Ninja binaries bundled with Visual Studio, thus a separate installation is not required.

**Known Limitations**

In addition to portability issues with several packages and external libraries, the classic Intel compilers are currently not able to compile the googletest libraries and thus enabling the `-DENABLE_TESTING` option will result in compilation failure. The LLVM based compilers are compatible.

**Note**

This is work in progress and you should contact the LAMMPS developers via GitHub or the LAMMPS forum at MatSci, if you have questions or LAMMPS specific problems.

### 3.10.6 Using a cross-compiler

If you need to provide custom LAMMPS binaries for Windows, but do not need to do the compilation on Windows, please consider using a Linux to Windows cross-compiler. This is how currently the Windows binary packages are created by the LAMMPS developers. Because of that, this is probably the currently best tested and supported way to build LAMMPS executables for Windows. A CMake preset selecting all packages compatible with this cross-compilation build is provided. The GPU package can only be compiled with OpenCL support. To compile with MPI support, a pre-compiled library and the corresponding header files are required. When building with CMake the matching package will be downloaded automatically, but MPI support has to be explicitly enabled with `-DBUILD_MPI=on`.

Please keep in mind, though, that this only applies to **compiling** LAMMPS. Whether the resulting binaries do work correctly is rarely tested by the LAMMPS developers. We instead rely on the feedback of the users of these pre-compiled LAMMPS packages for Windows. We will try to resolve issues to the best of our abilities if we become aware of them. However this is subject to time constraints and focus on HPC platforms.

## 3.11 Notes for saving disk space when building LAMMPS from source

LAMMPS is a large software project with a large number of source files, extensive documentation, and a large collection of example files. When downloading LAMMPS by cloning the git repository from GitHub this will by default also download the entire commit history since September 2006. Compiling LAMMPS will add the storage requirements of the compiled object files and libraries to the tally.

In a user account on an HPC cluster with filesystem quotas or in other environments with restricted disk space capacity it may be needed to reduce the storage requirements. Here are some suggestions:

- Create a so-called shallow repository by cloning only the last commit instead of the full project history by using `git clone git@github.com:lammps/lammps --depth=1 --branch=develop`. This reduces the downloaded size to about half. With `--depth=1` it is not possible to check out different versions/branches of LAMMPS, using `--depth=1000` will make multiple recent versions available at little extra storage needs (the entire git history had nearly 30,000 commits in fall 2021).
- Download a tar archive from either the download section on the LAMMPS homepage or from the LAMMPS releases page on GitHub these will not contain the git history at all.
- Build LAMMPS without the debug flag (remove `-g` from the machine makefile or use `-DCMAKE_BUILD_TYPE=Release`) or use the `strip` command on the LAMMPS executable when no more debugging would be needed. The strip command may also be applied to the LAMMPS shared library. The static library may be deleted entirely.
- Delete compiled object files and libraries after copying the LAMMPS executable to a permanent location. When using the traditional build process, one may use `make clean-<machine>` or `make clean-all` to delete object files in the src folder. For CMake based builds, one may use `make clean` or just delete the entire build folder.
- The folders containing the documentation tree (doc), the examples (examples) are not needed to build and run LAMMPS and can be safely deleted. Some files in the potentials folder are large and may be deleted, if not needed. The largest of those files (occupying about 120 MBytes combined) will only be downloaded on demand, when the corresponding package is installed.
- When using the CMake build procedure, the compilation can be done on a (local) scratch storage that will not count toward the quota. A local scratch file system may offer the additional benefit of speeding up creating object files and linking with libraries compared to a networked file system. Also with CMake (and unlike with the traditional make) it is possible to compile LAMMPS executables with different settings and packages included from the same source tree since all the configuration information is stored in the build folder. So it is not necessary to have multiple copies of LAMMPS.

## 3.12 Development build options

The build procedures in LAMMPS offers a few extra options which are useful during development, testing or debugging.


- *Monitor compilation flags (CMake only)*
- *Report missing and unneeded '#include' statements (CMake only)*
- *Address, Leak, Undefined Behavior, and Thread Sanitizer Support (CMake only)*
- *Code Coverage and Unit Testing (CMake only)*
  - *Unit tests for force styles*
  - *Unit tests for timestepping related fixes*
  - *Use custom linker for faster link times when ENABLE_TESTING is active*
  - *Tests for other components and utility functions*
  - *Collect and visualize code coverage metrics*
- *Coding style utilities*
- *Clang-format support*

- *GitHub command-line interface*

### 3.12.1 Monitor compilation flags (CMake only)

Sometimes it is necessary to verify the complete sequence of compilation flags generated by the CMake build. To enable a more verbose output during compilation you can use the following option.

```
-D CMAKE_VERBOSE_MAKEFILE=value    # value = no (default) or yes
```

Another way of doing this without reconfiguration is calling make with variable VERBOSE set to 1:

```
make VERBOSE=1
```

### 3.12.2 Report missing and unneeded '#include' statements (CMake only)

The conventions for how and when to use and order include statements in LAMMPS are documented in *LAMMPS programming style*. To assist with following these conventions one can use the Include What You Use tool. This tool is still under development and for large and complex projects like LAMMPS there are some false positives, so suggested changes need to be verified manually. It is recommended to use at least version 0.16, which has much fewer incorrect reports than earlier versions. To install the IWYU toolkit, you need to have the clang compiler **and** its development package installed. Download the IWYU version that matches the version of the clang compiler, configure, build, and install it.

The necessary steps to generate the report can be enabled via a CMake variable during CMake configuration.

```
-D ENABLE_IWYU=value    # value = no (default) or yes
```

This will check if the required binary (include-what-you-use or iwyu) and python script (iwyu-tool or iwyu_tool or iwyu_tool.py) can be found in the path. The analysis can then be started with:

```
make iwyu
```

This may first run some compilation, as the analysis is dependent on recording all commands required to do the compilation.

### 3.12.3 Address, Leak, Undefined Behavior, and Thread Sanitizer Support (CMake only)

Compilers such as GCC and Clang support generating instrumented binaries which use different sanitizer libraries to detect problems in the code during run-time. They can detect issues like:

- memory leaks
- undefined behavior
- data races

Please note that this kind of instrumentation usually comes with a performance hit (but much less than using tools like Valgrind with a more low level approach). To enable these features, additional compiler flags need to be added to the compilation and linking stages. This is done through setting the `ENABLE_SANITIZER` variable during configuration. Examples:

```
-D ENABLE_SANITIZER=none        # no sanitizer active (default)
-D ENABLE_SANITIZER=address     # enable address sanitizer / memory leak checker
-D ENABLE_SANITIZER=hwaddress   # enable hardware assisted address sanitizer / memory␣
↪leak checker
-D ENABLE_SANITIZER=leak        # enable memory leak checker (only)
-D ENABLE_SANITIZER=undefined   # enable undefined behavior sanitizer
-D ENABLE_SANITIZER=thread      # enable thread sanitizer
```

### 3.12.4 Code Coverage and Unit Testing (CMake only)

The LAMMPS code is subject to multiple levels of automated testing when pull requests are submitted to the LAMMPS repository on GitHub:

- Coding style compliance (see *Coding style utilities*)
- Integration testing (i.e. whether the code compiles on multiple platforms and with a variety of compilers and settings),
- Unit testing (i.e. whether certain functions or classes of the code produce the expected results for given inputs),
- Run testing (i.e. whether selected input decks can run to completion without crashing for multiple configurations),
- Regression testing (i.e. whether selected input examples reproduce the same results over a given number of steps and operations within a given error margin).

The tests are currently run as GitHub Actions and their configuration files are in the `.github/workflows/` folder of the LAMMPS git tree. The test status of these tests is reported with the corresponding pull request and the pull request *cannot* be merged without all tests passing.

In addition, there is a nightly test run using the `develop` branch to generate code coverage data for the included tests (see below), provided there have been changes to that tree. The results of this test run can be currently viewed at https://download.lammps.org/coverage/tests.html

Regression tests can also be performed locally with the *regression tester tool*. The tool checks if a given LAMMPS binary, when run with selected input examples produces, thermo output that is consistent with the provided log files. The script can be run in one pass over all available input files, but it can also first create multiple lists of inputs or folders that can then be run with multiple workers concurrently to speed things up. Another mode allows to do a quick check of inputs that contain commands that have changes in the current checkout branch relative to a git branch. This works similar to the two pass mode, but will select only shorter runs and no more than 100 inputs that are chosen randomly. This ensures that the quick test runs significantly faster compared to the full test run. These test runs can also be performed with instrumented LAMMPS binaries (see previous section).

The unit testing facility is integrated into the CMake build process of the LAMMPS source code distribution itself. It can be enabled by setting `-D ENABLE_TESTING=on` during the CMake configuration step. It requires the YAML library and matching development headers to compile (if those are not found locally a recent version of that library will be downloaded and compiled along with LAMMPS and the test programs) and will download and compile a specific version of the GoogleTest C++ test framework that is used to implement the tests. Those unit tests may be combined with memory access and leak checking with valgrind (see below for how to enable it). In that case, running so-called death tests will create a lot of false positives and thus they can be disabled by configuring compilation with the additional setting `-D SKIP_DEATH_TESTS=on`.

After compilation is complete, the unit testing is started in the build folder using the `ctest` command, which is part of the CMake software. The number of available tests will depend on the LAMMPS versions, installed LAMMPS packages, configuration settings, development environment, and operating system.

The output of the plain `ctest` command looks something like the following:

```
  $ ctest
  Test project /home/akohlmey/compile/lammps/build-testing
        Start   1: RunLammps
  1/563 Test   #1: RunLammps ..................................   Passed    0.28 sec
        Start   2: HelpMessage
  2/563 Test   #2: HelpMessage ................................   Passed    0.06 sec
        Start   3: InvalidFlag
  3/563 Test   #3: InvalidFlag ................................   Passed    0.06 sec
        Start   4: Tokenizer
  4/563 Test   #4: Tokenizer ..................................   Passed    0.05 sec
        Start   5: MemPool
  5/563 Test   #5: MemPool ....................................   Passed    0.05 sec
        Start   6: ArgUtils
  6/563 Test   #6: ArgUtils ...................................   Passed    0.05 sec
      [...]
        Start 561: ImproperStyle:zero
561/563 Test #561: ImproperStyle:zero ........................   Passed    0.07 sec
        Start 562: TestMliapPyUnified
562/563 Test #562: TestMliapPyUnified ........................   Passed    0.16 sec
        Start 563: TestPairList
563/563 Test #563: TestPairList ..............................   Passed    0.06 sec

100% tests passed, 0 tests failed out of 563

Label Time Summary:
generated    =   0.85 sec*proc (3 tests)
noWindows    =   4.16 sec*proc (2 tests)
slow         =  78.33 sec*proc (67 tests)
unstable     =  28.23 sec*proc (34 tests)

Total Test time (real) = 132.34 sec
```

The `ctest` command has many options, the most important ones are:

| Option | Function |
|---|---|
| `-V` | verbose output: display output of individual test runs |
| `-j <num>` | parallel run: run <num> tests in parallel |
| `--test-dir <path>` | provide path to the CMake build folder. By default `ctest` uses `.` |
| `-R <regex>` | run subset of tests matching the regular expression <regex> |
| `-E <regex>` | exclude subset of tests matching the regular expression <regex> |
| `-L <regex>` | run subset of tests with a label matching the regular expression <regex> |
| `-LE <regex>` | exclude subset of tests with a label matching the regular expression <regex> |
| `-N` | dry-run: display list of tests without running them |
| `-T memcheck` | run tests with valgrind memory checker (if available) |

In its full implementation, the unit test framework will consist of multiple kinds of tests implemented in different programming languages (C++, C, Python, Fortran) and testing different aspects of the LAMMPS software and its features. The tests will adapt to the compilation settings of LAMMPS, so that tests will be skipped if prerequisite features are not available in LAMMPS.

**Work in Progress**

The unit test framework was added in spring 2020 and is under active development. The coverage is not complete and will be expanded over time. Preference was given to test parts of the code base that are easy to test or commonly used.

Tests as shown by the `ctest` program are commands defined in the `CMakeLists.txt` files in the `unittest` directory tree. A few tests simply execute LAMMPS with specific command-line flags and check the output to the screen for expected content. A large number of unit tests are special tests programs using the GoogleTest framework and linked to the LAMMPS library that test individual functions or create a LAMMPS class instance, execute one or more commands and check data inside the LAMMPS class hierarchy. There are also tests for the C-library, Fortran, and Python module interfaces to LAMMPS. The Python tests use the Python "unittest" module in a similar fashion than the others use *GoogleTest*. These special test programs are structured to perform multiple individual tests internally and each of those contains several checks (aka assertions) for internal data being changed as expected.

Tests for force computing or modifying styles (e.g. styles for non-bonded and bonded interactions and selected fixes) are run by using a more generic test program that reads its input from files in YAML format. The YAML file provides the information on how to customized the test program to test a specific style and - if needed - with specific settings. To add a test for another, similar style (e.g. a new pair style) it is usually sufficient to add a suitable YAML file. *Detailed instructions for adding tests* are provided in the Programmer Guide part of the manual. A description of what happens during these tests is given below.

### Unit tests for force styles

A large part of LAMMPS are different "styles" for computing non-bonded and bonded interactions selected through the *pair_style command*, *bond_style command*, *angle_style command*, *dihedral_style command*, *improper_style command*, and *kspace_style command* commands. Since these styles all share common interfaces, it is possible to write generic test programs that will assemble LAMMPS inputs from templates with different settings and call those common interfaces for small test systems with less than 100 atoms and compare the results with pre-recorded reference results. A test run is then a collection of multiple individual test runs, each with many comparisons to reference results based on template input files, individual command settings, relative error margins, and reference data stored in a YAML format file with `.yaml` suffix. Currently the programs `test_pair_style`, `test_bond_style`, `test_angle_style`, `test_dihedral_style`, and `test_improper_style` are implemented. They will compare forces, energies and (global) stress for all atoms after a `run 0` calculation and after a few steps of MD with *fix nve*, each in multiple variants with different settings and also for multiple accelerated styles. If a prerequisite style or package is missing, the individual tests are skipped. All force style tests will be executed on a single MPI process, so using the CMake option `-D BUILD_MPI=off` can significantly speed up testing, since this will skip the MPI initialization for each test run.

Below is an example command and output for running a single test named `MolPairStyle:lj_cut` (argument to the `-R` option which selects tests by regular expression) and printing detailed output (the `-V` flag):

```
$ ctest -R MolPairStyle:lj_cut$ -V

[...]

   Start 199: MolPairStyle:lj_cut

199: Test command: /home/akohlmey/compile/lammps/build-test/test_pair_style "/home/
↪akohlmey/compile/lammps/unittest/force-styles/tests/mol-pair-lj_cut.yaml"
199: Working Directory: /home/akohlmey/compile/lammps/build-test/unittest/force-styles
199: Environment variables:
199:  PYTHONPATH=/home/akohlmey/compile/lammps/unittest/force-styles/tests:/home/
↪akohlmey/compile/lammps/python:
199:  PYTHONUNBUFFERED=1
```

(continues on next page)

(continued from previous page)

```
199:  PYTHONDONTWRITEBYTECODE=1
199:  OMP_PROC_BIND=false
199:  OMP_NUM_THREADS=4
199:  LAMMPS_POTENTIALS=/home/akohlmey/compile/lammps/potentials
199:  LD_LIBRARY_PATH=/home/akohlmey/compile/lammps/build-test:/usr/lib64/mpich/lib:/
↪home/akohlmey/.local/lib::
199: Test timeout computed to be: 1500
199: [==========] Running 9 tests from 1 test suite.
199: [----------] Global test environment set-up.
199: [----------] 9 tests from PairStyle
199: [ RUN      ] PairStyle.plain
199: [       OK ] PairStyle.plain (17 ms)
199: [ RUN      ] PairStyle.omp
199: [       OK ] PairStyle.omp (3 ms)
199: [ RUN      ] PairStyle.kokkos_omp
199: [       OK ] PairStyle.kokkos_omp (6 ms)
199: [ RUN      ] PairStyle.gpu
199: /home/akohlmey/compile/lammps/unittest/force-styles/test_pair_style.cpp:793: Skipped
199:
199:
199: [  SKIPPED ] PairStyle.gpu (0 ms)
199: [ RUN      ] PairStyle.intel
199: [       OK ] PairStyle.intel (2 ms)
199: [ RUN      ] PairStyle.opt
199: [       OK ] PairStyle.opt (2 ms)
199: [ RUN      ] PairStyle.single
199: [       OK ] PairStyle.single (2 ms)
199: [ RUN      ] PairStyle.extract
199: [       OK ] PairStyle.extract (1 ms)
199: [ RUN      ] PairStyle.extract_omp
199: [       OK ] PairStyle.extract_omp (1 ms)
199: [----------] 9 tests from PairStyle (37 ms total)
199:
199: [----------] Global test environment tear-down
199: [==========] 9 tests from 1 test suite ran. (37 ms total)
199: [  PASSED  ] 8 tests.
199: [  SKIPPED ] 1 test, listed below:
199: [  SKIPPED ] PairStyle.gpu
1/1 Test #199: MolPairStyle:lj_cut ..............   Passed    0.75 sec

The following tests passed:
        MolPairStyle:lj_cut

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.76 sec
```

In this particular case, 8 out of 9 sets of tests were conducted, the tests for the `lj/cut/gpu` pair style were skipped, since the LAMMPS library linked to the test executable did not include the GPU package. To learn what individual tests are performed, you (currently) need to read the source code. You can use code coverage recording (see next section) to confirm how well the tests cover the code paths in the individual source files.

The force style test programs have a common set of options:

| Option | Function |
|---|---|
| `-g <newfile>` | regenerate reference data in new YAML file |
| `-u` | update reference data in the original YAML file |
| `-s` | print error statistics for each group of comparisons |
| `-v` | verbose output: also print the executed LAMMPS commands |

Since the `ctest` tool has no mechanism to directly pass flags to the individual test programs, a workaround has been implemented where these flags can be set in an environment variable `TEST_ARGS`. Example:

```
env TEST_ARGS=-s ctest -V -R BondStyle
```

This adds output with statistics for the computed error of the various tests relative to the reference (e.g. the per-atom force components).

To add a test for a style that is not yet covered, it is usually best to copy a YAML file for a similar style to a new file, edit the details of the style (how to call it, how to set its coefficients) and then run test command with either the `-g` and the replace the initial test file with the regenerated one or the `-u` option. The `-u` option will destroy the original file, if the generation run does not complete, so using `-g` is recommended unless the YAML file is fully tested and working. To have the new test file recognized by `ctest`, you need to re-run cmake. You can verify that the new test is available by checking the output of `ctest -N`.

Some of the force style tests are rather slow to run and some are very sensitive to small differences like CPU architecture, compiler tool chain, compiler optimization. Those tests are flagged with a "slow" and/or "unstable" label, and thus those tests can be selectively excluded with the `-LE` flag to `ctest` (see description of the most commonly used `ctest` flags) or specifically selected using the `-L` flag.

**Recommendations and notes for YAML files**

- The reference results should be recorded without any code optimization or related compiler flags enabled.
- The `epsilon` parameter defines the relative precision with which the reference results must be met. The test geometries often have high and low energy parts and thus a significant impact from floating-point math truncation errors is to be expected. Some functional forms and potentials are more noisy than others, so this parameter needs to be adjusted. Typically a value around 1.0e-13 can be used, but it may need to be as large as 1.0e-8 in some cases.
- The tests for pair styles from OPT, OPENMP and INTEL are performed with automatically rescaled epsilon to account for additional loss of precision from code optimizations and different summation orders.
- When compiling with (aggressive) compiler optimization, some tests are likely to fail. It is recommended to inspect the individual tests in detail to decide, whether the specific error for a specific property is acceptable (it often is), or this may be an indication of mis-compiled code (or an undesired large loss of precision due to significant reordering of operations and thus less error cancellation).

### Unit tests for timestepping related fixes

A substantial subset of *fix styles* are invoked regularly during MD timestepping and manipulate per-atom properties like positions, velocities, and forces. For those fix styles, testing can be done in a very similar fashion as for force fields and thus there is a test program *test_fix_timestep* that shares a lot of code, properties, and command-line flags with the force field style testers described in the previous section.

This tester will set up a small molecular system run with verlet run style for 4 MD steps, then write a binary restart and continue for another 4 MD steps. At this point coordinates and velocities are recorded and compared to reference data. Then the system is cleared, restarted and running the second 4 MD steps again and the data is compared to the same reference. That is followed by another restart after which per atom type masses are replaced with per-atom masses and

the second 4 MD steps are repeated again and compared to the same reference. Also global scalar and vector data of the fix is recorded and compared. If the fix is a thermostat and thus the internal property `t_target` can be extracted, then this is compared to the reference data. The tests are repeated with the respa run style.

If the fix has a multi-threaded version in the OPENMP package, then the entire set of tests is repeated for that version as well.

For this to work, some additional conditions have to be met by the YAML format test inputs.

- The fix to be tested (and only this fix), should be listed in the `prerequisites:` section
- The fix to be tested must be specified in the `post_commands:` section with the fix-ID `test`. This section may contain other commands and other fixes (e.g. an instance of fix nve for testing a thermostat or force manipulation fix)
- For fixes that can tally contributions to the global virial, the line `fix_modify test virial yes` should be included in the `post_commands:` section of the test input.
- For thermostat fixes the target temperature should be ramped from an arbitrary value (e.g. 50K) to a pre-defined target temperature entered as `${t_target}`.
- For fixes that have thermostatting support included, but do not have it enabled in the input (e.g. fix rigid with default settings), the `post_commands:` section should contain the line `variable t_target delete` to disable the target temperature ramp check to avoid false positives.

### Use custom linker for faster link times when ENABLE_TESTING is active

When compiling LAMMPS with testing enabled, most test executables will need to be linked against the LAMMPS library and re-linked whenever there is a change to LAMMPS. Since this can be a very large library with many C++ objects when many packages are enabled, link times can become very long on machines that use the GNU BFD linker (e.g. Linux systems). Alternative linker programs like the `mold` linker, the `lld` linker of the LLVM project, or the `gold` linker available with GNU binutils can speed up this step substantially (in this order). CMake will by default test if any of the three can be enabled and use it when `ENABLE_TESTING` is active. The linker can also be selected manually through the `LAMMPS_CUSTOM_LINKER` CMake variable. Allowed values are `mold`, `lld`, `gold`, `bfd`, or `default`. The `default` option will use the system default linker otherwise, the linker is chosen explicitly. This option is only available for the GNU or Clang C++ compilers.

A small additional improvement can be obtained by building LAMMPS as a shared library with `-D BUILD_SHARED_LIBS=on`. But this is a small improvement due to reducing file I/O. Using an alternate linker has an algorithmic improvement through using symbol resolution algorithms with lower algorithmic complexity.

### Tests for other components and utility functions

Additional tests that validate utility functions or specific components of LAMMPS are implemented as standalone executable which may, or may not require creating a suitable LAMMPS instance. These tests are more specific and do not require YAML format input files. To add a test, either an existing source file needs to be extended or a new file added, which in turn requires additions to the `CMakeLists.txt` file in the source folder.

### Collect and visualize code coverage metrics

You can also collect code coverage metrics while running LAMMPS or the tests by enabling code coverage support during the CMake configuration:

```
-D ENABLE_COVERAGE=on  # enable coverage measurements (off by default)
```

This will instrument all object files to write information about which lines of code were accessed during execution in files next to the corresponding object files. These can be post-processed to visually show the degree of coverage and which code paths are accessed and which are not taken. When working on unit tests (see above), this can be extremely

helpful to determine which parts of the code are not executed and thus what kind of tests are still missing. The coverage data is cumulative, i.e. new data is added with each new run.

Enabling code coverage will also add the following build targets to generate coverage reports after running the LAMMPS executable or the unit tests:

```
make gen_coverage_html   # generate coverage report in HTML format
make gen_coverage_xml    # generate coverage report in XML format
make clean_coverage_html # delete folder with HTML format coverage report
make reset_coverage      # delete all collected coverage data and HTML output
```

These reports require GCOVR to be installed. The easiest way to do this to install it via pip:

```
python3 -m pip install gcovr
```

After post-processing with `gen_coverage_html` the results are in a folder `coverage_html` and can be viewed with a web browser. The images below illustrate how the data is presented. The coverage data for testing the current `develop` branch is generated nightly and currently available at: https://download.lammps.org/coverage/

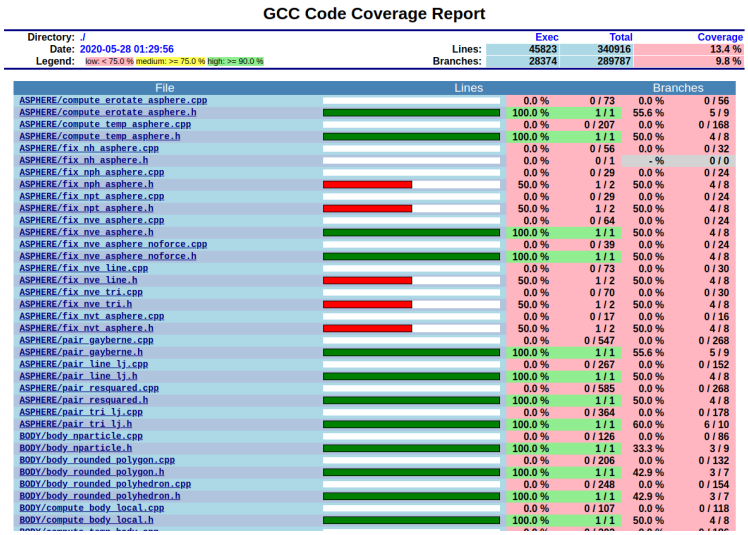



Fig. 1: Top of the overview page

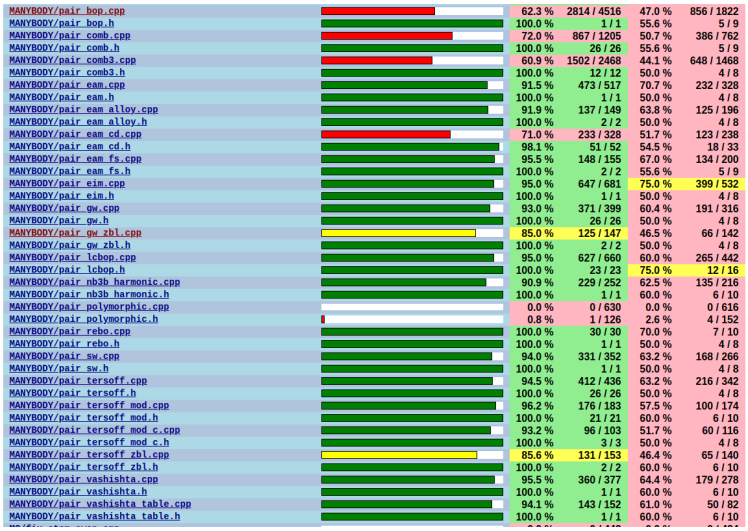
### 3.12.5 Coding style utilities

To aid with enforcing some of the coding style conventions in LAMMPS some additional build targets have been added. These require Python 3.5 or later and will only work properly on Unix-like operating and file systems.

The following options are available.

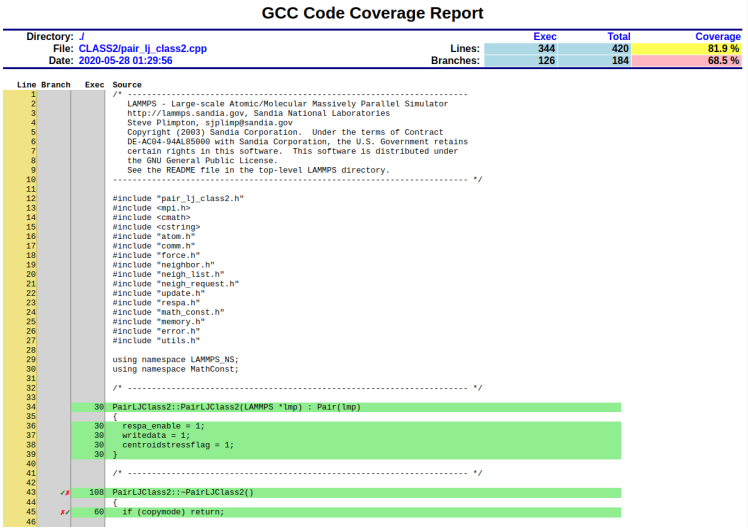
```
make check-whitespace    # search for files with whitespace issues
make fix-whitespace      # correct whitespace issues in files
make check-homepage      # search for files with old LAMMPS homepage URLs
make fix-homepage        # correct LAMMPS homepage URLs in files
make check-errordocs     # search for deprecated error docs in header files
make fix-errordocs       # remove error docs in header files
make check-permissions   # search for files with permissions issues
make fix-permissions     # correct permissions issues in files
make check-docs          # search for several issues in the manual
```

(continues on next page)



| | | | | | |
|---|---|---|---|---|---|
| MANYBODY/pair_bop.cpp | 62.3 % | 2814 / 4516 | 47.0 % | 856 / 1822 |
| MANYBODY/pair_bop.h | 100.0 % | 1 / 1 | 55.6 % | 5 / 9 |
| MANYBODY/pair_comb.cpp | 72.0 % | 867 / 1205 | 50.7 % | 386 / 762 |
| MANYBODY/pair_comb.h | 100.0 % | 26 / 26 | 55.6 % | 5 / 9 |
| MANYBODY/pair_comb3.cpp | 60.9 % | 1502 / 2468 | 44.1 % | 648 / 1468 |
| MANYBODY/pair_comb3.h | 100.0 % | 12 / 12 | 50.0 % | 4 / 8 |
| MANYBODY/pair_eam.cpp | 91.5 % | 473 / 517 | 70.7 % | 232 / 328 |
| MANYBODY/pair_eam.h | 100.0 % | 1 / 1 | 50.0 % | 4 / 8 |
| MANYBODY/pair_eam_alloy.cpp | 91.9 % | 137 / 149 | 63.8 % | 125 / 196 |
| MANYBODY/pair_eam_alloy.h | 100.0 % | 2 / 2 | 50.0 % | 4 / 8 |
| MANYBODY/pair_eam_cd.cpp | 71.0 % | 233 / 328 | 51.7 % | 123 / 238 |
| MANYBODY/pair_eam_cd.h | 98.1 % | 51 / 52 | 54.5 % | 18 / 33 |
| MANYBODY/pair_eam_fs.cpp | 95.5 % | 148 / 155 | 67.0 % | 134 / 200 |
| MANYBODY/pair_eam_fs.h | 100.0 % | 2 / 2 | 55.6 % | 5 / 9 |
| MANYBODY/pair_eim.cpp | 95.0 % | 647 / 681 | 75.0 % | 399 / 532 |
| MANYBODY/pair_eim.h | 100.0 % | 1 / 1 | 50.0 % | 4 / 8 |
| MANYBODY/pair_gw.cpp | 93.0 % | 371 / 399 | 60.4 % | 191 / 316 |
| MANYBODY/pair_gw.h | 100.0 % | 26 / 26 | 50.0 % | 4 / 8 |
| MANYBODY/pair_gw_zbl.cpp | 85.0 % | 125 / 147 | 46.5 % | 66 / 142 |
| MANYBODY/pair_gw_zbl.h | 100.0 % | 2 / 2 | 50.0 % | 4 / 8 |
| MANYBODY/pair_lcbop.cpp | 95.0 % | 627 / 660 | 60.0 % | 265 / 442 |
| MANYBODY/pair_lcbop.h | 100.0 % | 23 / 23 | 75.0 % | 12 / 16 |
| MANYBODY/pair_nb3b_harmonic.cpp | 90.9 % | 229 / 252 | 62.5 % | 135 / 216 |
| MANYBODY/pair_nb3b_harmonic.h | 100.0 % | 1 / 1 | 60.0 % | 6 / 10 |
| MANYBODY/pair_polymorphic.cpp | 0.0 % | 0 / 630 | 0.0 % | 0 / 616 |
| MANYBODY/pair_polymorphic.h | 0.8 % | 1 / 126 | 2.6 % | 4 / 152 |
| MANYBODY/pair_rebo.cpp | 100.0 % | 30 / 30 | 70.0 % | 7 / 10 |
| MANYBODY/pair_rebo.h | 100.0 % | 1 / 1 | 50.0 % | 4 / 8 |
| MANYBODY/pair_sw.cpp | 94.0 % | 331 / 352 | 63.2 % | 168 / 266 |
| MANYBODY/pair_sw.h | 100.0 % | 1 / 1 | 50.0 % | 4 / 8 |
| MANYBODY/pair_tersoff.cpp | 94.5 % | 412 / 436 | 63.2 % | 216 / 342 |
| MANYBODY/pair_tersoff.h | 100.0 % | 26 / 26 | 50.0 % | 4 / 8 |
| MANYBODY/pair_tersoff_mod.cpp | 96.2 % | 176 / 183 | 57.5 % | 100 / 174 |
| MANYBODY/pair_tersoff_mod.h | 100.0 % | 21 / 21 | 60.0 % | 6 / 10 |
| MANYBODY/pair_tersoff_mod_c.cpp | 93.2 % | 96 / 103 | 51.7 % | 60 / 116 |
| MANYBODY/pair_tersoff_mod_c.h | 100.0 % | 3 / 3 | 50.0 % | 4 / 8 |
| MANYBODY/pair_tersoff_zbl.cpp | 85.6 % | 131 / 153 | 46.4 % | 65 / 140 |
| MANYBODY/pair_tersoff_zbl.h | 100.0 % | 2 / 2 | 60.0 % | 6 / 10 |
| MANYBODY/pair_vashishta.cpp | 95.5 % | 360 / 377 | 64.4 % | 179 / 278 |
| MANYBODY/pair_vashishta.h | 100.0 % | 1 / 1 | 60.0 % | 6 / 10 |
| MANYBODY/pair_vashishta_table.cpp | 94.1 % | 143 / 152 | 61.0 % | 50 / 82 |
| MANYBODY/pair_vashishta_table.h | 100.0 % | 1 / 1 | 60.0 % | 6 / 10 |

Fig. 2: Styles with good coverage



**GCC Code Coverage Report**

Directory: ./
File: CLASS2/pair_lj_class2.cpp
Date: 2020-05-28 01:29:56

| | Exec | Total | Coverage |
|---|---|---|---|
| Lines: | 344 | 420 | 81.9 % |
| Branches: | 126 | 184 | 68.5 % |

```
Line Branch   Exec  Source
   1                /* ----------------------------------------------------------------------
   2                   LAMMPS - Large-scale Atomic/Molecular Massively Parallel Simulator
   3                   http://lammps.sandia.gov, Sandia National Laboratories
   4                   Steve Plimpton, sjplimp@sandia.gov
   5                   Copyright (2003) Sandia Corporation.  Under the terms of Contract
   6                   DE-AC04-94AL85000 with Sandia Corporation, the U.S. Government retains
   7                   certain rights in this software.  This software is distributed under
   8                   the GNU General Public License.
   9                   See the README file in the top-level LAMMPS directory.
  10                ------------------------------------------------------------------------- */
  11
  12                #include "pair_lj_class2.h"
  13                #include <mpi.h>
  14                #include <cmath>
  15                #include <cstring>
  16                #include "atom.h"
  17                #include "comm.h"
  18                #include "force.h"
  19                #include "neighbor.h"
  20                #include "neigh_list.h"
  21                #include "neigh_request.h"
  22                #include "update.h"
  23                #include "respa.h"
  24                #include "math_const.h"
  25                #include "memory.h"
  26                #include "error.h"
  27                #include "utils.h"
  28
  29                using namespace LAMMPS_NS;
  30                using namespace MathConst;
  31
  32                /* ---------------------------------------------------------------------- */
  33
  34       30       PairLJClass2::PairLJClass2(LAMMPS *lmp) : Pair(lmp)
  35                {
  36       30         respa_enable = 1;
  37       30         writedata = 1;
  38       30         centroidstressflag = 1;
  39       30       }
  40
  41                /* ---------------------------------------------------------------------- */
  42
  43    ✗  ✓  108   PairLJClass2::~PairLJClass2()
  44                {
  45    ✗  ✓   60     if (copymode) return;
  46
```

Fig. 3: Top of individual source page

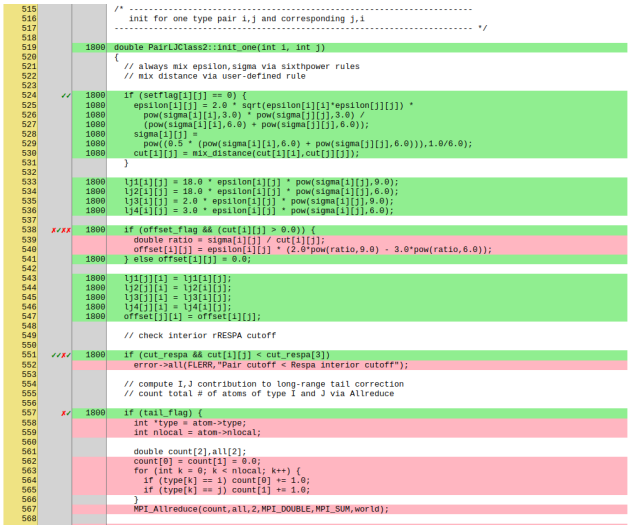


Fig. 4: Source page with branches

(continued from previous page)

```
make check-version       # list files with pending release version tags
make check               # run all check targets from above
```

These should help to make source and documentation files conforming to some the coding style preferences of the LAMMPS developers.

### 3.12.6 Clang-format support

For the code in the `unittest` and `src` trees we are transitioning to use the *clang-format* tool to assist with having a consistent source code formatting style. The *clang-format* command bundled with Clang version 8.0 or later is required. The configuration is in files called `.clang-format` in the respective folders. Since the modifications from *clang-format* can be significant and - especially for "legacy style code" - they are not always improving readability, a large number of files currently have a `// clang-format off` at the top, which will disable the processing. As of fall 2021 all files have been either "protected" this way or are enabled for full or partial *clang-format* processing. Over time, the "protected" files will be refactored and updated so that *clang-format* may be applied to them as well.

It is recommended for all newly contributed files to use the clang-format processing while writing the code or do the coding style processing (including the scripts mentioned in the previous paragraph)

If *clang-format* is available, files can be updated individually with commands like the following:

```
clang-format -i some_file.cpp
```

### 3.12.7 GitHub command-line interface

GitHub has developed a command-line tool to interact with the GitHub website via a command called `gh`. This is extremely convenient when working with a Git repository hosted on GitHub (like LAMMPS). It is thus highly recommended to install it when doing LAMMPS development. To use `gh` you must be within a git checkout of a repository and you must obtain an authentication token to connect your checkout with a GitHub user. This is done with the command: `gh auth login` where you then have to follow the prompts. Here are some examples:

| Command | Description |
|---|---|
| `gh pr list` | List currently open pull requests |
| `gh pr checks 404` | Shows the status of all checks for pull request #404 |
| `gh pr view 404` | Shows the description and recent comments for pull request #404 |
| `gh co 404` | Check out the branch from pull request #404; set up for pushing changes |
| `gh issue list` | List currently open issues |
| `gh issue view 430 --comments` | Shows the description and all comments for issue #430 |

The capabilities of the `gh` command are continually expanding, so for more details please see the documentation at https://cli.github.com/manual/ or use `gh --help` or `gh <command> --help` for embedded help.

CHAPTER

# FOUR

# RUN LAMMPS

These pages explain how to run LAMMPS once you have *installed an executable* or *downloaded the source code* and *built an executable*. The *Commands* doc page describes how input scripts are structured and the commands they can contain.

## 4.1 Basics of running LAMMPS

LAMMPS is run from the command-line, reading commands from a file via the `-in` command-line flag, or from standard input. Using the `-in in.file` variant is recommended (see note below). The name of the LAMMPS executable is either `lmp` or `lmp_<machine>` with *<machine>* being the machine string used when compiling LAMMPS. This is required when compiling LAMMPS with the traditional build system (e.g. with `make mpi`), but optional when using CMake to configure and build LAMMPS:

```
lmp_serial -in in.file
lmp_serial < in.file
lmp -in in.file
lmp < in.file
/path/to/lammps/src/lmp_serial -i in.file
mpirun -np 4 lmp_mpi -in in.file
mpiexec -np 4 lmp -in in.file
mpirun -np 8 /path/to/lammps/src/lmp_mpi -in in.file
mpiexec -n 6 /usr/local/bin/lmp -in in.file
```

You normally run the LAMMPS command in the directory where your input script is located. That is also where output files are produced by default, unless you provide specific other paths in your input script or on the command-line. As in some of the examples above, the LAMMPS executable itself can be placed elsewhere.

> **Note**
>
> The redirection operator "<" will not always work when running in parallel with `mpirun` or `mpiexec`; for those systems the -in form is required.

As LAMMPS runs it prints info to the screen and a logfile named `log.lammps`. More info about output is given on the *screen and logfile output* page.

If LAMMPS encounters errors in the input script or while running a simulation it will print an ERROR message and stop or a WARNING message and continue. See the *Common Problems* page for a discussion of the various kinds of errors LAMMPS can or can't detect, a list of all ERROR and WARNING messages, and what to do about them.

LAMMPS can run the same problem on any number of processors, including a single processor. In theory you should get identical answers on any number of processors and on any machine. In practice, numerical round-off due to using floating-point math can cause slight differences and an eventual divergence of molecular dynamics trajectories. See the *Errors common* page for discussion of this.

LAMMPS can run as large a problem as will fit in the physical memory of one or more processors. If you run out of memory, you must run on more processors or define a smaller problem. The amount of memory needed and how well it can be distributed across processors may vary based on the models and settings and commands used.

If you run LAMMPS in parallel via mpirun, you should be aware of the *processors* command, which controls how MPI tasks are mapped to the simulation box, as well as mpirun options that control how MPI tasks are assigned to physical cores of the node(s) of the machine you are running on. These settings can improve performance, though the defaults are often adequate.

For example, it is often important to bind MPI tasks (processes) to physical cores (processor affinity), so that the operating system does not migrate them during a simulation. If this is not the default behavior on your machine, the mpirun option `--bind-to core` (OpenMPI) or `-bind-to core` (MPICH) can be used.

If the LAMMPS command(s) you are using support multi-threading, you can set the number of threads per MPI task via the environment variable `OMP_NUM_THREADS`, before you launch LAMMPS:

```
export OMP_NUM_THREADS=2      # bash
setenv OMP_NUM_THREADS 2      # csh or tcsh
```

This can also be done via the *package* command or via the *-pk command-line switch* which invokes the package command. See the *package* command or *Speed* doc pages for more details about which accelerator packages and which commands support multi-threading.

---

You can experiment with running LAMMPS using any of the input scripts provided in the examples or bench directory. Input scripts are named `in.*` and sample outputs are named `log.*.P` where P is the number of processors it was run on.

Some of the examples or benchmarks require LAMMPS to be built with optional packages.

## 4.2 Command-line options

At run time, LAMMPS recognizes several optional command-line switches which may be used in any order. Either the full word or a one or two letter abbreviation can be used:

- *-e or -echo*
- *-h or -help*
- *-i or -in*
- *-k or -kokkos*
- *-l or -log*
- *-mdi*
- *-m or -mpicolor*
- *-c or -cite*
- *-nc or -nocite*
- *-nb or -nonbuf*
- *-pk or -package*

- *-p or -partition*
- *-pl or -plog*
- *-ps or -pscreen*
- *-ro or -reorder*
- *-r2data or -restart2data*
- *-r2dump or -restart2dump*
- *-r2info or -restart2info*
- *-sc or -screen*
- *-sr or skiprun*
- *-sf or -suffix*
- *-v or -var*

For example, the lmp_mpi executable might be launched as follows:

```
mpirun -np 16 lmp_mpi -v f tmp.out -l my.log -sc none -i in.alloy
mpirun -np 16 lmp_mpi -var f tmp.out -log my.log -screen none -in in.alloy
```

---

**-echo style**

Set the style of command echoing. The style can be *none* or *screen* or *log* or *both*. Depending on the style, each command read from the input script will be echoed to the screen and/or logfile. This can be useful to figure out which line of your script is causing an input error. The default value is *log*. The echo style can also be set by using the *echo* command in the input script itself.

---

**-help**

Print a brief help summary and a list of options compiled into this executable for each LAMMPS style (atom_style, fix, compute, pair_style, bond_style, etc). This can tell you if the command you want to use was included via the appropriate package at compile time. LAMMPS will print the info and immediately exit if this switch is used.

---

**-in file**

Specify a file to use as an input script. This is currently an optional but recommended switch when running LAMMPS in the default one-partition mode. If it is not specified, LAMMPS reads its script from standard input, typically from a script via I/O redirection; e.g. `lmp_linux < in.run`. With many MPI implementations (but not all of them), I/O redirection also works in parallel, but using the `-in` flag will *always* work.

This is a **required** switch when running LAMMPS in multi-partition mode (see below), since multiple pools of MPI processes cannot all read from standard input concurrently. The file name may be "none" for starting multi-partition calculations without reading an initial input file when using the library interface.

---

**-kokkos on/off keyword/value ...**

Explicitly enable or disable KOKKOS support, as provided by the KOKKOS package. Even if LAMMPS is built with this package, as described in the *the KOKKOS package page*, this switch must be set to enable running with KOKKOS-enabled styles the package provides. If the switch is not set (the default), LAMMPS will operate as if the KOKKOS

package were not installed; i.e. you can run standard LAMMPS or with the GPU or OPENMP packages, for testing or benchmarking purposes.

Additional optional keyword/value pairs can be specified which determine how Kokkos will use the underlying hardware on your platform. These settings apply to each MPI task you launch via the `mpirun` or `mpiexec` command. You may choose to run one or more MPI tasks per physical node. Note that if you are running on a desktop machine, you typically have one physical node. On a cluster or supercomputer there may be dozens or 1000s of physical nodes.

Either the full word or an abbreviation can be used for the keywords. Note that the keywords do not use a leading minus sign. I.e. the keyword is "t", not "-t". Also note that each of the keywords has a default setting. Examples of when to use these options and what settings to use on different platforms is given on the *KOKKOS package* doc page.

- d or device
- g or gpus
- t or threads

```
device Nd
```

This option is only relevant if you built LAMMPS with CUDA=yes, you have more than one GPU per node, and if you are running with only one MPI task per node. The Nd setting is the ID of the GPU on the node to run on. By default Nd = 0. If you have multiple GPUs per node, they have consecutive IDs numbered as 0,1,2,etc. This setting allows you to launch multiple independent jobs on the node, each with a single MPI task per node, and assign each job to run on a different GPU.

```
gpus Ng Ns
```

This option is only relevant if you built LAMMPS with CUDA=yes, you have more than one GPU per node, and you are running with multiple MPI tasks per node (up to one per GPU). The Ng setting is how many GPUs you will use. The Ns setting is optional. If set, it is the ID of a GPU to skip when assigning MPI tasks to GPUs. This may be useful if your desktop system reserves one GPU to drive the screen and the rest are intended for computational work like running LAMMPS. By default Ng = 1 and Ns is not set.

Depending on which flavor of MPI you are running, LAMMPS will look for one of these 4 environment variables

```
SLURM_LOCALID (various MPI variants compiled with SLURM support)
MPT_LRANK (HPE MPI)
MV2_COMM_WORLD_LOCAL_RANK (Mvapich)
OMPI_COMM_WORLD_LOCAL_RANK (OpenMPI)
```

which are initialized by the `srun`, `mpirun`, or `mpiexec` commands. The environment variable setting for each MPI rank is used to assign a unique GPU ID to the MPI task.

```
threads Nt
```

This option assigns Nt number of threads to each MPI task for performing work when Kokkos is executing in OpenMP or pthreads mode. The default is Nt = 1, which essentially runs in MPI-only mode. If there are Np MPI tasks per physical node, you generally want Np*Nt = the number of physical cores per node, to use your available hardware optimally. This also sets the number of threads used by the host when LAMMPS is compiled with CUDA=yes.

Deprecated since version 22Dec2022.

Support for the "numa" or "n" option was removed as its functionality was ignored in Kokkos for some time already.

---

**-log file**

Specify a log file for LAMMPS to write status information to. In one-partition mode, if the switch is not used, LAMMPS writes to the file log.lammps. If this switch is used, LAMMPS writes to the specified file. In multi-partition mode, if the switch is not used, a log.lammps file is created with high-level status information. Each partition also writes to a log.lammps.N file where N is the partition ID. If the switch is specified in multi-partition mode, the high-level logfile is named "file" and each partition also logs information to a file.N. For both one-partition and multi-partition mode, if the specified file is "none", then no log files are created. Using a *log* command in the input script will override this setting. Option -plog will override the name of the partition log files file.N.

---

**-mdi 'multiple flags'**

This flag is only recognized and used when LAMMPS has support for the MolSSI Driver Interface (MDI) included as part of the *MDI* package. This flag is specific to the MDI library and controls how LAMMPS interacts with MDI. There are usually multiple flags that have to follow it and those have to be placed in quotation marks. For more information about how to launch LAMMPS in MDI client/server mode please refer to the *MDI Howto*.

---

**-mpicolor color**

If used, this must be the first command-line argument after the LAMMPS executable name. It is only used when LAMMPS is launched by an mpirun command which also launches another executable(s) at the same time (The other executable could be LAMMPS as well.). The *color* is an integer value which should be different for each executable (another application may set this value in a different way). LAMMPS and the other executable(s) perform an MPI_Comm_split() with their own colors to replace the `MPI_COMM_WORLD` communicator with a new communicator using the subset of MPI processes they are actually running on.

---

**-cite style** or **file name**

Select how and where to output a reminder about citing contributions to the LAMMPS code that were used during the run. Available keywords for styles are "both", "none", "screen", or "log". Any other keyword will be considered a file name to write the detailed citation info to instead of logfile or screen. Default is the "log" style where there is a short summary in the screen output and detailed citations in BibTeX format in the logfile. The option "both" selects the detailed output for both, "none", the short output for both, and "screen" will write the detailed info to the screen and the short version to the log file. If a dedicated citation info file is requested, the screen and log file output will be in the short format (same as with "none").

See the *citation page* for more details on how to correctly reference and cite LAMMPS.

---

**-nocite**

Disable generating a citation reminder (see above) at all.

---

**-nonbuf**

Added in version 15Sep2022.

Turn off buffering for screen and logfile output. For performance reasons, output to the screen and logfile is usually buffered, i.e. output is only written to a file if its buffer - typically 4096 bytes - has been filled. When LAMMPS crashes for some reason, however, that can mean that there is important output missing. With this flag the buffering can be turned off (only for screen and logfile output) and any output will be committed immediately. Note that when running in parallel with MPI, the screen output may still be buffered by the MPI library and this cannot be changed by LAMMPS. This flag should only be used for debugging and not for production simulations as the performance impact can be significant, especially for large parallel runs.

**-package style args ....**

Invoke the *package* command with style and args. The syntax is the same as if the command appeared at the top of the input script. For example `-package gpu 2` or `-pk gpu 2` is the same as *package gpu 2* in the input script. The possible styles and args are documented on the *package* doc page. This switch can be used multiple times, e.g. to set options for the INTEL and OPENMP packages which can be used together.

Along with the `-suffix` command-line switch, this is a convenient mechanism for invoking accelerator packages and their options without having to edit an input script.

**-partition 8x2 4 5 ...**

Invoke LAMMPS in multi-partition mode. When LAMMPS is run on *P* MPI processes and this switch is not used, LAMMPS runs in *one partition*, i.e. all *P* MPI processes run a single simulation with the same settings. If this switch *is* used, the *P* MPI processes are split into separate partitions and each partition runs its own simulation. The arguments to the switch specify the number of MPI processes in each partition. Arguments of the form *MxN* mean *M* partitions, each with *N* MPI processes. Arguments of the form *N* mean a single partition with *N* MPI processes. The sum of MPI processes in all partitions must equal *P*. Thus the command `-partition 8x2 4 5` has 10 partitions (eight with 2 MPI processes, one with 4 and one with 5) and runs on a total of 25 MPI processes.

Running with multiple partitions can be useful for running *multi-replica simulations*, where each replica runs on one or a few MPI processes.

**Note**

With MPI installed on a standalone machine (e.g. your desktop or laptop), you can run on more (virtual) MPI processes than you have physical processors (for testing purposes), but some MPI implementations (for instance OpenMPI) may require an additional command line flag to enable this so-called oversubscription. You may also have to disable processor affinity or else the performance may be exceptionally bad when oversubscribing processors.

To run multiple independent simulations from one input script, using multiple partitions, see the *Howto multiple* page. World- and universe-style *variables* are useful in this context.

**-plog file**

Specify the base name for the partition log files, so partition *N* writes log information to `file.N`. If *file* is *none*, then no partition log files are created. This overrides the filename specified in the *-log* command-line option. This option is useful when working with large numbers of partitions, allowing the partition log files to be suppressed (`-plog none`) or placed in a subdirectory (`-plog replica_files/log.lammps`). If this option is not used, the log file for partition *N* is `log.lammps.N` or whatever is specified by the *-log* command-line option.

**-pscreen file**

Specify the base name for the partition screen file, so partition *N* writes screen information to `file.N`. If *file* is *none*, then no partition screen files are created. This overrides the filename specified in the *-screen* command-line option. This option is useful when working with large numbers of partitions, allowing the partition screen files to be suppressed (`-pscreen none`) or placed in a subdirectory (`-pscreen replica_files/screen`). If this option is not used, the screen file for partition *N* is `screen.N` or whatever is specified by the *-screen* command-line option.

**-reorder**

This option has 2 forms:

```
-reorder nth N
-reorder custom filename
```

Reorder the ranks in the MPI communicator used to instantiate LAMMPS, in one of several ways. The original MPI communicator ranks all *P* MPI processes from *0* to *P-1*. The mapping of these ranks to physical processors is done by the MPI library before LAMMPS begins. It may be useful in some cases to alter the order of the ranks, for example to ensure that cores within each node are ranked in a desired order. Or when using the *run_style verlet/split* command with 2 partitions to ensure that a specific Kspace processor (in the second partition) is matched up with a specific set of processors in the first partition. See the *General tips* page for more details.

If the keyword *nth* is used with a setting *N*, then it means every Nth processor will be moved to the end of the ranking. This is useful when using the *run_style verlet/split* command with 2 partitions via the *-partition* command-line switch. The first set of processors will be in the first partition, the second set in the second partition. The *-reorder* command-line switch can alter this so that the first *N* MPI processes in the first partition and one MPI process in the second partition will be ordered consecutively, e.g. as the cores on one physical node. This can boost performance. For example, if you use `-reorder nth 4` and `-partition 9 3` and you are running on 12 processors, the MPI process ranks will be reordered from

```
0 1 2 3 4 5 6 7 8 9 10 11
```

to

```
0 1 2 4 5 6 8 9 10 3 7 11
```

so that the processors in each partition will be

```
0 1 2 4 5 6 8 9 10
3 7 11
```

See the "processors" command for how to ensure processors from each partition could then be grouped optimally for quad-core nodes.

If the keyword is *custom*, then a file that specifies a permutation of the processor ranks is also specified. The format of the reorder file is as follows. Any number of initial blank or comment lines (starting with a "#" character) can be present. These should be followed by *P* lines of the form:

```
I J
```

where *P* is the number of processors LAMMPS was launched with. Note that if running in multi-partition mode (see the -partition switch above) *P* is the total number of MPI processes in all partitions. The *I* and *J* values describe a permutation of the *P* MPI process ranks. Every *I* and *J* should be values from *0* to *P-1* inclusive. In the set of *P I* values, every MPI rank ID should appear exactly once. Ditto for the set of *P J* values. A single *I*, *J* pairing means that the physical processor with MPI rank *I* in the original MPI communicator will have rank *J* in the reordered MPI communicator.

Note that rank ordering can also be specified by many MPI implementations, either by environment variables that specify how to order physical processors, or by config files that specify what physical processors to assign to each MPI rank. The *-reorder* switch simply gives you a portable way to do this without relying on MPI itself. See the *processors file* command for how to output info on the final assignment of physical processors to the LAMMPS simulation domain.

---

**-restart2data restartfile datafile keyword value ...**

Convert the restart file into a data file and immediately exit. This is the same operation as if the following 2-line input script were run:

```
read_restart restartfile
write_data datafile keyword value ...
```

The specified restartfile and/or datafile name may contain the wild-card character "*". The restartfile name may also contain the wild-card character "%". The meaning of these characters is explained on the *read_restart* and *write_data* doc pages. The use of "%" means that a parallel restart file can be read. Note that a filename such as file.* may need to be enclosed in quotes or the "*" character prefixed with a backslash ("") to avoid shell expansion of the "*" character.

The syntax following restartfile, namely

```
datafile keyword value ...
```

is identical to the arguments of the *write_data* command. See its documentation page for details. This includes its optional keyword/value settings.

---

**-restart2dump restartfile group-ID dumpstyle dumpfile arg1 arg2 ...**

Convert the restart file into a dump file and immediately exit. This is the same operation as if the following 2-line input script were run:

```
read_restart restartfile
write_dump group-ID dumpstyle dumpfile arg1 arg2 ...
```

Note that the specified restartfile and dumpfile names may contain wild-card characters ("*" or "%") as explained on the *read_restart* and *write_dump* doc pages. The use of "%" means that a parallel restart file and/or parallel dump file can be read and/or written. Note that a filename such as file.* may need to be enclosed in quotes or the "*" character prefixed with a backslash ("") to avoid shell expansion of the "*" character.

The syntax following restartfile, namely

```
group-ID dumpstyle dumpfile arg1 arg2 ...
```

is identical to the arguments of the *write_dump* command. See its documentation page for details. This includes what per-atom fields are written to the dump file and optional dump_modify settings, including ones that affect how parallel dump files are written, e.g. the *nfile* and *fileper* keywords. See the *dump_modify* page for details.

---

**-restart2info restartfile keyword ...**

Added in version 29Aug2024.

Write out some info about the restart file and and immediately exit. This is the same operation as if the following 2-line input script were run:

```
read_restart restartfile
info system group computes fixes
```

The specified restartfile name may contain the wild-card character "*". The restartfile name may also contain the wild-card character "%". The meaning of these characters is explained on the *read_restart* documentation. The use of "%" means that a parallel restart file can be read. Note that a filename such as file.* may need to be enclosed in quotes or the "*" character prefixed with a backslash ("") to avoid shell expansion of the "*" character.

Optional keywords may follow the restartfile argument. These must be valid keywords for the *info command*. The most useful ones - *system*, *group*, *computes*, and *fixes* - are already applied. Appending keywords like *coeffs* or *communication* may provide additional useful information stored in the restart file.

---

**-screen file**

Specify a file for LAMMPS to write its screen information to. In one-partition mode, if the switch is not used, LAMMPS writes to the screen. If this switch is used, LAMMPS writes to the specified file instead and you will see no screen output. In multi-partition mode, if the switch is not used, high-level status information is written to the screen. Each partition also writes to a screen.N file where N is the partition ID. If the switch is specified in multi-partition mode, the high-level screen dump is named "file" and each partition also writes screen information to a file.N. For both one-partition and multi-partition mode, if the specified file is "none", then no screen output is performed. Option -pscreen will override the name of the partition screen files file.N.

---

**-skiprun**

Insert the command *timer timeout 0 every 1* at the beginning of an input file or after a *clear* command. This has the effect that the entire LAMMPS input script is processed without executing actual *run* or *minimize* and similar commands (their main loops are skipped). This can be helpful and convenient to test input scripts of long running calculations for correctness to avoid having them crash after a long time due to a typo or syntax error in the middle or at the end.

---

**-suffix style args**

Use variants of various styles if they exist. The specified style can be *gpu*, *intel*, *kk*, *omp*, *opt*, or *hybrid*. These refer to optional packages that LAMMPS can be built with, as described in *Accelerate performance*. The "gpu" style corresponds to the GPU package, the "intel" style to the INTEL package, the "kk" style to the KOKKOS package, the "opt" style to the OPT package, and the "omp" style to the OPENMP package. The hybrid style is the only style that accepts arguments. It allows for two packages to be specified. The first package specified is the default and will be used if it is available. If no style is available for the first package, the style for the second package will be used if available. For example, `-suffix hybrid intel omp` will use styles from the INTEL package if they are installed and available, but styles for the OPENMP package otherwise.

Along with the `-package` command-line switch, this is a convenient mechanism for invoking accelerator packages and their options without having to edit an input script.

As an example, all of the packages provide a *pair_style lj/cut* variant, with style names lj/cut/gpu, lj/cut/intel, lj/cut/kk, lj/cut/omp, and lj/cut/opt. A variant style can be specified explicitly in your input script, e.g. pair_style lj/cut/gpu. If the -suffix switch is used the specified suffix (gpu,intel,kk,omp,opt) is automatically appended whenever your input script command creates a new *atom style*, *pair style*, *fix*, *compute*, or *run style*. If the variant version does not exist, the standard version is created.

For the GPU package, using this command-line switch also invokes the default GPU settings, as if the command "package gpu 1" were used at the top of your input script. These settings can be changed by using the `-package gpu` command-line switch or the *package gpu* command in your script.

For the INTEL package, using this command-line switch also invokes the default INTEL settings, as if the command "package intel 1" were used at the top of your input script. These settings can be changed by using the `-package intel` command-line switch or the *package intel* command in your script. If the OPENMP package is also installed, the hybrid style with "intel omp" arguments can be used to make the omp suffix a second choice, if a requested style is not available in the INTEL package. It will also invoke the default OPENMP settings, as if the command "package omp 0" were used at the top of your input script. These settings can be changed by using the `-package omp` command-line switch or the *package omp* command in your script.

For the KOKKOS package, using this command-line switch also invokes the default KOKKOS settings, as if the command "package kokkos" were used at the top of your input script. These settings can be changed by using the `-package kokkos` command-line switch or the *package kokkos* command in your script.

For the OMP package, using this command-line switch also invokes the default OMP settings, as if the command "package omp 0" were used at the top of your input script. These settings can be changed by using the `-package omp` command-line switch or the *package omp* command in your script.

The *suffix* command can also be used within an input script to set a suffix, or to turn off or back on any suffix setting made via the command-line.

---

**-var name value1 value2 ...**

Specify a variable that will be defined for substitution purposes when the input script is read. This switch can be used multiple times to define multiple variables. "Name" is the variable name which can be a single character (referenced as $x in the input script) or a full string (referenced as ${abc}). An *index-style variable* will be created and populated with the subsequent values, e.g. a set of filenames. Using this command-line option is equivalent to putting the line "variable name index value1 value2 ..." at the beginning of the input script. Defining an index variable as a command-line argument overrides any setting for the same index variable in the input script, since index variables cannot be re-defined.

See the *variable* command for more info on defining index and other kinds of variables and the *Parsing rules* page for more info on using variables in input scripts.

**Note**

Currently, the command-line parser looks for arguments that start with "-" to indicate new switches. Thus you cannot specify multiple variable values if any of them start with a "-", e.g. a negative numeric value. It is OK if the first value1 starts with a "-", since it is automatically skipped.

## 4.3 Screen and logfile output

As LAMMPS reads an input script, it prints information to both the screen and a log file about significant actions it takes to setup a simulation. When the simulation is ready to begin, LAMMPS performs various initializations, and prints info about the run it is about to perform, including the amount of memory (in MBytes per processor) that the simulation requires. It also prints details of the initial thermodynamic state of the system. During the run itself, thermodynamic information is printed periodically, every few timesteps. When the run concludes, LAMMPS prints the final thermodynamic state and a total run time for the simulation. It also appends statistics about the CPU time and storage requirements for the simulation. An example set of statistics is shown here:

```
Loop time of 0.942801 on 4 procs for 300 steps with 2004 atoms

Performance: 54.985 ns/day, 0.436 hours/ns, 318.201 timesteps/s, 637.674 katom-step/s
195.2% CPU use with 2 MPI tasks x 2 OpenMP threads

MPI task timing breakdown:
Section |  min time  |  avg time  |  max time  |%varavg| %total
---------------------------------------------------------------
Pair    | 0.61419    | 0.62872    | 0.64325    |   1.8 | 66.69
Bond    | 0.0028608  | 0.0028899  | 0.002919   |   0.1 |  0.31
Kspace  | 0.12652    | 0.14048    | 0.15444    |   3.7 | 14.90
Neigh   | 0.10242    | 0.10242    | 0.10242    |   0.0 | 10.86
Comm    | 0.026753   | 0.027593   | 0.028434   |   0.5 |  2.93
Output  | 0.00018341 | 0.00030942 | 0.00043542 |   0.0 |  0.03
```

```
Modify  | 0.039117   | 0.039348   | 0.039579   |   0.1 |  4.17
Other   |            | 0.001041   |            |       |  0.11

Nlocal:           1002 ave        1006 max         998 min
Histogram: 1 0 0 0 0 0 0 0 0 1
Nghost:         8670.5 ave        8691 max        8650 min
Histogram: 1 0 0 0 0 0 0 0 0 1
Neighs:         354010 ave      357257 max      350763 min
Histogram: 1 0 0 0 0 0 0 0 0 1

Total # of neighbors = 708020
Ave neighs/atom = 353.30339
Ave special neighs/atom = 2.3403194
Neighbor list builds = 26
Dangerous builds = 0
```

---

The first section provides a global loop timing summary. The *loop time* is the total wall-clock time for the MD steps of the simulation run, excluding the time for initialization and setup (i.e. the parts that may be skipped with *run N pre no*). The *Performance* line is provided for convenience to help predict how long it will take to run a desired physical simulation and to have numbers useful for performance comparison between different simulation settings or system sizes. The *CPU use* line provides the CPU utilization per MPI task; it should be close to 100% times the number of OpenMP threads (or 1 if not using OpenMP). Lower numbers correspond to delays due to file I/O or insufficient thread utilization from parts of the code that have not been multi-threaded.

---

The *MPI task* section gives the breakdown of the CPU run time (in seconds) into major categories:

- *Pair* = non-bonded force computations
- *Bond* = bonded interactions: bonds, angles, dihedrals, impropers
- *Kspace* = long-range interactions: Ewald, PPPM, MSM
- *Neigh* = neighbor list construction
- *Comm* = inter-processor communication of atoms and their properties
- *Output* = output of thermodynamic info and dump files
- *Modify* = fixes and computes invoked by fixes
- *Other* = all the remaining time

For each category, there is a breakdown of the least, average and most amount of wall time any processor spent on this category of computation. The "%varavg" is the percentage by which the max or min varies from the average. This is an indication of load imbalance. A percentage close to 0 is perfect load balance. A large percentage is imbalance. The final "%total" column is the percentage of the total loop time is spent in this category.

When using the *timer full* setting, an additional column is added that also prints the CPU utilization in percent. In addition, when using *timer full* and the *package omp* command are active, a similar timing summary of time spent in threaded regions to monitor thread utilization and load balance is provided. A new *Thread timings* section is also added, which lists the time spent in reducing the per-thread data elements to the storage for non-threaded computation. These thread timings are measured for the first MPI rank only and thus, because the breakdown for MPI tasks can change from MPI rank to MPI rank, this breakdown can be very different for individual ranks. Here is an example output for this section:

```
Thread timings breakdown (MPI rank 0):
Total threaded time 0.6846 / 90.6%
```

```
Section |  min time  |  avg time  |  max time  |%varavg| %total
---------------------------------------------------------------
Pair    | 0.5127     | 0.5147     | 0.5167     |   0.3 | 75.18
Bond    | 0.0043139  | 0.0046779  | 0.0050418  |   0.5 |  0.68
Kspace  | 0.070572   | 0.074541   | 0.07851    |   1.5 | 10.89
Neigh   | 0.084778   | 0.086969   | 0.089161   |   0.7 | 12.70
Reduce  | 0.0036485  | 0.003737   | 0.0038254  |   0.1 |  0.55
```

The third section above lists the number of owned atoms (Nlocal), ghost atoms (Nghost), and pairwise neighbors stored per processor. The max and min values give the spread of these values across processors with a 10-bin histogram showing the distribution. The total number of histogram counts is equal to the number of processors.

---

The last section gives aggregate statistics (across all processors) for pairwise neighbors and special neighbors that LAMMPS keeps track of (see the *special_bonds* command). This section will not always contain data, for example when there has not been a neighbor rebuild, or the neighbor list was constructed on the GPU or when a hybrid pair style was used and LAMMPS cannot determine a suitable (base) neighbor list to draw the statistics from.

The number of times neighbor lists were rebuilt is tallied, as is the number of potentially *dangerous* rebuilds. If atom movement triggered neighbor list rebuilding (see the *neigh_modify* command), then dangerous reneighborings are those that were triggered on the first timestep atom movement was checked for. If this count is non-zero you may wish to reduce the delay factor to ensure no force interactions are missed by atoms moving beyond the neighbor skin distance before a rebuild takes place.

---

If an energy minimization was performed via the *minimize* command, additional information is printed, e.g.

```
Minimization stats:
  Stopping criterion = linesearch alpha is zero
  Energy initial, next-to-last, final =
         -6372.3765206     -8328.46998942     -8328.46998942
  Force two-norm initial, final = 1059.36 5.36874
  Force max component initial, final = 58.6026 1.46872
  Final line search alpha, max atom move = 2.7842e-10 4.0892e-10
  Iterations, force evaluations = 701 1516
```

The first line prints the criterion that determined minimization was converged. The next line lists the initial and final energy, as well as the energy on the next-to-last iteration. The next 2 lines give a measure of the gradient of the energy (force on all atoms). The 2-norm is the "length" of this 3N-component force vector; the largest component (x, y, or z) of force (infinity-norm) is also given. Then information is provided about the line search and statistics on how many iterations and force-evaluations the minimizer required. Multiple force evaluations are typically done at each iteration to perform a 1d line minimization in the search direction. See the *minimize* page for more details.

---

If a *kspace_style* long-range Coulombics solver that performs FFTs was used during the run (PPPM, Ewald), then additional information is printed, e.g.

```
FFT time (% of Kspce) = 0.200313 (8.34477)
FFT Gflps 3d 1d-only = 2.31074 9.19989
```

The first line is the time spent doing 3d FFTs (several per timestep) and the fraction it represents of the total KSpace time (listed above). Each 3d FFT requires computation (3 sets of 1d FFTs) and communication (transposes). The total flops performed is 5Nlog_2(N), where N is the number of points in the 3d grid. The FFTs are timed with and without the communication and a Gflop rate is computed. The 3d rate is with communication; the 1d rate is without (just the 1d FFTs). Thus you can estimate what fraction of your FFT time was spent in communication, roughly 75% in the example above.

## 4.4 Error message output

Depending on the error function arguments when it is called in the source code, there will be one to four lines of error output.

### 4.4.1 A single line

The line starts with "ERROR: ", followed by the error message and information about the location in the source where the error function was called in parenthesis on the right (here: line 131 of the file src/fix_print.cpp). Example:

```
ERROR: Fix print timestep variable nevery returned a bad timestep: 9900 (src/fix_print.
↪cpp:131)
```

### 4.4.2 Two lines

In addition to the single line output, also the last line of the input will be repeated. If a command is spread over multiple lines in the input using the continuation character '&', then the error will print the entire concatenated line. For readability all whitespace is compressed to single blanks. Example:

```
ERROR: Unrecognized fix style 'printf' (src/modify.cpp:924)
Last input line: fix 0 all printf v_nevery "Step: $(step) ${step}"
```

### 4.4.3 Three lines

In addition to the two line output from above, a third line is added that uses caret character markers '^' to indicate which "word" in the input failed. Example:

```
ERROR: Illegal fix print nevery value -100; must be > 0 (src/fix_print.cpp:41)
Last input line: fix 0 all print -100 "Step: $(step) ${stepx}"
                                 ^^^^
```

### 4.4.4 Four lines

The three line output is expanded to four lines, if the input is modified through input pre-processing, e.g. when substituting variables. Now the last command is printed once in the original form and a second time after substitutions are applied. The caret character markers '^' are applied to the second version. Example:

```
ERROR: Illegal fix print nevery value -100; must be > 0 (src/fix_print.cpp:41)
Last input line: fix 0 all print ${nevery} 'Step: $(step) ${step}'
--> parsed line: fix 0 all print -100 "Step: $(step) ${step}"
                                 ^^^^
```

## 4.5 File formats used by LAMMPS

This page provides a general overview of the kinds of files and file formats that LAMMPS is reading and writing.

**On this page**


- *Character Encoding*
- *Number Formatting*

- *Input File*
- *Data File*
- *Molecule File*
- *Restart File*
- *JSON Dump Files*

### 4.5.1 Character Encoding

For processing text files, the LAMMPS source code assumes ASCII character encoding which represents the digits 0 to 9, the lower and upper case letters a to z, some common punctuation and other symbols and a few whitespace characters including a regular "space character", "line feed", "carriage return", "tabulator". These characters are all represented by single bytes with a value smaller than 128 and only 95 of those 128 values represent printable characters. This list is sufficient to represent most English text, but misses accented characters or umlauts or Greek symbols and more.

Modern text often uses UTF-8 character encoding instead. This encoding is a way to represent many more different characters as defined by the Unicode standard. UTF-8 is compatible with ASCII, since the first 128 values are identical with the ASCII encoding. It is important to note, however, that there are Unicode characters that *look* similar to ASCII characters, but have a different binary representation. As a general rule, these characters may not be correctly recognized by LAMMPS. For some parts of LAMMPS' text processing, translation tables with known "lookalike" characters are used. The tables are used to substitute non-ASCII characters with their ASCII equivalents. Non-ASCII lookalike characters are often used by web browsers or PDF viewers to improve the readability of text. Thus, when using copy and paste to transfer text from such an application to your input file, you may unintentionally create text that is not exclusively using ASCII encoding and may cause errors when LAMMPS is trying to read it.

Lines with non-printable and non-ASCII characters in text files can be detected for example with a (Linux) command like the following:

```
env LC_ALL=C grep -n '[^ -~]' some_file.txt
```

### 4.5.2 Number Formatting

Different countries and languages have different conventions to format numbers. While in some regions commas are used for fractions and points to indicate thousand, million and so on, this is reversed in other regions. Modern operating systems have facilities to adjust input and output accordingly that are collectively referred to as "native language support" (NLS). The exact rules are often applied according to the value of the `$LANG` environment variable (e.g. "en_US.utf8" for English text in UTF-8 encoding).

For the sake of simplicity of the implementation and transferability of results, LAMMPS does not support this and instead expects numbers being formatted in the generic or "C" locale. The "C" locale has no punctuation for thousand, million and so on and uses a decimal point for fractions. One thousand would be represented as "1000.0" and not as "1,000.0" nor as "1.000,0". Having native language support enabled for a locale other than "C" will result in different behavior when converting or formatting numbers that can trigger unexpected errors.

LAMMPS also only accepts integer numbers when an integer is required, so using floating point equivalents like "1.0" are not accepted; you *must* use "1" instead.

For floating point numbers in scientific notation, the Fortran double precision notation "1.1d3" is not accepted; you have to use "1100", "1100.0" or "1.1e3".

### 4.5.3 Input File

A LAMMPS input file is a text file with commands. It is read line-by-line and each line is processed *immediately*. Before looking for commands and executing them, there is a pre-processing step where comments (non-quoted text starting with a pound sign '#') are removed, `${variable}` and `$(expression)` constructs are expanded or evaluated, and lines that end in the ampersand character '&' are combined with the next line (similar to Fortran 90 free-format source code). After the pre-processing, lines are split into "words" and evaluated. The first word must be a *command* and all following words are arguments. Below are some example lines:

```
# full line comment

# some global settings
units           lj
atom_style      atomic
# ^^ command    ^^ argument(s)

variable        x index 1       # may be overridden from command line with -var x <value>
variable        xx equal 20*$x  # variable "xx" is always 20 times "x"

lattice         fcc 0.8442

# example of a command written across multiple lines
# the "region" command uses spacing from "lattice" command, unless "units box" is␣
↪specified
region          box block 0.0 ${xx} &
                          0.0 40.0  &
                          0.0 30.0
# create simulation box and fill with atoms according to lattice setting
create_box      1 box
create_atoms    1 box

# set force field and parameters
mass            1 1.0
pair_style      lj/cut 2.5
pair_coeff      1 1 1.0 1.0 2.5

# run simulation
fix             1 all nve
run             1000
```

The pivotal command in this example input is the *create_box command*. It defines the simulation system and many parameters that go with it: units, atom style, number of atom types (and other types) and more. Those settings are *locked in* after the box is created. Commands that change these kind of settings are only allowed **before** a simulation box is created and many other commands are only allowed **after** the simulation box is defined (e.g. *pair_coeff*). Very few commands (e.g. *pair_style*) may be used in either part of the input. The *read_data* and *read_restart* commands also create the system box and thus have a similar pivotal function.

The LAMMPS input syntax has minimal support for conditionals and loops, but if more complex operations are required, it is recommended to use the library interface, e.g. *from Python using the LAMMPS Python module*.

There is a frequent misconception about the *if command*: this is a command for conditional execution **outside** a run or minimization. To trigger actions on specific conditions **during** a run is a non-trivial operation that usually requires adopting one of the available "fix" commands or creating a new "fix" command.

LAMMPS commands change the internal state and thus the order of commands matters and reordering them can produce different results. For example, the region defined by the *region command* in the example above depends on

the *lattice setting* and thus its dimensions will be different depending on the order of the two commands.

Each line must have an "end-of-line" character (line feed or carriage return plus line feed). Some text editors do not automatically insert one which may cause LAMMPS to ignore the last command. It is thus recommended to always have an empty line at the end of an input file.

The specific details describing how LAMMPS input is processed and parsed are explained in *Parsing rules for input scripts*.

### 4.5.4 Data File

A LAMMPS data file contains a description of a system suitable for reading with the *read_data command*. Data files are commonly used for setting up complex molecular systems that can be difficult to achieve with the commands *create_box* and *create_atoms* alone. Also, data files can be used as a portable alternatives to a *binary restart file*. A restart file can be converted into a data file from the *command line*.

Data files have a header section at the very beginning of the file and multiple titled sections such as "Atoms", "Masses", "Pair Coeffs", and so on. Header keywords can only be used *before* the first title section.

The data file **always** starts with a "title" line, which will be **ignored** by LAMMPS. Omitting the title line can lead to unexpected behavior because a line of the header with an actual setting may be ignored. In this case, the mistakenly ignored line often contains the "atoms" keyword, which results in LAMMPS assuming that there are no atoms in the data file and thus throwing an error on the contents of the "Atoms" section. The title line may contain some keywords that can be used by external programs to convey information about the system (included as comments), that is not required and not read by LAMMPS.

The line following a section title is also **ignored**. An error will occur if an empty line is not placed after a section title. The number of lines in titled sections depends on header keywords, like the number of atom types, the number of atoms, the number of bond types, the number of bonds, and so on. The data in those sections has to be complete. A special case are the "Pair Coeffs" and "PairIJ Coeffs" sections; the former is for force fields and pair styles that use mixing of non-bonded potential parameters, the latter for pair styles and force fields requiring explicit coefficients. Thus with $N$ being the number of atom types, the "Pair Coeffs" section has $N$ entries while "PairIJ Coeffs" has $N \cdot (N-1)$ entries. Internally, these sections will be converted to *pair_coeff* commands. Thus the corresponding *pair style* must have been set *before* the *read_data command* reads the data file.

Data files may contain comments, which start with the pound sign '#'. There must be at least one blank between a valid keyword and the pound sign. Below is a simple example case of a data file for *atom style full*.

```
LAMMPS Title line (ignored)
# full line comment

        10  atoms # comment
         4  atom types

 -36.840194 64.211560 xlo xhi
 -41.013691 68.385058 ylo yhi
 -29.768095 57.139462 zlo zhi

Masses

  1 12.0110
  2 12.0110
  3 15.9990
  4  1.0080

Pair Coeffs  # this section is optional
```

(continues on next page)

(continued from previous page)

```
  1    0.110000    3.563595    0.110000    3.563595
  2    0.080000    3.670503    0.010000    3.385415
  3    0.120000    3.029056    0.120000    2.494516
  4    0.022000    2.351973    0.022000    2.351973

Atoms # full

      1      1       1       0.560   43.99993  58.52678  36.78550   0   0   0
      2      1       2      -0.270   45.10395  58.23499  35.86693   0   0   0
      3      1       3      -0.510   43.81519  59.54928  37.43995   0   0   0
      4      1       4       0.090   45.71714  57.34797  36.13434   0   0   0
      5      1       4       0.090   45.72261  59.13657  35.67007   0   0   0
      6      1       4       0.090   44.66624  58.09539  34.85538   0   0   0
      7      1       3      -0.470   43.28193  57.47427  36.91953   0   0   0
      8      1       4       0.070   42.07157  57.45486  37.62418   0   0   0
      9      1       1       0.510   42.19985  57.57789  39.12163   0   0   0
     10      1       1       0.510   41.88641  58.62251  39.70398   0   0   0
#  ^^atomID ^^molID ^^type  ^^charge ^^xcoord  ^^ycoord  ^^ycoord  ^^image^^flags␣
↪(optional)

Velocities # this section is optional

      1  0.0050731  -0.00398928  0.00391473
      2 -0.0175184   0.0173484  -0.00489207
      3  0.00597225 -0.00202006  0.00166454
      4 -0.010395   -0.0082582   0.00316419
      5 -0.00390877  0.00470331 -0.00226911
      6 -0.00111157 -0.00374545 -0.0169374
      7  0.00209054 -0.00594936 -0.000124563
      8  0.00635002 -0.0120093  -0.0110999
      9 -0.004955   -0.0123375   0.000403422
     10  0.00265028 -0.00189329 -0.00293198
```

The common problem is processing the "Atoms" section, since its format depends on the *atom style* used, and that setting must be done in the input file *before* reading the data file. To assist with detecting incompatible data files, a comment is appended to the "Atoms" title indicating the atom style used (or intended) when *writing* the data file. For example, below is an "Atoms" section for *atom style charge*, which omits the molecule ID column.

```
Atoms # charge

      1      1       0.560   43.99993  58.52678  36.78550
      2      2      -0.270   45.10395  58.23499  35.86693
      3      3      -0.510   43.81519  59.54928  37.43995
      4      4       0.090   45.71714  57.34797  36.13434
      5      4       0.090   45.72261  59.13657  35.67007
      6      4       0.090   44.66624  58.09539  34.85538
      7      3      -0.470   43.28193  57.47427  36.91953
      8      4       0.070   42.07157  57.45486  37.62418
      9      1       0.510   42.19985  57.57789  39.12163
     10      1       0.510   41.88641  58.62251  39.70398
#  ^^atomID ^^type  ^^charge ^^xcoord  ^^ycoord  ^^ycoord
```

Another source of confusion about the "Atoms" section format is the ordering of columns. The three atom style variants *atom_style full*, *atom_style hybrid charge molecular*, and *atom_style hybrid molecular charge* all carry the same per-atom information. However, in data files, the Atoms section has the columns 'Atom-ID Molecule-ID Atom-type Charge X Y Z' for atom style full, but for hybrid atom styles the first columns are always 'Atom-ID Atom-type X Y Z' followed by any *additional* data added by the hybrid styles, for example, 'Charge Molecule-ID' for the first hybrid style and 'Molecule-ID Charge' in the second hybrid style variant. Finally, an alternative to a hybrid atom style is to use fix property/atom, e.g. to add molecule IDs to atom style charge. In this case the "Atoms" section is formatted according to atom style charge and a new section, "Molecules" is added that contains lines with 'Atom-ID Molecule-ID', one for each atom in the system. For adding charges to atom style molecular with fix property/atom, the "Atoms" section is now formatted according to the atom style and a "Charges" section is added.

### 4.5.5 Molecule File

Molecule files for use with the *molecule command* look quite similar to data files but they do not have a compatible format, i.e., one cannot use a data file as molecule file and vice versa. Below is a simple example for a water molecule (SPC/E model). Same as a data file, there is an ignored title line and you can use comments. However, there is no information about the number of types or the box dimensions. These parameters are set when the simulation box is created. Thus the header only has the count of atoms, bonds, and so on.

Molecule files have a header followed by sections (just as in data files), but the section names are different than those of a data file. There is no "Atoms" section and the section formats in molecule files is independent of the atom style. Its information is split across multiple sections, like "Coords", "Types", and "Charges". Note that no "Masses" section is needed here. The atom masses are by default tied to the atom type and set with a data file or the *mass command*. A "Masses" section would only be required for atom styles with per-atom masses, e.g. atom style sphere, where in data files you would provide the density and the diameter instead of the mass.

Since the entire file is a 'molecule', LAMMPS will assign a new molecule-ID (if supported by the atom style) when atoms are instantiated from a molecule file, e.g. with the *create_atoms command*. It is possible to include a "Molecules" section to indicate that the atoms belong to multiple 'molecules'. Atom-IDs and molecule-IDs in the molecule file are relative for the file (i.e. starting from 1) and will be translated into actual atom-IDs also when the atoms from the molecule are created.

```
# Water molecule. SPC/E model.

3 atoms
2 bonds
1 angles

Coords

1    1.12456   0.09298   1.27452
2    1.53683   0.75606   1.89928
3    0.49482   0.56390   0.65678

Types

1        1
2        2
3        2

Charges

1       -0.8472
2        0.4236
3        0.4236
```

(continues on next page)

(continued from previous page)

```
Bonds

1   1       1       2
2   1       1       3

Angles

1   1       2       1       3
```

There are also optional sections, e.g. about *SHAKE* and *special bonds*. Those sections are only needed if the molecule command is issued *before* the simulation box is defined. Otherwise, the molecule command can derive the required settings internally.

### 4.5.6 Restart File

LAMMPS restart files are binary files and not available in text format. They can be identified by the first few bytes that contain the (C-style) string `LammpS RestartT` as magic string. This string is followed by a 16-bit integer of the number 1 used for detecting whether the computer writing the restart has the same endianness as the computer reading it. If not, the file cannot be read correctly. This integer is followed by a 32-bit integer indicating the file format revision (currently 3), which can be used to implement backward compatibility for reading older revisions.

This information has been added to the *Unix "file" command's <https://www.darwinsys.com/file/>* "magic" file so that restart files can be identified without opening them. If you have a fairly recent version, it should already be included. If you have an older version, the LAMMPS source package *contains a file with the necessary additions*.

The rest of the file is organized in sections of a 32-bit signed integer constant indicating the kind of content and the corresponding value (or values). If those values are arrays (including C-style strings), then the integer constant is followed by a 32-bit integer indicating the length of the array. This mechanism will read the data regardless of the ordering of the sections. Symbolic names of the section constants are in the `lmprestart.h` header file.

LAMMPS restart files are not expected to be portable between platforms or LAMMPS versions, but changes to the file format are rare.

### 4.5.7 JSON Dump Files

LAMMPS can print information about molecules and other sets of atoms during a run, in JSON format. Several fixes currently support dumping JSON files in the 'molecules' style, including *fix bond/react* and the *delete* keyword of *fix reaxff/species*. The JSON 'dump molecules' format lists sets of atoms in the style of the *JSON molecule file*, where more discussion of JSON schema can be found. Here is an generic example of a JSON output file that dumped one water molecule on the first timestep:

```
{
    "application": "LAMMPS",
    "units": "real",
    "format": "dump",
    "style": "molecules",
    "revision": 1,
    "timesteps": [
        {
            "timestep": 1,
            "molecules": [
                {
                    "types": {
```

(continues on next page)

(continued from previous page)

```
                        "format": ["atom-id", "type"],
                        "data": [
                            [1368, "H"],
                            [1366, "O"],
                            [1367, "H"]
                        ]
                    },
                    "coords": {
                        "format": ["atom-id", "x", "y", "z"],
                        "data": [
                            [1368, 26.787767440427466, 29.785528640296768, 25.
↪85197353660144],
                            [1366, 26.641801222582824, 29.868106247702887, 24.
↪91285138212243],
                            [1367, 25.69611192416744, 30.093425787807448, 24.
↪914380215672846]
                        ]
                    }
                }
            ]
        }
    ]
}
```

The required first-level keys of the JSON format output are "application", "format", "style", "revision", and "timesteps", and optional keys are "units" and "title". The value of the "timesteps" key is an array of objects that contain data for each timestep on which a molecule was dumped, and the other first-level keys identify this JSON schema. The objects in "timesteps" contains two mandatory keys, "timestep" and "molecules". The "molecules" key is an array of *LAMMPS molecule JSON* objects, and may contain other keys that contain metadata for each molecule. The "format" keys within molecule JSON objects are only printed once per output file, for brevity. The "atom-id" values are atom IDs from the simulation, and the "type" values are atom types. In the above example, the types were reported as strings corresponding to elements using *type labels*.

## 4.6 Running LAMMPS on Windows

To run a serial (non-MPI) executable, follow these steps:

- Install a LAMMPS installer package from https://packages.lammps.org/windows.html
- Open the "Command Prompt" or "Terminal" app.
- Change to the directory where you have your input script, (e.g. by typing: cd "Documents").
- At the command prompt, type "lmp -in in.file.lmp", where `in.file.lmp` is the name of your LAMMPS input script.

Note that the serial executable includes support for multi-threading parallelization from the styles in the OPENMP and KOKKOS packages. To run with 4 threads, you can type this:

```
lmp -in in.lj.lmp -pk omp 4 -sf omp
lmp -in in.lj.lmp -k on t 4 -sf kk
```

Alternately, you can also install a package with LAMMPS-GUI included and open the LAMMPS-GUI app (the package includes the command-line version of LAMMPS as well) and open the input file in the GUI and run it from there. For details on LAMMPS-GUI, see https://lammps-gui.lammps.org/

For the MS-MPI executables, which allow you to run LAMMPS under Windows in parallel using MPI rather than multi-threading, follow these steps.

Download and install the MS-MPI runtime package `msmpisetup.exe` from https://www.microsoft.com/en-us/download/details.aspx?id=105289 (Note that the `msmpisdk.msi` is **only** required for **compilation** of LAMMPS from source on Windows using Microsoft Visual Studio). After installation of MS-MPI perform a reboot.

Then you can run the executable in serial like in the example above or in parallel using MPI with one of the following commands:

```
mpiexec -localonly 4 lmp -in in.file.lmp
mpiexec -np 4 lmp -in in.file.lmp
```

where `in.file.lmp` is the name of your LAMMPS input script. For the latter case, you may be prompted to enter the password that you set during installation of the MPI library software.

In this mode, output may not immediately show up on the screen, so if your input script takes a long time to execute, you may need to be patient before the output shows up.

Note that the parallel executable also includes OpenMP multi-threading through both the OPENMP and the KOKKOS package, which can be combined with MPI using something like:

```
mpiexec -localonly 2 lmp -in in.lj.lmp -pk omp 2 -sf omp
mpiexec -localonly 2 lmp -in in.lj.lmp -kokkos on t 2 -sf kk
```

MPI parallelization will work for *all* functionality in LAMMPS and in many cases the MPI parallelization is more efficient than multi-threading since LAMMPS was designed from the ground up for MPI parallelization using domain decomposition. Multi-threading is only available for selected styles and implemented on top of the MPI parallelization. Multi-threading is most useful for systems with large load imbalances when using domain decomposition and a smaller number of threads (<= 8).

CHAPTER

FIVE

# ERRORS

These doc pages describe many of the error and warning message you can encounter when using LAMMPS. The common problems include conceptual issues. The messages and warnings doc pages give complete lists of all the messages the code may generate, with additional details for many of them.

## 5.1 Common issues that are often regarded as bugs

The list below are some random notes on behavior of LAMMPS that is sometimes unexpected or even considered a bug. Most of the time, these are just issues of understanding how LAMMPS is implemented and parallelized. Please also have a look at the *Error details discussions page* that contains recommendations for tracking down issues and explanations for error messages that may sometimes be confusing or need additional explanations.

- A LAMMPS simulation typically has two stages, 1) issuing commands and 2) run or minimize. Most LAMMPS errors are detected in stage 1), others at the beginning of stage 2), and finally others like a bond stretching too far may or lost atoms or bonds may not occur until the middle of a run.
- If two LAMMPS runs do not produce the exact same answer on different machines or different numbers of processors, this is typically not a bug. In theory you should get identical answers on any number of processors and on any machine. In practice, numerical round-off can cause slight differences and eventual divergence of molecular dynamics phase space trajectories within a few 100s or few 1000s of timesteps. This can be triggered by different ordering of atoms due to different domain decompositions, but also through different CPU architectures, different operating systems, different compilers or compiler versions, different compiler optimization levels, different FFT libraries. However, the statistical properties of the two runs (e.g. average energy or temperature) should still be the same.
- If the *velocity* command is used to set initial atom velocities, a particular atom can be assigned a different velocity when the problem is run on a different number of processors or on different machines. If this happens, the phase space trajectories of the two simulations will rapidly diverge. See the discussion of the *loop* option in the *velocity* command for details and options that avoid this issue.
- Similarly, the *create_atoms* command generates a lattice of atoms. For the same physical system, the ordering and numbering of atoms by atom ID may be different depending on the number of processors.
- Some commands use random number generators which may be setup to produce different random number streams on each processor and hence will produce different effects when run on different numbers of processors. A commonly-used example is the *fix langevin* command for thermostatting.
- LAMMPS tries to flag errors and print informative error messages so you can fix the problem. For most errors it will also print the last input script command that it was processing or even point to the keyword that is causing troubles. Of course, LAMMPS cannot figure out your physics or numerical mistakes, like choosing too big a timestep, specifying erroneous force field coefficients, or putting 2 atoms on top of each other! Also, LAMMPS does not know what you *intend* to do, but very strictly applies the syntax as described in the documentation. If you run into errors that LAMMPS does not catch that you think it should flag, please send an email to the developers or create a new topic on the dedicated MatSci forum section.

- If you get an error message about an invalid command in your input script, you can determine what command is causing the problem by looking in the log.lammps file or using the *echo command* to see it on the screen. If you get an error like "Invalid ... style", with ... being fix, compute, pair, etc, it means that you mistyped the style name or that the command is part of an optional package which was not compiled into your executable. The list of available styles in your executable can be listed by using *the -h command-line switch*. The installation and compilation of optional packages is explained on the *Build packages* doc page.
- For a given command, LAMMPS expects certain arguments in a specified order. If you mess this up, LAMMPS will often flag the error, but it may also simply read a bogus argument and assign a value that is valid, but not what you wanted. E.g. trying to read the string "abc" as an integer value of 0. Careful reading of the associated doc page for the command should allow you to fix these problems. In most cases, where LAMMPS expects to read a number, either integer or floating point, it performs a stringent test on whether the provided input actually is an integer or floating-point number, respectively, and reject the input with an error message (for instance, when an integer is required, but a floating-point number 1.0 is provided):

```
ERROR: Expected integer parameter instead of '1.0' in input script or data file
```

- Some commands allow for using variable references in place of numeric constants so that the value can be evaluated and may change over the course of a run. This is typically done with the syntax *v_name* for a parameter, where name is the name of the variable. On the other hand, immediate variable expansion with the syntax ${name} is performed while reading the input and before parsing commands,

> **Note**
>
> Using a variable reference (i.e. *v_name*) is only allowed if the documentation of the corresponding command explicitly says it is. Otherwise, you will receive an error message of this kind:

```
ERROR: Expected floating point parameter instead of 'v_name' in input script or␣
↪data file
```

- Generally, LAMMPS will print a message to the screen and logfile and exit gracefully when it encounters a fatal error. When running in parallel this message may be stuck in an I/O buffer and LAMMPS will be terminated before that buffer is printed. In that case you can try adding the `-nonblock` or `-nb` command-line flag to turn off that buffering. Please note that this should not be used for production runs, since turning off buffering usually has a significant negative impact on performance (even worse than *thermo_modify flush yes*). Sometimes LAMMPS will print a WARNING to the screen and logfile and continue on; you can decide if the WARNING is important or not, but as a general rule do not ignore warnings that you not understand. A WARNING message that is generated in the middle of a run is only printed to the screen, not to the logfile, to avoid cluttering up thermodynamic output. If LAMMPS crashes or hangs without generating an error message first then it could be a bug (see *this section*).
- LAMMPS runs in the available memory a processor allows to be allocated. Most reasonable MD runs are compute limited, not memory limited, so this should not be a bottleneck on most platforms. Almost all large memory allocations in the code are done via C-style malloc's which will generate an error message if you run out of memory. Smaller chunks of memory are allocated via C++ "new" statements. If you are unlucky you could run out of memory just when one of these small requests is made, in which case the code will crash or hang (in parallel).
- Illegal arithmetic can cause LAMMPS to run slow or crash. This is typically due to invalid physics and numerics that your simulation is computing. If you see wild thermodynamic values or NaN values in your LAMMPS output, something is wrong with your simulation. If you suspect this is happening, it is a good idea to print out thermodynamic info frequently (e.g. every timestep) via the *thermo* so you can monitor what is happening. Visualizing the atom movement is also a good idea to ensure your model is behaving as you expect.
- When running in parallel with MPI, one way LAMMPS can hang is because LAMMPS has come across an error condition, but only on one or a few MPI processes and not all of them. LAMMPS has two different "stop with

an error message" functions and the correct one has to be called or else it will hang.

## 5.2 Errors and warnings details

Many errors and warnings that LAMMPS outputs are self-explanatory and thus straightforward to resolve. However, there are also cases where there is no single cause or simple explanation that can be provided in a short message printed by LAMMPS. Therefore, more detailed discussions of such scenarios are provided here; first on a more general level and then for specific errors. In the latter cases, LAMMPS will output a short message and then provide a URL that links to a specific section on this page.

---

**Individual paragraphs**


- *General troubleshooting advice*
  - *Create a small test system*
  - *Visualize your trajectory*
  - *Parallel versus serial*
  - *Segmentation Fault*
  - *Fast moving atoms*
  - *Ignoring lost atoms*
  - *Pressure, forces, positions becoming NaN or Inf*
  - *Communication cutoff*
  - *Neighbor list settings*
  - *Units*
  - *No error message printed*
  - *Errors before or after the simulation box is created*
  - *Illegal … command*
- *Unknown identifier in data file*
- *Incorrect format in … section of data file*
- *Illegal variable command: expected X arguments but found Y*
- *Out of range atoms - cannot compute …*
- *Bond (or angle, dihedral, improper, cmap, or shake) atoms missing*
- *Non-numeric atom coords or pressure or box dimensions - simulation unstable*
- *Fix used in … not computed at compatible time*
- *Lost atoms …*
- *Too many neighbor bins*
- *Unrecognized … style …*
- *Energy or stress was not tallied by pair style*
- *fmt::format_error*

- *Substitution for illegal variable*
- *Bond atom missing in image check or box size check*
- *Cannot use neighbor bins - box size << cutoff*
- *Did not assign all atoms correctly*
- *Domain too large for neighbor bins*
- *Step X: (h)bondchk failed*
- *Numeric index X is out of bounds*
- *Compute, fix, or variable vector or array is accessed out-of-range*
- *Incorrect args for pair coefficients (also bond/angle/dihedral/improper coefficients)*
- *Energy was not tallied on needed timestep (also virial, per-atom energy, per-atom virial)*
- *Molecule auto special bond generation overflow*
- *Molecule topology/atom exceeds system topology/atom*
- *Molecule topology type exceeds system topology type*
- *Molecule attributes do not match system attributes*
- *Inconsistent image flags*
- *No fixes with time integration, atoms won't move*
- *System is not charge neutral, net charge = ...*
- *Variable evaluation before simulation box is defined*
- *Invalid thermo keyword 'X' in variable formula*
- *One or more atoms are time integrated more than once*
- *XXX command before simulation box is defined*
- *XXX command after simulation box is defined*
- *Error messages ending in 'Please contact the LAMMPS developers'*
- *Neighbor list overflow, boost neigh_modify one*
- *Variable ...: Compute/Fix ... does not compute requested property*
- *The ... style ... is no longer available*

### 5.2.1 General troubleshooting advice

Below are suggestions that can help to understand the causes of problems with simulations leading to errors or unexpected results.

#### Create a small test system

Debugging problems often requires running a simulation many times with small modifications, thus it can be a huge time saver to first assemble a small test system input that has the same issue, but will take much less time until it triggers the error condition. Also, it will be easier to see what happens when visualizing the system or looking at output files.

### Visualize your trajectory

To better understand what is causing problems, it is often very useful to visualize the system close to the point of failure. It may be necessary to have LAMMPS output trajectory frames rather frequently. To avoid gigantic files, you can use *dump_modify delay* to delay output until the critical section is reached, and you can use a smaller test system (see above).

### Parallel versus serial

Issues where something is "lost" or "missing" often exhibit that issue *only* when running in parallel. That doesn't mean there is no problem when running in serial, only the symptoms are not triggering an error. This may be because there is no domain decomposition with just one processor and thus all atoms are accessible, or it may be because the problem will manifest faster with smaller subdomains. Correspondingly, errors may be triggered faster with more processors and thus smaller sub-domains.

### Segmentation Fault

A segmentation fault is an error reported by the **operating system** and not LAMMPS itself. It happens when a process tries to access a memory address that is not available. This can have **many** reasons: memory has not been allocated, a memory buffer is not large enough, a memory address is computed from an incorrect index, a memory buffer is used after it has been freed, some general memory corruption. When investigating a segmentation fault (aka segfault), it is important to determine which process is causing it; it may not always be LAMMPS. For example, some MPI library implementations report a segmentation fault from their "mpirun" or "mpiexec" command when the application has been terminated unexpectedly.

While a segmentation fault is likely an indication of a bug in LAMMPS, it need not always be; it can also be the consequence of too aggressive simulation settings. For time critical code paths, LAMMPS will assume the user has chosen the settings carefully and will not make any checks to avoid performance penalties.

A crucial step in resolving a segmentation fault is to identify the exact location in the code where it happens. Please see *Debugging crashes* for a couple of examples showing how to do this on a Linux machine. With this information – a simple way to reproduce the segmentation fault and the exact *LAMMPS version* and platform you are running on – you can contact the LAMMPS developers or post in the LAMMPS forum to get assistance.

### Fast moving atoms

Fast moving atoms may be "lost" or "missing" when their velocity becomes so large that they can cross a sub-domain within one timestep. This often happens when atoms are too close, but atoms may also "move" too fast from sub-domain to sub-domain if the box changes rapidly. E.g. when setting a large initial box with *shrink-wrap boundary conditions* that collapses on the first step (in this case the solution is often using 'm' instead of 's' as a boundary condition).

To reduce the impact of "close contacts", one can remove those atoms or molecules with something like *delete_atoms overlap 0.1 all all*. With periodic boundaries, a close contact pair of atoms may be on opposite sides of the simulation box. Another option would be to first run a minimization (aka quench) before starting the MD. Reducing the time step can also help. Many times, one just needs to "ease" the system into a balanced state and can then switch to more aggressive settings.

The speed of atoms during an MD run depends on the steepness of the potential function and their mass. Since the positions and velocities of atoms are computed with finite timesteps, the timestep needs to be small enough for stable numeric integration of the trajectory. If the timestep is too large during initialization (or other instances of extreme dynamics), using *fix nve/limit* or *fix dt/reset* temporarily can help to avoid too large updates or adapt the timestep according to the displacements.

### Ignoring lost atoms

It is tempting to use the *thermo_modify lost ignore* to avoid LAMMPS aborting with an error on lost atoms. This setting should, however, *only* be used when atoms *should* leave the system. In general, ignoring a problem does not solve it.

### Pressure, forces, positions becoming NaN or Inf

Some potentials can overflow or have a division by zero with close contacts or bad geometries (for the given force styles in use) leading to forces that can no longer be represented as numbers. Those will show as "NaN" or "Inf". On most machines, the program will continue, but there is no way to recover from it and those NaN or Inf values will propagate.

If the "NaN" or "Inf" appears in the first simulation step, the most common cause is overlapping atoms. Note that when atoms are *very* close, this cannot be seen when visualizing the geometry, since the atoms are effectively sitting on top of each other. A good test is to insert a command like *delete_atoms 0.1 all all* and then monitor the output to see how many atoms are deleted, if any. A non-zero number would be an indication of overlapping atoms. Note that atoms can also overlap through periodic boundaries when the box dimensions are too small (e.g. determined by min/max position of atoms without padding).

So-called *"soft-core" potentials* or the *"soft" repulsive-only pair style* are less prone for this behavior (depending on the settings in use) and can be used at the beginning of a simulation. Also, single precision numbers can overflow much faster, so for the GPU, KOKKOS, or INTEL package it may be beneficial to run with double precision initially before switching to mixed or single precision for faster execution when the system has relaxed.

### Communication cutoff

The communication cutoff determines the "overlap" between sub-domains and atoms in these regions are referred to in LAMMPS as "ghost atoms". This region has to be large enough to contain all atoms of a bond, angle, dihedral, or improper with just one atom in the actual sub-domain. Typically, this cutoff is set to the largest cutoff from the *pair style(s)* plus the *neighbor list skin distance* and will typically be sufficient for all bonded interactions. But if the pair style cutoff is small (e.g. with a repulsive-only Lennard-Jones potential) this may not be enough. It is even worse if there is no pair style defined (or the pair style is set to "none"), since then there will be no ghost atoms created at all.

The communication cutoff can be set or adjusted with *comm_modify cutoff <value>*, but setting this too large will waste CPU time and memory. LAMMPS will print warnings in these cases. For bonds it uses some heuristic based on the equilibrium bond length, but that still may not be sufficient for cases where the force constants are small and thus bonds may be stretched very far.

### Neighbor list settings

Every time LAMMPS rebuilds the neighbor lists, LAMMPS will also check for "lost" or "missing" atoms. Thus it can help to use very conservative *neighbor list settings* and then examine the neighbor list statistics if the neighbor list rebuild can be safely delayed. Rebuilding the neighbor list less frequently (i.e. through increasing the *delay* or *every*) setting has diminishing returns and increasing risks.

### Units

A frequent cause for a variety of problems is due to using the wrong *units* settings for a particular potentials, especially when reading them from a potential file. Most of the (example) potentials bundled with LAMMPS have a "UNITS:" tag that allows LAMMPS to check of the units are consistent with what is intended, but potential files from publications or potential parameter databases may lack this metadata information and thus will not error out or warn when using the wrong setting. Most potential files usually use "metal" units, but some are parameterized for other settings, most notably *ReaxFF potentials* that use "real" units.

Also, individual parameters for *pair_coeff* commands taken from publications or other MD software may need to be converted and sometimes in unexpected ways. Thus some careful checking is recommended.

#### No error message printed

In some cases – especially when running in parallel with MPI – LAMMPS may stop without displaying an error. But the fact that nothing was displayed does not mean there was not an error message. Instead it is highly likely that the message was written to a buffer and LAMMPS was aborted before the buffer was output. Usually, output buffers are output for every line of output, but sometimes this is delayed until 4096 or 8192 bytes of output have been accumulated. This buffering for screen and logfile output can be disabled by using the *-nb or -nonbuf* command-line flag. This is most often needed when debugging crashing multi-replica calculations.

#### Errors before or after the simulation box is created

As critical step in a LAMMPS input is when the simulation box is defined, either with a *create_box command*, a *read_data command*, or a *read_restart command*. After this step, certain settings are locked in (e.g. units, or number of atom, bond, angle, dihedral, improper types) and cannot be changed after that. Consequently, commands that change such settings (e.g. *units*) are only allowed before the box is defined. Very few commands can be used before and after, like *pair_style* (but not *pair_coeff*). Most LAMMPS commands must be used after the simulation box is created.

Consequently, LAMMPS will stop with an error, if a command is used in the wrong place. This is not always obvious. So index or string style *variables* can be expanded anywhere in the input, but equal style (or similar) variables can only be expanded before the box is defined if they do not reference anything that cannot be defined before the box (e.g. a compute or fix reference or a thermo keyword).

#### Illegal ... command

These are catchall error messages that used to be used a *lot* in LAMMPS (also programmers are sometimes lazy). They usually include the name of the source file and the line where the error happened. This can be used to track down what caused the error (most often some form of syntax error) by looking at the source code. However, this has two disadvantages: 1. one has to check the source file from the exact same LAMMPS version, or else the line number would be different or the core may have been rewritten and that specific error does not exist anymore.

The LAMMPS developers are committed to replace these too generic error messages with more descriptive errors, e.g. listing *which* keyword was causing the error, so that it will be much simpler to look up the correct syntax in the manual (and without referring to the source code).

---

### 5.2.2 Unknown identifier in data file

This error happens when LAMMPS encounters a line of text with an unexpected keyword while *reading a data file*. This would be either header keywords or section header keywords. This is most commonly due to a mistyped keyword or due to a keyword that is inconsistent with the *atom style* used.

The header section informs LAMMPS how many entries or lines are expected in the various sections (like Atoms, Masses, Pair Coeffs, *etc.*) of the data file. If there is a mismatch, LAMMPS will either keep reading beyond the end of a section or stop reading before the section has ended. In that case the next line will not contain a recognized keyword.

Such a mismatch can also happen when the first line of the data is *not* a comment as required by the format, but a line with a valid header keyword. That would result in LAMMPS expecting, for instance, 0 atoms because the "atoms" header line is the first line and thus treated as a comment.

Another possibility to trigger this error is to have a keyword in the data file that corresponds to a fix (e.g. *fix cmap*) but the *read_data* command is missing the (optional) arguments that identify the fix and its header and section keywords. Alternatively, those arguments are inconsistent with the keywords in the data file.

### 5.2.3 Incorrect format in ... section of data file

This error happens when LAMMPS reads the contents of a section of a *data file* and the number of parameters in the line differs from what is expected. This most commonly happens when the atom style is different from what is expected for a specific data file since changing the atom style usually changes the format of the line.

This error can also occur when the number of entries indicated in the header of a data file (e.g. the number of atoms) is larger than the number of lines provided (e.g. in the corresponding Atoms section) causing LAMMPS to continue reading into the next section which has a completely different format.

### 5.2.4 Illegal variable command: expected X arguments but found Y

This error indicates that a variable command has either incorrectly formatted arguments or the wrong number of arguments. A common reason for this is that a variable expression contains whitespace, but is not enclosed in single or double quotes.

To explain, the LAMMPS input parser reads and processes lines. The resulting line is broken down into "words". Those are usually individual commands, labels, names, and values separated by whitespace (a space or tab character). For "words" that may contain whitespace, they have to be enclosed in single (') or double (") quotes. The parser will then remove the outermost pair of quotes and pass that string as single argument to the variable command.

Thus missing quotes or accidental extra whitespace will trigger this error because the unquoted whitespace will result in the text being broken into more "words" than expected, i.e. the variable expression being split.

### 5.2.5 Out of range atoms - cannot compute ...

The PPPM (and also PPPMDisp and MSM) methods need to assemble a grid of electron density data derived from the (partial) charges assigned to the atoms. These charges are smeared out across multiple grid points (see *kspace_modify order*). When running in parallel with MPI, LAMMPS uses a *domain decomposition scheme* where each processor manages a subset of atoms and thus also a grid representing the density. The processor's grid covers the actual volume of the sub-domain and some extra space corresponding to the *neighbor list skin*. These are then *combined and redistributed* for parallel processing of the long-range component of the Coulomb interaction.

The `Out of range atoms` error can happen when atoms move too fast, the neighbor list skin is too small, or the neighbor lists are not updated frequently enough. The smeared charges cannot then be fully assigned to the density grid for all atoms. LAMMPS checks for this condition and stops with an error. Most of the time, this is an indication of a system with very high forces, often at the beginning of a simulation or when boundary conditions are changed. The error becomes more likely with more MPI processes.

There are multiple options to explore for avoiding the error. The best choice depends strongly on the individual system, and often a combination of changes is required. For example, more conservative MD parameter settings can be used (larger neighbor skin, shorter time step, more frequent neighbor list updates). Sometimes, it helps to revisit the system generation and avoid close contacts when building it. Otherwise one can use the *delete_atoms overlap* command to delete those close contact atoms or run a minimization before the MD. It can also help to temporarily use a cutoff-Coulomb pair style and no kspace style until the system has somewhat equilibrated and then switch to the long-range solver.

### 5.2.6 Bond (or angle, dihedral, improper, cmap, or shake) atoms missing

The second atom needed to compute a particular bond (or the third or fourth atom for angle, dihedral, or improper) is missing on the indicated timestep and processor. Typically, this is because the two bonded atoms have become too far apart relative to the communication cutoff distance for ghost atoms. By default, the communication cutoff is set by the pair cutoff. However, to accommodate larger distances between topologically connected atoms, it can be manually adjusted using *comm_modify* at the cost of increased communication and more ghost atoms. However, missing bond atoms may also indicate that there are unstable dynamics which caused the atoms to blow apart. In this scenario, increasing the communication distance will not solve the underlying issue. Rather, see *Fast moving atoms* and *Neighbor list settings* in the general troubleshooting section above for ideas to fix unstable dynamics.

If atoms are intended to be lost during a simulation (e.g. due to open boundary conditions or *fix evaporate*) such that two bonded atoms may be lost at different times from each other, this error can be converted to a warning or turned off using the *lost/bond* keyword in the *thermo_modify* command.

### 5.2.7 Non-numeric atom coords or pressure or box dimensions - simulation unstable

This error usually occurs due to overly aggressive simulation settings or issues with the system geometry or the potential. See *Pressure, forces, positions becoming NaN or Inf* above in the general troubleshooting section. This error is more likely to happen during equilibration, so it can help to do a minimization before or even add a second or third minimization after running a few equilibration MD steps. It also is more likely when directly using a Nose-Hoover (or other) barostat, and thus it may be advisable to run with only a thermostat for a bit until the potential energy has stabilized.

### 5.2.8 Fix used in . . . not computed at compatible time

Many fix styles are invoked only every *nevery* timesteps, which means their data is only valid on those steps. When data from a fix is used as input for a compute, a dump, another fix, or thermo output, it must read that data at timesteps when the fix in question was invoked, i.e. on timesteps that are multiples of its *nevery* setting. If this is not the case, LAMMPS will stop with an error. To remedy this, it may be required to change the output frequency or the *nevery* setting of the fix.

### 5.2.9 Lost atoms . . .

A simulation stopping with an error due to lost atoms can have multiple causes. By default, LAMMPS checks for whether the total number of atoms is consistent with the sum of atoms "owned" by MPI processors every time that thermodynamic output is written. In the majority of cases, lost atoms are unexpected and a result of extremely high velocities causing instabilities in the system. Such velocities can result from a variety of issues. For ideas on how to track down issues with unexpected lost atoms, see *Fast moving atoms* and *Neighbor list settings* in the general troubleshooting section above. In specific situations however, losing atoms is expected material behavior (e.g. with sputtering and surface evaporation simulations), and an unwanted crash can be avoided by changing the *thermo_modify lost* keyword from the default 'error' to 'warn' or 'ignore' (though heed the advice in *Ignoring lost atoms* above!).

### 5.2.10 Too many neighbor bins

The simulation box is or has become too large relative to the size of a neighbor bin (which in turn depends on the largest pair-wise cutoff by default) such that LAMMPS is unable to store the needed number of bins. This typically implies the simulation box has expanded too far. That can occur when some atoms move rapidly apart with shrink-wrap boundaries or when a fix (like fix deform or a barostat) excessively grows the simulation box. This can also happen if the largest pair-wise cutoff is small. In this case, the error can be avoided by using the *neigh_modify command* to either (a) set the bin width to a suitably large value or (b) use a hash table to store the bins for each atom. The latter option currently only supports the *multi* neighbor style and is intended for systems where there is a low particle density and a small cutoff. Namely, systems with very large ratios of minimum to maximum particle sizes/cutoffs. This may require adjustment of *atom sort settings*.

### 5.2.11 Unrecognized . . . style . . .

There are multiple variants of this error message. The most common case is that there is a typo or syntax error in the input file and the style name of a command was not found in the LAMMPS executable.

Another case is that the input is using the correct style command, but the LAMMPS executable in use was not compiled with the package containing that specific style. LAMMPS executables include tables of all available packages and styles in the distribution, and thus will print in this case a message indicating which package is missing. This indicates that the executable needs to be re-built after enabling the correct package. See the *LAMMPS build instructions* for more details on including packages. One can check if the expected package and style is present in the executable by running

it with the `-help` (or `-h`) flag on the command line. One common oversight, especially for LAMMPS users with limited experience in compiling software from source, is enabling the package but forgetting to rebuild or install the executable. One can also check the documentation for the style in question for a "Restrictions" section, which should indicate which requirements apply to a given command.

Finally, there is the case that the necessary package is included, but the missing style depends also on *another* style from a *different* package and *this* package is missing. In that case, LAMMPS does not know which package is missing, and it is necessary to check the documentation for the missing style in the manual.

If this error occurs with an executable that the user does not control (e.g., through a module on HPC clusters), the user will need to get in contact with the relevant person or people who can update the executable. In rare cases, there may be licensing or portability issues that prevent including a package in publicly accessible binaries or in a specific environment.

### 5.2.12 Energy or stress was not tallied by pair style

This warning can be printed by computes from the *TALLY package*. Those use a callback mechanism that only work for regular pair-wise additive pair styles like *Lennard-Jones*, *Morse*, *Born-Meyer-Huggins*, and similar. Such required callbacks have not been implemented for many-body potentials so one would have to implement them to add compatibility with these computes (which may be difficult to do in a generic fashion). Whether this warning indicates that contributions to the computed properties are missing depends on the groups used. At any rate, careful testing of the results is advised when this warning appears.

### 5.2.13 fmt::format_error

LAMMPS uses the {fmt} library for advanced string formatting tasks. This is similar to the `printf()` family of functions from the standard C library, but more flexible. If there is a bug in the LAMMPS code and the format string does not match the list of arguments or has some other error, this error message will be shown. You should contact the LAMMPS developers and report the bug as a GitHub Bug Report Issue along with sufficient information to easily reproduce it.

### 5.2.14 Substitution for illegal variable

A variable in an input script or a variable expression was not found in the list of valid variables. The most common reason for this is a typo somewhere in the input file such that the expression uses an invalid variable name. The second most common reason is omitting the curly braces for a direct variable with a name that is not a single letter. For example:

```
variable cutoff index 10.0
pair_style lj/cut ${cutoff}  # this is correct
pair_style lj/cut $cutoff    # this is incorrect, LAMMPS looks for 'c' instead of 'cutoff'
variable c      index 5.0    # if $c is defined, LAMMPS substitutes only '$c' and reads:␣
↪5utoff
```

Another potential source of this error may be invalid command line variables (-var or -v argument) used when launching LAMMPS from an interactive shell or shell scripts. An uncommon source for this error is using the *next command* to advance through a list of values provided by an index style variable. If there is no remaining element in the list, LAMMPS will delete the variable and any following expansion or reference attempt will trigger the error.

Users with harder-to-track variable errors might also find reading the *Parsing rules for input scripts* helpful.

### 5.2.15 Bond atom missing in image check or box size check

This can be either an error or a warning depending on your *thermo_modify settings*. It is flagged in a part of the LAMMPS code where it updates the domain decomposition and before it builds the neighbor lists. It checks that both atoms of a bond are within the communication cutoff of a subdomain. It is usually caused by atoms moving too fast (see

the *paragraph on fast moving atoms*), or by the *communication cutoff being too small*, or by waiting too long between *sub-domain and neighbor list updates*.

### 5.2.16 Cannot use neighbor bins - box size << cutoff

LAMMPS is unable to build neighbor bins since the size of the box is much smaller than an interaction cutoff in at least one of its dimensions. Typically, this error is triggered when the simulation box has one very thin dimension. If a cubic neighbor bin had to fit exactly within the thin dimension, then an inordinate amount of bins would be created to fill space. This error can be avoided using the generally slower *nsq neighbor style* or by increasing the size of the smallest box lengths.

### 5.2.17 Did not assign all atoms correctly

This error happens most commonly when *reading a data file* under *non-periodic boundary conditions*. Only atoms with positions **inside** the simulation box will be read and thus any atoms outside the box will be skipped and the total atom count will not match, which triggers the error. This does not happen with periodic boundary conditions where atoms outside the principal box will be "wrapped" into the principal box and their image flags set accordingly.

Similar errors can happen with the *replicate command* or the *read_restart command*. In these cases the cause may be a problematic geometry, an insufficient communication cutoff, or a bug in the LAMMPS source code. In these cases it is advisable to set up *small test case* for testing and debugging. This will be required in case you need to get help from a LAMMPS developer.

### 5.2.18 Domain too large for neighbor bins

The domain has become extremely large so that neighbor bins cannot be used. Too many neighbor bins would need to be created to fill space. Most likely, one or more atoms have been blown a great distance out of the simulation box or a fix (like fix deform or a barostat) has excessively grown the simulation box.

### 5.2.19 Step X: (h)bondchk failed

This error is a consequence of the heuristic memory allocations for buffers of the regular ReaxFF version. In ReaxFF simulations, the lists of bonds and hydrogen bonds can change due to chemical reactions. The default approach, however, assumes that these changes are not very large, so it allocates buffers for the current system setup plus a safety margin. This can be adjusted with the *safezone, mincap, and minhbonds settings of the pair style*, but only to some extent. When equilibrating a new system, or simulating a sparse system in parallel, this can be difficult to control and become wasteful. A simple workaround is often to break a simulation down in multiple chunks. A better approach, however, is to compile and use the KOKKOS package version of ReaxFF (you do not need a GPU for that, but can also compile it in serial or OpenMP mode), which uses a more robust memory allocation approach.

### 5.2.20 Numeric index X is out of bounds

This error most commonly happens when setting force field coefficients with either the *pair_coeff*, the *bond_coeff*, the *angle_coeff*, the *dihedral_coeff*, or the *improper_coeff* command. These commands accept type labels, explicit numbers, and wildcards for ranges of numbers. If the numeric value of any of these is outside the valid range (defined by the number of corresponding types), LAMMPS will stop with this error. A few other commands and styles also allow ranges of numbers and check using the same method and thus print the same kind of error.

The cause is almost always a typo in the input or a logic error when defining the values or ranges. So one needs to carefully review the input. Along with the error, LAMMPS will print the valid range as a hint.

### 5.2.21 Compute, fix, or variable vector or array is accessed out-of-range

When accessing an individual element of a global vector or array or a per-atom vector or array provided by a compute or fix or atom-style or vector-style variable or data from a specific atom, an index in square brackets ("[ ]") (or two indices) must be provided to determine which element to access and it must be in a valid range or else LAMMPS would access invalid data or crash with a segmentation fault. In the two most common cases, where this data is accessed, *variable expressions* and *thermodynamic output*, LAMMPS will check for valid indices and stop with an error otherwise.

While LAMMPS is written in C++ (which uses 0 based indexing) these indices start at 1 (i.e. similar to Fortran). Any index smaller than 1 or larger than the maximum allowed value should trigger this error. Since this kind of error frequently happens with rather complex expressions, it is recommended to test these with small test systems, where the values can be tracked with output files for all relevant properties at every step.

### 5.2.22 Incorrect args for pair coefficients (also bond/angle/dihedral/improper coefficients)

The parameters in the *pair_coeff* command for a specified *pair_style* have a missing or erroneous argument. The same applies when seeing this error for *bond_coeff*, *angle_coeff*, *dihedral_coeff*, or *improper_coeff* and their respective style commands when using the MOLECULE or EXTRA-MOLECULE packages. The cases below describe some ways to approach pair coefficient errors, but the same strategies apply to bonded systems as well.

Outside of normal typos, this error can have several sources. In all cases, the first step is to compare the command arguments to the expected format found in the corresponding *pair_style* page. This can reveal cases where, for example, a pair style was changed, but the pair coefficients were not updated. This can happen especially with pair style variants such as *pair_style eam* vs. *pair_style eam/alloy* that look very similar but accept different parameters (the latter 'eam/alloy' variant takes element type names while 'eam' does not).

Another common source of coefficient errors is when using multiple pair styles with commands such as *pair_style hybrid*. Using hybrid pair styles requires adding an extra "label" argument in the coefficient commands that designates which pair style the command line refers to. Moreover, if the same pair style is used multiple times, this label must be followed by an additional numeric argument. Also, different pair styles may require different arguments.

This error message might also require a close look at other LAMMPS input files that are read in by the input script, such as data files or restart files.

### 5.2.23 Energy was not tallied on needed timestep (also virial, per-atom energy, per-atom virial)

This error is generated when LAMMPS attempts to access an out-of-date or non-existent energy, pressure, or virial. For efficiency reasons, LAMMPS does *not* calculate these quantities when the forces are calculated on every timestep or iteration. Global quantities are only calculated when they are needed for *thermo* output (at the beginning, end, and at regular intervals specified by the *thermo* command). Similarly, per-atom quantities are only calculated if they are needed to write per-atom energy or virial to a dump file. This system works fine for simple input scripts. However, the many user-specified *variable*, *fix*, and *compute* commands that LAMMPS provides make it difficult to anticipate when a quantity will be requested. In some use cases, LAMMPS will figure out that a quantity is needed and arrange for it to be calculated on that timestep e.g. if it is requested by *fix ave/time* or similar commands. If that fails, it can be detected by a mismatch between the current timestep and when a quantity was last calculated, in which case an error message of this type is generated.

The most common cause of this type of error is requesting a quantity before the start of the simulation.

```
# run 0 post no               # this will fix the error
variable e equal pe           # requesting energy compute
print "Potential energy = $e" # this will generate the error
run 1000                      # start of simulation
```

This situation can be avoided by adding in a "run 0" command, as explained in more detail in the "Variable Accuracy" section of the *variable* doc page.

Another cause is requesting a quantity on a timestep that is not a thermo or dump output timestep. This can often be remedied by increasing the frequency of thermo or dump output.

### 5.2.24 Molecule auto special bond generation overflow

In order to correctly apply the *special_bonds* settings (also known as "exclusions"), LAMMPS needs to maintain for each atom a list of atoms that are connected to this atom, either directly with a bond or indirectly through bonding with an intermediate atom(s). The purpose is to either remove or tag those pairs of atoms in the neighbor list. This information is stored with individual atoms and thus the maximum number of such "special" neighbors is set when the simulation box is created. When reading (relative) geometry and topology of a 'molecule' from a *molecule file*, LAMMPS will build the list of such "special" neighbors for the molecule atom (if not given in the molecule file explicitly). The error is triggered when the resulting list is too long for the space reserved when creating the simulation box. The solution is to increase the corresponding setting. Overestimating this value will only consume more memory, and is thus a safe choice.

### 5.2.25 Molecule topology/atom exceeds system topology/atom

LAMMPS uses *domain decomposition* to distribute data (i.e. atoms) across the MPI processes in parallel runs. This includes topology data about bonds, angles, dihedrals, impropers and *"special" neighbors*. This information is stored with either one or all atoms involved in such a topology entry (which of the two option applies depends on the *newton* setting for bonds). When reading a data file, LAMMPS analyzes the requirements for this file and then the values are "locked in" and cannot be extended.

So loading a molecule file that requires more of the topology per atom storage or adding a data file with such needs will lead to an error. To avoid the error, one or more of the *extra/XXX/per/atom* keywords are required to extend the corresponding storage. It is no problem to choose those numbers generously and have more storage reserved than actually needed, but having these numbers set too small will lead to an error.

### 5.2.26 Molecule topology type exceeds system topology type

The total number of atom, bond, angle, dihedral, and improper types is "locked in" when LAMMPS creates the simulation box. This can happen through either the *create_box*, the *read_data*, or the *read_restart* command. After this it is not possible to refer to an additional type. So loading a molecule file that uses additional types or adding a data file that would require additional types will lead to an error. To avoid the error, one or more of the *extra/XXX/types* keywords are required to extend the maximum number of the individual types.

### 5.2.27 Molecule attributes do not match system attributes

Choosing an *atom_style* in LAMMPS determines which per-atom properties are available. In a *molecule file*, however, it is possible to add sections (for example Masses or Charges) that are not supported by the atom style. Masses for example, are usually not a per-atom property, but defined through the atom type. Thus it would not be required to have a Masses section and the included data would be ignored. LAMMPS prints this warning to inform about this case.

### 5.2.28 Inconsistent image flags

This warning happens when the distance between the *unwrapped* x-, y-, or z-components of the coordinates of a bond is larger than half the box with periodic boundaries or larger than the box with non-periodic boundaries. It means that the positions and image flags have become inconsistent. LAMMPS will still compute bonded interactions based on the closest periodic images of the atoms and thus in most cases the results will be correct. However they can cause problems when such atoms are used with the fix rigid or replicate commands. Thus, it is good practice to update the system so that the message does not appear. It will help with future manipulations of the system.

There is one case where this warning *must* appear: when you have a chain of connected bonds that pass through the entire box and connect back to the first atom in the chain through periodic boundaries, i.e. some kind of "infinite polymer". In that case, the bond image flags *must* be inconsistent for the one bond that reaches back to the beginning of the chain.

### 5.2.29 No fixes with time integration, atoms won't move

This warning will be issued if LAMMPS encounters a *run* command that does not have a preceding *fix* command that updates atom/object positions and velocities per step. In other words, there are no fixes detected that perform velocity-Verlet time integration, such as *fix nve*. Note that this alert does not mean that there are no active fixes. LAMMPS has a very wide variety of fixes, many of which do not move objects but also operate through steps, such as printing outputs (e.g. *fix print*), performing calculations (e.g. *fix ave/time*), or changing other system parameters (e.g. *fix dt/reset*). It is up to the user to determine whether the lack of a time-integrating fix is intentional or not.

### 5.2.30 System is not charge neutral, net charge = ...

The sum of charges in the system is not zero. When a system is not charge-neutral, methods that evolve/manipulate per-atom charges, evaluate Coulomb interactions, evaluate Coulomb forces, or evaluate/manipulate other properties relying on per-atom charges may raise this warning. A non-zero net charge most commonly arises after setting per-atom charges *set* such that the sum is non-zero or by reading in a system through *read_data* where the per-atom charges do not sum to zero. However, a loss of charge neutrality may occur in other less common ways, like when charge equilibration methods (e.g., *fix qeq*) fail.

A similar warning/error may be raised when using certain charge equilibration methods: *fix qeq*, *fix qeq/comb*, *fix qeq/reaxff*, and *fix qtpie/reaxff*. In such cases, this warning/error will be raised for the fix *group* when the group has a non-zero net charge.

When the system is expected to be charge-neutral, this warning often arises due to an error in the lammps input (e.g., an incorrect *set* command, error in the data file read by *read_data*, incorrectly grouping atoms with charge, etc.). If the system is NOT expected to be charge-neutral, the user should make sure that the method(s) used are appropriate for systems with a non-zero net charge. Some commonly used fixes for charge equilibration *fix qeq*, pair styles that include charge interactions *pair_style coul/XXX*, and kspace methods *kspace_style* can, in theory, support systems with non-zero net charge. However, non-zero net charge can lead to spurious artifacts. The severity of these artifacts depends on the magnitude of total charge, system size, and methods used. Before running simulations or calculations for systems with non-zero net charge, users should test for artifacts and convergence of properties.

### 5.2.31 Variable evaluation before simulation box is defined

This error happens, when trying to expand or use an equal- or atom-style variable (or an equivalent style), where the expression contains a reference to something (e.g. a compute reference, a property of an atom, or a thermo keyword) that is not allowed to be used before the simulation box is defined. See the paragraph on *errors before or after the simulation box is created* for additional information.

### 5.2.32 Invalid thermo keyword 'X' in variable formula

This error message is often misleading. It is caused when evaluating a *variable command* expression and LAMMPS comes across a string that it does not recognize. LAMMPS first checks if a string is a reference to a compute, fix, custom property, or another variable by looking at the first 2-3 characters (and if it is, it checks whether the referenced item exists). Next LAMMPS checks if the string matches one of the available functions or constants. If that fails, LAMMPS will assume that this string is a *thermo keyword* and let the code for printing thermodynamic output return the corresponding value. However, if this fails too, since the string is not a thermo keyword, LAMMPS stops with the 'Invalid thermo keyword' error. But it is also possible, that there is just a typo in the name of a valid variable function. Thus it is recommended to check the failing variable expression very carefully.

### 5.2.33 One or more atoms are time integrated more than once

This is probably an error since you typically do not want to advance the positions or velocities of an atom more than once per timestep. This typically happens when there are multiple fix commands that advance atom positions with overlapping groups. Also, for some fix styles it is not immediately obvious that they include time integration. Please check the documentation carefully.

### 5.2.34 XXX command before simulation box is defined

This error occurs when trying to execute a LAMMPS command that requires information about the system dimensions, or the number atom, bond, angle, dihedral, or improper types, or the number of atoms or similar data that is only available *after* the simulation box has been created. See the paragraph on *errors before or after the simulation box is created* for additional information.

### 5.2.35 XXX command after simulation box is defined

This error occurs when trying to execute a LAMMPS command that changes a global setting *after* it is locked in when the simulation box is created (for instance defining the *atom style*, *dimension*, *newton*, or *units* setting). These settings may only be changed *before* the simulation box has been created. See the paragraph on *errors before or after the simulation box is created* for additional information.

### 5.2.36 Error messages ending in 'Please contact the LAMMPS developers'

Such error messages indicate that something unexpected has happened and that it will require a good understanding of the details of the design of LAMMPS to resolve this. This can be due to some bug in contributed code, an oversight when updating functionality, a feature that is scheduled to be removed or reaching a combination of flags and settings that should not be possible or similar.

Even if you find a way to work around this error or warning, you should contact the LAMMPS developers and prepare a minimal set of inputs that can be used to reproduce this error or warning. By providing the input, the LAMMPS developers can then assess whether additional action is needed and who else to contact about this, if needed.

There are multiple ways to get into contact and report your issue. In order of preference there are:

- Submit a bug report issue in the LAMMPS GitHub repository
- Post a message in the "LAMMPS Development" forum in the MatSci Community Discourse
- Send an email to `developers@lammps.org`
- Send an email to an *individual LAMMPS developer* that you know and trust

### 5.2.37 Neighbor list overflow, boost neigh_modify one

The neighbor list code in LAMMPS uses a special memory allocation strategy to speed up building and accessing neighbor lists.

Instead of making a memory allocation for each list of neighbors of the atoms LAMMPS allocates "pages" that have room for several neighbor lists. This has two main advantages:

1. It is not needed to first count how many neighbors there are for an atom to determine the storage required. Since the pages are much larger than individual lists, LAMMPS just "fills up" the page until there is not enough space left and then allocates a new page.
2. There are fewer calls to the memory allocator functions (which can be time consuming for long-running jobs and fragmented memory space) and the resulting neighbor lists are close to each other physically which improves cache efficiency.

This is controlled by the two parameters "one" and "page", respectively, that can be set via the *neigh_modify command*. The parameter "one" is the maximum number of entries in a list of neighbors for a single atom. If an atom has more

neighbors as the "one" parameter allows, the "overflow" error message is triggered. The parameter "page" sets the size of the page. The neighbor list code checks, if there are "one" entries left in the current page. If not, a new page is allocated.

The default settings are suitable for most systems. They need to be changed, for instance, when simulating a system with a very high density or when setting a very long cutoff (e.g. $\gtrapprox$ 15Å with *units real*). The value of "page" **must** be at least 10x the value of "one", but 50x to 100x are recommended to avoid wasting memory. The neighbor list storage is typically the largest amount of RAM required by a LAMMPS calculation.

Even though the LAMMPS error message recommends to increase the "one" parameter, this may not always be the correct solution. The neighbor list overflow can also be a symptom for some other error that cannot be easily detected. For example, a frequent reason for an (unexpected) high density are incorrect box dimensions (since LAMMPS wraps atoms back into the principal box with periodic boundaries) or coordinates provided as fractional coordinates (LAMMPS does not support this for data files). In both cases, LAMMPS cannot easily know whether the input geometry has such a high density (and thus requiring more neighbor list storage per atom) on purpose or by accident. Rather than blindly increasing the "one" parameter, it is thus worth checking if this is justified by the combination of density and cutoff. This is particularly recommended when using some tool(s) to convert input or data files.

When boosting (= increasing) the "one" parameter, it is recommended to also increase the value for the "page" parameter to maintain the ratio between "one" and "page" to reduce waste of memory. For some more details, please check out the documentation for the *neigh_modify command*.

### 5.2.38 Variable . . . : Compute/Fix . . . does not compute requested property

Compute and fix styles can compute different kinds of properties: for example, global scalars, vectors, or arrays, or per-atom vectors or arrays. In equal-style or similar variable, only scalar properties can be used, so to access a particular element in a vector one has to use square brackets with a suitable index to select it. However, not all fixes and computes provide all types of properties. So this error message will be shown if there is a mismatch, of if there are not enough or too many square brackets. To differentiate between accessing an element of a global array or a per-atom array element of a specific atom, one has to use a reference with a lower case 'c' (e.g. 'c_name') for the former and upper case 'C' (e.g. 'C_name') for the latter. The same applies to fix styles. The full details are in the documentation for the *variable command*.

### 5.2.39 The . . . style . . . is no longer available

While the LAMMPS developers try to keep the software backward compatible as far as input files and file formats are concerned, this is not always desired and changes are made and commands renamed or removed. In that case an error message is printed describing why the command cannot be executed. More details can be found on the manual page *Removed commands and packages*.

## 5.3 Reporting bugs

If you are confident that you have found a bug in LAMMPS, please follow the steps outlined below:

- Check the New features and bug fixes section of the LAMMPS WWW site or the GitHub Releases page to see if the bug has already been addressed in a patch release.
- Check that your issue can be reproduced with the latest development version of LAMMPS.
- Check the manual carefully to verify that the unexpected behavior you are observing is indeed in conflict with the documentation
- Check the GitHub Issue page if your issue has already been reported and if it is still open.
- Check the GitHub Pull Requests page to see if there is already a fix for your bug pending.
- Check the LAMMPS forum at MatSci to see if the issue has been discussed before.

If none of these steps yields any useful information, please file a new bug report on the GitHub Issue page. The website will offer you to select a suitable template with explanations and then you should replace those explanations with the information that you can provide to reproduce your issue.

The most useful thing you can do to help us verify and fix a bug is to isolate the problem. Run it on the smallest number of atoms and fewest number of processors with the simplest input script that reproduces the bug. Try to identify what command or combination of commands is causing the problem and upload the complete input deck as a tar or zip archive. Please avoid using binary restart files unless the issue requires it. In the latter case you should also include an input deck to quickly generate this restart from a data file or a simple additional input. This input deck can be used with tools like a debugger or valgrind to further *debug the crash*.

You may also post a message in the development category of the LAMMPS forum at MatSci describing the problem with the same kind of information. The forum can provide a faster response, especially if the bug reported is actually expected behavior or other LAMMPS users have come across it before.

## 5.4 Debugging crashes

If LAMMPS crashes with a "segmentation fault" or a "bus error" or similar message, then you can use the following two methods to further narrow down the origin of the issue. This will help the LAMMPS developers (or yourself) to understand the reason for the crash and apply a fix (either to the input script or the source code). This requires that your LAMMPS executable includes the required *debug information*. Otherwise it is not possible to look up the names of functions or variables.

The following patch will introduce a bug into the code for pair style *lj/cut* when using the `examples/melt/in.melt` input. We use it to show how to identify the origin of a segmentation fault.

```
--- a/src/pair_lj_cut.cpp
+++ b/src/pair_lj_cut.cpp
@@ -81,6 +81,7 @@ void PairLJCut::compute(int eflag, int vflag)
   int nlocal = atom->nlocal;
   double *special_lj = force->special_lj;
   int newton_pair = force->newton_pair;
+  double comx = 0.0;

   inum = list->inum;
   ilist = list->ilist;
@@ -134,8 +135,10 @@ void PairLJCut::compute(int eflag, int vflag)
                              evdwl,0.0,fpair,delx,dely,delz);
       }
     }
-  }

+    comx += atom->rmass[i]*x[i][0]; /* BUG */
+  }
+  printf("comx = %g\n",comx);
   if (vflag_fdotr) virial_fdotr_compute();
 }
```

After recompiling LAMMPS and running the input you should get something like this:

```
$ ./lmp -in in.melt
LAMMPS (19 Mar 2020)
  using 1 OpenMP thread(s) per MPI task
Lattice spacing in x,y,z = 1.6796 1.6796 1.6796
```

(continues on next page)

(continued from previous page)

```
Created orthogonal box = (0 0 0) to (16.796 16.796 16.796)
  1 by 1 by 1 MPI processor grid
Created 4000 atoms
  create_atoms CPU = 0.000432253 secs
Neighbor list info ...
  update every 20 steps, delay 0 steps, check no
  max neighbors/atom: 2000, page size: 100000
  master list distance cutoff = 2.8
  ghost atom cutoff = 2.8
  binsize = 1.4, bins = 12 12 12
  1 neighbor lists, perpetual/occasional/extra = 1 0 0
  (1) pair lj/cut, perpetual
      attributes: half, newton on
      pair build: half/bin/atomonly/newton
      stencil: half/bin/3d/newton
      bin: standard
Setting up Verlet run ...
  Unit style    : lj
  Current step  : 0
  Time step     : 0.005
Segmentation fault (core dumped)
```

### 5.4.1 Using the GDB debugger to get a stack trace

There are two options to use the GDB debugger for identifying the origin of the segmentation fault or similar crash. The GDB debugger has many more features and options, as can be seen for example its online documentation.

#### Run LAMMPS from within the debugger

Running LAMMPS under the control of the debugger as shown below only works for a single MPI rank (for debugging a program running in parallel you usually need a parallel debugger program). A simple way to launch GDB is to prefix the LAMMPS command-line with `gdb --args` and then type the command "run" at the GDB prompt. This will launch the debugger, load the LAMMPS executable and its debug info, and then run it. When it reaches the code causing the segmentation fault, it will stop with a message why it stopped, print the current line of code, and drop back to the GDB prompt.

```
(gdb) run
[...]
Setting up Verlet run ...
  Unit style    : lj
  Current step  : 0
  Time step     : 0.005

Program received signal SIGSEGV, Segmentation fault.
0x00000000006653ab in LAMMPS_NS::PairLJCut::compute (this=0x829740, eflag=1, vflag=
↪<optimized out>) at /home/akohlmey/compile/lammps/src/pair_lj_cut.cpp:139
139      comx += atom->rmass[i]*x[i][0]; /* BUG */
(gdb)
```

Now typing the command "where" will show the stack of functions starting from the current function back to "main()".

```
(gdb) where
#0  0x00000000006653ab in LAMMPS_NS::PairLJCut::compute (this=0x829740, eflag=1, vflag=
↪<optimized out>) at /home/akohlmey/compile/lammps/src/pair_lj_cut.cpp:139
#1  0x00000000004cf0a2 in LAMMPS_NS::Verlet::setup (this=0x7e6c90, flag=1) at /home/
↪akohlmey/compile/lammps/src/verlet.cpp:131
#2  0x000000000049db42 in LAMMPS_NS::Run::command (this=this@entry=0x7fffffffcca0,␣
↪narg=narg@entry=1, arg=arg@entry=0x7e8750)
    at /home/akohlmey/compile/lammps/src/run.cpp:177
#3  0x000000000041258a in LAMMPS_NS::Input::command_creator<LAMMPS_NS::Run> (lmp=
↪<optimized out>, narg=1, arg=0x7e8750)
    at /home/akohlmey/compile/lammps/src/input.cpp:878
#4  0x0000000000410ad3 in LAMMPS_NS::Input::execute_command (this=0x7d1410) at /home/
↪akohlmey/compile/lammps/src/input.cpp:864
#5  0x00000000004111fb in LAMMPS_NS::Input::file (this=0x7d1410) at /home/akohlmey/
↪compile/lammps/src/input.cpp:229
#6  0x000000000040933a in main (argc=<optimized out>, argv=<optimized out>) at /home/
↪akohlmey/compile/lammps/src/main.cpp:65
(gdb)
```

You can also print the value of variables and see if there is anything unexpected. Segmentation faults, for example, commonly happen when a pointer variable is not assigned and still initialized to NULL.

```
(gdb) print x
$1 = (double **) 0x7ffff7ca1010
(gdb) print i
$2 = 0
(gdb) print x[0]
$3 = (double *) 0x7ffff6d80010
(gdb) print x[0][0]
$4 = 0
(gdb) print x[1][0]
$5 = 0.83979809569125363
(gdb) print atom->rmass
$6 = (double *) 0x0
(gdb)
```

### Inspect a core dump file with the debugger

When an executable crashes with a "core dumped" message, it creates a file "core" or "core.<PID#>" which contains the information about the current state. This file may be located in the folder where you ran LAMMPS or in some hidden folder managed by the systemd daemon. In the latter case, you need to "extract" the core file with the `coredumpctl` utility to the current folder. Example: `coredumpctl -o core dump lmp`. Now you can launch the debugger to load the executable, its debug info and the core dump and drop you to a prompt like before.

```
$ gdb lmp core
Reading symbols from lmp...
[New LWP 1928535]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Core was generated by `./lmp -in in.melt'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x00000000006653ab in LAMMPS_NS::PairLJCut::compute (this=0x1b10740, eflag=1, vflag=
↪<optimized out>)
```

(continues on next page)

(continued from previous page)

```
    at /home/akohlmey/compile/lammps/src/pair_lj_cut.cpp:139
139       comx += atom->rmass[i]*x[i][0]; /* BUG */
(gdb)
```

From here on, you use the same commands as shown before to get a stack trace and print current values of (pointer) variables.

### 5.4.2 Using valgrind to get a stack trace

The valgrind suite of tools allows to closely inspect the behavior of a compiled program by essentially emulating a CPU and instrumenting the program while running. This slows down execution quite significantly, but can also report issues that are not resulting in a crash. The default valgrind tool is a memory checker and you can use it by prefixing the normal command-line with `valgrind`. Unlike GDB, this will also work for parallel execution, but it is recommended to redirect the valgrind output to a file (e.g. with `--log-file=crash-%p.txt`, the %p will be substituted with the process ID) so that the messages of the multiple valgrind instances to the console are not mixed.

```
$ valgrind ./lmp -in in.melt
==1933642== Memcheck, a memory error detector
==1933642== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1933642== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==1933642== Command: ./lmp -in in.melt
==1933642==
LAMMPS (19 Mar 2020)
OMP_NUM_THREADS environment is not set. Defaulting to 1 thread. (src/comm.cpp:94)
  using 1 OpenMP thread(s) per MPI task
Lattice spacing in x,y,z = 1.6796 1.6796 1.6796
Created orthogonal box = (0 0 0) to (16.796 16.796 16.796)
  1 by 1 by 1 MPI processor grid
Created 4000 atoms
  create_atoms CPU = 0.032964 secs
Neighbor list info ...
  update every 20 steps, delay 0 steps, check no
  max neighbors/atom: 2000, page size: 100000
  master list distance cutoff = 2.8
  ghost atom cutoff = 2.8
  binsize = 1.4, bins = 12 12 12
  1 neighbor lists, perpetual/occasional/extra = 1 0 0
  (1) pair lj/cut, perpetual
      attributes: half, newton on
      pair build: half/bin/atomonly/newton
      stencil: half/bin/3d/newton
      bin: standard
Setting up Verlet run ...
  Unit style    : lj
  Current step  : 0
  Time step     : 0.005
==1933642== Invalid read of size 8
==1933642==    at 0x6653AB: LAMMPS_NS::PairLJCut::compute(int, int) (pair_lj_cut.cpp:139)
==1933642==    by 0x4CF0A1: LAMMPS_NS::Verlet::setup(int) (verlet.cpp:131)
==1933642==    by 0x49DB41: LAMMPS_NS::Run::command(int, char**) (run.cpp:177)
==1933642==    by 0x412589: void LAMMPS_NS::Input::command_creator<LAMMPS_NS::Run>
↪(LAMMPS_NS::LAMMPS*, int, char**) (input.cpp:881)
```

(continues on next page)

(continued from previous page)

```
==1933642==    by 0x410AD2: LAMMPS_NS::Input::execute_command() (input.cpp:864)
==1933642==    by 0x4111FA: LAMMPS_NS::Input::file() (input.cpp:229)
==1933642==    by 0x409339: main (main.cpp:65)
==1933642==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==1933642==
```

As you can see, the stack trace information is similar to that obtained from GDB. In addition you get a more specific hint about what cause the segmentation fault, i.e. that it is a NULL pointer dereference. To find out which pointer exactly was NULL, you need to use the debugger, though.

## 5.5 Debugging when LAMMPS appears to be stuck

Sometimes the LAMMPS calculation appears to be stuck, that is the LAMMPS process or processes are active, but there is no visible progress. This can have multiple reasons:

- The selected styles are slow and require a lot of CPU time and the system is large. When extrapolating the expected speed from smaller systems, one has to factor in that not all models scale linearly with system size, e.g. *kspace styles like ewald or pppm*. There is very little that can be done in this case.
- The output interval is not set or set to a large value with the *thermo* command. In the first case, there will be output only at the first and last step.
- The output is block-buffered and instead of line-buffered. The output will only be written to the screen after 4096 or 8192 characters of output have accumulated. This most often happens for files but also with MPI parallel executables for output to the screen, since the output to the screen is handled by the MPI library so that output from all processes can be shown. This can be suppressed by using the `-nonblock` or `-nb` command-line flag, which turns off buffering for screen and logfile output.
- An MPI parallel calculation has a bug where a collective MPI function is called (e.g. `MPI_Barrier()`, `MPI_Bcast()`, `MPI_Allreduce()` and so on) before pending point-to-point communications are completed or when the collective function is only called from a subset of the MPI processes. This also applies to some internal LAMMPS functions like `Error::all()` which uses `MPI_Barrier()` and thus `Error::one()` must be called, if the error condition does not happen on all MPI processes simultaneously.
- Some function in LAMMPS has a bug where a `for` or `while` loop does not trigger the exit condition and thus will loop forever. This can happen when the wrong variable is incremented or when one value in a comparison becomes `NaN` due to an overflow.

In the latter two cases, further information and stack traces (see above) can be obtain by attaching a debugger to a running process. For that the process ID (PID) is needed; this can be found on Linux machines with the `top`, `htop`, `ps`, or `pstree` commands.

Then running the (GNU) debugger `gdb` with the `-p` flag followed by the process id will attach the process to the debugger and stop execution of that specific process. From there on it is possible to issue all debugger commands in the same way as when LAMMPS was started from the debugger (see above). Most importantly it is possible to obtain a stack trace with the `where` command and thus determine where in the execution of a timestep this process is. Also internal data can be printed and execution single stepped or continued. When the debugger is exited, the calculation will resume normally.

## 5.6 Error messages

This is an alphabetic list of some of the ERROR messages LAMMPS prints out and the reason why. If the explanation here is not sufficient, the documentation for the offending command may help. This is a historic list and no longer updated. Instead the LAMMPS developers are trying to provide more details right with the error message or link to a paragraph with *detailed explanations*.

Error messages also list the source file and line number where the error was generated. For example, a message like this:

```
ERROR: Illegal velocity command (velocity.cpp:78)
```

means that line #78 in the file src/velocity.cpp generated the error. Looking in the source code may help you figure out what went wrong.

Please also see the page with *Warning messages*.

---

***Accelerator sharing is not currently supported on system***
Multiple MPI processes cannot share the accelerator on your system. For NVIDIA GPUs, see the nvidia-smi command to change this setting.

***All angle coeffs are not set***
All angle coefficients must be set in the data file or by the angle_coeff command before running a simulation.

***All bond coeffs are not set***
All bond coefficients must be set in the data file or by the bond_coeff command before running a simulation.

***All dihedral coeffs are not set***
All dihedral coefficients must be set in the data file or by the dihedral_coeff command before running a simulation.

***All improper coeffs are not set***
All improper coefficients must be set in the data file or by the improper_coeff command before running a simulation.

***All masses are not set***
For atom styles that define masses for each atom type, all masses must be set in the data file or by the mass command before running a simulation. They must also be set before using the velocity command.

***All mol IDs should be set for fix gcmc group atoms***
The molecule flag is on, yet not all molecule ids in the fix group have been set to non-zero positive values by the user. This is an error since all atoms in the fix gcmc group are eligible for deletion, rotation, and translation and therefore must have valid molecule ids.

***All pair coeffs are not set***
All pair coefficients must be set in the data file or by the pair_coeff command before running a simulation.

***All read_dump x,y,z fields must be specified for scaled, triclinic coords***
For triclinic boxes and scaled coordinates you must specify all 3 of the x,y,z fields, else LAMMPS cannot reconstruct the unscaled coordinates.

***Angle atom missing in delete_bonds***
The delete_bonds command cannot find one or more atoms in a particular angle on a particular processor. The pairwise cutoff is too short or the atoms are too far apart to make a valid angle.

***Angle atom missing in set command***
The set command cannot find one or more atoms in a particular angle on a particular processor. The pairwise cutoff is too short or the atoms are too far apart to make a valid angle.

***Angle coeff for hybrid has invalid style***
Angle style hybrid uses another angle style as one of its coefficients. The angle style used in the angle_coeff command or read from a restart file is not recognized.

***Angle coeffs are not set***
No angle coefficients have been assigned in the data file or via the angle_coeff command.

***Angle extent > half of periodic box length***
This error was detected by the neigh_modify check yes setting. It is an error because the angle atoms are so far apart it is ambiguous how it should be defined.

***Angle potential must be defined for SHAKE***
When shaking angles, an angle_style potential must be used.

***Angle table parameters did not set N***
List of angle table parameters must include N setting.

***Angle_coeff command before angle_style is defined***
Coefficients cannot be set in the data file or via the angle_coeff command until an angle_style has been assigned.

***Angle_coeff command before simulation box is defined***
The angle_coeff command cannot be used before a read_data, read_restart, or create_box command.

***Angle_coeff command when no angles allowed***
The chosen atom style does not allow for angles to be defined.

***Angle_style command when no angles allowed***
The chosen atom style does not allow for angles to be defined.

***Angles assigned incorrectly***
Angles read in from the data file were not assigned correctly to atoms. This means there is something invalid about the topology definitions.

***Angles defined but no angle types***
The data file header lists angles but no angle types.

***Append boundary must be shrink/minimum***
The boundary style of the face where atoms are added must be of type m (shrink/minimum).

***Arccos of invalid value in variable formula***
Argument of arccos() must be between -1 and 1.

***Arcsin of invalid value in variable formula***
Argument of arcsin() must be between -1 and 1.

***Atom IDs must be used for molecular systems***
Atom IDs are used to identify and find partner atoms in bonds.

***Atom count changed in fix neb***
This is not allowed in a NEB calculation.

***Atom count is inconsistent, cannot write data file***
The sum of atoms across processors does not equal the global number of atoms. Probably some atoms have been lost.

***Atom count is inconsistent, cannot write restart file***
Sum of atoms across processors does not equal initial total count. This is probably because you have lost some atoms.

***Atom sort did not operate correctly***
This is an internal LAMMPS error. Please report it to the developers.

***Atom style template molecule must have atom types***
The defined molecule(s) does not specify atom types.

***Atom style was redefined after using fix property/atom***
This is not allowed.

***Atom vector in equal-style variable formula***
Atom vectors generate one value per atom which is not allowed in an equal-style variable.

***Atom-style variable in equal-style variable formula***
Atom-style variables generate one value per atom which is not allowed in an equal-style variable.

***Atom_modify id command after simulation box is defined***
The atom_modify id command cannot be used after a read_data, read_restart, or create_box command.

***Atom_modify map command after simulation box is defined***
The atom_modify map command cannot be used after a read_data, read_restart, or create_box command.

***Atom_style command after simulation box is defined***
The atom_style command cannot be used after a read_data, read_restart, or create_box command.

***Atomfile variable could not read values***
Check the file assigned to the variable.

***Attempt to pop empty stack in fix box/relax***
Internal LAMMPS error. Please report it to the developers.

***Attempt to push beyond stack limit in fix box/relax***
Internal LAMMPS error. Please report it to the developers.

***Attempting to rescale a 0.0 temperature***
Cannot rescale a temperature that is already 0.0.

***Bad FENE bond***
Two atoms in a FENE bond have become so far apart that the bond cannot be computed.

***Bad TIP4P angle type for PPPM/TIP4P***
Specified angle type is not valid.

***Bad TIP4P angle type for PPPMDisp/TIP4P***
Specified angle type is not valid.

***Bad TIP4P bond type for PPPM/TIP4P***
Specified bond type is not valid.

***Bad TIP4P bond type for PPPMDisp/TIP4P***
Specified bond type is not valid.

***Bad fix ID in fix append/atoms command***
The value of the fix_id for keyword spatial must start with "f_".

***Bad grid of processors***
The 3d grid of processors defined by the processors command does not match the number of processors LAMMPS is being run on.

***Bad kspace_modify kmax/ewald parameter***
Kspace_modify values for the kmax/ewald keyword must be integers > 0

***Bad kspace_modify slab parameter***
Kspace_modify value for the slab/volume keyword must be >= 2.0.

***Bad matrix inversion in mldivide3***
This error should not occur unless the matrix is badly formed.

***Bad principal moments***
Fix rigid did not compute the principal moments of inertia of a rigid group of atoms correctly.

***Bad quadratic solve for particle/line collision***
This is an internal error. It should normally not occur.

***Bad quadratic solve for particle/tri collision***
This is an internal error. It should normally not occur.

***Bad real space Coulombic cutoff in fix tune/kspace***
Fix tune/kspace tried to find the optimal real space Coulombic cutoff using the Newton-Rhaphson method, but found a non-positive or NaN cutoff

***Balance command before simulation box is defined***
The balance command cannot be used before a read_data, read_restart, or create_box command.

***Balance produced bad splits***
This should not occur. It means two or more cutting plane locations are on top of each other or out of order. Report the problem to the developers.

***Balance rcb cannot be used with comm_style brick***
Comm_style tiled must be used instead.

***Balance shift string is invalid***
The string can only contain the characters "x", "y", or "z".

***Bias compute does not calculate a velocity bias***
The specified compute must compute a bias for temperature.

***Bias compute does not calculate temperature***
The specified compute must compute temperature.

***Bias compute group does not match compute group***
The specified compute must operate on the same group as the parent compute.

***Big particle in fix srd cannot be point particle***
Big particles must be extended spheroids or ellipsoids.

***Bigint setting in lmptype.h is invalid***
Size of bigint is less than size of tagint.

***Bigint setting in lmptype.h is not compatible***
Format of bigint stored in restart file is not consistent with LAMMPS version you are running. See the settings in src/lmptype.h

***Bitmapped lookup tables require int/float be same size***
Cannot use pair tables on this machine, because of word sizes. Use the pair_modify command with table 0 instead.

***Bitmapped table in file does not match requested table***
Setting for bitmapped table in pair_coeff command must match table in file exactly.

***Bitmapped table is incorrect length in table file***
Number of table entries is not a correct power of 2.

***Bond and angle potentials must be defined for TIP4P***
Cannot use TIP4P pair potential unless bond and angle potentials are defined.

***Bond atom missing in box size check***
The second atom needed to compute a particular bond is missing on this processor. Typically this is because the pairwise cutoff is set too short or the bond has blown apart and an atom is too far away.

***Bond atom missing in delete_bonds***
The delete_bonds command cannot find one or more atoms in a particular bond on a particular processor. The pairwise cutoff is too short or the atoms are too far apart to make a valid bond.

***Bond atom missing in set command***
The set command cannot find one or more atoms in a particular bond on a particular processor. The pairwise cutoff is too short or the atoms are too far apart to make a valid bond.

***Bond coeff for hybrid has invalid style***
Bond style hybrid uses another bond style as one of its coefficients. The bond style used in the bond_coeff command or read from a restart file is not recognized.

***Bond coeffs are not set***
No bond coefficients have been assigned in the data file or via the bond_coeff command.

***Bond extent > half of periodic box length***
This error was detected by the neigh_modify check yes setting. It is an error because the bond atoms are so far apart it is ambiguous how it should be defined.

***Bond potential must be defined for SHAKE***
Cannot use fix shake unless bond potential is defined.

***Bond style quartic cannot be used with 3,4-body interactions***
No angle, dihedral, or improper styles can be defined when using bond style quartic.

***Bond style quartic cannot be used with atom style template***
This bond style can change the bond topology which is not allowed with this atom style.

***Bond style quartic requires special_bonds = 1,1,1***
This is a restriction of the current bond quartic implementation.

***Bond table parameters did not set N***
List of bond table parameters must include N setting.

***Bond table values are not increasing***
The values in the tabulated file must be monotonically increasing.

***BondAngle coeff for hybrid angle has invalid format***
No "ba" field should appear in data file entry.

***BondBond coeff for hybrid angle has invalid format***
No "bb" field should appear in data file entry.

***Bond_coeff command before bond_style is defined***
Coefficients cannot be set in the data file or via the bond_coeff command until an bond_style has been assigned.

***Bond_coeff command before simulation box is defined***
The bond_coeff command cannot be used before a read_data, read_restart, or create_box command.

***Bond_coeff command when no bonds allowed***
The chosen atom style does not allow for bonds to be defined.

***Bond_style command when no bonds allowed***
The chosen atom style does not allow for bonds to be defined.

***Bonds assigned incorrectly***
Bonds read in from the data file were not assigned correctly to atoms. This means there is something invalid about the topology definitions.

***Bonds defined but no bond types***
The data file header lists bonds but no bond types.

***Both sides of boundary must be periodic***
Cannot specify a boundary as periodic only on the lo or hi side. Must be periodic on both sides.

***Boundary command after simulation box is defined***
The boundary command cannot be used after a read_data, read_restart, or create_box command.

***Box bounds are invalid***
The box boundaries specified in the read_data file are invalid. The lo value must be less than the hi value for all 3 dimensions.

***Box command after simulation box is defined***
The box command cannot be used after a read_data, read_restart, or create_box command.

***CPU neighbor lists must be used for ellipsoid/sphere mix.***
When using Gay-Berne or RE-squared pair styles with both ellipsoidal and spherical particles, the neighbor list must be built on the CPU

***Can not specify Pxy/Pxz/Pyz in fix box/relax with non-triclinic box***
Only triclinic boxes can be used with off-diagonal pressure components. See the region prism command for details.

***Can not specify Pxy/Pxz/Pyz in fix nvt/npt/nph with non-triclinic box***
Only triclinic boxes can be used with off-diagonal pressure components. See the region prism command for details.

***Cannot (yet) do analytic differentiation with pppm/gpu***
This is a current restriction of this command.

***Cannot (yet) request ghost atoms with Kokkos half neighbor list***
This feature is not yet supported.

***Cannot (yet) use 'electron' units with dipoles***
This feature is not yet supported.

***Cannot (yet) use Ewald with triclinic box and slab correction***
This feature is not yet supported.

***Cannot (yet) use K-space slab correction with compute group/group for triclinic systems***
This option is not yet supported.

***Cannot (yet) use MSM with 2d simulation***
This feature is not yet supported.

***Cannot (yet) use PPPM with triclinic box and TIP4P***
This feature is not yet supported.

***Cannot (yet) use PPPM with triclinic box and kspace_modify diff ad***
This feature is not yet supported.

***Cannot (yet) use PPPM with triclinic box and slab correction***
This feature is not yet supported.

***Cannot (yet) use kspace slab correction with long-range dipoles and non-neutral systems or per-atom energy***
This feature is not yet supported.

***Cannot (yet) use kspace_modify diff ad with compute group/group***
This option is not yet supported.

***Cannot (yet) use kspace_style pppm/stagger with triclinic systems***
This feature is not yet supported.

***Cannot (yet) use single precision with MSM (remove -DFFT_SINGLE from Makefile and re-compile)***
Single precision cannot be used with MSM.

***Cannot add atoms to fix move variable***
Atoms can not be added afterwards to this fix option.

***Cannot append atoms to a triclinic box***
The simulation box must be defined with edges aligned with the Cartesian axes.

***Cannot change box ortho/triclinic with certain fixes defined***
This is because those fixes store the shape of the box. You need to use unfix to discard the fix, change the box, then redefine a new fix.

***Cannot change box ortho/triclinic with dumps defined***
This is because some dumps store the shape of the box. You need to use undump to discard the dump, change the box, then redefine a new dump.

***Cannot change box tilt factors for orthogonal box***
Cannot use tilt factors unless the simulation box is non-orthogonal.

***Cannot change dump_modify every for dump dcd***
The frequency of writing dump dcd snapshots cannot be changed.

***Cannot change dump_modify every for dump xtc***
The frequency of writing dump xtc snapshots cannot be changed.

***Cannot change timestep once fix srd is setup***
This is because various SRD properties depend on the timestep size.

***Cannot change timestep with fix pour***
This is because fix pour pre-computes the time delay for particles to fall out of the insertion volume due to gravity.

***Cannot change_box after reading restart file with per-atom info***
This is because the restart file info cannot be migrated with the atoms. You can get around this by performing a 0-timestep run which will assign the restart file info to actual atoms.

***Cannot clear group all***
This operation is not allowed.

***Cannot compute initial g_ewald_disp***
LAMMPS failed to compute an initial guess for the PPPM_disp g_ewald_6 factor that partitions the computation between real space and k-space for Dispersion interactions.

***Cannot create an atom map unless atoms have IDs***
The simulation requires a mapping from global atom IDs to local atoms, but the atoms that have been defined have no IDs.

***Cannot create atoms with undefined lattice***
Must use the lattice command before using the create_atoms command.

***Cannot create/grow a vector/array of pointers for %s***
LAMMPS code is making an illegal call to the templated memory allocators, to create a vector or array of pointers.

***Cannot create_atoms after reading restart file with per-atom info***
The per-atom info was stored to be used when by a fix that you may re-define. If you add atoms before re-defining the fix, then there will not be a correct amount of per-atom info.

***Cannot create_box after simulation box is defined***
A simulation box can only be defined once.

***Cannot currently use pair reax with pair hybrid***
This is not yet supported.

***Cannot displace_atoms after reading restart file with per-atom info***
This is because the restart file info cannot be migrated with the atoms. You can get around this by performing a 0-timestep run which will assign the restart file info to actual atoms.

***Cannot do GCMC on atoms in atom_modify first group***
This is a restriction due to the way atoms are organized in a list to enable the atom_modify first command.

***Cannot do atom/swap on atoms in atom_modify first group***
This is a restriction due to the way atoms are organized in a list to enable the atom_modify first command.

***Cannot dump sort when multiple dump files are written***
In this mode, each processor dumps its atoms to a file, so no sorting is allowed.

***Cannot embed Python when also extending Python with LAMMPS***
When running LAMMPS via Python through the LAMMPS library interface you cannot also user the input script python command.

***Cannot evaporate atoms in atom_modify first group***
This is a restriction due to the way atoms are organized in a list to enable the atom_modify first command.

***Cannot find delete_bonds group ID***
Group ID used in the delete_bonds command does not exist.

***Cannot have both pair_modify shift and tail set to yes***
These 2 options are contradictory.

***Cannot intersect groups using a dynamic group***
This operation is not allowed.

***Cannot open ADP potential file %s***
The specified ADP potential file cannot be opened. Check that the path and name are correct.

***Cannot open AIREBO potential file %s***
The specified AIREBO potential file cannot be opened. Check that the path and name are correct.

***Cannot open BOP potential file %s***
The specified BOP potential file cannot be opened. Check that the path and name are correct.

***Cannot open COMB potential file %s***
The specified COMB potential file cannot be opened. Check that the path and name are correct.

***Cannot open COMB3 lib.comb3 file***
The COMB3 library file cannot be opened. Check that the path and name are correct.

***Cannot open COMB3 potential file %s***
The specified COMB3 potential file cannot be opened. Check that the path and name are correct.

***Cannot open EAM potential file %s***
The specified EAM potential file cannot be opened. Check that the path and name are correct.

***Cannot open EIM potential file %s***
The specified EIM potential file cannot be opened. Check that the path and name are correct.

***Cannot open LCBOP potential file %s***
The specified LCBOP potential file cannot be opened. Check that the path and name are correct.

***Cannot open MEAM potential file %s***
The specified MEAM potential file cannot be opened. Check that the path and name are correct.

***Cannot open SNAP coefficient file %s***
The specified SNAP coefficient file cannot be opened. Check that the path and name are correct.

***Cannot open SNAP parameter file %s***
The specified SNAP parameter file cannot be opened. Check that the path and name are correct.

***Cannot open Stillinger-Weber potential file %s***
The specified SW potential file cannot be opened. Check that the path and name are correct.

***Cannot open Tersoff potential file %s***
The specified potential file cannot be opened. Check that the path and name are correct.

***Cannot open Vashishta potential file %s***
The specified Vashishta potential file cannot be opened. Check that the path and name are correct.

***Cannot open coul/streitz potential file %s***
The specified coul/streitz potential file cannot be opened. Check that the path and name are correct.

***Cannot open data file %s***
The specified file cannot be opened. Check that the path and name are correct.

***Cannot open dir to search for restart file***
Using a "*" in the name of the restart file will open the current directory to search for matching file names.

***Cannot open dump file %s***
The output file for the dump command cannot be opened. Check that the path and name are correct.

***Cannot open file %s***
The specified file cannot be opened. Check that the path and name are correct. If the file is a compressed file, also check that the gzip executable can be found and run.

***Cannot open file variable file %s***
The specified file cannot be opened. Check that the path and name are correct.

***Cannot open fix ave/chunk file %s***
The specified file cannot be opened. Check that the path and name are correct.

***Cannot open fix ave/correlate file %s***
The specified file cannot be opened. Check that the path and name are correct.

***Cannot open fix ave/histo file %s***
The specified file cannot be opened. Check that the path and name are correct.

***Cannot open fix ave/time file %s***
The specified file cannot be opened. Check that the path and name are correct.

***Cannot open fix print file %s***
The output file generated by the fix print command cannot be opened

***Cannot open fix qeq parameter file %s***
The specified file cannot be opened. Check that the path and name are correct.

***Cannot open fix qeq/comb file %s***
The output file for the fix qeq/combs command cannot be opened. Check that the path and name are correct.

***Cannot open fix reax/bonds file %s***
The output file for the fix reax/bonds command cannot be opened. Check that the path and name are correct.

***Cannot open fix rigid infile %s***
The specified file cannot be opened. Check that the path and name are correct.

***Cannot open fix rigid restart file %s***
The specified file cannot be opened. Check that the path and name are correct.

***Cannot open fix rigid/small infile %s***
The specified file cannot be opened. Check that the path and name are correct.

***Cannot open fix tmd file %s***
The output file for the fix tmd command cannot be opened. Check that the path and name are correct.

***Cannot open fix ttm file %s***
The output file for the fix ttm command cannot be opened. Check that the path and name are correct.

***Cannot open gzipped file***
LAMMPS was compiled without support for reading and writing gzipped files through a pipeline to the gzip program with -DLAMMPS_GZIP.

***Cannot open log.cite file***
This file is created when you use some LAMMPS features, to indicate what paper you should cite on behalf of those who implemented the feature. Check that you have write privileges into the directory you are running in.

***Cannot open log.lammps for writing***
The default LAMMPS log file cannot be opened. Check that the directory you are running in allows for files to be created.

***Cannot open logfile***
The LAMMPS log file named in a command-line argument cannot be opened. Check that the path and name are correct.

***Cannot open logfile %s***
The LAMMPS log file specified in the input script cannot be opened. Check that the path and name are correct.

***Cannot open molecule file %s***
The specified file cannot be opened. Check that the path and name are correct.

***Cannot open nb3b/harmonic potential file %s***
The specified potential file cannot be opened. Check that the path and name are correct.

***Cannot open pair_write file***
The specified output file for pair energies and forces cannot be opened. Check that the path and name are correct.

***Cannot open polymorphic potential file %s***
The specified polymorphic potential file cannot be opened. Check that the path and name are correct.

***Cannot open screen file***
The screen file specified as a command-line argument cannot be opened. Check that the directory you are running in allows for files to be created.

***Cannot open universe log file***
For a multi-partition run, the master log file cannot be opened. Check that the directory you are running in allows for files to be created.

***Cannot open universe screen file***
For a multi-partition run, the master screen file cannot be opened. Check that the directory you are running in allows for files to be created.

***Cannot read_restart after simulation box is defined***
The read_restart command cannot be used after a read_data, read_restart, or create_box command.

***Cannot redefine variable as a different style***
An equal-style variable can be re-defined but only if it was originally an equal-style variable.

***Cannot replicate 2d simulation in z dimension***
The replicate command cannot replicate a 2d simulation in the z dimension.

***Cannot replicate with fixes that store atom quantities***
Either fixes are defined that create and store atom-based vectors or a restart file was read which included atom-based vectors for fixes. The replicate command cannot duplicate that information for new atoms. You should use the replicate command before fixes are applied to the system.

***Cannot reset timestep with a dynamic region defined***
Dynamic regions (see the region command) have a time dependence. Thus you cannot change the timestep when one or more of these are defined.

***Cannot reset timestep with a time-dependent fix defined***
You cannot reset the timestep when a fix that keeps track of elapsed time is in place.

***Cannot run 2d simulation with non-periodic Z dimension***
Use the boundary command to make the z dimension periodic in order to run a 2d simulation.

***Cannot set bond topology types for atom style template***
The bond, angle, etc types cannot be changed for this atom style since they are static settings in the molecule template files.

***Cannot set both respa pair and inner/middle/outer***
In the rRESPA integrator, you must compute pairwise potentials either all together (pair), or in pieces (inner/middle/outer). You can't do both.

***Cannot set mass for this atom style***
This atom style does not support mass settings for each atom type. Instead they are defined on a per-atom basis in the data file.

***Cannot set respa hybrid and any of pair/inner/middle/outer***
In the rRESPA integrator, you must compute pairwise potentials either all together (pair), with different cutoff regions (inner/middle/outer), or per hybrid sub-style (hybrid). You cannot mix those.

***Cannot set respa middle without inner/outer***
In the rRESPA integrator, you must define both a inner and outer setting in order to use a middle setting.

***Cannot set temperature for fix rigid/nph***
The temp keyword cannot be specified.

***Cannot set this attribute for this atom style***
The attribute being set does not exist for the defined atom style.

***Cannot subtract groups using a dynamic group***
This operation is not allowed.

***Cannot union groups using a dynamic group***
This operation is not allowed.

***Cannot use Ewald with 2d simulation***
The kspace style ewald cannot be used in 2d simulations. You can use 2d Ewald in a 3d simulation; see the kspace_modify command.

***Cannot use Ewald/disp solver on system with no charge, dipole, or LJ particles***
No atoms in system have a non-zero charge or dipole, or are LJ particles. Change charges/dipoles or change options of the kspace solver/pair style.

***Cannot use EwaldDisp with 2d simulation***
This is a current restriction of this command.

***Cannot use NEB unless atom map exists***
Use the atom_modify command to create an atom map.

***Cannot use NEB with atom_modify sort enabled***
This is current restriction for NEB implemented in LAMMPS.

***Cannot use PPPM with 2d simulation***
The kspace style pppm cannot be used in 2d simulations. You can use 2d PPPM in a 3d simulation; see the kspace_modify command.

***Cannot use PPPMDisp with 2d simulation***
The kspace style pppm/disp cannot be used in 2d simulations. You can use 2d pppm/disp in a 3d simulation; see the kspace_modify command.

***Cannot use PRD with a changing box***
The current box dimensions are not copied between replicas

***Cannot use PRD with a time-dependent fix defined***
PRD alters the timestep in ways that will mess up these fixes.

***Cannot use PRD with a time-dependent region defined***
PRD alters the timestep in ways that will mess up these regions.

***Cannot use PRD with atom_modify sort enabled***
This is a current restriction of PRD. You must turn off sorting, which is enabled by default, via the atom_modify command.

***Cannot use PRD with multi-processor replicas unless atom map exists***
Use the atom_modify command to create an atom map.

***Cannot use TAD unless atom map exists for NEB***
See atom_modify map command to set this.

***Cannot use TAD with a single replica for NEB***
NEB requires multiple replicas.

***Cannot use TAD with atom_modify sort enabled for NEB***
This is a current restriction of NEB.

***Cannot use a damped dynamics min style with fix box/relax***
This is a current restriction in LAMMPS. Use another minimizer style.

***Cannot use a damped dynamics min style with per-atom DOF***
This is a current restriction in LAMMPS. Use another minimizer style.

***Cannot use append/atoms in periodic dimension***
The boundary style of the face where atoms are added can not be of type p (periodic).

***Cannot use chosen neighbor list style with buck/kk***
That style is not supported by Kokkos.

***Cannot use chosen neighbor list style with coul/cut/kk***
That style is not supported by Kokkos.

***Cannot use chosen neighbor list style with coul/dsf/kk***
That style is not supported by Kokkos.

***Cannot use chosen neighbor list style with coul/wolf/kk***
That style is not supported by Kokkos.

***Cannot use chosen neighbor list style with lj/cut/coul/cut/kk***
That style is not supported by Kokkos.

***Cannot use chosen neighbor list style with lj/cut/coul/long/kk***
That style is not supported by Kokkos.

***Cannot use chosen neighbor list style with lj/cut/kk***
That style is not supported by Kokkos.

***Cannot use chosen neighbor list style with lj/spica/kk***
That style is not supported by Kokkos.

***Cannot use chosen neighbor list style with pair eam/kk***
That style is not supported by Kokkos.

***Cannot use compute cluster/atom unless atoms have IDs***
Atom IDs are used to identify clusters.

***Cannot use create_bonds unless atoms have IDs***
This command requires a mapping from global atom IDs to local atoms, but the atoms that have been defined have no IDs.

***Cannot use cwiggle in variable formula between runs***
This is a function of elapsed time.

***Cannot use delete_atoms bond yes with atom_style template***
This is because the bonds for that atom style are hardwired in the molecule template.

***Cannot use delete_atoms unless atoms have IDs***
Your atoms do not have IDs, so the delete_atoms command cannot be used.

***Cannot use delete_bonds with non-molecular system***
Your choice of atom style does not have bonds.

***Cannot use dynamic group with fix adapt atom***
This is not yet supported.

***Cannot use fix TMD unless atom map exists***
Using this fix requires the ability to lookup an atom index, which is provided by an atom map. An atom map does not exist (by default) for non-molecular problems. Using the atom_modify map command will force an atom map to be created.

***Cannot use fix bond/break with non-molecular systems***
Only systems with bonds that can be changed can be used. Atom_style template does not qualify.

***Cannot use fix bond/create with non-molecular systems***
Only systems with bonds that can be changed can be used. Atom_style template does not qualify.

***Cannot use fix bond/swap with non-molecular systems***
Only systems with bonds that can be changed can be used. Atom_style template does not qualify.

***Cannot use fix box/relax on a 2nd non-periodic dimension***
When specifying an off-diagonal pressure component, the second of the two dimensions must be periodic. E.g. if the xy component is specified, then the y dimension must be periodic.

***Cannot use fix box/relax on a non-periodic dimension***
When specifying a diagonal pressure component, the dimension must be periodic.

***Cannot use fix box/relax with both relaxation and scaling on a tilt factor***
When specifying scaling on a tilt factor component, that component can not also be controlled by the barostat. E.g. if scalexy yes is specified and also keyword tri or xy, this is wrong.

***Cannot use fix box/relax with tilt factor scaling on a 2nd non-periodic dimension***
When specifying scaling on a tilt factor component, the second of the two dimensions must be periodic. E.g. if the xy component is specified, then the y dimension must be periodic.

***Cannot use fix deform on a shrink-wrapped boundary***
The x, y, z options cannot be applied to shrink-wrapped dimensions.

***Cannot use fix deform tilt on a shrink-wrapped 2nd dim***
This is because the shrink-wrapping will change the value of the strain implied by the tilt factor.

***Cannot use fix deform trate on a box with zero tilt***
The trate style alters the current strain.

***Cannot use fix deposit rigid and shake***
These two attributes are conflicting.

***Cannot use fix gcmc in a 2d simulation***
Fix gcmc is set up to run in 3d only. No 2d simulations with fix gcmc are allowed.

***Cannot use fix npt and fix deform on same component of stress tensor***
This would be changing the same box dimension twice.

***Cannot use fix nvt/npt/nph on a 2nd non-periodic dimension***
When specifying an off-diagonal pressure component, the second of the two dimensions must be periodic. E.g. if the xy component is specified, then the y dimension must be periodic.

***Cannot use fix nvt/npt/nph on a non-periodic dimension***
When specifying a diagonal pressure component, the dimension must be periodic.

***Cannot use fix nvt/npt/nph with xy scaling when y is non-periodic dimension***
The second dimension in the barostatted tilt factor must be periodic.

***Cannot use fix nvt/npt/nph with xz scaling when z is non-periodic dimension***
The second dimension in the barostatted tilt factor must be periodic.

***Cannot use fix nvt/npt/nph with yz scaling when z is non-periodic dimension***
The second dimension in the barostatted tilt factor must be periodic.

***Cannot use fix pour rigid and shake***
These two attributes are conflicting.

***Cannot use fix pour with triclinic box***
This option is not yet supported.

***Cannot use fix press/berendsen and fix deform on same component of stress tensor***
These commands both change the box size/shape, so you cannot use both together.

***Cannot use fix rigid npt/nph and fix deform on same component of stress tensor***
This would be changing the same box dimension twice.

***Cannot use fix rigid npt/nph on a non-periodic dimension***
When specifying a diagonal pressure component, the dimension must be periodic.

***Cannot use fix rigid/small npt/nph on a non-periodic dimension***
When specifying a diagonal pressure component, the dimension must be periodic.

***Cannot use fix shake with non-molecular system***
Your choice of atom style does not have bonds.

***Cannot use fix ttm with 2d simulation***
This is a current restriction of this fix due to the grid it creates.

***Cannot use fix ttm with triclinic box***
This is a current restriction of this fix due to the grid it creates.

***Cannot use fix tune/kspace without a pair style***
This fix (tune/kspace) can only be used when a pair style has been specified.

***Cannot use fix wall/srd more than once***
Nor is their a need to since multiple walls can be specified in one command.

***Cannot use lines with fix srd unless overlap is set***
This is because line segments are connected to each other.

***Cannot use neigh_modify exclude with GPU neighbor builds***
This is a current limitation of the GPU implementation in LAMMPS.

***Cannot use non-zero forces in an energy minimization***
Fix setforce cannot be used in this manner. Use fix addforce instead.

***Cannot use non-periodic boundares with fix ttm***
This fix requires a fully periodic simulation box.

***Cannot use non-periodic boundaries with Ewald***
For kspace style ewald, all 3 dimensions must have periodic boundaries unless you use the kspace_modify command to define a 2d slab with a non-periodic z dimension.

***Cannot use non-periodic boundaries with EwaldDisp***
For kspace style ewald/disp, all 3 dimensions must have periodic boundaries unless you use the kspace_modify command to define a 2d slab with a non-periodic z dimension.

***Cannot use non-periodic boundaries with PPPM***
For kspace style pppm, all 3 dimensions must have periodic boundaries unless you use the kspace_modify command to define a 2d slab with a non-periodic z dimension.

***Cannot use non-periodic boundaries with PPPMDisp***
For kspace style pppm/disp, all 3 dimensions must have periodic boundaries unless you use the kspace_modify command to define a 2d slab with a non-periodic z dimension.

***Cannot use package gpu neigh yes with triclinic box***
This is a current restriction in LAMMPS.

***Cannot use pair tail corrections with 2d simulations***
The correction factors are only currently defined for 3d systems.

***Cannot use processors part command without using partitions***
See the command-line -partition switch.

***Cannot use ramp in variable formula between runs***
This is because the ramp() function is time dependent.

***Cannot use region INF or EDGE when box does not exist***
Regions that extend to the box boundaries can only be used after the create_box command has been used.

***Cannot use set atom with no atom IDs defined***
Atom IDs are not defined, so they cannot be used to identify an atom.

***Cannot use swiggle in variable formula between runs***
This is a function of elapsed time.

***Cannot use tris with fix srd unless overlap is set***
This is because triangles are connected to each other.

***Cannot use variable energy with constant force in fix addforce***
This is because for constant force, LAMMPS can compute the change in energy directly.

***Cannot use variable every setting for dump dcd***
The format of DCD dump files requires snapshots be output at a constant frequency.

***Cannot use variable every setting for dump xtc***
The format of this file requires snapshots at regular intervals.

***Cannot use vdisplace in variable formula between runs***
This is a function of elapsed time.

***Cannot use velocity create loop all unless atoms have IDs***
Atoms in the simulation to do not have IDs, so this style of velocity creation cannot be performed.

***Cannot wiggle and shear fix wall/gran***
Cannot specify both options at the same time.

***Cannot yet use KSpace solver with grid with comm style tiled***
This is current restriction in LAMMPS.

***Cannot yet use compute tally with Kokkos***
This feature is not yet supported.

***Cannot yet use fix bond/break with this improper style***
This is a current restriction in LAMMPS.

***Cannot yet use fix bond/create with this improper style***
This is a current restriction in LAMMPS.

***Cannot zero Langevin force of 0 atoms***
The group has zero atoms, so you cannot request its force be zeroed.

***Cannot zero gld force for zero atoms***
There are no atoms currently in the group.

***Change_box volume used incorrectly***
The "dim volume" option must be used immediately following one or two settings for "dim1 ..." (and optionally "dim2 ...") and must be for a different dimension, i.e. dim != dim1 and dim != dim2.

***Comm tiled invalid index in box drop brick***
Internal error check in comm_style tiled which should not occur. Contact the developers.

***Comm tiled mis-match in box drop brick***
Internal error check in comm_style tiled which should not occur. Contact the developers.

***Communication cutoff too small for SNAP micro load balancing***
This can happen if you change the neighbor skin after your pair_style command or if your box dimensions grow during a run. You can set the cutoff explicitly via the comm_modify cutoff command.

***Compute %s does not allow use of dynamic group***
Dynamic groups have not yet been enabled for this compute.

***Compute angle/local used when angles are not allowed***
The atom style does not support angles.

***Compute angmom/chunk does not use chunk/atom compute***
The style of the specified compute is not chunk/atom.

***Compute bond/local used when bonds are not allowed***
The atom style does not support bonds.

***Compute centro/atom requires a pair style be defined***
This is because the computation of the centro-symmetry values uses a pairwise neighbor list.

***Compute chunk/atom bin/cylinder radius is too large for periodic box***
Radius cannot be bigger than 1/2 of a non-axis periodic dimension.

***Compute chunk/atom bin/sphere radius is too large for periodic box***
Radius cannot be bigger than 1/2 of any periodic dimension.

***Compute chunk/atom ids once but nchunk is not once***
You cannot assign chunks IDs to atom permanently if the number of chunks may change.

***Compute chunk/atom stores no IDs for compute property/chunk***
It will only store IDs if its compress option is enabled.

***Compute chunk/atom stores no coord1 for compute property/chunk***
Only certain binning options for compute chunk/atom store coordinates.

***Compute chunk/atom stores no coord2 for compute property/chunk***
Only certain binning options for compute chunk/atom store coordinates.

***Compute chunk/atom stores no coord3 for compute property/chunk***
Only certain binning options for compute chunk/atom store coordinates.

***Compute chunk/atom without bins cannot use discard mixed***
That discard option only applies to the binning styles.

***Compute cluster/atom cutoff is longer than pairwise cutoff***
Cannot identify clusters beyond cutoff.

***Compute cluster/atom requires a pair style be defined***
This is so that the pair style defines a cutoff distance which is used to find clusters.

***Compute com/chunk does not use chunk/atom compute***
The style of the specified compute is not chunk/atom.

***Compute coord/atom cutoff is longer than pairwise cutoff***
Cannot compute coordination at distances longer than the pair cutoff, since those atoms are not in the neighbor list.

***Compute damage/atom requires peridynamic potential***
Damage is a Peridynamic-specific metric. It requires you to be running a Peridynamics simulation.

***Compute dihedral/local used when dihedrals are not allowed***
The atom style does not support dihedrals.

***Compute does not allow an extra compute or fix to be reset***
This is an internal LAMMPS error. Please report it to the developers.

***Compute erotate/asphere requires extended particles***
This compute cannot be used with point particles.

***Compute event/displace has invalid fix event assigned***
This is an internal LAMMPS error. Please report it to the developers.

***Compute gyration/chunk does not use chunk/atom compute***
The style of the specified compute is not chunk/atom.

***Compute hexorder/atom cutoff is longer than pairwise cutoff***
Cannot compute order parameter beyond cutoff.

***Compute improper/local used when impropers are not allowed***
The atom style does not support impropers.

***Compute inertia/chunk does not use chunk/atom compute***
The style of the specified compute is not chunk/atom.

***Compute msd/chunk does not use chunk/atom compute***
The style of the specified compute is not chunk/atom.

***Compute msd/chunk nchunk is not static***
This is required because the MSD cannot be computed consistently if the number of chunks is changing. Compute chunk/atom allows setting nchunk to be static.

***Compute omega/chunk does not use chunk/atom compute***
The style of the specified compute is not chunk/atom.

***Compute orientorder/atom cutoff is longer than pairwise cutoff***
Cannot compute order parameter beyond cutoff.

***Compute pair must use group all***
Pair styles accumulate energy on all atoms.

***Compute pe must use group all***
Energies computed by potentials (pair, bond, etc) are computed on all atoms.

***Compute pressure must use group all***
Virial contributions computed by potentials (pair, bond, etc) are computed on all atoms.

***Compute pressure requires temperature ID to include kinetic energy***
The keflag cannot be used unless a temperature compute is provided.

***Compute pressure temperature ID does not compute temperature***
The compute ID assigned to a pressure computation must compute temperature.

***Compute property/atom floating point vector does not exist***
The command is accessing a vector added by the fix property/atom command, that does not exist.

***Compute property/atom integer vector does not exist***
The command is accessing a vector added by the fix property/atom command, that does not exist.

***Compute property/chunk does not use chunk/atom compute***
The style of the specified compute is not chunk/atom.

***Compute property/local cannot use these inputs together***
Only inputs that generate the same number of datums can be used together. E.g. bond and angle quantities cannot be mixed.

***Compute reduce compute calculates global values***
A compute that calculates peratom or local values is required.

***Compute reduce fix calculates global values***
A fix that calculates peratom or local values is required.

***Compute stress/atom temperature ID does not compute temperature***
The specified compute must compute temperature.

***Compute temp/asphere requires extended particles***
This compute cannot be used with point particles.

***Compute temp/body requires bodies***
This compute can only be applied to body particles.

***Compute temp/chunk does not use chunk/atom compute***
The style of the specified compute is not chunk/atom.

***Compute temp/cs requires ghost atoms store velocity***
Use the comm_modify vel yes command to enable this.

***Compute temp/cs used when bonds are not allowed***
This compute only works on pairs of bonded particles.

***Compute torque/chunk does not use chunk/atom compute***
The style of the specified compute is not chunk/atom.

***Compute used in dump between runs is not current***
The compute was not invoked on the current timestep, therefore it cannot be used in a dump between runs.

***Compute used in variable between runs is not current***
Computes cannot be invoked by a variable in between runs. Thus they must have been evaluated on the last timestep of the previous run in order for their value(s) to be accessed. See the page for the variable command for more info.

***Compute used in variable thermo keyword between runs is not current***
Some thermo keywords rely on a compute to calculate their value(s). Computes cannot be invoked by a variable in between runs. Thus they must have been evaluated on the last timestep of the previous run in order for their value(s) to be accessed. See the page for the variable command for more info.

***Compute vcm/chunk does not use chunk/atom compute***
The style of the specified compute is not chunk/atom.

***Computed temperature for fix temp/rescale cannot be 0.0***
Cannot rescale the temperature to a new value if the current temperature is 0.0.

***Core/shell partner atom not found***
Could not find one of the atoms in the bond pair.

***Core/shell partners were not all found***
Could not find or more atoms in the bond pairs.

***Could not adjust g_ewald_6***
The Newton-Raphson solver failed to converge to a good value for g_ewald. This error should not occur for typical problems. Please send an email to the developers.

***Could not compute g_ewald***
The Newton-Raphson solver failed to converge to a good value for g_ewald. This error should not occur for typical problems. Please send an email to the developers.

***Could not compute grid size***
The code is unable to compute a grid size consistent with the desired accuracy. This error should not occur for typical problems. Please send an email to the developers.

***Could not compute grid size for Coulomb interaction***
The code is unable to compute a grid size consistent with the desired accuracy. This error should not occur for typical problems. Please send an email to the developers.

***Could not compute grid size for Dispersion***
The code is unable to compute a grid size consistent with the desired accuracy. This error should not occur for typical problems. Please send an email to the developers.

***Could not create 3d FFT plan***
The FFT setup for the PPPM solver failed, typically due to lack of memory. This is an unusual error. Check the size of the FFT grid you are requesting.

***Could not create 3d grid of processors***
The specified constraints did not allow a Px by Py by Pz grid to be created where Px * Py * Pz = P = total number of processors.

***Could not create 3d remap plan***
The FFT setup in pppm failed.

***Could not create Python function arguments***
This is an internal Python error, possibly because the number of inputs to the function is too large.

***Could not create numa grid of processors***
The specified constraints did not allow this style of grid to be created. Usually this is because the total processor count is not a multiple of the cores/node or the user specified processor count is > 1 in one of the dimensions.

***Could not create twolevel 3d grid of processors***
The specified constraints did not allow this style of grid to be created.

***Could not find Python function***
The provided Python code was run successfully, but it not define a callable function with the required name.

***Could not find change_box group ID***
Group ID used in the change_box command does not exist.

***Could not find compute msd/chunk fix ID***
The compute creates an internal fix, which has been deleted.

***Could not find compute pressure temperature ID***
The compute ID for calculating temperature does not exist.

***Could not find delete_atoms group ID***
Group ID used in the delete_atoms command does not exist.

***Could not find delete_atoms region ID***
Region ID used in the delete_atoms command does not exist.

***Could not find displace_atoms group ID***
Group ID used in the displace_atoms command does not exist.

***Could not find dump group ID***
A group ID used in the dump command does not exist.

***Could not find fix adapt storage fix ID***
This should not happen unless you explicitly deleted a secondary fix that fix adapt created internally.

***Could not find fix group ID***
A group ID used in the fix command does not exist.

***Could not find fix recenter group ID***
A group ID used in the fix recenter command does not exist.

***Could not find fix rigid group ID***
A group ID used in the fix rigid command does not exist.

***Could not find fix_modify ID***
A fix ID used in the fix_modify command does not exist.

***Could not find fix_modify pressure ID***
The compute ID for computing pressure does not exist.

***Could not find fix_modify temperature ID***
The compute ID for computing temperature does not exist.

***Could not find pair fix ID***
A fix is created internally by the pair style to store shear history information. You cannot delete it.

***Could not find set group ID***
Group ID specified in set command does not exist.

***Could not find thermo compute ID***
Compute ID specified in thermo_style command does not exist.

***Could not find thermo custom compute ID***
The compute ID needed by thermo style custom to compute a requested quantity does not exist.

***Could not find thermo custom fix ID***
The fix ID needed by thermo style custom to compute a requested quantity does not exist.

***Could not find thermo fix ID***
Fix ID specified in thermo_style command does not exist.

***Could not find thermo_modify pressure ID***
The compute ID needed by thermo style custom to compute pressure does not exist.

***Could not find thermo_modify temperature ID***
The compute ID needed by thermo style custom to compute temperature does not exist.

***Could not find undump ID***
A dump ID used in the undump command does not exist.

***Could not find velocity group ID***
A group ID used in the velocity command does not exist.

***Could not find velocity temperature ID***
The compute ID needed by the velocity command to compute temperature does not exist.

***Could not find/initialize a specified accelerator device***
Could not initialize at least one of the devices specified for the gpu package

***Could not initialize embedded Python***
The main module in Python was not accessible.

***Could not open Python file***
The specified file of Python code cannot be opened. Check that the path and name are correct.

***Could not process Python file***
The Python code in the specified file was not run successfully by Python, probably due to errors in the Python code.

***Could not process Python string***
The Python code in the here string was not run successfully by Python, probably due to errors in the Python code.

***Coulomb PPPMDisp order has been reduced below minorder***
The default minimum order is 2. This can be reset by the kspace_modify minorder command.

***Coulombic cutoff not supported in pair_style buck/long/coul/coul***
Must use long-range Coulombic interactions.

***Coulombic cutoff not supported in pair_style lj/long/coul/long***
Must use long-range Coulombic interactions.

***Coulombic cutoff not supported in pair_style lj/long/tip4p/long***
Must use long-range Coulombic interactions.

***Coulombic cutoffs of pair hybrid sub-styles do not match***
If using a Kspace solver, all Coulombic cutoffs of long pair styles must be the same.

***Coulombic cut not supported in pair_style lj/long/dipole/long***
Must use long-range Coulombic interactions.

***Create_atoms command before simulation box is defined***
The create_atoms command cannot be used before a read_data, read_restart, or create_box command.

***Create_atoms molecule has atom IDs, but system does not***
The atom_style id command can be used to force atom IDs to be stored.

***Create_atoms molecule must have atom types***
The defined molecule does not specify atom types.

***Create_atoms molecule must have coordinates***
The defined molecule does not specify coordinates.

***Create_atoms region ID does not exist***
A region ID used in the create_atoms command does not exist.

***Create_bonds command requires no kspace_style be defined***
This is so that atom pairs that are already bonded to not appear in the neighbor list.

***Create_bonds command requires special_bonds 1-2 weights be 0.0***
This is so that atom pairs that are already bonded to not appear in the neighbor list.

***Create_bonds max distance > neighbor cutoff***
Can only create bonds for atom pairs that will be in neighbor list.

***Create_box region does not support a bounding box***
Not all regions represent bounded volumes. You cannot use such a region with the create_box command.

***Custom floating point vector for fix store/state does not exist***
The command is accessing a vector added by the fix property/atom command, that does not exist.

***Custom integer vector for fix store/state does not exist***
The command is accessing a vector added by the fix property/atom command, that does not exist.

***Degenerate lattice primitive vectors***
Invalid set of 3 lattice vectors for lattice command.

***Delete_atoms command before simulation box is defined***
The delete_atoms command cannot be used before a read_data, read_restart, or create_box command.

***Delete_atoms cutoff > max neighbor cutoff***
Can only delete atoms in atom pairs that will be in neighbor list.

***Delete_atoms mol yes requires atom attribute molecule***
Cannot use this option with a non-molecular system.

***Delete_atoms requires a pair style be defined***
This is because atom deletion within a cutoff uses a pairwise neighbor list.

***Delete_bonds command before simulation box is defined***
The delete_bonds command cannot be used before a read_data, read_restart, or create_box command.

***Delete_bonds command with no atoms existing***
No atoms are yet defined so the delete_bonds command cannot be used.

***Did not assign all restart atoms correctly***
Atoms read in from the restart file were not assigned correctly to processors. This is likely due to some atom coordinates being outside a non-periodic simulation box. Normally this should not happen. You may wish to use the "remap" option on the read_restart command to see if this helps.

***Did not find all elements in MEAM library file***
Some requested elements were not found in the MEAM file. Check spelling etc.

***Did not find fix shake partner info***
Could not find bond partners implied by fix shake command. This error can be triggered if the delete_bonds command was used before fix shake, and it removed bonds without resetting the 1-2, 1-3, 1-4 weighting list via the special keyword.

***Did not find keyword in table file***
Keyword used in pair_coeff command was not found in table file.

***Did not set pressure for fix rigid/nph***
The press keyword must be specified.

***Did not set temperature for fix rigid/nvt***
The temp keyword must be specified.

***Did not set temperature or pressure for fix rigid/npt***
The temp and press keywords must be specified.

***Dihedral atom missing in delete_bonds***
The delete_bonds command cannot find one or more atoms in a particular dihedral on a particular processor. The pairwise cutoff is too short or the atoms are too far apart to make a valid dihedral.

***Dihedral atom missing in set command***
The set command cannot find one or more atoms in a particular dihedral on a particular processor. The pairwise cutoff is too short or the atoms are too far apart to make a valid dihedral.

***Dihedral charmm is incompatible with Pair style***
Dihedral style charmm must be used with a pair style charmm in order for the 1-4 epsilon/sigma parameters to be defined.

***Dihedral coeff for hybrid has invalid style***
Dihedral style hybrid uses another dihedral style as one of its coefficients. The dihedral style used in the dihedral_coeff command or read from a restart file is not recognized.

***Dihedral coeffs are not set***
No dihedral coefficients have been assigned in the data file or via the dihedral_coeff command.

***Dihedral/improper extent > half of periodic box length***
This error was detected by the neigh_modify check yes setting. It is an error because the dihedral atoms are so far apart it is ambiguous how it should be defined.

***Dihedral_coeff command before dihedral_style is defined***
Coefficients cannot be set in the data file or via the dihedral_coeff command until an dihedral_style has been assigned.

***Dihedral_coeff command before simulation box is defined***
The dihedral_coeff command cannot be used before a read_data, read_restart, or create_box command.

***Dihedral_coeff command when no dihedrals allowed***
The chosen atom style does not allow for dihedrals to be defined.

***Dihedral_style command when no dihedrals allowed***
The chosen atom style does not allow for dihedrals to be defined.

***Dihedrals assigned incorrectly***
Dihedrals read in from the data file were not assigned correctly to atoms. This means there is something invalid about the topology definitions.

***Dihedrals defined but no dihedral types***
The data file header lists dihedrals but no dihedral types.

***Dimension command after simulation box is defined***
The dimension command cannot be used after a read_data, read_restart, or create_box command.

***Dispersion PPPMDisp order has been reduced below minorder***
The default minimum order is 2. This can be reset by the kspace_modify minorder command.

***Displace_atoms command before simulation box is defined***
The displace_atoms command cannot be used before a read_data, read_restart, or create_box command.

***Divide by 0 in influence function***
This should not normally occur. It is likely a problem with your model.

***Divide by 0 in influence function of pair peri/lps***
This should not normally occur. It is likely a problem with your model.

***Dump atom/gz only writes compressed files***
The dump atom/gz output file name must have a .gz suffix.

***Dump cfg arguments must start with 'mass type xs ys zs' or 'mass type xsu ysu zsu'***
This is a requirement of the CFG output format. See the dump cfg doc page for more details.

***Dump cfg requires one snapshot per file***
Use the wildcard "*" character in the filename.

***Dump cfg/gz only writes compressed files***
The dump cfg/gz output file name must have a .gz suffix.

***Dump custom and fix not computed at compatible times***
The fix must produce per-atom quantities on timesteps that dump custom needs them.

***Dump custom variable is not atom-style variable***
Only atom-style variables generate per-atom quantities, needed for dump output.

***Dump custom/gz only writes compressed files***
The dump custom/gz output file name must have a .gz suffix.

***Dump dcd of non-matching # of atoms***
Every snapshot written by dump dcd must contain the same # of atoms.

***Dump dcd requires sorting by atom ID***
Use the dump_modify sort command to enable this.

***Dump every variable returned a bad timestep***
The variable must return a timestep greater than the current timestep.

***Dump image requires one snapshot per file***
Use a "*" in the filename.

***Dump local and fix not computed at compatible times***
The fix must produce per-atom quantities on timesteps that dump local needs them.

***Dump local cannot sort by atom ID***
This is because dump local does not really dump per-atom info.

***Dump local count is not consistent across input fields***
Every column of output must be the same length.

***Dump modify element names do not match atom types***
Number of element names must equal number of atom types.

***Dump xtc requires sorting by atom ID***
Use the dump_modify sort command to enable this.

***Dump xyz/gz only writes compressed files***
The dump xyz/gz output file name must have a .gz suffix.

***Dump_modify format string is too short***
There are more fields to be dumped in a line of output than your format string specifies.

***Dumping an atom property that is not allocated***
The chosen atom style does not define the per-atom quantity being dumped.

***Duplicate particle in PeriDynamic bond - simulation box is too small***
This is likely because your box length is shorter than 2 times the bond length.

***Electronic temperature dropped below zero***
Something has gone wrong with the fix ttm electron temperature model.

***Element not defined in potential file***
The specified element is not in the potential file.

***Empty brackets in variable***
There is no variable syntax that uses empty brackets. Check the variable doc page.

***Epsilon or sigma reference not set by pair style in ewald/n***
The pair style is not providing the needed epsilon or sigma values.

***Error in vdw spline: inner radius > outer radius***
A pre-tabulated spline is invalid. Likely a problem with the potential parameters.

***Error writing averaged chunk data***
Something in the output to the file triggered an error.

***Error writing file header***
Something in the output to the file triggered an error.

***Error writing out correlation data***
Something in the output to the file triggered an error.

***Error writing out histogram data***
Something in the output to the file triggered an error.

***Error writing out time averaged data***
Something in the output to the file triggered an error.

***Failed to allocate %ld bytes for array %s***
Your LAMMPS simulation has run out of memory. You need to run a smaller simulation or on more processors.

***Failed to open FFmpeg pipeline to file %s***
The specified file cannot be opened. Check that the path and name are correct and writable and that the FFmpeg executable can be found and run.

***Failed to reallocate %ld bytes for array %s***
Your LAMMPS simulation has run out of memory. You need to run a smaller simulation or on more processors.

***Fewer SRD bins than processors in some dimension***
This is not allowed. Make your SRD bin size smaller.

***File variable could not read value***
Check the file assigned to the variable.

***Fix %s does not allow use of dynamic group***
Dynamic groups have not yet been enabled for this fix.

***Fix SRD: bad bin assignment for SRD advection***
Something has gone wrong in your SRD model; try using more conservative settings.

***Fix SRD: bad search bin assignment***
Something has gone wrong in your SRD model; try using more conservative settings.

***Fix SRD: bad stencil bin for big particle***
Something has gone wrong in your SRD model; try using more conservative settings.

***Fix SRD: too many big particles in bin***
Reset the ATOMPERBIN parameter at the top of fix_srd.cpp to a larger value, and re-compile the code.

***Fix SRD: too many walls in bin***
This should not happen unless your system has been setup incorrectly.

***Fix adapt interface to this pair style not supported***
New coding for the pair style would need to be done.

***Fix adapt pair style param not supported***
The pair style does not know about the parameter you specified.

***Fix adapt requires atom attribute charge***
The atom style being used does not specify an atom charge.

***Fix adapt requires atom attribute diameter***
The atom style being used does not specify an atom diameter.

***Fix append/atoms requires a lattice be defined***
Use the lattice command for this purpose.

***Fix ave/atom compute does not calculate a per-atom vector***
A compute used by fix ave/atom must generate per-atom values.

***Fix ave/atom compute does not calculate per-atom values***
A compute used by fix ave/atom must generate per-atom values.

***Fix ave/atom fix does not calculate a per-atom vector***
A fix used by fix ave/atom must generate per-atom values.

***Fix ave/atom fix does not calculate per-atom values***
A fix used by fix ave/atom must generate per-atom values.

***Fix ave/atom variable is not atom-style variable***
A variable used by fix ave/atom must generate per-atom values.

***Fix ave/chunk does not use chunk/atom compute***
The specified compute is not for a compute chunk/atom command.

***Fix ave/histo inputs are not all global, peratom, or local***
All inputs in a single fix ave/histo command must be of the same style.

***Fix ave/time cannot set output array intensive/extensive from these inputs***
One of more of the vector inputs has individual elements which are flagged as intensive or extensive. Such an input cannot be flagged as all intensive/extensive when turned into an array by fix ave/time.

***Fix ave/time cannot use variable with vector mode***
Variables produce scalar values.

***Fix balance rcb cannot be used with comm_style brick***
Comm_style tiled must be used instead.

***Fix balance shift string is invalid***
The string can only contain the characters "x", "y", or "z".

***Fix bond/break needs ghost atoms from further away***
This is because the fix needs to walk bonds to a certain distance to acquire needed info, The comm_modify cutoff command can be used to extend the communication range.

***Fix bond/create cutoff is longer than pairwise cutoff***
This is not allowed because bond creation is done using the pairwise neighbor list.

***Fix bond/create induced too many angles/dihedrals/impropers per atom***
See the read_data command for info on using the "extra/angle/per/atom", (or dihedral, improper) keywords to allow for additional angles, dihedrals, and impropers to be formed.

***Fix bond/create needs ghost atoms from further away***
This is because the fix needs to walk bonds to a certain distance to acquire needed info, The comm_modify cutoff command can be used to extend the communication range.

***Fix bond/react: Cannot use fix bond/react with non-molecular systems***
Only systems with bonds that can be changed can be used. Atom_style template does not qualify.

***Fix bond/react: Invalid template atom ID in map file***
Atom IDs in molecule templates range from 1 to the number of atoms in the template.

***Fix bond/react: Rmax cutoff is longer than pairwise cutoff***
This is not allowed because bond creation is done using the pairwise neighbor list.

***Fix bond/react: Molecule template ID for fix bond/react does not exist***
A valid molecule template must have been created with the molecule command.

***Fix bond/react: Reaction templates must contain the same number of atoms***
There should be a one-to-one correspondence between atoms in the pre-reacted and post-reacted templates, as specified by the map file.

***Fix bond/react: Unknown section in map file***
Please ensure reaction map files are properly formatted.

***Fix bond/react: Atom/Bond type affected by reaction too close to template edge***
This means an atom which changes type or connectivity during the reaction is too close to an 'edge' atom defined in the map file. This could cause incorrect assignment of bonds, angle, etc. Generally, this means you must include more atoms in your templates, such that there are at least two atoms between each atom involved in the reaction and an edge atom.

***Fix bond/react: Fix bond/react needs ghost atoms from farther away***
This is because a processor needs to map the entire unreacted molecule template onto simulation atoms it knows about. The comm_modify cutoff command can be used to extend the communication range.

***Fix bond/react: Molecule template 'Coords' section required for chiralIDs keyword***
The coordinates of atoms in the pre-reacted template are used to determine chirality.

***Fix bond/react special bond generation overflow***
The number of special bonds per-atom created by a reaction exceeds the system setting. See the read_data or create_box command for how to specify this value.

***Fix bond/react topology/atom exceed system topology/atom***
The number of bonds, angles etc per-atom created by a reaction exceeds the system setting. See the read_data or create_box command for how to specify this value.

***Fix bond/swap cannot use dihedral or improper styles***
These styles cannot be defined when using this fix.

***Fix box/relax generated negative box length***
The pressure being applied is likely too large. Try applying it incrementally, to build to the high pressure.

***Fix command before simulation box is defined***
The fix command cannot be used before a read_data, read_restart, or create_box command.

***Fix deform cannot use yz variable with xy***
The yz setting cannot be a variable if xy deformation is also specified. This is because LAMMPS cannot determine if the yz setting will induce a box flip which would be invalid if xy is also changing.

***Fix deform is changing yz too much with xy***
When both yz and xy are changing, it induces changes in xz if the box must flip from one tilt extreme to another. Thus it is not allowed for yz to grow so much that a flip is induced.

***Fix deform tilt factors require triclinic box***
Cannot deform the tilt factors of a simulation box unless it is a triclinic (non-orthogonal) box.

***Fix deform volume setting is invalid***
Cannot use volume style unless other dimensions are being controlled.

***Fix deposit molecule must have atom types***
The defined molecule does not specify atom types.

***Fix deposit molecule must have coordinates***
The defined molecule does not specify coordinates.

***Fix deposit molecule template ID must be same as atom_style template ID***
When using atom_style template, you cannot deposit molecules that are not in that template.

***Fix deposit region cannot be dynamic***
Only static regions can be used with fix deposit.

***Fix deposit region does not support a bounding box***
Not all regions represent bounded volumes. You cannot use such a region with the fix deposit command.

***Fix efield requires atom attribute q or mu***
The atom style defined does not have this attribute.

***Fix efield with dipoles cannot use atom-style variables***
This option is not supported.

***Fix evaporate molecule requires atom attribute molecule***
The atom style being used does not define a molecule ID.

***Fix external callback function not set***
This must be done by an external program in order to use this fix.

***Fix for fix ave/atom not computed at compatible time***
Fixes generate their values on specific timesteps. Fix ave/atom is requesting a value on a non-allowed timestep.

***Fix for fix ave/chunk not computed at compatible time***
Fixes generate their values on specific timesteps. Fix ave/chunk is requesting a value on a non-allowed timestep.

***Fix for fix ave/correlate not computed at compatible time***
Fixes generate their values on specific timesteps. Fix ave/correlate is requesting a value on a non-allowed timestep.

***Fix for fix ave/histo not computed at compatible time***
Fixes generate their values on specific timesteps. Fix ave/histo is requesting a value on a non-allowed timestep.

***Fix for fix ave/spatial not computed at compatible time***
Fixes generate their values on specific timesteps. Fix ave/spatial is requesting a value on a non-allowed timestep.

***Fix for fix ave/time not computed at compatible time***
Fixes generate their values on specific timesteps. Fix ave/time is requesting a value on a non-allowed timestep.

***Fix for fix store/state not computed at compatible time***
Fixes generate their values on specific timesteps. Fix store/state is requesting a value on a non-allowed timestep.

***Fix for fix vector not computed at compatible time***
Fixes generate their values on specific timesteps. Fix vector is requesting a value on a non-allowed timestep.

***Fix freeze requires atom attribute torque***
The atom style defined does not have this attribute.

***Fix gcmc cannot exchange individual atoms belonging to a molecule***
This is an error since you should not delete only one atom of a molecule. The user has specified atomic (non-molecular) gas exchanges, but an atom belonging to a molecule could be deleted.

***Fix gcmc molecule command requires that atoms have molecule attributes***
Should not choose the gcmc molecule feature if no molecules are being simulated. The general molecule flag is off, but gcmc's molecule flag is on.

***Fix gcmc molecule must have atom types***
The defined molecule does not specify atom types.

***Fix gcmc molecule must have coordinates***
The defined molecule does not specify coordinates.

***Fix gcmc molecule template ID must be same as atom_style template ID***
When using atom_style template, you cannot insert molecules that are not in that template.

***Fix gcmc put atom outside box***
This should not normally happen. Contact the developers.

***Fix gcmc ran out of available atom IDs***
See the setting for tagint in the src/lmptype.h file.

***Fix gcmc ran out of available molecule IDs***
See the setting for tagint in the src/lmptype.h file.

***Fix gcmc region cannot be dynamic***
Only static regions can be used with fix gcmc.

***Fix gcmc region does not support a bounding box***
Not all regions represent bounded volumes. You cannot use such a region with the fix gcmc command.

***Fix heat kinetic energy of an atom went negative***
This will cause the velocity rescaling about to be performed by fix heat to be invalid.

***Fix heat kinetic energy went negative***
This will cause the velocity rescaling about to be performed by fix heat to be invalid.

***Fix in variable not computed at compatible time***
Fixes generate their values on specific timesteps. The variable is requesting the values on a non-allowed timestep.

***Fix langevin angmom is not yet implemented with kokkos***
This option is not yet available.

***Fix langevin angmom requires extended particles***
This fix option cannot be used with point particles.

***Fix langevin gjf cannot have period equal to dt/2***
If the period is equal to dt/2 then division by zero will happen.

***Fix langevin omega is not yet implemented with kokkos***
This option is not yet available.

***Fix langevin omega requires extended particles***
One of the particles has radius 0.0.

***Fix langevin period must be > 0.0***
The time window for temperature relaxation must be > 0

***Fix npt/nph has tilted box too far in one step - periodic cell is too far from equilibrium state***
Self-explanatory. The change in the box tilt is too extreme on a short timescale.

***Fix numdiff requires an atom map, see atom_modify***
Self-explanatory. Efficient loop over all atoms for numerical difference requires an atom map.

***Fix numdiff requires consecutive atom IDs***
Self-explanatory. Efficient loop over all atoms for numerical difference requires consecutive atom IDs.

***Fix numdiff/virial must use group all***
Virial contributions computed by this fix are computed on all atoms.

***Fix nve/asphere requires extended particles***
This fix can only be used for particles with a shape setting.

***Fix nve/asphere/noforce requires extended particles***
One of the particles is not an ellipsoid.

***Fix nve/body requires bodies***
This fix can only be used for particles that are bodies.

***Fix nve/sphere dipole requires atom attribute mu***
An atom style with this attribute is needed.

***Fix nve/sphere requires extended particles***
This fix can only be used for particles of a finite size.

***Fix nvt/nph/npt asphere requires extended particles***
The shape setting for a particle in the fix group has shape = 0.0, which means it is a point particle.

***Fix nvt/sphere requires extended particles***
This fix can only be used for particles of a finite size.

***Fix orient/fcc file open failed***
The fix orient/fcc command could not open a specified file.

***Fix orient/fcc file read failed***
The fix orient/fcc command could not read the needed parameters from a specified file.

***Fix orient/fcc found self twice***
The neighbor lists used by fix orient/fcc are messed up. If this error occurs, it is likely a bug, so send an email to the developers.

***Fix peri neigh does not exist***
Somehow a fix that the pair style defines has been deleted.

***Fix pour molecule must have atom types***
The defined molecule does not specify atom types.

***Fix pour molecule must have coordinates***
The defined molecule does not specify coordinates.

***Fix pour molecule template ID must be same as atom style template ID***
When using atom_style template, you cannot pour molecules that are not in that template.

***Fix pour region cannot be dynamic***
Only static regions can be used with fix pour.

***Fix pour region does not support a bounding box***
Not all regions represent bounded volumes. You cannot use such a region with the fix pour command.

***Fix pour requires atom attributes radius, rmass***
The atom style defined does not have these attributes.

***Fix property/atom vector name already exists***
The name for an integer or floating-point vector must be unique.

***Fix qeq/comb requires atom attribute q***
An atom style with charge must be used to perform charge equilibration.

***Fix qeq/point has insufficient QEq matrix size***
Occurs when number of neighbor atoms for an atom increased too much during a run. Increase SAFE_ZONE and MIN_CAP in fix_qeq.h and re-compile.

***Fix qeq/shielded has insufficient QEq matrix size***
Occurs when number of neighbor atoms for an atom increased too much during a run. Increase SAFE_ZONE and MIN_CAP in fix_qeq.h and re-compile.

***Fix qeq/slater could not extract params from pair coul/streitz***
This should not happen unless pair coul/streitz has been altered.

***Fix qeq/slater has insufficient QEq matrix size***
Occurs when number of neighbor atoms for an atom increased too much during a run. Increase SAFE_ZONE and MIN_CAP in fix_qeq.h and re-compile.

***Fix reax/bonds numbonds > nsbmax_most***
The limit of the number of bonds expected by the ReaxFF force field was exceeded.

***Fix rigid atom has non-zero image flag in a non-periodic dimension***
Image flags for non-periodic dimensions should not be set.

***Fix rigid npt/nph does not yet allow triclinic box***
This is a current restriction in LAMMPS.

***Fix rigid/small atom has non-zero image flag in a non-periodic dimension***
Image flags for non-periodic dimensions should not be set.

***Fix rigid/small molecule must have atom types***
The defined molecule does not specify atom types.

***Fix rigid/small molecule must have coordinates***
The defined molecule does not specify coordinates.

***Fix rigid: Bad principal moments***
The principal moments of inertia computed for a rigid body are not within the required tolerances.

***Fix shake molecule template must have shake info***
The defined molecule does not specify SHAKE information.

***Fix srd can only currently be used with comm_style brick***
This is a current restriction in LAMMPS.

***Fix srd lamda must be >= 0.6 of SRD grid size***
This is a requirement for accuracy reasons.

***Fix srd no-slip requires atom attribute torque***
This is because the SRD collisions will impart torque to the solute particles.

***Fix srd requires ghost atoms store velocity***
Use the comm_modify vel yes command to enable this.

***Fix store/state compute does not calculate a per-atom array***
The compute calculates a per-atom vector.

***Fix store/state compute does not calculate a per-atom vector***
The compute calculates a per-atom vector.

***Fix store/state compute does not calculate per-atom values***
Computes that calculate global or local quantities cannot be used with fix store/state.

***Fix store/state fix does not calculate a per-atom array***
The fix calculates a per-atom vector.

***Fix store/state fix does not calculate a per-atom vector***
The fix calculates a per-atom array.

***Fix store/state fix does not calculate per-atom values***
Fixes that calculate global or local quantities cannot be used with fix store/state.

***Fix store/state variable is not atom-style variable***
Only atom-style variables calculate per-atom quantities.

***Fix temp/csld is not compatible with fix rattle or fix shake***
These two commands cannot currently be used together with fix temp/csld.

***Fix tfmc is not compatible with fix shake***
These two commands cannot currently be used together.

***Fix tmd must come after integration fixes***
Any fix tmd command must appear in the input script after all time integration fixes (nve, nvt, npt). See the fix tmd documentation for details.

***Fix used in compute chunk/atom not computed at compatible time***
The chunk/atom compute cannot query the output of the fix on a timestep it is needed.

***Fix used in compute reduce not computed at compatible time***
Fixes generate their values on specific timesteps. Compute reduce is requesting a value on a non-allowed timestep.

***Fix used in compute slice not computed at compatible time***
Fixes generate their values on specific timesteps. Compute slice is requesting a value on a non-allowed timestep.

***Fix vector cannot set output array intensive/extensive from these inputs***
The inputs to the command have conflicting intensive/extensive attributes. You need to use more than one fix vector command.

***Fix wall/colloid requires extended particles***
One of the particles has radius 0.0.

***Fix wall/gran is incompatible with Pair style***
Must use a granular pair style to define the parameters needed for this fix.

***Fix wall/piston command only available at zlo***
The face keyword must be zlo.

***Fix wall/region colloid requires extended particles***
One of the particles has radius 0.0.

***Fix_modify pressure ID does not compute pressure***
The compute ID assigned to the fix must compute pressure.

***Fix_modify temperature ID does not compute temperature***
The compute ID assigned to the fix must compute temperature.

***For triclinic deformation, specified target stress must be hydrostatic***
Triclinic pressure control is allowed using the tri keyword, but non-hydrostatic pressure control can not be used in this case.

***Found no restart file matching pattern***
When using a "*" in the restart file name, no matching file was found.

***GPU particle split must be set to 1 for this pair style.***
For this pair style, you cannot run part of the force calculation on the host. See the package command.

***GPUs are requested but Kokkos has not been compiled for CUDA***
Re-compile Kokkos with CUDA support to use GPUs.

***Ghost velocity forward comm not yet implemented with Kokkos***
This is a current restriction.

***Gmask function in equal-style variable formula***
Gmask is per-atom operation.

***Gravity changed since fix pour was created***
The gravity vector defined by fix gravity must be static.

***Grmask function in equal-style variable formula***
Grmask is per-atom operation.

***Group ID does not exist***
A group ID used in the group command does not exist.

***Group all cannot be made dynamic***
This operation is not allowed.

***Group command before simulation box is defined***
The group command cannot be used before a read_data, read_restart, or create_box command.

***Group region ID does not exist***
A region ID used in the group command does not exist.

***If read_dump purges it cannot replace or trim***
These operations are not compatible. See the read_dump doc page for details.

***Illegal . . . command***
Self-explanatory. Check the input script syntax and compare to the documentation for the command. You can use -echo screen as a command-line option when running LAMMPS to see the offending line.

***Illegal COMB parameter***
One or more of the coefficients defined in the potential file is invalid.

***Illegal COMB3 parameter***
One or more of the coefficients defined in the potential file is invalid.

***Illegal Stillinger-Weber parameter***
One or more of the coefficients defined in the potential file is invalid.

***Illegal Tersoff parameter***
One or more of the coefficients defined in the potential file is invalid.

***Illegal Vashishta parameter***
One or more of the coefficients defined in the potential file is invalid.

***Illegal coul/streitz parameter***
One or more of the coefficients defined in the potential file is invalid.

***Illegal fix gcmc gas mass <= 0***
The computed mass of the designated gas molecule or atom type was less than or equal to zero.

***Illegal fix tfmc random seed***
Seeds can only be nonzero positive integers.

***Illegal fix wall/piston velocity***
The piston velocity must be positive.

***Illegal nb3b/harmonic parameter***
One or more of the coefficients defined in the potential file is invalid.

***Illegal number of angle table entries***
There must be at least 2 table entries.

***Illegal number of bond table entries***
There must be at least 2 table entries.

***Illegal number of pair table entries***
There must be at least 2 table entries.

***Illegal or unset periodicity in restart***
This error should not normally occur unless the restart file is invalid.

***Illegal range increment value***
The increment must be >= 1.

***Illegal simulation box***
The lower bound of the simulation box is greater than the upper bound.

***Illegal size double vector read requested***
This error should not normally occur unless the restart file is invalid.

***Illegal size integer vector read requested***
This error should not normally occur unless the restart file is invalid.

***Illegal size string or corrupt restart***
This error should not normally occur unless the restart file is invalid.

***Imageint setting in lmptype.h is invalid***
Imageint must be as large or larger than smallint.

***Imageint setting in lmptype.h is not compatible***
Format of imageint stored in restart file is not consistent with LAMMPS version you are running. See the settings in src/lmptype.h

***Improper atom missing in delete_bonds***
The delete_bonds command cannot find one or more atoms in a particular improper on a particular processor. The pairwise cutoff is too short or the atoms are too far apart to make a valid improper.

***Improper atom missing in set command***
The set command cannot find one or more atoms in a particular improper on a particular processor. The pairwise cutoff is too short or the atoms are too far apart to make a valid improper.

***Improper coeff for hybrid has invalid style***
Improper style hybrid uses another improper style as one of its coefficients. The improper style used in the improper_coeff command or read from a restart file is not recognized.

***Improper coeffs are not set***
No improper coefficients have been assigned in the data file or via the improper_coeff command.

***Improper_coeff command before improper_style is defined***
Coefficients cannot be set in the data file or via the improper_coeff command until an improper_style has been assigned.

***Improper_coeff command before simulation box is defined***
The improper_coeff command cannot be used before a read_data, read_restart, or create_box command.

***Improper_coeff command when no impropers allowed***
The chosen atom style does not allow for impropers to be defined.

***Improper_style command when no impropers allowed***
The chosen atom style does not allow for impropers to be defined.

***Impropers assigned incorrectly***
Impropers read in from the data file were not assigned correctly to atoms. This means there is something invalid about the topology definitions.

***Impropers defined but no improper types***
The data file header lists improper but no improper types.

***Incompatible KIM Simulator Model***
The requested KIM Simulator Model was defined for a different MD code and thus is not compatible with LAMMPS.

***Incompatible units for KIM Simulator Model***
The selected unit style is not compatible with the requested KIM Simulator Model.

***Incomplete use of variables in create_atoms command***
The var and set options must be used together.

***Inconsistent iparam/jparam values in fix bond/create command***
If itype and jtype are the same, then their maxbond and newtype settings must also be the same.

***Inconsistent line segment in data file***
The end points of the line segment are not equal distances from the center point which is the atom coordinate.

***Inconsistent triangle in data file***
The centroid of the triangle as defined by the corner points is not the atom coordinate.

***Inconsistent use of finite-size particles by molecule template molecules***
Not all of the molecules define a radius for their constituent particles.

***Incorrect # of floating-point values in Bodies section of data file***
See page for body style.

***Incorrect # of integer values in Bodies section of data file***
See page for body style.

***Incorrect %s format in data file***
A section of the data file being read by fix property/atom does not have the correct number of values per line.

***Incorrect SNAP parameter file***
The file cannot be parsed correctly, check its internal syntax.

***Incorrect atom format in data file***
Number of values per atom line in the data file is not consistent with the atom style.

***Incorrect atom format in neb file***
The number of fields per line is not what expected.

***Incorrect bonus data format in data file***
See the read_data page for a description of how various kinds of bonus data must be formatted for certain atom styles.

***Incorrect boundaries with slab Ewald***
Must have periodic x,y dimensions and non-periodic z dimension to use 2d slab option with Ewald.

***Incorrect boundaries with slab EwaldDisp***
Must have periodic x,y dimensions and non-periodic z dimension to use 2d slab option with Ewald.

***Incorrect boundaries with slab PPPM***
Must have periodic x,y dimensions and non-periodic z dimension to use 2d slab option with PPPM.

***Incorrect boundaries with slab PPPMDisp***
Must have periodic x,y dimensions and non-periodic z dimension to use 2d slab option with pppm/disp.

***Incorrect conversion in format string***
A format style variable was not using either a %f, a %g, or a %e conversion. Or an immediate variable with format suffix was not using either a %f, a %g or a %e conversion in the format suffix.

***Incorrect element names in ADP potential file***
The element names in the ADP file do not match those requested.

***Incorrect element names in EAM potential file***
The element names in the EAM file do not match those requested.

***Incorrect format in COMB potential file***
Incorrect number of words per line in the potential file.

***Incorrect format in COMB3 potential file***
Incorrect number of words per line in the potential file.

***Incorrect format in MEAM library file***
Incorrect number of words per line in the potential file.

***Incorrect format in SNAP coefficient file***
Incorrect number of words per line in the coefficient file.

***Incorrect format in SNAP parameter file***
Incorrect number of words per line in the parameter file.

***Incorrect format in Stillinger-Weber potential file***
Incorrect number of words per line in the potential file.

***Incorrect format in TMD target file***
Format of file read by fix tmd command is incorrect.

***Incorrect format in Tersoff potential file***
Incorrect number of words per line in the potential file.

***Incorrect format in Vashishta potential file***
Incorrect number of words per line in the potential file.

***Incorrect format in coul/streitz potential file***
Incorrect number of words per line in the potential file.

***Incorrect format in nb3b/harmonic potential file***
Incorrect number of words per line in the potential file.

***Incorrect integer value in Bodies section of data file***
See page for body style.

***Incorrect rigid body format in fix rigid file***
The number of fields per line is not what expected.

***Incorrect rigid body format in fix rigid/small file***
The number of fields per line is not what expected.

***Incorrect velocity format in data file***
Each atom style defines a format for the Velocity section of the data file. The read-in lines do not match.

***Indexed per-atom vector in variable formula without atom map***
Accessing a value from an atom vector requires the ability to lookup an atom index, which is provided by an atom map. An atom map does not exist (by default) for non-molecular problems. Using the atom_modify map command will force an atom map to be created.

***Input line quote not followed by white-space***
An end quote must be followed by white-space.

***Insufficient Jacobi rotations for body nparticle***
Eigensolve for rigid body was not sufficiently accurate.

***Insufficient Jacobi rotations for rigid body***
Eigensolve for rigid body was not sufficiently accurate.

***Insufficient Jacobi rotations for rigid molecule***
Eigensolve for rigid body was not sufficiently accurate.

***Insufficient Jacobi rotations for triangle***
The calculation of the inertia tensor of the triangle failed. This should not happen if it is a reasonably shaped triangle.

***Insufficient memory on accelerator***
There is insufficient memory on one of the devices specified for the gpu package

***Internal error in atom_style body***
This error should not occur. Contact the developers.

***Invalid LAMMPS restart file***
The file does not appear to be a LAMMPS restart file since it does not contain the correct magic string at the beginning.

***Invalid molecule ID in molecule file***
Molecule ID must be a non-zero positive integer.

***Invalid REAX atom type***
There is a mis-match between LAMMPS atom types and the elements listed in the ReaxFF force field file.

***Invalid angle table length***
Length must be 2 or greater.

***Invalid angle type in Angles section of data file***
Angle type must be positive integer and within range of specified angle types.

***Invalid args for non-hybrid pair coefficients***
"NULL" is only supported in pair_coeff calls when using pair hybrid

***Invalid argument to factorial %d***
N must be >= 0 and <= 167, otherwise the factorial result is too large.

***Invalid atom ID in %s section of data file***
An atom in a section of the data file being read by fix property/atom has an invalid atom ID that is <= 0 or > the maximum existing atom ID.

***Invalid atom ID in Angles section of data file***
Atom IDs must be positive integers and within range of defined atoms.

***Invalid atom ID in Atoms section of data file***
Atom IDs must be positive integers.

***Invalid atom ID in Bodies section of data file***
Atom IDs must be positive integers and within range of defined atoms.

***Invalid atom ID in Bonds section of data file***
Atom IDs must be positive integers and within range of defined atoms.

***Invalid atom ID in Bonus section of data file***
Atom IDs must be positive integers and within range of defined atoms.

***Invalid atom ID in Dihedrals section of data file***
Atom IDs must be positive integers and within range of defined atoms.

***Invalid atom ID in Impropers section of data file***
Atom IDs must be positive integers and within range of defined atoms.

***Invalid atom ID in Velocities section of data file***
Atom IDs must be positive integers and within range of defined atoms.

***Invalid atom IDs in neb file***
An ID in the file was not found in the system.

***Invalid atom diameter in molecule file***
Diameters must be >= 0.0.

***Invalid atom mass for fix shake***
Mass specified in fix shake command must be > 0.0.

***Invalid atom mass in molecule file***
Masses must be > 0.0.

***Invalid atom type in Atoms section of data file***
Atom types must range from 1 to specified # of types.

***Invalid atom type in create_atoms command***
The create_box command specified the range of valid atom types. An invalid type is being requested.

***Invalid atom type in create_atoms mol command***
The atom types in the defined molecule are added to the value specified in the create_atoms command, as an offset. The final value for each atom must be between 1 to N, where N is the number of atom types.

***Invalid atom type in fix atom/swap command***
The atom type specified in the atom/swap command does not exist.

***Invalid atom type in fix deposit mol command***
The atom types in the defined molecule are added to the value specified in the create_atoms command, as an offset. The final value for each atom must be between 1 to N, where N is the number of atom types.

***Invalid atom type in fix gcmc command***
The atom type specified in the gcmc command does not exist.

***Invalid atom type in fix pour mol command***
The atom types in the defined molecule are added to the value specified in the create_atoms command, as an offset. The final value for each atom must be between 1 to N, where N is the number of atom types.

***Invalid atom type in molecule file***
Atom types must range from 1 to specified # of types.

***Invalid atom type in neighbor exclusion list***
Atom types must range from 1 to Ntypes inclusive.

***Invalid atom type index for fix shake***
Atom types must range from 1 to Ntypes inclusive.

***Invalid atom types in pair_write command***
Atom types must range from 1 to Ntypes inclusive.

***Invalid atom vector in variable formula***
The atom vector is not recognized.

***Invalid atom_style body command***
No body style argument was provided.

***Invalid basis setting in create_atoms command***
The basis index must be between 1 to N where N is the number of basis atoms in the lattice. The type index must be between 1 to N where N is the number of atom types.

***Invalid basis setting in fix append/atoms command***
The basis index must be between 1 to N where N is the number of basis atoms in the lattice. The type index must be between 1 to N where N is the number of atom types.

***Invalid bin bounds in compute chunk/atom***
The lo/hi values are inconsistent.

***Invalid bin bounds in fix ave/spatial***
The lo/hi values are inconsistent.

***Invalid body nparticle command***
Arguments in atom-style command are not correct.

***Invalid bond table length***
Length must be 2 or greater.

***Invalid bond type in Bonds section of data file***
Bond type must be positive integer and within range of specified bond types.

***Invalid coeffs for this dihedral style***
Cannot set class 2 coeffs in data file for this dihedral style.

***Invalid color in dump_modify command***
The specified color name was not in the list of recognized colors. See the dump_modify doc page.

***Invalid color map min/max values***
The min/max values are not consistent with either each other or with values in the color map.

***Invalid command-line argument***
One or more command-line arguments is invalid. Check the syntax of the command you are using to launch LAMMPS.

***Invalid compute ID in variable formula***
The compute is not recognized.

***Invalid create_atoms rotation vector for 2d model***
The rotation vector can only have a z component.

***Invalid custom OpenCL parameter string.***
There are not enough or too many parameters in the custom string for package GPU.

***Invalid cutoff in comm_modify command***
Specified cutoff must be >= 0.0.

***Invalid cutoffs in pair_write command***
Inner cutoff must be larger than 0.0 and less than outer cutoff.

***Invalid d1 or d2 value for pair colloid coeff***
Neither d1 or d2 can be < 0.

***Invalid data file section: Angle Coeffs***
Atom style does not allow angles.

***Invalid data file section: AngleAngle Coeffs***
Atom style does not allow impropers.

***Invalid data file section: AngleAngleTorsion Coeffs***
Atom style does not allow dihedrals.

***Invalid data file section: AngleTorsion Coeffs***
Atom style does not allow dihedrals.

***Invalid data file section: Angles***
Atom style does not allow angles.

***Invalid data file section: Bodies***
Atom style does not allow bodies.

***Invalid data file section: Bond Coeffs***
Atom style does not allow bonds.

***Invalid data file section: BondAngle Coeffs***
Atom style does not allow angles.

***Invalid data file section: BondBond Coeffs***
Atom style does not allow angles.

***Invalid data file section: BondBond13 Coeffs***
Atom style does not allow dihedrals.

***Invalid data file section: Bonds***
Atom style does not allow bonds.

***Invalid data file section: Dihedral Coeffs***
Atom style does not allow dihedrals.

***Invalid data file section: Dihedrals***
Atom style does not allow dihedrals.

***Invalid data file section: Ellipsoids***
Atom style does not allow ellipsoids.

***Invalid data file section: EndBondTorsion Coeffs***
Atom style does not allow dihedrals.

***Invalid data file section: Improper Coeffs***
Atom style does not allow impropers.

***Invalid data file section: Impropers***
Atom style does not allow impropers.

***Invalid data file section: Lines***
Atom style does not allow lines.

***Invalid data file section: MiddleBondTorsion Coeffs***
Atom style does not allow dihedrals.

***Invalid data file section: Triangles***
Atom style does not allow triangles.

***Invalid delta_conf in tad command***
The value must be between 0 and 1 inclusive.

***Invalid density in Atoms section of data file***
Density value cannot be <= 0.0.

***Invalid density in set command***
Density must be > 0.0.

***Invalid dihedral type in Dihedrals section of data file***
Dihedral type must be positive integer and within range of specified dihedral types.

***Invalid displace_atoms rotate axis for 2d***
Axis must be in z direction.

***Invalid dump dcd filename***
Filenames used with the dump dcd style cannot be binary or compressed or cause multiple files to be written.

***Invalid dump frequency***
Dump frequency must be 1 or greater.

***Invalid dump image element name***
The specified element name was not in the standard list of elements. See the dump_modify doc page.

***Invalid dump image filename***
The file produced by dump image cannot be binary and must be for a single processor.

***Invalid dump image theta value***
Theta must be between 0.0 and 180.0 inclusive.

***Invalid dump image zoom value***
Zoom value must be > 0.0.

***Invalid dump movie filename***
The file produced by dump movie cannot be binary or compressed and must be a single file for a single processor.

***Invalid dump xtc filename***
Filenames used with the dump xtc style cannot be binary or compressed or cause multiple files to be written.

***Invalid dump xyz filename***
Filenames used with the dump xyz style cannot be binary or cause files to be written by each processor.

***Invalid dump_modify threshold operator***
Operator keyword used for threshold specification in not recognized.

***Invalid fix ID in variable formula***
The fix is not recognized.

***Invalid fix box/relax command for a 2d simulation***
Fix box/relax styles involving the z dimension cannot be used in a 2d simulation.

***Invalid fix box/relax command pressure settings***
If multiple dimensions are coupled, those dimensions must be specified.

***Invalid fix box/relax pressure settings***
Settings for coupled dimensions must be the same.

***Invalid fix nvt/npt/nph command for a 2d simulation***
Cannot control z dimension in a 2d model.

***Invalid fix nvt/npt/nph command pressure settings***
If multiple dimensions are coupled, those dimensions must be specified.

***Invalid fix nvt/npt/nph pressure settings***
Settings for coupled dimensions must be the same.

***Invalid fix press/berendsen for a 2d simulation***
The z component of pressure cannot be controlled for a 2d model.

***Invalid fix press/berendsen pressure settings***
Settings for coupled dimensions must be the same.

***Invalid fix qeq parameter file***
Element index > number of atom types.

***Invalid fix rigid npt/nph command for a 2d simulation***
Cannot control z dimension in a 2d model.

***Invalid fix rigid npt/nph command pressure settings***
If multiple dimensions are coupled, those dimensions must be specified.

***Invalid fix rigid/small npt/nph command for a 2d simulation***
Cannot control z dimension in a 2d model.

***Invalid fix rigid/small npt/nph command pressure settings***
If multiple dimensions are coupled, those dimensions must be specified.

***Invalid flag in force field section of restart file***
Unrecognized entry in restart file.

***Invalid flag in header section of restart file***
Unrecognized entry in restart file.

***Invalid flag in peratom section of restart file***
The format of this section of the file is not correct.

***Invalid flag in type arrays section of restart file***
Unrecognized entry in restart file.

***Invalid frequency in temper command***
Nevery must be > 0.

***Invalid group ID in neigh_modify command***
A group ID used in the neigh_modify command does not exist.

***Invalid group function in variable formula***
Group function is not recognized.

***Invalid image up vector***
Up vector cannot be (0,0,0).

***Invalid immediate variable***
Syntax of immediate value is incorrect.

***Invalid improper type in Impropers section of data file***
Improper type must be positive integer and within range of specified improper types.

***Invalid index for non-body particles in compute body/local command***
Only indices 1,2,3 can be used for non-body particles.

***Invalid keyword in pair table parameters***
Keyword used in list of table parameters is not recognized.

***Invalid option in lattice command for non-custom style***
Certain lattice keywords are not supported unless the lattice style is “custom”.

***Invalid order of forces within respa levels***
For respa, ordering of force computations within respa levels must obey certain rules. E.g. bonds cannot be compute less frequently than angles, pairwise forces cannot be computed less frequently than kspace, etc.

***Invalid pair table cutoff***
Cutoffs in pair_coeff command are not valid with read-in pair table.

***Invalid pair table length***
Length of read-in pair table is invalid

***Invalid param file for fix qeq/shielded***
Invalid value of gamma.

***Invalid param file for fix qeq/slater***
Zeta value is 0.0.

***Invalid partitions in processors part command***
Valid partitions are numbered 1 to N and the sender and receiver cannot be the same partition.

***Invalid python command***
Self-explanatory. Check the input script syntax and compare to the documentation for the command. You can use -echo screen as a command-line option when running LAMMPS to see the offending line.

***Invalid radius in Atoms section of data file***
Radius must be >= 0.0.

***Invalid random number seed in fix ttm command***
Random number seed must be > 0.

***Invalid random number seed in set command***
Random number seed must be > 0.

***Invalid rigid body ID in fix rigid file***
The ID does not match the number of an existing ID of rigid bodies that are defined by the fix rigid command.

***Invalid rigid body ID in fix rigid/small file***
The ID does not match the number of an existing ID of rigid bodies that are defined by the fix rigid/small command.

***Invalid run command N value***
The number of timesteps must fit in a 32-bit integer. If you want to run for more steps than this, perform multiple shorter runs.

***Invalid seed for Marsaglia random # generator***
The initial seed for this random number generator must be a positive integer less than or equal to 900 million.

***Invalid seed for Park random # generator***
The initial seed for this random number generator must be a positive integer.

***Invalid shape in Triangles section of data file***
Two or more of the triangle corners are duplicate points.

***Invalid t_event in tad command***
The value must be greater than 0.

***Invalid template atom in Atoms section of data file***
The atom indices must be between 1 to N, where N is the number of atoms in the template molecule the atom belongs to.

***Invalid template index in Atoms section of data file***
The template indices must be between 1 to N, where N is the number of molecules in the template.

***Invalid threads_per_atom specified.***
For 3-body potentials on the GPU, the threads_per_atom setting cannot be greater than 4 for NVIDIA GPUs.

***Invalid timestep reset for fix ave/atom***
Resetting the timestep has invalidated the sequence of timesteps this fix needs to process.

***Invalid timestep reset for fix ave/chunk***
Resetting the timestep has invalidated the sequence of timesteps this fix needs to process.

***Invalid timestep reset for fix ave/correlate***
Resetting the timestep has invalidated the sequence of timesteps this fix needs to process.

***Invalid timestep reset for fix ave/histo***
Resetting the timestep has invalidated the sequence of timesteps this fix needs to process.

***Invalid timestep reset for fix ave/spatial***
Resetting the timestep has invalidated the sequence of timesteps this fix needs to process.

***Invalid timestep reset for fix ave/time***
Resetting the timestep has invalidated the sequence of timesteps this fix needs to process.

***Invalid tmax in tad command***
The value must be greater than 0.0.

***Invalid type for mass set***
Mass command must set a type from 1-N where N is the number of atom types.

***Invalid label2type() function syntax in variable formula***
The first argument must be a label map kind (atom, bond, angle, dihedral, or improper) and the second argument must be a valid type label that has been assigned to a numeric type.

***Invalid use of library file() function***
This function is called through the library interface. This error should not occur. Contact the developers if it does.

***Invalid value in set command***
The value specified for the setting is invalid, likely because it is too small or too large.

***Invalid variable evaluation in variable formula***
A variable used in a formula could not be evaluated.

***Invalid variable name***
Variable name used in an input script line is invalid.

***Invalid variable name in variable formula***
Variable name is not recognized.

***Invalid variable style in special function next***
Only file-style or atomfile-style variables can be used with next().

***Invalid variable style with next command***
Variable styles *equal* and *world* cannot be used in a next command.

***Invalid volume in set command***
Volume must be > 0.0.

***Invoked pair single on pair style none***
A command (e.g. a dump) attempted to invoke the single() function on a pair style none, which is illegal. You are probably attempting to compute per-atom quantities with an undefined pair style.

***Invoking coulombic in pair style lj/coul requires atom attribute q***
The atom style defined does not have this attribute.

***Invoking coulombic in pair style lj/long/dipole/long requires atom attribute q***
The atom style defined does not have these attributes.

***KIM Simulator Model has no Model definition***
There is no model definition (key: model-defn) in the KIM Simulator Model. Please contact the OpenKIM database maintainers to verify and potentially correct this.

***KSpace accuracy must be > 0***
The kspace accuracy designated in the input must be greater than zero.

***KSpace accuracy too large to estimate G vector***
Reduce the accuracy request or specify gewald explicitly via the kspace_modify command.

***KSpace accuracy too low***
Requested accuracy must be less than 1.0.

***KSpace solver requires a pair style***
No pair style is defined.

***KSpace style does not yet support triclinic geometries***
The specified kspace style does not allow for non-orthogonal simulation boxes.

***KSpace style has not yet been set***
Cannot use kspace_modify command until a kspace style is set.

***KSpace style is incompatible with Pair style***
Setting a kspace style requires that a pair style with matching long-range Coulombic or dispersion components be used.

***Kokkos has been compiled for CUDA but no GPUs are requested***
One or more GPUs must be used when Kokkos is compiled for CUDA.

***Kspace_modify mesh parameter must be all zero or all positive***
Valid kspace mesh parameters are >0. The code will try to auto-detect suitable values when all three mesh sizes are set to zero (the default).

***Kspace_modify mesh/disp parameter must be all zero or all positive***
Valid kspace mesh/disp parameters are >0. The code will try to auto-detect suitable values when all three mesh sizes are set to zero **and** the required accuracy via *force/disp/real* as well as *force/disp/kspace* is set.

***Kspace style requires atom attribute q***
The atom style defined does not have these attributes.

***LAMMPS is not built with Python embedded***
This is done by including the PYTHON package before LAMMPS is built. This is required to use python-style variables.

***Label map is incomplete: all types must be assigned a unique type label***
For a given type-kind (atom types, bond types, etc.) to be written to the data file, all associated types must be assigned a type label, and each type label can be assigned to only one numeric type.

***Labelmap command before simulation box is defined***
The labelmap command cannot be used before a read_data, read_restart, or create_box command.

***Lattice orient vectors are not orthogonal***
The three specified lattice orientation vectors must be mutually orthogonal.

***Lattice orient vectors are not right-handed***
The three specified lattice orientation vectors must create a right-handed coordinate system such that a1 cross a2 = a3.

***Lattice primitive vectors are collinear***
The specified lattice primitive vectors do not for a unit cell with non-zero volume.

***Lattice settings are not compatible with 2d simulation***
One or more of the specified lattice vectors has a non-zero z component.

***Lattice spacings are invalid***
Each x,y,z spacing must be > 0.

***Lattice style incompatible with simulation dimension***
2d simulation can use sq, sq2, or hex lattice. 3d simulation can use sc, bcc, or fcc lattice.

***Lost atoms via balance: original %ld current %ld***
This should not occur. Report the problem to the developers.

***MEAM library error %d***
A call to the MEAM Fortran library returned an error.

***MPI_LMP_BIGINT and bigint in lmptype.h are not compatible***
The size of the MPI datatype does not match the size of a bigint.

***MPI_LMP_TAGINT and tagint in lmptype.h are not compatible***
The size of the MPI datatype does not match the size of a tagint.

***MSM can only currently be used with comm_style brick***
This is a current restriction in LAMMPS.

***MSM grid is too large***
The global MSM grid is larger than OFFSET in one or more dimensions. OFFSET is currently set to 16384. You likely need to decrease the requested accuracy.

***MSM order must be 4, 6, 8, or 10***
This is a limitation of the MSM implementation in LAMMPS: the MSM order can only be 4, 6, 8, or 10.

***Mass command before simulation box is defined***
The mass command cannot be used before a read_data, read_restart, or create_box command.

***Matrix factorization to split dispersion coefficients failed***
This should not normally happen. Contact the developers.

***Min_style command before simulation box is defined***
The min_style command cannot be used before a read_data, read_restart, or create_box command.

***Minimization could not find thermo_pe compute***
This compute is created by the thermo command. It must have been explicitly deleted by a uncompute command.

***Minimize command before simulation box is defined***
The minimize command cannot be used before a read_data, read_restart, or create_box command.

***Mismatched compute in variable formula***
A compute is referenced incorrectly or a compute that produces per-atom values is used in an equal-style variable formula.

***Mismatched fix in variable formula***
A fix is referenced incorrectly or a fix that produces per-atom values is used in an equal-style variable formula.

***Mismatched parameter in MEAM library file: z!=lat***
The coordination number and lattice do not match, check that consistent values are given.

***Mismatched variable in variable formula***
A variable is referenced incorrectly or an atom-style variable that produces per-atom values is used in an equal-style variable formula.

***Molecule IDs too large for compute chunk/atom***
The IDs must not be larger than can be stored in a 32-bit integer since chunk IDs are 32-bit integers.

***Molecule file shake flags not before shake atoms***
The order of the two sections is important.

***Molecule file shake flags not before shake bonds***
The order of the two sections is important.

***Molecule file shake info is incomplete***
All 3 SHAKE sections are needed.

***Molecule file special list does not match special count***
The number of values in an atom's special list does not match count.

***More than one fix deform***
Only one fix deform can be defined at a time.

***More than one fix freeze***
Only one of these fixes can be defined, since the granular pair potentials access it.

***More than one fix shake***
Only one fix shake can be defined.

***Must define angle_style before Angle Coeffs***
Must use an angle_style command before reading a data file that defines Angle Coeffs.

***Must define angle_style before BondAngle Coeffs***
Must use an angle_style command before reading a data file that defines Angle Coeffs.

***Must define angle_style before BondBond Coeffs***
Must use an angle_style command before reading a data file that defines Angle Coeffs.

***Must define bond_style before Bond Coeffs***
Must use a bond_style command before reading a data file that defines Bond Coeffs.

***Must define dihedral_style before AngleAngleTorsion Coeffs***
Must use a dihedral_style command before reading a data file that defines AngleAngleTorsion Coeffs.

***Must define dihedral_style before AngleTorsion Coeffs***
Must use a dihedral_style command before reading a data file that defines AngleTorsion Coeffs.

***Must define dihedral_style before BondBond13 Coeffs***
Must use a dihedral_style command before reading a data file that defines BondBond13 Coeffs.

***Must define dihedral_style before Dihedral Coeffs***
Must use a dihedral_style command before reading a data file that defines Dihedral Coeffs.

***Must define dihedral_style before EndBondTorsion Coeffs***
Must use a dihedral_style command before reading a data file that defines EndBondTorsion Coeffs.

***Must define dihedral_style before MiddleBondTorsion Coeffs***
Must use a dihedral_style command before reading a data file that defines MiddleBondTorsion Coeffs.

***Must define improper_style before AngleAngle Coeffs***
Must use an improper_style command before reading a data file that defines AngleAngle Coeffs.

***Must define improper_style before Improper Coeffs***
Must use an improper_style command before reading a data file that defines Improper Coeffs.

***Must define pair_style before Pair Coeffs***
Must use a pair_style command before reading a data file that defines Pair Coeffs.

***Must define pair_style before PairIJ Coeffs***
Must use a pair_style command before reading a data file that defines PairIJ Coeffs.

***Must have more than one processor partition to temper***
Cannot use the temper command with only one processor partition. Use the -partition command-line option.

***Must read Angle Type Labels before Angles***
An Angle Type Labels section of a data file must come before the Angles section.

***Must read Atom Type Labels before Atoms***
An Atom Type Labels section of a data file must come before the Atoms section.

***Must read Atoms before Angles***
The Atoms section of a data file must come before an Angles section.

***Must read Atoms before Bodies***
The Atoms section of a data file must come before a Bodies section.

***Must read Atoms before Bonds***
The Atoms section of a data file must come before a Bonds section.

***Must read Atoms before Dihedrals***
The Atoms section of a data file must come before a Dihedrals section.

***Must read Atoms before Ellipsoids***
The Atoms section of a data file must come before a Ellipsoids section.

***Must read Atoms before Impropers***
The Atoms section of a data file must come before an Impropers section.

***Must read Atoms before Lines***
The Atoms section of a data file must come before a Lines section.

***Must read Atoms before Triangles***
The Atoms section of a data file must come before a Triangles section.

***Must read Atoms before Velocities***
The Atoms section of a data file must come before a Velocities section.

***Must read Bond Type Labels before Bonds***
A Bond Type Labels section of a data file must come before the Bonds section.

***Must read Dihedral Type Labels before Dihedrals***
An Dihedral Type Labels section of a data file must come before the Dihedrals section.

***Must read Improper Type Labels before Impropers***
An Improper Type Labels section of a data file must come before the Impropers section.

***Must re-specify non-restarted pair style (xxx) after read_restart***
For pair styles, that do not store their settings in a restart file, it must be defined with a new 'pair_style' command after read_restart.

***Must set both respa inner and outer***
Cannot use just the inner or outer option with respa without using the other.

***Must set number of threads via package omp command***
Because you are using the OPENMP package, set the number of threads via its settings, not by the pair_style snap nthreads setting.

***Must shrink-wrap piston boundary***
The boundary style of the face where the piston is applied must be of type s (shrink-wrapped).

***Must specify a region in fix deposit***
The region keyword must be specified with this fix.

***Must use 'kspace_modify pressure/scalar no' for rRESPA with kspace_style MSM***
The kspace scalar pressure option cannot (yet) be used with rRESPA.

***Must use 'kspace_modify pressure/scalar no' for tensor components with kspace_style msm***
Otherwise MSM will compute only a scalar pressure. See the kspace_modify command for details on this setting.

***Must use 'kspace_modify pressure/scalar no' to obtain per-atom virial with kspace_style MSM***
The kspace scalar pressure option cannot be used to obtain per-atom virial.

***Must use 'kspace_modify pressure/scalar no' with GPU MSM Pair styles***
The kspace scalar pressure option is not (yet) compatible with GPU MSM Pair styles.

***Must use 'kspace_modify pressure/scalar no' with kspace_style msm/cg***
The kspace scalar pressure option is not compatible with kspace_style msm/cg.

***Must use -in switch with multiple partitions***
A multi-partition simulation cannot read the input script from stdin. The -in command-line option must be used to specify a file.

***Must use Kokkos half/thread or full neighbor list with threads or GPUs***
Using Kokkos half-neighbor lists with threading is not allowed.

***Must use a bond style with TIP4P potential***
TIP4P potentials assume bond lengths in water are constrained by a fix shake command.

***Must use an angle style with TIP4P potential***
TIP4P potentials assume angles in water are constrained by a fix shake command.

***Must use atom map style array with Kokkos***
See the atom_modify map command.

***Must use variable energy with fix addforce***
Must define an energy variable when applying a dynamic force during minimization.

***Must use variable energy with fix efield***
You must define an energy when performing a minimization with a variable E-field.

***NEB requires damped dynamics minimizer***
Use a different minimization style.

***NL ramp in wall/piston only implemented in zlo for now***
The ramp keyword can only be used for piston applied to face zlo.

***Needed bonus data not in data file***
Some atom styles require bonus data. See the read_data page for details.

***Needed molecular topology not in data file***
The header of the data file indicated bonds, angles, etc would be included, but they are not present.

***Neighbor delay must be 0 or multiple of every setting***
The delay and every parameters set via the neigh_modify command are inconsistent. If the delay setting is non-zero, then it must be a multiple of the every setting.

***Neighbor include group not allowed with ghost neighbors***
This is a current restriction within LAMMPS.

***Neighbor list overflow, boost neigh_modify one***
There are too many neighbors of a single atom. Use the neigh_modify command to increase the max number of neighbors allowed for one atom. You may also want to boost the page size.

***Neighbor multi not yet enabled for ghost neighbors***
This is a current restriction within LAMMPS.

***Neighbor page size must be >= 10x the one atom setting***
This is required to prevent wasting too much memory.

***New atom IDs exceed maximum allowed ID***
See the setting for tagint in the src/lmptype.h file.

***New bond exceeded bonds per atom in create_bonds***
See the read_data command for info on using the "extra/bond/per/atom" keyword to allow for additional bonds to be formed

***New bond exceeded bonds per atom in fix bond/create***
See the read_data command for info on using the "extra/bond/per/atom" keyword to allow for additional bonds to be formed

***New bond exceeded special list size in fix bond/create***
See the "read_data extra/special/per/atom" command (or the "create_box extra/special/per/atom" command) for info on how to leave space in the special bonds list to allow for additional bonds to be formed.

***Newton bond change after simulation box is defined***
The newton command cannot be used to change the newton bond value after a read_data, read_restart, or create_box command.

***Next command must list all universe and uloop variables***
This is to ensure they stay in sync.

***No OpenMP support compiled in***
An OpenMP flag is set, but LAMMPS was not built with OpenMP support.

***No atoms in data file***
The header of the data file indicated that atoms would be included, but they are not present.

***No basis atoms in lattice***
Basis atoms must be defined for lattice style user.

***No count or invalid atom count in molecule file***
The number of atoms must be specified.

***No dump custom arguments specified***
The dump custom command requires that atom quantities be specified to output to dump file.

***No fix gravity defined for fix pour***
Gravity is required to use fix pour.

***No matching element in ADP potential file***
The ADP potential file does not contain elements that match the requested elements.

***No matching element in EAM potential file***
The EAM potential file does not contain elements that match the requested elements.

***No molecule topology allowed with atom style template***
The data file cannot specify the number of bonds, angles, etc, because this info if inferred from the molecule templates.

***No pair coul/streitz for fix qeq/slater***
These commands must be used together.

***No pair style defined for compute group/group***
Cannot calculate group interactions without a pair style defined.

***No rigid bodies defined***
The fix specification did not end up defining any rigid bodies.

***Non integer # of swaps in temper command***
Swap frequency in temper command must evenly divide the total # of timesteps.

***Number of core atoms != number of shell atoms***
There must be a one-to-one pairing of core and shell atoms.

***One or more Atom IDs is negative***
Atom IDs must be positive integers.

***One or more atom IDs is too big***
The limit on atom IDs is set by the SMALLBIG, BIGBIG setting in your LAMMPS build. See the *Build settings* page for more info.

***One or more atom IDs is zero***
Either all atoms IDs must be zero or none of them.

***One or more atoms belong to multiple rigid bodies***
Two or more rigid bodies defined by the fix rigid command cannot contain the same atom.

***One or zero atoms in rigid body***
Any rigid body defined by the fix rigid command must contain 2 or more atoms.

***Overflow of allocated fix vector storage***
This should not normally happen if the fix correctly calculated how long the vector will grow to. Contact the developers.

***Overlapping large/large in pair colloid***
This potential is infinite when there is an overlap.

***Overlapping small/large in pair colloid***
This potential is infinite when there is an overlap.

***PPPM can only currently be used with comm_style brick***
This is a current restriction in LAMMPS.

***PPPM grid is too large***
The global PPPM grid is larger than OFFSET in one or more dimensions. OFFSET is currently set to 4096. You likely need to decrease the requested accuracy.

***PPPM grid stencil extends beyond nearest neighbor processor***
This is not allowed if the kspace_modify overlap setting is no.

***PPPM order < minimum allowed order***
The default minimum order is 2. This can be reset by the kspace_modify minorder command.

***PPPM order cannot be < 2 or > than %d***
This is a limitation of the PPPM implementation in LAMMPS.

***PPPMDisp Coulomb grid is too large***
The global PPPM grid is larger than OFFSET in one or more dimensions. OFFSET is currently set to 4096. You likely need to decrease the requested accuracy.

***PPPMDisp Dispersion grid is too large***
The global PPPM grid is larger than OFFSET in one or more dimensions. OFFSET is currently set to 4096. You likely need to decrease the requested accuracy.

***PPPMDisp can only currently be used with comm_style brick***
This is a current restriction in LAMMPS.

***PPPMDisp coulomb order cannot be greater than %d***
This is a limitation of the PPPM implementation in LAMMPS.

***PPPMDisp used but no parameters set, for further information please see the pppm/disp documentation***
An efficient and accurate usage of the pppm/disp requires settings via the kspace_modify command. Please see the pppm/disp documentation for further instructions.

***PRD command before simulation box is defined***
The prd command cannot be used before a read_data, read_restart, or create_box command.

***Package command after simulation box is defined***
The package command cannot be used after a read_data, read_restart, or create_box command.

***Package gpu command without GPU package installed***
The GPU package must be installed via "make yes-gpu" before LAMMPS is built.

***Package intel command without INTEL package installed***
The INTEL package must be installed via "make yes-intel" before LAMMPS is built.

***Package kokkos command without KOKKOS package enabled***
The KOKKOS package must be installed via "make yes-kokkos" before LAMMPS is built, and the "-k on" must be used to enable the package.

***Package omp command without OPENMP package installed***
The OPENMP package must be installed via "make yes-openmp" before LAMMPS is built.

***Pair body requires body style nparticle***
This pair style is specific to the nparticle body style.

***Pair brownian requires extended particles***
One of the particles has radius 0.0.

***Pair brownian requires monodisperse particles***
All particles must be the same finite size.

***Pair brownian/poly requires extended particles***
One of the particles has radius 0.0.

***Pair coeff for hybrid has invalid style***
Style in pair coeff must have been listed in pair_style command.

***Pair coul/wolf requires atom attribute q***
The atom style defined does not have this attribute.

***Pair cutoff < Respa interior cutoff***
One or more pairwise cutoffs are too short to use with the specified rRESPA cutoffs.

***Pair dipole/cut requires atom attributes q, mu, torque***
The atom style defined does not have these attributes.

***Pair dipole/cut/gpu requires atom attributes q, mu, torque***
The atom style defined does not have this attribute.

***Pair dipole/long requires atom attributes q, mu, torque***
The atom style defined does not have these attributes.

***Pair dipole/sf/gpu requires atom attributes q, mu, torque***
The atom style defined does not one or more of these attributes.

***Pair distance < table inner cutoff***
Two atoms are closer together than the pairwise table allows.

***Pair distance > table outer cutoff***
Two atoms are further apart than the pairwise table allows.

***Pair dpd requires ghost atoms store velocity***
Use the comm_modify vel yes command to enable this.

***Pair gayberne epsilon a,b,c coeffs are not all set***
Each atom type involved in pair_style gayberne must have these 3 coefficients set at least once.

***Pair granular requires atom attributes radius, rmass***
The atom style defined does not have these attributes.

***Pair granular requires ghost atoms store velocity***
Use the comm_modify vel yes command to enable this.

***Pair granular with shear history requires newton pair off***
This is a current restriction of the implementation of pair granular styles with history.

***Pair hybrid sub-style does not support single call***
You are attempting to invoke a single() call on a pair style that does not support it.

***Pair hybrid sub-style is not used***
No pair_coeff command used a sub-style specified in the pair_style command.

***Pair inner cutoff < Respa interior cutoff***
One or more pairwise cutoffs are too short to use with the specified rRESPA cutoffs.

***Pair inner cutoff >= Pair outer cutoff***
The specified cutoffs for the pair style are inconsistent.

***Pair lj/long/dipole/long requires atom attributes mu, torque***
The atom style defined does not have these attributes.

***Pair lubricate requires ghost atoms store velocity***
Use the comm_modify vel yes command to enable this.

***Pair lubricate requires monodisperse particles***
All particles must be the same finite size.

***Pair lubricate/poly requires extended particles***
One of the particles has radius 0.0.

***Pair lubricate/poly requires ghost atoms store velocity***
Use the comm_modify vel yes command to enable this.

***Pair lubricateU requires ghost atoms store velocity***
Use the comm_modify vel yes command to enable this.

***Pair lubricateU requires monodisperse particles***
All particles must be the same finite size.

***Pair lubricateU/poly requires ghost atoms store velocity***
Use the comm_modify vel yes command to enable this.

***Pair peri lattice is not identical in x, y, and z***
The lattice defined by the lattice command must be cubic.

***Pair peri requires a lattice be defined***
Use the lattice command for this purpose.

***Pair peri requires an atom map, see atom_modify***
Even for atomic systems, an atom map is required to find Peridynamic bonds. Use the atom_modify command to define one.

***Pair style AIREBO requires atom IDs***
This is a requirement to use the AIREBO potential.

***Pair style AIREBO requires newton pair on***
See the newton command. This is a restriction to use the AIREBO potential.

***Pair style BOP requires atom IDs***
This is a requirement to use the BOP potential.

***Pair style BOP requires newton pair on***
See the newton command. This is a restriction to use the BOP potential.

***Pair style COMB requires atom IDs***
This is a requirement to use the AIREBO potential.

***Pair style COMB requires newton pair on***
See the newton command. This is a restriction to use the COMB potential.

***Pair style COMB3 requires atom IDs***
This is a requirement to use the COMB3 potential.

***Pair style COMB3 requires newton pair on***
See the newton command. This is a restriction to use the COMB3 potential.

***Pair style LCBOP requires atom IDs***
This is a requirement to use the LCBOP potential.

***Pair style LCBOP requires newton pair on***
See the newton command. This is a restriction to use the Tersoff potential.

***Pair style MEAM requires newton pair on***
See the newton command. This is a restriction to use the MEAM potential.

***Pair style SNAP requires newton pair on***
See the newton command. This is a restriction to use the SNAP potential.

***Pair style Stillinger-Weber requires atom IDs***
This is a requirement to use the SW potential.

***Pair style Stillinger-Weber requires newton pair on***
See the newton command. This is a restriction to use the SW potential.

***Pair style Tersoff requires atom IDs***
This is a requirement to use the Tersoff potential.

***Pair style Tersoff requires newton pair on***
See the newton command. This is a restriction to use the Tersoff potential.

***Pair style Vashishta requires atom IDs***
This is a requirement to use the Vashishta potential.

***Pair style Vashishta requires newton pair on***
See the newton command. This is a restriction to use the Vashishta potential.

***Pair style bop requires comm ghost cutoff at least 3x larger than %g***
Use the communicate ghost command to set this. See the pair bop page for more details.

***Pair style born/coul/long requires atom attribute q***
An atom style that defines this attribute must be used.

***Pair style born/coul/long/gpu requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style born/coul/wolf requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style buck/coul/cut requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style buck/coul/long requires atom attribute q***
The atom style defined does not have these attributes.

***Pair style buck/coul/long/gpu requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style buck/long/coul/long requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style coul/cut requires atom attribute q***
The atom style defined does not have these attributes.

***Pair style coul/cut/gpu requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style coul/debye/gpu requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style coul/dsf requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style coul/dsf/gpu requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style coul/long/gpu requires atom attribute q***
The atom style defined does not have these attributes.

***Pair style does not have extra field requested by compute pair/local***
The pair style does not support the pN value requested by the compute pair/local command.

***Pair style does not support bond_style quartic***
The pair style does not have a single() function, so it can not be invoked by bond_style quartic.

***Pair style does not support compute group/group***
The pair_style does not have a single() function, so it cannot be invoked by the compute group/group command.

***Pair style does not support compute pair/local***
The pair style does not have a single() function, so it can not be invoked by compute pair/local.

***Pair style does not support compute property/local***
The pair style does not have a single() function, so it can not be invoked by fix bond/swap.

***Pair style does not support fix bond/swap***
The pair style does not have a single() function, so it can not be invoked by fix bond/swap.

***Pair style does not support pair_write***
The pair style does not have a single() function, so it can not be invoked by pair write.

***Pair style does not support rRESPA inner/middle/outer***
You are attempting to use rRESPA options with a pair style that does not support them.

***Pair style granular with history requires atoms have IDs***
Atoms in the simulation do not have IDs, so history effects cannot be tracked by the granular pair potential.

***Pair style hbond/dreiding requires newton pair on***
See the newton command for details.

***Pair style is incompatible with KSpace style***
If a pair style with a long-range Coulombic component is selected, then a kspace style must also be used.

***Pair style is incompatible with TIP4P KSpace style***
The pair style does not have the requires TIP4P settings.

***Pair style lj/charmm/coul/charmm requires atom attribute q***
The atom style defined does not have these attributes.

***Pair style lj/charmm/coul/long requires atom attribute q***
The atom style defined does not have these attributes.

***Pair style lj/charmm/coul/long/gpu requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style lj/class2/coul/cut requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style lj/class2/coul/long requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style lj/class2/coul/long/gpu requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style lj/cut/coul/cut requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style lj/cut/coul/cut/gpu requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style lj/cut/coul/debye/gpu requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style lj/cut/coul/dsf requires atom attribute q***
The atom style defined does not have these attributes.

***Pair style lj/cut/coul/dsf/gpu requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style lj/cut/coul/long requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style lj/cut/coul/long/gpu requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style lj/cut/tip4p/cut requires atom IDs***
This is a requirement to use this potential.

***Pair style lj/cut/tip4p/cut requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style lj/cut/tip4p/cut requires newton pair on***
See the newton command. This is a restriction to use this potential.

***Pair style lj/cut/tip4p/long requires atom IDs***
There are no atom IDs defined in the system and the TIP4P potential requires them to find O,H atoms with a water molecule.

***Pair style lj/cut/tip4p/long requires atom attribute q***
The atom style defined does not have these attributes.

***Pair style lj/cut/tip4p/long requires newton pair on***
This is because the computation of constraint forces within a water molecule adds forces to atoms owned by other processors.

***Pair style lj/gromacs/coul/gromacs requires atom attribute q***
An atom_style with this attribute is needed.

***Pair style lj/long/dipole/long does not currently support respa***
This feature is not yet supported.

***Pair style lj/long/tip4p/long requires atom IDs***
There are no atom IDs defined in the system and the TIP4P potential requires them to find O,H atoms with a water molecule.

***Pair style lj/long/tip4p/long requires atom attribute q***
The atom style defined does not have these attributes.

***Pair style lj/long/tip4p/long requires newton pair on***
This is because the computation of constraint forces within a water molecule adds forces to atoms owned by other processors.

***Pair style lj/spica/coul/long/gpu requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style nb3b/harmonic requires atom IDs***
This is a requirement to use this potential.

***Pair style nb3b/harmonic requires newton pair on***
See the newton command. This is a restriction to use this potential.

***Pair style nm/cut/coul/cut requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style nm/cut/coul/long requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style polymorphic requires atom IDs***
This is a requirement to use the polymorphic potential.

***Pair style polymorphic requires newton pair on***
See the newton command. This is a restriction to use the polymorphic potential.

***Pair style reax requires atom IDs***
This is a requirement to use the ReaxFF potential.

***Pair style reax requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style reax requires newton pair on***
This is a requirement to use the ReaxFF potential.

***Pair style requires a KSpace style***
No kspace style is defined.

***Pair style sw/gpu requires atom IDs***
This is a requirement to use this potential.

***Pair style sw/gpu requires newton pair off***
See the newton command. This is a restriction to use this potential.

***Pair style vashishta/gpu requires atom IDs***
This is a requirement to use this potential.

***Pair style vashishta/gpu requires newton pair off***
See the newton command. This is a restriction to use this potential.

***Pair style tersoff/gpu requires atom IDs***
This is a requirement to use the tersoff/gpu potential.

***Pair style tersoff/gpu requires newton pair off***
See the newton command. This is a restriction to use this pair style.

***Pair style tip4p/cut requires atom IDs***
This is a requirement to use this potential.

***Pair style tip4p/cut requires atom attribute q***
The atom style defined does not have this attribute.

***Pair style tip4p/cut requires newton pair on***
See the newton command. This is a restriction to use this potential.

***Pair style tip4p/long requires atom IDs***
There are no atom IDs defined in the system and the TIP4P potential requires them to find O,H atoms with a water molecule.

***Pair style tip4p/long requires atom attribute q***
The atom style defined does not have these attributes.

***Pair style tip4p/long requires newton pair on***
This is because the computation of constraint forces within a water molecule adds forces to atoms owned by other processors.

***Pair table cutoffs must all be equal to use with KSpace***
When using pair style table with a long-range KSpace solver, the cutoffs for all atom type pairs must all be the same, since the long-range solver starts at that cutoff.

***Pair table parameters did not set N***
List of pair table parameters must include N setting.

***Pair tersoff/zbl requires metal or real units***
This is a current restriction of this pair potential.

***Pair tersoff/zbl/kk requires metal or real units***
This is a current restriction of this pair potential.

***PairKIM only works with 3D problems***
This is a current limitation.

***Pair_coeff command before simulation box is defined***
The pair_coeff command cannot be used before a read_data, read_restart, or create_box command.

***Pair_modify special setting for pair hybrid incompatible with global special_bonds setting***
Cannot override a setting of 0.0 or 1.0 or change a setting between 0.0 and 1.0.

***Particle on or inside fix wall surface***
Particles must be "exterior" to the wall in order for energy/force to be calculated.

***Particle outside surface of region used in fix wall/region***
Particles must be inside the region for energy/force to be calculated. A particle outside the region generates an error.

***Per-atom compute in equal-style variable formula***
Equal-style variables cannot use per-atom quantities.

***Per-atom fix in equal-style variable formula***
Equal-style variables cannot use per-atom quantities.

***Per-processor system is too big***
The number of owned atoms plus ghost atoms on a single processor must fit in 32-bit integer.

***Potential energy ID for fix nvt/nph/npt does not exist***
A compute for potential energy must be defined.

***Potential file has duplicate entry***
The potential file has more than one entry for the same element.

***Potential file is missing an entry***
The potential file does not have a needed entry.

***Pressure ID for fix box/relax does not exist***
The compute ID needed to compute pressure for the fix does not exist.

***Pressure ID for fix press/berendsen does not exist***
The compute ID needed to compute pressure for the fix does not exist.

***Pressure ID for thermo does not exist***
The compute ID needed to compute pressure for thermodynamics does not exist.

***Pressure control must be used with fix nphug***
A pressure control keyword (iso, aniso, tri, x, y, or z) must be provided.

***Processor partitions do not match number of allocated processors***
The total number of processors in all partitions must match the number of processors LAMMPS is running on.

***Processors command after simulation box is defined***
The processors command cannot be used after a read_data, read_restart, or create_box command.

***Processors custom grid file is inconsistent***
The vales in the custom file are not consistent with the number of processors you are running on or the Px,Py,Pz settings of the processors command. Or there was not a setting for every processor.

***Processors grid numa and map style are incompatible***
Using numa for gstyle in the processors command requires using cart for the map option.

***Processors part option and grid style are incompatible***
Cannot use gstyle numa or custom with the part option.

***Python function evaluation failed***
The Python function did not run successfully and/or did not return a value (if it is supposed to return a value). This is probably due to some error condition in the function.

***Python function is not callable***
The provided Python code was run successfully, but it not define a callable function with the required name.

***Python invoke of undefined function***
Cannot invoke a function that has not been previously defined.

***Python variable does not match Python function***
This matching is defined by the python-style variable and the python command.

***Python variable has no function***
No python command was used to define the function associated with the python-style variable.

***QEQ with 'newton pair off' not supported***
See the newton command. This is a restriction to use the QEQ fixes.

***R0 < 0 for fix spring command***
Equilibrium spring length is invalid.

***RATTLE determinant = 0.0***
The determinant of the matrix being solved for a single cluster specified by the fix rattle command is numerically invalid.

***RATTLE failed***
Certain constraints were not satisfied.

***Read data add offset is too big***
It cannot be larger than the size of atom IDs, e.g. the maximum 32-bit integer.

***Read_data shrink wrap did not assign all atoms correctly***
This is typically because the box-size specified in the data file is large compared to the actual extent of atoms in

a shrink-wrapped dimension. When LAMMPS shrink-wraps the box atoms will be lost if the processor they are re-assigned to is too far away. Choose a box size closer to the actual extent of the atoms.

***Read_dump command before simulation box is defined***
The read_dump command cannot be used before a read_data, read_restart, or create_box command.

***Read_dump triclinic status does not match simulation***
Both the dump snapshot and the current LAMMPS simulation must be using either an orthogonal or triclinic box.

***Reax_defs.h setting for NATDEF is too small***
Edit the setting in the ReaxFF library and re-compile the library and re-build LAMMPS.

***Reax_defs.h setting for NNEIGHMAXDEF is too small***
Edit the setting in the ReaxFF library and re-compile the library and re-build LAMMPS.

***Receiving partition in processors part command is already a receiver***
Cannot specify a partition to be a receiver twice.

***Region union or intersect cannot be dynamic***
The sub-regions can be dynamic, but not the combined region.

***Region union region ID does not exist***
One or more of the region IDs specified by the region union command does not exist.

***Replacing a fix, but new style != old style***
A fix ID can be used a second time, but only if the style matches the previous fix. In this case it is assumed you want to reset a fix's parameters. This error may mean you are mistakenly re-using a fix ID when you do not intend to.

***Replicate command before simulation box is defined***
The replicate command cannot be used before a read_data, read_restart, or create_box command.

***Replicate did not assign all atoms correctly***
Atoms replicated by the replicate command were not assigned correctly to processors. This is likely due to some atom coordinates being outside a non-periodic simulation box.

***Replicated system atom IDs are too big***
See the setting for tagint in the src/lmptype.h file.

***Replicated system is too big***
See the setting for bigint in the src/lmptype.h file.

***Required border comm not yet implemented with Kokkos***
There are various limitations in the communication options supported by Kokkos.

***Rerun command before simulation box is defined***
The rerun command cannot be used before a read_data, read_restart, or create_box command.

***Resetting timestep size is not allowed with fix move***
This is because fix move is moving atoms based on elapsed time.

***Respa inner cutoffs are invalid***
The first cutoff must be <= the second cutoff.

***Respa middle cutoffs are invalid***
The first cutoff must be <= the second cutoff.

***Restart file byte ordering is not recognized***
The file does not appear to be a LAMMPS restart file since it does not contain a recognized byte-ordering flag at the beginning.

***Restart file byte ordering is swapped***
The file was written on a machine with different byte-ordering than the machine you are reading it on. Convert it to a text data file instead, on the machine you wrote it on.

***Restart file incompatible with current version***
This is probably because you are trying to read a file created with a version of LAMMPS that is too old compared to the current version. Use your older version of LAMMPS and convert the restart file to a data file.

***Restart file is a multi-proc file***
The file is inconsistent with the filename you specified for it.

***Restart file is not a multi-proc file***
The file is inconsistent with the filename you specified for it.

***Restart variable returned a bad timestep***
The variable must return a timestep greater than the current timestep.

***Restrain atoms %d %d %d %d missing on proc %d at step %ld***
The 4 atoms in a restrain dihedral specified by the fix restrain command are not all accessible to a processor. This probably means an atom has moved too far.

***Restrain atoms %d %d %d missing on proc %d at step %ld***
The three atoms in a restrain angle specified by the fix restrain command are not all accessible to a processor. This probably means an atom has moved too far.

***Restrain atoms %d %d missing on proc %d at step %ld***
The two atoms in a restrain bond specified by the fix restrain command are not all accessible to a processor. This probably means an atom has moved too far.

***Reuse of compute ID***
A compute ID cannot be used twice.

***Reuse of dump ID***
A dump ID cannot be used twice.

***Reuse of molecule template ID***
The template IDs must be unique.

***Reuse of region ID***
A region ID cannot be used twice.

***Rigid body atoms %d %d missing on proc %d at step %ld***
This means that an atom cannot find the atom that owns the rigid body it is part of, or vice versa. The solution is to use the communicate cutoff command to ensure ghost atoms are acquired from far enough away to encompass the max distance printed when the fix rigid/small command was invoked.

***Rigid fix must come before NPT/NPH fix***
NPT/NPH fix must be defined in input script after all rigid fixes, else the rigid fix contribution to the pressure virial is incorrect.

***Rmask function in equal-style variable formula***
Rmask is per-atom operation.

***Run command before simulation box is defined***
The run command cannot be used before a read_data, read_restart, or create_box command.

***Run_style command before simulation box is defined***
The run_style command cannot be used before a read_data, read_restart, or create_box command.

***SRD bin size for fix srd differs from user request***
Fix SRD had to adjust the bin size to fit the simulation box. See the cubic keyword if you want this message to be an error vs warning.

***SRD bins for fix srd are not cubic enough***
The bin shape is not within tolerance of cubic. See the cubic keyword if you want this message to be an error vs warning.

***SRD particle %d started inside big particle %d on step %ld bounce %d***
See the inside keyword if you want this message to be an error vs warning.

***SRD particle %d started inside wall %d on step %ld bounce %d***
See the inside keyword if you want this message to be an error vs warning.

***Sending partition in processors part command is already a sender***
Cannot specify a partition to be a sender twice.

***Set command before simulation box is defined***
The set command cannot be used before a read_data, read_restart, or create_box command.

***Set command with no atoms existing***
No atoms are yet defined so the set command cannot be used.

***Set region ID does not exist***
Region ID specified in set command does not exist.

***Shake angles have different bond types***
All 3-atom angle-constrained SHAKE clusters specified by the fix shake command that are the same angle type, must also have the same bond types for the two bonds in the angle.

***Shake atoms %d %d %d %d missing on proc %d at step %ld***
The 4 atoms in a single shake cluster specified by the fix shake command are not all accessible to a processor. This probably means an atom has moved too far.

***Shake atoms %d %d %d missing on proc %d at step %ld***
The three atoms in a single shake cluster specified by the fix shake command are not all accessible to a processor. This probably means an atom has moved too far.

***Shake atoms %d %d missing on proc %d at step %ld***
The two atoms in a single shake cluster specified by the fix shake command are not all accessible to a processor. This probably means an atom has moved too far.

***Shake cluster of more than 4 atoms***
A single cluster specified by the fix shake command can have no more than 4 atoms.

***Shake clusters are connected***
A single cluster specified by the fix shake command must have a single central atom with up to 3 other atoms bonded to it.

***Shake determinant = 0.0***
The determinant of the matrix being solved for a single cluster specified by the fix shake command is numerically invalid.

***Shake fix must come before NPT/NPH fix***
NPT fix must be defined in input script after SHAKE fix, else the SHAKE fix contribution to the pressure virial is incorrect.

***Shear history overflow, boost neigh_modify one***
There are too many neighbors of a single atom. Use the neigh_modify command to increase the max number of neighbors allowed for one atom. You may also want to boost the page size.

***Small to big integers are not sized correctly***
This error occurs when the sizes of smallint, imageint, tagint, bigint, as defined in src/lmptype.h are not what is expected. Contact the developers if this occurs.

***Smallint setting in lmptype.h is invalid***
It has to be the size of an integer.

***Smallint setting in lmptype.h is not compatible***
Smallint stored in restart file is not consistent with LAMMPS version you are running.

***Special list size exceeded in fix bond/create***
See the "read_data extra/special/per/atom" command (or the "create_box extra/special/per/atom" command) for info on how to leave space in the special bonds list to allow for additional bonds to be formed.

***Species XXX is not supported by this KIM Simulator Model***
The kim_style define command was referencing a species that is not present in the requested KIM Simulator Model.

***Specified processors != physical processors***
The 3d grid of processors defined by the processors command does not match the number of processors LAMMPS is being run on.

***Subsequent read data induced too many angles per atom***
See the extra/angle/per/atom keyword for the create_box or the read_data command to set this limit larger

***Subsequent read data induced too many bonds per atom***
See the extra/bond/per/atom keyword for the create_box or the read_data command to set this limit larger

***Subsequent read data induced too many dihedrals per atom***
See the extra/dihedral/per/atom keyword for the create_box or the read_data command to set this limit larger

***Subsequent read data induced too many impropers per atom***
See the extra/improper/per/atom keyword for the create_box or the read_data command to set this limit larger

***Support for writing images in JPEG format not included***
LAMMPS was not built with the -DLAMMPS_JPEG switch in the Makefile.

***Support for writing images in PNG format not included***
LAMMPS was not built with the -DLAMMPS_PNG switch in the Makefile.

***Support for writing movies not included***
LAMMPS was not built with the -DLAMMPS_FFMPEG switch in the Makefile

***System in data file is too big***
See the setting for bigint in the src/lmptype.h file.

***System is not charge neutral, net charge = %g***
The total charge on all atoms on the system is not 0.0. For some KSpace solvers this is an error.

***TIP4P hydrogen has incorrect atom type***
The TIP4P pairwise computation found an H atom whose type does not agree with the specified H type.

***TIP4P hydrogen is missing***
The TIP4P pairwise computation failed to find the correct H atom within a water molecule.

***TMD target file did not list all group atoms***
The target file for the fix tmd command did not list all atoms in the fix group.

***Tagint setting in lmptype.h is invalid***
Tagint must be as large or larger than smallint.

***Tagint setting in lmptype.h is not compatible***
Format of tagint stored in restart file is not consistent with LAMMPS version you are running. See the settings in src/lmptype.h

***Temper command before simulation box is defined***
The temper command cannot be used before a read_data, read_restart, or create_box command.

***Temperature compute degrees of freedom < 0***
This should not happen if you are calculating the temperature on a valid set of atoms.

***Temperature control must be used with fix nphug***
The temp keyword must be provided.

***Temperature for fix nvt/sllod does not have a bias***
The specified compute must compute temperature with a bias.

***Tempering could not find thermo_pe compute***
This compute is created by the thermo command. It must have been explicitly deleted by a uncompute command.

***Tempering fix ID is not defined***
The fix ID specified by the temper command does not exist.

***Tempering temperature fix is not valid***
The fix specified by the temper command is not one that controls temperature (nvt or langevin).

***Test_descriptor_string already allocated***
This is an internal error. Contact the developers.

***Thermo and fix not computed at compatible times***
Fixes generate values on specific timesteps. The thermo output does not match these timesteps.

***Thermo custom variable is not equal-style variable***
Only equal-style variables can be output with thermodynamics, not atom-style variables.

***Thermo every variable returned a bad timestep***
The variable must return a timestep greater than the current timestep.

***Thermo keyword in variable requires thermo to use/init pe***
You are using a thermo keyword in a variable that requires potential energy to be calculated, but your thermo output does not use it. Add it to your thermo output.

***Thermo keyword in variable requires thermo to use/init press***
You are using a thermo keyword in a variable that requires pressure to be calculated, but your thermo output does not use it. Add it to your thermo output.

***Thermo keyword in variable requires thermo to use/init temp***
You are using a thermo keyword in a variable that requires temperature to be calculated, but your thermo output does not use it. Add it to your thermo output.

***Thermo style does not use press***
Cannot use thermo_modify to set this parameter since the thermo_style is not computing this quantity.

***Thermo style does not use temp***
Cannot use thermo_modify to set this parameter since the thermo_style is not computing this quantity.

***Thermo_modify every variable returned a bad timestep***
The returned timestep is less than or equal to the current timestep.

***Thermo_modify pressure ID does not compute pressure***
The specified compute ID does not compute pressure.

***Thermo_modify temperature ID does not compute temperature***
The specified compute ID does not compute temperature.

***Thermo_style command before simulation box is defined***
The thermo_style command cannot be used before a read_data, read_restart, or create_box command.

***This variable thermo keyword cannot be used between runs***
Keywords that refer to time (such as cpu, elapsed) do not make sense in between runs.

***Threshold for an atom property that is not allocated***
A dump threshold has been requested on a quantity that is not defined by the atom style used in this simulation.

***Timestep must be >= 0***
Specified timestep is invalid.

***Too big a problem to use velocity create loop all***
The system size must fit in a 32-bit integer to use this option.

***Too big a timestep for dump dcd***
The timestep must fit in a 32-bit integer to use this dump style.

***Too big a timestep for dump xtc***
The timestep must fit in a 32-bit integer to use this dump style.

***Too few bits for lookup table***
Table size specified via pair_modify command does not work with your machine's floating point representation.

***Too many -pk arguments in command-line***
The string formed by concatenating the arguments is too long. Use a package command in the input script instead.

***Too many MSM grid levels***
The max number of MSM grid levels is hardwired to 10.

***Too many args in variable function***
More args are used than any variable function allows.

***Too many atom pairs for pair bop***
The number of atomic pairs exceeds the expected number. Check your atomic structure to ensure that it is realistic.

***Too many atom sorting bins***
This is likely due to an immense simulation box that has blown up to a large size.

***Too many atom triplets for pair bop***
The number of three atom groups for angle determinations exceeds the expected number. Check your atomic structure to ensure that it is realistic.

***Too many atoms for dump dcd***
The system size must fit in a 32-bit integer to use this dump style.

***Too many atoms for dump xtc***
The system size must fit in a 32-bit integer to use this dump style.

***Too many elements extracted from MEAM library.***
Increase 'maxelt' in meam.h and recompile.

***Too many exponent bits for lookup table***
Table size specified via pair_modify command does not work with your machine's floating point representation.

***Too many groups***
The maximum number of atom groups (including the "all" group) is given by MAX_GROUP in group.cpp and is 32.

***Too many iterations***
You must use a number of iterations that fit in a 32-bit integer for minimization.

***Too many lines in one body in data file - boost MAXBODY***
MAXBODY is a setting at the top of the src/read_data.cpp file. Set it larger and re-compile the code.

***Too many local+ghost atoms for neighbor list***
The number of nlocal + nghost atoms on a processor is limited by the size of a 32-bit integer with 2 bits removed for masking 1-2, 1-3, 1-4 neighbors.

***Too many mantissa bits for lookup table***
Table size specified via pair_modify command does not work with your machine's floating point representation.

***Too many masses for fix shake***
The fix shake command cannot list more masses than there are atom types.

***Too many molecules for fix rigid***
The limit is 2^31 = ~2 billion molecules.

***Too many timesteps***
The cumulative timesteps must fit in a 64-bit integer.

***Too many timesteps for NEB***
You must use a number of timesteps that fit in a 32-bit integer for NEB.

***Too many total atoms***
See the setting for bigint in the src/lmptype.h file.

***Too many total bits for bitmapped lookup table***
Table size specified via pair_modify command is too large. Note that a value of N generates a 2^N size table.

***Too much buffered per-proc info for dump***
The size of the buffered string must fit in a 32-bit integer for a dump.

***Too much per-proc info for dump***
Number of local atoms times number of columns must fit in a 32-bit integer for dump.

***Topology type exceeds system topology type***
The number of bond, angle, etc types exceeds the system setting. See the create_box or read_data command for how to specify these values.

***Trying to build an occasional neighbor list before initialization completed***
This is not allowed. Source code caller needs to be modified.

***Two fix ave commands using same compute chunk/atom command in incompatible ways***
They are both attempting to "lock" the chunk/atom command so that the chunk assignments persist for some number of timesteps, but are doing it in different ways.

***The %s type label %s is already in use for type %s***
For a given type-kind (atom types, bond types, etc.), a given type label can be assigned to only one numeric type.

***Type label string %s for %s type %s is invalid***
See the labelmap command documentation for valid type labels.

***Unable to initialize accelerator for use***
There was a problem initializing an accelerator for the gpu package

***Unbalanced quotes in input line***
No matching end double quote was found following a leading double quote.

***Unexpected empty line in Angle Coeffs section***
Read a blank line where there should be coefficient data.

***Unexpected empty line in Bond Coeffs section***
Read a blank line where there should be coefficient data.

***Unexpected empty line in Dihedral Coeffs section***
Read a blank line where there should be coefficient data.

***Unexpected empty line in Improper Coeffs section***
Read a blank line where there should be coefficient data.

***Unexpected empty line in Pair Coeffs section***
Read a blank line where there should be coefficient data.

***Unexpected end of data file***
LAMMPS hit the end of the data file while attempting to read a section. Something is wrong with the format of the data file.

***Unexpected end of dump file***
A read operation from the file failed.

***Unexpected end of fix rigid file***
A read operation from the file failed.

***Unexpected end of fix rigid/small file***
A read operation from the file failed.

***Unexpected end of neb file***
A read operation from the file failed.

***Units command after simulation box is defined***
The units command cannot be used after a read_data, read_restart, or create_box command.

***Universe/uloop variable count < # of partitions***
A universe or uloop style variable must specify a number of values >= to the number of processor partitions.

***Unknown keyword in thermo_style custom command***
One or more specified keywords are not recognized.

***Unknown pair_modify hybrid sub-style***
The choice of sub-style is unknown.

***Unknown table style in pair_style command***
Style of table is invalid for use with pair_style table command.

***Unrecognized lattice type in MEAM library file***
The lattice type in an entry of the MEAM library file is not valid.

***Unrecognized lattice type in MEAM parameter file***
The lattice type in an entry of the MEAM parameter file is not valid.

***Unsupported mixing rule in kspace_style ewald/disp***
Only geometric mixing is supported.

***Unsupported order in kspace_style ewald/disp***
Only 1/r^6 dispersion or dipole terms are supported.

***Unsupported order in kspace_style pppm/disp, pair_style %s***
Only pair styles with 1/r and 1/r^6 dependence are currently supported.

***Use cutoff keyword to set cutoff in single mode***
Mode is single so cutoff/multi keyword cannot be used.

***Use cutoff/multi keyword to set cutoff in multi mode***
Mode is multi so cutoff keyword cannot be used.

***Using fix nvt/sllod with inconsistent fix deform remap option***
Fix nvt/sllod requires that deforming atoms have a velocity profile provided by "remap v" as a fix deform option.

***Using fix srd with inconsistent fix deform remap option***
When shearing the box in an SRD simulation, the remap v option for fix deform needs to be used.

***Using pair lubricate with inconsistent fix deform remap option***
Must use remap v option with fix deform with this pair style.

***Using pair lubricate/poly with inconsistent fix deform remap option***
If fix deform is used, the remap v option is required.

***Variable atom ID is too large***
Specified ID is larger than the maximum allowed atom ID.

***Variable evaluation before simulation box is defined***
Cannot evaluate a compute or fix or atom-based value in a variable before the simulation has been setup.

***Variable evaluation in fix wall gave bad value***
The returned value for epsilon or sigma < 0.0.

***Variable evaluation in region gave bad value***
Variable returned a radius < 0.0.

***Variable for create_atoms is invalid style***
The variables must be equal-style variables.

***Variable for displace_atoms is invalid style***
It must be an equal-style or atom-style variable.

***Variable for dump every is invalid style***
Only equal-style variables can be used.

***Variable for dump image center is invalid style***
Must be an equal-style variable.

***Variable for dump image phi is invalid style***
Must be an equal-style variable.

***Variable for dump image theta is invalid style***
Must be an equal-style variable.

***Variable for dump image zoom is invalid style***
Must be an equal-style variable.

***Variable for fix adapt is invalid style***
Only equal-style variables can be used.

***Variable for fix aveforce is invalid style***
Only equal-style variables can be used.

***Variable for fix deform is invalid style***
The variable must be an equal-style variable.

***Variable for fix efield is invalid style***
The variable must be an equal- or atom-style variable.

***Variable for fix gravity is invalid style***
Only equal-style variables can be used.

***Variable for fix heat is invalid style***
Only equal-style or atom-style variables can be used.

***Variable for fix indent is invalid style***
Only equal-style variables can be used.

***Variable for fix indent is not equal style***
Only equal-style variables can be used.

***Variable for fix langevin is invalid style***
It must be an equal-style variable.

***Variable for fix move is invalid style***
Only equal-style variables can be used.

***Variable for fix setforce is invalid style***
Only equal-style variables can be used.

***Variable for fix temp/berendsen is invalid style***
Only equal-style variables can be used.

***Variable for fix temp/csld is invalid style***
Only equal-style variables can be used.

***Variable for fix temp/csvr is invalid style***
Only equal-style variables can be used.

***Variable for fix temp/rescale is invalid style***
Only equal-style variables can be used.

***Variable for fix wall is invalid style***
Only equal-style variables can be used.

***Variable for fix wall/reflect is invalid style***
Only equal-style variables can be used.

***Variable for fix wall/srd is invalid style***
Only equal-style variables can be used.

***Variable for group dynamic is invalid style***
The variable must be an atom-style variable.

***Variable for group is invalid style***
Only atom-style variables can be used.

***Variable for region cylinder is invalid style***
Only equal-style variables are allowed.

***Variable for region is invalid style***
Only equal-style variables can be used.

***Variable for region sphere is invalid style***
Only equal-style variables are allowed.

***Variable for restart is invalid style***
Only equal-style variables can be used.

***Variable for set command is invalid style***
Only atom-style variables can be used.

***Variable for thermo every is invalid style***
Only equal-style variables can be used.

***Variable for velocity set is invalid style***
Only atom-style variables can be used.

***Variable has circular dependency***
A circular dependency is when variable "a" in used by variable "b" and variable "b" is also used by variable "a". Circular dependencies with longer chains of dependence are also not allowed.

***Velocity command before simulation box is defined***
The velocity command cannot be used before a read_data, read_restart, or create_box command.

***Velocity command with no atoms existing***
A velocity command has been used, but no atoms yet exist.

***Velocity temperature ID does calculate a velocity bias***
The specified compute must compute a bias for temperature.

***Velocity temperature ID does not compute temperature***
The compute ID given to the velocity command must compute temperature.

***Verlet/split can only currently be used with comm_style brick***
This is a current restriction in LAMMPS.

***Verlet/split does not yet support TIP4P***
This is a current limitation.

***Verlet/split requires 2 partitions***
See the -partition command-line switch.

***Verlet/split requires Rspace partition layout be multiple of Kspace partition layout in each dim***
This is controlled by the processors command.

***Verlet/split requires Rspace partition size be multiple of Kspace partition size***
This is so there is an equal number of Rspace processors for every Kspace processor.

***Voro++ error: narea and neigh have a different size***
This error is returned by the Voro++ library.

***Water H epsilon must be 0.0 for pair style lj/cut/tip4p/cut***
This is because LAMMPS does not compute the Lennard-Jones interactions with these particles for efficiency reasons.

***Water H epsilon must be 0.0 for pair style lj/cut/tip4p/long***
This is because LAMMPS does not compute the Lennard-Jones interactions with these particles for efficiency reasons.

***Water H epsilon must be 0.0 for pair style lj/long/tip4p/long***
This is because LAMMPS does not compute the Lennard-Jones interactions with these particles for efficiency reasons.

***World variable count does not match # of partitions***
A world-style variable must specify a number of values equal to the number of processor partitions.

***Write_restart command before simulation box is defined***
The write_restart command cannot be used before a read_data, read_restart, or create_box command.

## 5.7 Warning messages

This is an alphabetic list of some of the WARNING messages LAMMPS prints out and the reason why. If the explanation here is not sufficient, the documentation for the offending command may help. This is a historic list and no longer updated. Instead the LAMMPS developers are trying to provide more details right with the error message or link to a paragraph with *detailed explanations*.

Warning messages also list the source file and line number where the warning was generated. For example, a message like this:

```
WARNING: Bond atom missing in box size check (domain.cpp:187)
```

means that line #187 in the file src/domain.cpp generated the error. Looking in the source code may help you figure out what went wrong.

Please also see the page with *Error messages*

---

***Adjusting Coulombic cutoff for MSM, new cutoff = %g***
The adjust/cutoff command is turned on and the Coulombic cutoff has been adjusted to match the user-specified accuracy.

***Angle atoms missing at step %ld***
One or more of three atoms needed to compute a particular angle are missing on this processor. Typically this is because the pairwise cutoff is set too short or the angle has blown apart and an atom is too far away.

***Angles are defined but no angle style is set***
The topology contains angles, but there are no angle forces computed since there was no angle_style command.

***Bond atom missing in box size check***
The second atom needed to compute a particular bond is missing on this processor. Typically this is because the pairwise cutoff is set too short or the bond has blown apart and an atom is too far away.

***Bond atom missing in image check***
The second atom in a particular bond is missing on this processor. Typically this is because the pairwise cutoff is set too short or the bond has blown apart and an atom is too far away.

***Bond atoms missing at step %ld***
The second atom needed to compute a particular bond is missing on this processor. Typically this is because the pairwise cutoff is set too short or the bond has blown apart and an atom is too far away.

***Bonds are defined but no bond style is set***
The topology contains bonds, but there are no bond forces computed since there was no bond_style command.

***Bond/angle/dihedral extent > half of periodic box length***
This is a restriction because LAMMPS can be confused about which image of an atom in the bonded interaction is the correct one to use. "Extent" in this context means the maximum end-to-end length of the bond/angle/dihedral. LAMMPS computes this by taking the maximum bond length, multiplying by the number of bonds in the interaction (e.g. 3 for a dihedral) and adding a small amount of stretch.

***Calling write_dump before a full system init.***
The write_dump command is used before the system has been fully initialized as part of a 'run' or 'minimize' command. Not all dump styles and features are fully supported at this point and thus the command may fail or produce incomplete or incorrect output. Insert a "run 0" command, if a full system init is required.

***Cannot count rigid body degrees-of-freedom before bodies are fully initialized***
This means the temperature associated with the rigid bodies may be incorrect on this timestep.

***Cannot count rigid body degrees-of-freedom before bodies are initialized***
This means the temperature associated with the rigid bodies may be incorrect on this timestep.

***Communication cutoff is 0.0. No ghost atoms will be generated. Atoms may get lost***
The communication cutoff defaults to the maximum of what is inferred from pair and bond styles (will be zero, if none are defined) and what is specified via *comm_modify cutoff* (defaults to 0.0). If this results to 0.0, no ghost atoms will be generated and LAMMPS may lose atoms or use incorrect periodic images of atoms in interaction lists. To avoid, either use *pair style zero* with a suitable cutoff or use *comm_modify cutoff*.

***Communication cutoff is shorter than a bond length based estimate. This may lead to errors.***
Since LAMMPS stores topology data with individual atoms, all atoms comprising a bond, angle, dihedral or improper must be present on any subdomain that "owns" the atom with the information, either as a local or a ghost atom. The communication cutoff is what determines up to what distance from a subdomain boundary ghost atoms are created. The communication cutoff is by default the largest non-bonded cutoff plus the neighbor skin distance, but for short or non-bonded cutoffs and/or long bonds, this may not be sufficient. This warning indicates that there is an increased risk of a simulation stopping unexpectedly because of Bond/Angle/Dihedral/Improper atoms missing. It can be silenced by manually setting the communication cutoff via *comm_modify cutoff*. However, since the heuristic used to determine the estimate is not always accurate, it is not changed automatically and the warning may be ignored depending on the specific system being simulated.

***Compute cna/atom cutoff may be too large to find ghost atom neighbors***
The neighbor cutoff used may not encompass enough ghost atoms to perform this operation correctly.

***Computing temperature of portions of rigid bodies***
The group defined by the temperature compute does not encompass all the atoms in one or more rigid bodies, so the change in degrees-of-freedom for the atoms in those partial rigid bodies will not be accounted for.

***Create_bonds max distance > minimum neighbor cutoff***
This means atom pairs for some atom types may not be in the neighbor list and thus no bond can be created between them.

***Delete_atoms cutoff > minimum neighbor cutoff***
This means atom pairs for some atom types may not be in the neighbor list and thus an atom in that pair cannot be deleted.

***Dihedral atoms missing at step %ld***
One or more of 4 atoms needed to compute a particular dihedral are missing on this processor. Typically this is because the pairwise cutoff is set too short or the dihedral has blown apart and an atom is too far away.

***Dihedral problem***
Conformation of the 4 listed dihedral atoms is extreme; you may want to check your simulation geometry.

***Dihedral problem: %d %ld %d %d %d %d***
Conformation of the 4 listed dihedral atoms is extreme; you may want to check your simulation geometry.

***Dihedrals are defined but no dihedral style is set***
The topology contains dihedrals, but there are no dihedral forces computed since there was no dihedral_style command.

***Dump dcd/xtc timestamp may be wrong with fix dt/reset***
If the fix changes the timestep, the dump dcd file will not reflect the change.

***Energy due to X extra global DOFs will be included in minimizer energies***
When using fixes like box/relax, the potential energy used by the minimizer is augmented by an additional energy provided by the fix. Thus the printed converged energy may be different from the total potential energy.

***Estimated error in splitting of dispersion coeffs is %g***
Error is greater than 0.0001 percent.

***FENE bond too long***
A FENE bond has stretched dangerously far. It's interaction strength will be truncated to attempt to prevent the bond from blowing up.

***FENE bond too long: %ld %d %d %g***
A FENE bond has stretched dangerously far. It's interaction strength will be truncated to attempt to prevent the bond from blowing up.

***FENE bond too long: %ld %g***
A FENE bond has stretched dangerously far. It's interaction strength will be truncated to attempt to prevent the bond from blowing up.

***Fix SRD walls overlap but fix srd overlap not set***
You likely want to set this in your input script.

***Fix bond/create is used multiple times or with fix bond/break - may not work as expected***
When using fix bond/create multiple times or in combination with fix bond/break, the individual fix instances do not share information about changes they made at the same time step and thus it may result in unexpected behavior.

***Fix bond/react: Atom affected by reaction too close to template edge***
This means an atom which changes type or connectivity during the reaction is too close to an 'edge' atom defined in the superimpose file. This could cause incorrect assignment of bonds, angle, etc. Generally, this means you must include more atoms in your templates, such that there are at least two atoms between each atom involved in the reaction and an edge atom.

***Fix bond/swap will ignore defined angles***
See the page for fix bond/swap for more info on this restriction.

***Fix deposit near setting < possible overlap separation %g***
This test is performed for finite size particles with a diameter, not for point particles. The near setting is smaller than the particle diameter which can lead to overlaps.

***Fix evaporate may delete atom with non-zero molecule ID***
This is probably an error, since you should not delete only one atom of a molecule.

***Fix gcmc using full_energy option***
Fix gcmc has automatically turned on the full_energy option since it is required for systems like the one specified by the user. User input included one or more of the following: kspace, triclinic, a hybrid pair style, an eam pair style, or no "single" function for the pair style.

***Fix langevin gjf using random gaussians is not implemented with kokkos***
This will most likely cause errors in kinetic fluctuations.

***Fix property/atom mol or charge w/out ghost communication***
A model typically needs these properties defined for ghost atoms.

***Fix qeq has non-zero lower Taper radius cutoff***
Absolute value must be <= 0.01.

***Fix qeq has very low Taper radius cutoff***
Value should typically be >= 5.0.

***Fix rattle should come after all other integration fixes***
This fix is designed to work after all other integration fixes change atom positions. Thus it should be the last integration fix specified. If not, it will not satisfy the desired constraints as well as it otherwise would.

***Fix recenter should come after all other integration fixes***
Other fixes may change the position of the center-of-mass, so fix recenter should come last.

***Fix srd SRD moves may trigger frequent reneighboring***
This is because the SRD particles may move long distances.

***Fix srd grid size > 1/4 of big particle diameter***
This may cause accuracy problems.

***Fix srd particle moved outside valid domain***
This may indicate a problem with your simulation parameters.

***Fix srd particles may move > big particle diameter***
This may cause accuracy problems.

***Fix srd viscosity < 0.0 due to low SRD density***
This may cause accuracy problems.

***Fixes cannot send data in Kokkos communication, switching to classic communication***
This is current restriction with Kokkos.

***For better accuracy use 'pair_modify table 0'***
The user-specified force accuracy cannot be achieved unless the table feature is disabled by using 'pair_modify table 0'.

***Group for fix_modify temp != fix group***
The fix_modify command is specifying a temperature computation that computes a temperature on a different group of atoms than the fix itself operates on. This is probably not what you want to do.

***H matrix size has been exceeded: m_fill=%d H.m=%dn***
This is the size of the matrix.

***Ignoring unknown or incorrect info command flag***
Self-explanatory. An unknown argument was given to the info command. Compare your input with the documentation.

***Improper atoms missing at step %ld***
One or more of 4 atoms needed to compute a particular improper are missing on this processor. Typically this is because the pairwise cutoff is set too short or the improper has blown apart and an atom is too far away.

***Improper problem: %d %ld %d %d %d %d***
Conformation of the 4 listed improper atoms is extreme; you may want to check your simulation geometry.

***Impropers are defined but no improper style is set***
The topology contains impropers, but there are no improper forces computed since there was no improper_style command.

***Increasing communication cutoff for GPU style***
The pair style has increased the communication cutoff to be consistent with the communication cutoff requirements for this pair style when run on the GPU.

***Kspace_modify slab param < 2.0 may cause unphysical behavior***
The kspace_modify slab parameter should be larger to ensure periodic grids padded with empty space do not overlap.

***Less insertions than requested***
The fix pour command was unsuccessful at finding open space for as many particles as it tried to insert.

***Library error in lammps_gather_atoms***
This library function cannot be used if atom IDs are not defined or are not consecutively numbered.

***Library error in lammps_scatter_atoms***
This library function cannot be used if atom IDs are not defined or are not consecutively numbered, or if no atom map is defined. See the atom_modify command for details about atom maps.

***Likewise 1-2 special neighbor interactions != 1.0***
The topology contains bonds, but there is no bond style defined and a 1-2 special neighbor scaling factor was not 1.0. This means that pair style interactions may have scaled or missing pairs in the neighbor list in expectation of interactions for those pairs being computed from the bond style.

***Likewise 1-3 special neighbor interactions != 1.0***
The topology contains angles, but there is no angle style defined and a 1-3 special neighbor scaling factor was not 1.0. This means that pair style interactions may have scaled or missing pairs in the neighbor list in expectation of interactions for those pairs being computed from the angle style.

***Likewise 1-4 special neighbor interactions != 1.0***
The topology contains dihedrals, but there is no dihedral style defined and a 1-4 special neighbor scaling factor was not 1.0. This means that pair style interactions may have scaled or missing pairs in the neighbor list in expectation of interactions for those pairs being computed from the dihedral style.

***Lost atoms via change_box: original %ld current %ld***
The command options you have used caused atoms to be lost.

***Lost atoms via displace_atoms: original %ld current %ld***
The command options you have used caused atoms to be lost.

***Lost atoms: original %ld current %ld***
Lost atoms are checked for each time thermo output is done. See the thermo_modify lost command for options. Lost atoms usually indicate bad dynamics, e.g. atoms have been blown far out of the simulation box, or moved further than one processor's subdomain away before reneighboring.

***Mismatch between velocity and compute groups***
The temperature computation used by the velocity command will not be on the same group of atoms that velocities

are being set for.

***Molecule has bond topology but no special bond settings***
This means the bonded atoms will not be excluded in pairwise interactions.

***Molecule template for create_atoms has multiple molecules***
The create_atoms command will only create molecules of a single type, i.e. the first molecule in the template.

***Molecule template for fix gcmc has multiple molecules***
The fix gcmc command will only create molecules of a single type, i.e. the first molecule in the template.

***Molecule template for fix shake has multiple molecules***
The fix shake command will only recognize molecules of a single type, i.e. the first molecule in the template.

***More than one compute centro/atom***
It is not efficient to use compute centro/atom more than once.

***More than one compute cluster/atom***
It is not efficient to use compute cluster/atom more than once.

***More than one compute cna/atom defined***
It is not efficient to use compute cna/atom more than once.

***More than one compute contact/atom***
It is not efficient to use compute contact/atom more than once.

***More than one compute coord/atom***
It is not efficient to use compute coord/atom more than once.

***More than one compute damage/atom***
It is not efficient to use compute ke/atom more than once.

***More than one compute erotate/sphere/atom***
It is not efficient to use compute erorate/sphere/atom more than once.

***More than one compute hexorder/atom***
It is not efficient to use compute hexorder/atom more than once.

***More than one compute ke/atom***
It is not efficient to use compute ke/atom more than once.

***More than one compute orientorder/atom***
It is not efficient to use compute orientorder/atom more than once.

***More than one fix rigid***
It is not efficient to use fix rigid more than once.

***Neighbor exclusions used with KSpace solver may give inconsistent Coulombic energies***
This is because excluding specific pair interactions also excludes them from long-range interactions which may not be the desired effect. The special_bonds command handles this consistently by ensuring excluded (or weighted) 1-2, 1-3, 1-4 interactions are treated consistently by both the short-range pair style and the long-range solver. This is not done for exclusions of charged atom pairs via the neigh_modify exclude command.

***New thermo_style command, previous thermo_modify settings will be lost***
If a thermo_style command is used after a thermo_modify command, the settings changed by the thermo_modify command will be reset to their default values. This is because the thermo_modify command acts on the currently defined thermo style, and a thermo_style command creates a new style.

***No Kspace calculation with verlet/split***
The second partition performs a kspace calculation so the kspace_style command must be used.

***No automatic unit conversion to XTC file format conventions possible for units lj***
This means no scaling will be performed.

***No fixes defined, atoms won't move***
If you are not using a fix like nve, nvt, npt then atom velocities and coordinates will not be updated during timestepping.

***Not using real units with pair reaxff***
This is most likely an error, unless you have created your own ReaxFF parameter file in a different set of units.

***Number of MSM mesh points changed to be a multiple of 2***
MSM requires that the number of grid points in each direction be a multiple of two and the number of grid points in one or more directions have been adjusted to meet this requirement.

***OMP_NUM_THREADS environment is not set.***
This environment variable must be set appropriately to use the OPENMP package.

***One or more atoms are time integrated more than once***
This is probably an error since you typically do not want to advance the positions or velocities of an atom more than once per timestep.

***One or more chunks do not contain all atoms in molecule***
This may not be what you intended.

***One or more dynamic groups may not be updated at correct point in timestep***
If there are other fixes that act immediately after the initial stage of time integration within a timestep (i.e. after atoms move), then the command that sets up the dynamic group should appear after those fixes. This will ensure that dynamic group assignments are made after all atoms have moved.

***One or more respa levels compute no forces***
This is computationally inefficient.

***Pair COMB charge %.10f with force %.10f hit max barrier***
Something is possibly wrong with your model.

***Pair COMB charge %.10f with force %.10f hit min barrier***
Something is possibly wrong with your model.

***Pair dsmc: num_of_collisions > number_of_A***
Collision model in DSMC is breaking down.

***Pair dsmc: num_of_collisions > number_of_B***
Collision model in DSMC is breaking down.

***Pair style restartinfo set but has no restart support***
This pair style has a bug, where it does not support reading and writing information to a restart file, but does not set the member variable "restartinfo" to 0 as required in that case.

***Particle deposition was unsuccessful***
The fix deposit command was not able to insert as many atoms as needed. The requested volume fraction may be too high, or other atoms may be in the insertion region.

***Proc subdomain size < neighbor skin, could lead to lost atoms***
The decomposition of the physical domain (likely due to load balancing) has led to a processor's subdomain being smaller than the neighbor skin in one or more dimensions. Since reneighboring is triggered by atoms moving the skin distance, this may lead to lost atoms, if an atom moves all the way across a neighboring processor's subdomain before reneighboring is triggered.

***Reducing PPPM order b/c stencil extends beyond nearest neighbor processor***
This may lead to a larger grid than desired. See the kspace_modify overlap command to prevent changing of the PPPM order.

***Reducing PPPMDisp Coulomb order b/c stencil extends beyond neighbor processor***
This may lead to a larger grid than desired. See the kspace_modify overlap command to prevent changing of the PPPM order.

***Reducing PPPMDisp dispersion order b/c stencil extends beyond neighbor processor***
This may lead to a larger grid than desired. See the kspace_modify overlap command to prevent changing of the PPPM order.

***Replacing a fix, but new group != old group***
The ID and style of a fix match for a fix you are changing with a fix command, but the new group you are specifying does not match the old group.

***Replicating in a non-periodic dimension***
The parameters for a replicate command will cause a non-periodic dimension to be replicated; this may cause unwanted behavior.

***Resetting reneighboring criteria during PRD***
A PRD simulation requires that neigh_modify settings be delay = 0, every = 1, check = yes. Since these settings were not in place, LAMMPS changed them and will restore them to their original values after the PRD simulation.

***Resetting reneighboring criteria during TAD***
A TAD simulation requires that neigh_modify settings be delay = 0, every = 1, check = yes. Since these settings were not in place, LAMMPS changed them and will restore them to their original values after the PRD simulation.

***Resetting reneighboring criteria during minimization***
Minimization requires that neigh_modify settings be delay = 0, every = 1, check = yes. Since these settings were not in place, LAMMPS changed them and will restore them to their original values after the minimization.

***Restart file used different # of processors***
The restart file was written out by a LAMMPS simulation running on a different number of processors. Due to round-off, the trajectories of your restarted simulation may diverge a little more quickly than if you ran on the same # of processors.

***Restart file used different 3d processor grid***
The restart file was written out by a LAMMPS simulation running on a different 3d grid of processors. Due to round-off, the trajectories of your restarted simulation may diverge a little more quickly than if you ran on the same # of processors.

***Restart file used different boundary settings, using restart file values***
Your input script cannot change these restart file settings.

***Restart file used different newton bond setting, using restart file value***
The restart file value will override the setting in the input script.

***Restart file used different newton pair setting, using input script value***
The input script value will override the setting in the restart file.

***Restrain problem: %d %ld %d %d %d %d***
Conformation of the 4 listed dihedral atoms is extreme; you may want to check your simulation geometry.

***Running PRD with only one replica***
This is allowed, but you will get no parallel speed-up.

***SRD bin shifting turned on due to small lamda***
This is done to try to preserve accuracy.

***SRD bin size for fix srd differs from user request***
Fix SRD had to adjust the bin size to fit the simulation box. See the cubic keyword if you want this message to be an error vs warning.

***SRD bins for fix srd are not cubic enough***
The bin shape is not within tolerance of cubic. See the cubic keyword if you want this message to be an error vs warning.

***SRD particle %d started inside big particle %d on step %ld bounce %d***
See the inside keyword if you want this message to be an error vs warning.

***SRD particle %d started inside wall %d on step %ld bounce %d***
See the inside keyword if you want this message to be an error vs warning.

***Shake determinant < 0.0***
The determinant of the quadratic equation being solved for a single cluster specified by the fix shake command is numerically suspect. LAMMPS will set it to 0.0 and continue.

***Shell command returned with non-zero status***
This may indicate the shell command did not operate as expected.

***Should not allow rigid bodies to bounce off reflecting walls***
LAMMPS allows this, but their dynamics are not computed correctly.

***Should not use fix nve/limit with fix shake or fix rattle***
This will lead to invalid constraint forces in the SHAKE/RATTLE computation.

***Slab correction not needed for MSM***
Slab correction is intended to be used with Ewald or PPPM and is not needed by MSM.

***System is not charge neutral, net charge = %g***
The total charge on all atoms on the system is not 0.0. For some KSpace solvers this is only a warning.

***Table inner cutoff >= outer cutoff***
You specified an inner cutoff for a Coulombic table that is longer than the global cutoff. Probably not what you wanted.

***Temperature for MSST is not for group all***
User-assigned temperature to MSST fix does not compute temperature for all atoms. Since MSST computes a global pressure, the kinetic energy contribution from the temperature is assumed to also be for all atoms. Thus the pressure used by MSST could be inaccurate.

***Temperature for NPT is not for group all***
User-assigned temperature to NPT fix does not compute temperature for all atoms. Since NPT computes a global pressure, the kinetic energy contribution from the temperature is assumed to also be for all atoms. Thus the pressure used by NPT could be inaccurate.

***Temperature for fix modify is not for group all***
The temperature compute is being used with a pressure calculation which does operate on group all, so this may be inconsistent.

***Temperature for thermo pressure is not for group all***
User-assigned temperature to thermo via the thermo_modify command does not compute temperature for all atoms. Since thermo computes a global pressure, the kinetic energy contribution from the temperature is assumed to also be for all atoms. Thus the pressure printed by thermo could be inaccurate.

***The minimizer does not re-orient dipoles when using fix efield***
This means that only the atom coordinates will be minimized, not the orientation of the dipoles.

***Too many common neighbors in CNA %d times***
More than the maximum # of neighbors was found multiple times. This was unexpected.

***Too many neighbors in CNA for %d atoms***
More than the maximum # of neighbors was found multiple times. This was unexpected.

***Use special bonds = 0,1,1 with bond style fene***
Most FENE models need this setting for the special_bonds command.

***Use special bonds = 0,1,1 with bond style fene/expand***
Most FENE models need this setting for the special_bonds command.

***Using a many-body potential with bonds/angles/dihedrals and special_bond exclusions***
This is likely not what you want to do. The exclusion settings will eliminate neighbors in the neighbor list, which the many-body potential needs to calculated its terms correctly.

***Using compute temp/deform with inconsistent fix deform remap option***
Fix nvt/sllod assumes deforming atoms have a velocity profile provided by "remap v" or "remap none" as a fix deform option.

***Using compute temp/deform with no fix deform defined***
This is probably an error, since it makes little sense to use compute temp/deform in this case.

***Using fix srd with box deformation but no SRD thermostat***
The deformation will heat the SRD particles so this can be dangerous.

***Using pair potential shift with pair_modify compute no***
The shift effects will thus not be computed.

***Using pair tail corrections with nonperiodic system***
This is probably a bogus thing to do, since tail corrections are computed by integrating the density of a periodic system out to infinity.

***Using pair tail corrections with pair_modify compute no***
The tail corrections will thus not be computed.

CHAPTER

SIX

# COMMANDS

These pages describe how a LAMMPS input script is formatted and the commands in it are used to define a LAMMPS simulation.

## 6.1 LAMMPS input scripts

LAMMPS executes calculations by reading commands from a input script (text file), one line at a time. When the input script ends, LAMMPS exits. This is different from programs that read and process the entire input before starting a calculation.

Each command causes LAMMPS to take some immediate action without regard for any commands that may be processed later. Commands may set an internal variable, read in a file, or run a simulation. These actions can be grouped into three categories:

a) commands that change a global setting (examples: *timestep*, *newton*, *echo*, *log*, *thermo*, *restart*),

b) commands that add, modify, remove, or replace "styles" that are executed during a "run" (examples: *pair_style*, *fix*, *compute*, *dump*, *thermo_style*, *pair_modify*), and

c) commands that execute a "run" or perform some other computation or operation (examples: *print*, *run*, *minimize*, *temper*, *write_dump*, *rerun*, *read_data*, *read_restart*)

Commands in category a) have default settings, which means you only need to use the command if you wish to change the defaults.

In many cases, the ordering of commands in an input script is not important, but can have consequences when the global state is changed between commands in the c) category. The following rules apply:

(1) LAMMPS does not read your entire input script and then perform a simulation with all the settings. Rather, the input script is read one line at a time and each command takes effect when it is read. Thus this sequence of commands:

```
timestep 0.5
run      100
run      100
```

does something different than this sequence:

```
run      100
timestep 0.5
run      100
```

In the first case, the specified timestep (0.5 fs) is used for two simulations of 100 timesteps each. In the second case, the default timestep (1.0 fs) is used for the first 100 step simulation and a 0.5 fs timestep is used for the second one.

(2) Some commands are only valid when they follow other commands. For example you cannot set the temperature of a group of atoms until atoms have been defined and a group command is used to define which atoms belong to the group.

(3) Sometimes command B will use values that can be set by command A. This means command A must precede command B in the input script if it is to have the desired effect. For example, the *read_data* command initializes the system by setting up the simulation box and assigning atoms to processors. If default values are not desired, the *processors* and *boundary* commands need to be used before `read_data` to tell LAMMPS how to map processors to the simulation box.

Many input script errors are detected by LAMMPS and an ERROR or WARNING message is printed. The *Errors* page gives more information on what errors mean. The documentation for each command lists restrictions on how the command can be used.

You can use the *-skiprun* command-line flag to have LAMMPS skip the execution of any `run`, `minimize`, or similar commands to check the entire input for correct syntax to avoid crashes on typos or syntax errors in long runs.

## 6.2 Parsing rules for input scripts

Each non-blank line in the input script is treated as a command. LAMMPS commands are case sensitive. Command names are lower-case, as are specified command arguments. Upper case letters may be used in file names or user-chosen ID strings.

Here are 6 rules for how each line in the input script is parsed by LAMMPS:

1. If the last printable character on the line is a "&" character, the command is assumed to continue on the next line. The next line is concatenated to the previous line by removing the "&" character and line break. This allows long commands to be continued across two or more lines. See the discussion of triple quotes in *6* for how to continue a command across multiple line without using "&" characters.

2. All characters from the first "#" character onward are treated as comment and discarded. The exception to this rule is described in *6*. Note that a comment after a trailing "&" character will prevent the command from continuing on the next line. Also note that for multi-line commands a single leading "#" will comment out the entire command.

   ```
   # this is a comment
   timestep  1.0   # this is also a comment
   ```

3. The line is searched repeatedly for $ characters, which indicate variables that are replaced with a text string. The exception to this rule is described in *6*.

   If the $ is followed by text in curly brackets '{}', then the variable name is the text inside the curly brackets. If no curly brackets follow the $, then the variable name is the single character immediately following the $. Thus `${myTemp}` and `$x` refer to variables named "myTemp" and "x", while `$xx` will be interpreted as a variable named "x" followed by an "x" character.

   How the variable is converted to a text string depends on what style of variable it is; see the *variable* page for details. It can be a variable that stores multiple text strings, and return one of them. The returned text string can be multiple "words" (space separated) which will then be interpreted as multiple arguments in the input command. The variable can also store a numeric formula which will be evaluated and its numeric result returned as a string.

   As a special case, if the $ is followed by parenthesis "()", then the text inside the parenthesis is treated as an "immediate" variable and evaluated as an *equal-style variable*. This is a way to use numeric formulas in an input script without having to assign them to variable names. For example, these 3 input script lines:

```
variable X equal (xlo+xhi)/2+sqrt(v_area)
region 1 block $X 2 INF INF EDGE EDGE
variable X delete
```

can be replaced by:

```
region 1 block $((xlo+xhi)/2+sqrt(v_area)) 2 INF INF EDGE EDGE
```

so that you do not have to define (or discard) a temporary variable, "X" in this case.

Additionally, the entire "immediate" variable expression may be followed by a colon, followed by a C-style format string, e.g. `:%f` or `:%.10g`. The format string must be appropriate for a double-precision floating-point value. The format string is used to output the result of the variable expression evaluation. If a format string is not specified, a high-precision `%.20g` is used as the default format.

This can be useful for formatting print output to a desired precision:

```
print "Final energy per atom: $(v_ke_per_atom+v_pe_per_atom:%10.3f) eV/atom"
```

Note that neither the curly-bracket or immediate form of variables can contain nested $ characters for other variables to substitute for. Thus you may **NOT** do this:

```
variable        a equal 2
variable        b2 equal 4
print           "B2 = ${b$a}"
```

Nor can you specify an expression like `$($x-1.0)` for an immediate variable, but you could use `$(v_x-1.0)`, since the latter is valid syntax for an *equal-style variable*.

See the *variable* command for more details of how strings are assigned to variables and evaluated, and how they can be used in input script commands.

4. The line is broken into "words" separated by white-space (tabs, spaces). Note that words can thus contain letters, digits, underscores, or punctuation characters.

5. The first word is the command name. All successive words in the line are arguments.

6. If you want text with spaces to be treated as a single argument, it can be enclosed in either single (') or double (") or triple ("""") quotes. A long single argument enclosed in single or double quotes can span multiple lines if the "&" character is used, as described in *1* above. When the lines are concatenated together by LAMMPS (and the "&" characters and line breaks removed), the combined text will become a single line. If you want multiple lines of an argument to retain their line breaks, the text can be enclosed in triple quotes, in which case "&" characters are not needed and do not function as line continuation character. For example:

```
print "Volume = $v"
print 'Volume = $v'
if "${steps} > 1000" then quit
variable a string "red green blue &
                   purple orange cyan"
print """
System volume = $v
System temperature = $t
"""
```

In each of these cases, the single, double, or triple quotes are removed and the enclosed text stored internally as a single argument.

See the *dump modify format*, *print*, *if*, and *python* commands for examples.

A "#" or "$" character that is between quotes will not be treated as a comment indicator in *2* or substituted for as a variable in *3*.

**Note**

If the argument is itself a command that requires a quoted argument (e.g. using a *print* command as part of an *if* or *run every* command), then single, double, or triple quotes can be nested in the usual manner. See the doc pages for those commands for examples. Only one level of nesting is allowed, but that should be sufficient for most use cases.

**ASCII versus UTF-8**

LAMMPS expects and processes 7-bit ASCII format text internally. Many modern environments use UTF-8 encoding, which is a superset of the 7-bit ASCII character table and thus mostly compatible. However, there are several non-ASCII characters that can look very similar to their ASCII equivalents or are invisible (so they look like a blank), but are encoded differently. Web browsers, PDF viewers, document editors are known to sometimes replace one with the other for a better looking output. However, that can lead to problems, for instance, when using cut-n-paste of input file examples from web pages, or when using a document editor (not a dedicated plain text editor) for writing LAMMPS inputs. LAMMPS will try to detect this and substitute the non-ASCII characters with their ASCII equivalents where known. There also is going to be a warning printed, if this occurs. It is recommended to avoid such characters altogether in LAMMPS input, data and potential files. The replacement tables are likely incomplete and dependent on users reporting problems processing correctly looking input containing UTF-8 encoded non-ASCII characters.

## 6.3 Input script structure

This page describes the structure of a typical LAMMPS input script. The examples directory in the LAMMPS distribution contains many sample input scripts; it is discussed on the *Examples* doc page.

A LAMMPS input script typically has 4 parts:

1. *Initialization*
2. *System definition*
3. *Simulation settings*
4. *Run a simulation*

The last 2 parts can be repeated as many times as desired. I.e. run a simulation, change some settings, run some more, etc. Each of the 4 parts is now described in more detail. Remember that almost all commands need only be used if a non-default value is desired.

### 6.3.1 Initialization

Set parameters that need to be defined before atoms are created or read-in from a file.

The relevant commands are *units*, *dimension*, *newton*, *processors*, *boundary*, *atom_style*, *atom_modify*.

If force-field parameters appear in the files that will be read, these commands tell LAMMPS what kinds of force fields are being used: *pair_style*, *bond_style*, *angle_style*, *dihedral_style*, *improper_style*.

### 6.3.2 System definition

There are 3 ways to define the simulation cell and reserve space for force field info and fill it with atoms in LAMMPS. Read them in from (1) a data file or (2) a restart file via the *read_data* or *read_restart* commands, respectively. These files can also contain molecular topology information. Or (3) create a simulation cell and fill it with atoms on a lattice (with no molecular topology), using these commands: *lattice*, *region*, *create_box*, *create_atoms* or *read_dump*.

The entire set of atoms can be duplicated to make a larger simulation using the *replicate* command.

### 6.3.3 Simulation settings

Once atoms and molecular topology are defined, a variety of settings can be specified: force field coefficients, simulation parameters, output options, and more.

Force field coefficients are set by these commands (they can also be set in the read-in files): *pair_coeff*, *bond_coeff*, *angle_coeff*, *dihedral_coeff*, *improper_coeff*, *kspace_style*, *dielectric*, *special_bonds*.

Various simulation parameters are set by these commands: *neighbor*, *neigh_modify*, *group*, *timestep*, *reset_timestep*, *run_style*, *min_style*, *min_modify*.

Fixes impose a variety of boundary conditions, time integration, and diagnostic options. The *fix* command comes in many flavors.

Various computations can be specified for execution during a simulation using the *compute*, *compute_modify*, and *variable* commands.

Output options are set by the *thermo*, *dump*, and *restart* commands.

### 6.3.4 Run a simulation

A molecular dynamics simulation is run using the *run* command. Energy minimization (molecular statics) is performed using the *minimize* command. A parallel tempering (replica-exchange) simulation can be run using the *temper* command.

## 6.4 Commands by category

This page lists most of the LAMMPS commands, grouped by category. The *General commands* page lists all general commands alphabetically. Style options for entries like fix, compute, pair etc. have their own pages where they are listed alphabetically.

### 6.4.1 Initialization

| | | | | |
|---|---|---|---|---|
| *newton* | *package* | *processors* | *suffix* | *units* |

### 6.4.2 Setup simulation box

| | | | |
|---|---|---|---|
| *boundary* | *change_box* | *create_box* | *dimension* |
| *lattice* | *region* | | |

### 6.4.3 Setup atoms

| | | | |
|---|---|---|---|
| *atom_modify* | *atom_style* | *balance* | *create_atoms* |
| *create_bonds* | *delete_atoms* | *delete_bonds* | *displace_atoms* |
| *group* | *mass* | *molecule* | *read_data* |
| *read_dump* | *read_restart* | *replicate* | *set* |
| *velocity* | | | |

### 6.4.4 Force fields

| | | | |
|---|---|---|---|
| *angle_coeff* | *angle_style* | *bond_coeff* | *bond_style* |
| *bond_write* | *dielectric* | *dihedral_coeff* | *dihedral_style* |
| *improper_coeff* | *improper_style* | *kspace_modify* | *kspace_style* |
| *pair_coeff* | *pair_modify* | *pair_style* | *pair_write* |
| *special_bonds* | | | |

### 6.4.5 Settings

| | | | |
|---|---|---|---|
| *comm_modify* | *comm_style* | *info* | *min_modify* |
| *min_style* | *neigh_modify* | *neighbor* | *partition* |
| *reset_timestep* | *run_style* | *timer* | *timestep* |

### 6.4.6 Operations within timestepping (fixes) and diagnostics (computes)

| | | | |
|---|---|---|---|
| *compute* | *compute_modify* | *fix* | *fix_modify* |
| *uncompute* | *unfix* | | |

### 6.4.7 Output

| | | | |
|---|---|---|---|
| *dump image* | *dump movie* | *dump* | *dump_modify* |
| *restart* | *thermo* | *thermo_modify* | *thermo_style* |
| *undump* | *write_coeff* | *write_data* | *write_dump* |
| *write_restart* | | | |

### 6.4.8 Actions

| | | | | | |
|---|---|---|---|---|---|
| *minimize* | *neb* | *neb_spin* | *prd* | *rerun* | *run* |
| *tad* | *temper* | | | | |

### 6.4.9 Input script control

| | | | | | | |
|---|---|---|---|---|---|---|
| *clear* | *echo* | *if* | *include* | *info* | *jump* | *label* |
| *log* | *next* | *print* | *python* | *quit* | *shell* | *variable* |

## 6.5 General commands

An alphabetic list of general LAMMPS commands.

| | | | | | |
|---|---|---|---|---|---|
| *angle_coeff* | *angle_style* | *angle_write* | *atom_modify* | *atom_style* | *balance* |
| *bond_coeff* | *bond_style* | *bond_write* | *boundary* | *change_box* | *clear* |
| *comm_modify* | *comm_style* | *compute* | *compute_modify* | *create_atoms* | *create_bonds* |
| *create_box* | *delete_atoms* | *delete_bonds* | *dielectric* | *dihedral_coeff* | *dihedral_style* |
| *dihedral_write* | *dimension* | *displace_atoms* | *dump* | *dump_modify* | *echo* |
| *fix* | *fix_modify* | *geturl* | *group* | *if* | *improper_coeff* |
| *improper_style* | *include* | *info* | *jump* | *kspace_modify* | *kspace_style* |
| *label* | *labelmap* | *lattice* | *log* | *mass* | *minimize* |
| *min_modify* | *min_style* | *molecule* | *neigh_modify* | *neighbor* | *newton* |
| *next* | *package* | *pair_coeff* | *pair_modify* | *pair_style* | *pair_write* |
| *partition* | *print* | *processors* | *quit* | *read_data* | *read_dump* |
| *read_restart* | *region* | *replicate* | *rerun* | *reset_atoms* | *reset_timestep* |
| *restart* | *run* | *run_style* | *set* | *shell* | *special_bonds* |
| *suffix* | *thermo* | *thermo_modify* | *thermo_style* | *timer* | *timestep* |
| *uncompute* | *undump* | *unfix* | *units* | *variable* | *velocity* |
| *write_coeff* | *write_data* | *write_dump* | *write_molecule* | *write_restart* | |

Additional general LAMMPS commands provided by packages. A few commands have accelerated versions. This is indicated by an additional letter in parentheses: k = KOKKOS.

| | | | | | |
|---|---|---|---|---|---|
| *dynamical_matrix (k)* | *group2ndx* | *hyper* | *kim* | *fitpod* | *mdi* |
| *ndx2group* | *neb* | *neb/spin* | *plugin* | *prd* | *python* |
| *region2vmd* | *tad* | *temper* | *temper/grem* | *temper/npt* | *third_order (k)* |

## 6.6 Fix styles

An alphabetic list of all LAMMPS *fix* commands. Some styles have accelerated versions. This is indicated by additional letters in parenthesis: g = GPU, i = INTEL, k = KOKKOS, o = OPENMP, t = OPT.

| | | | |
|---|---|---|---|
| *accelerate/cos* | *acks2/reaxff (k)* | *adapt* | *adapt/fep* |
| *addforce (k)* | *add/heat* | *addtorque/atom* | *addtorque/group* |
| *alchemy* | *align/self* | *amoeba/bitorsion* | *amoeba/pitorsion* |
| *append/atoms* | *atom/swap* | *atom_weight/apip* | *ave/atom* |
| *ave/chunk* | *ave/correlate* | *ave/correlate/long* | *ave/grid* |
| *ave/histo* | *ave/histo/weight* | *ave/moments* | *ave/time* |
| *aveforce (k)* | *balance* | *baoab* | *bocs* |
| *bond/break* | *bond/create* | *bond/create/angle* | *bond/react* |
| *bond/swap* | *box/relax* | *brownian* | *brownian/asphere* |
| *brownian/sphere* | *charge/regulation* | *cmap (k)* | *colvars* |
| *controller* | *damping/cundall* | *deform (k)* | *deform/pressure* |
| *deposit* | *dpd/energy (k)* | *drag (k)* | *drude* |
| *drude/transform/direct* | *drude/transform/inverse* | *dt/reset (k)* | *edpd/source* |
| *efield (k)* | *efield/lepton* | *efield/tip4p* | *ehex* |
| *electrode/conp (i)* | *electrode/conq (i)* | *electrode/thermo (i)* | *electron/stopping (k)* |
| *electron/stopping/fit* | *enforce2d (k)* | *eos/cv* | *eos/table* |
| *eos/table/rx (k)* | *evaporate* | *external (k)* | *ffl* |

continues on next page

Table 1 – continued from previous page

| | | | |
|---|---|---|---|
| *filter/corotate* | *flow/gauss* | *freeze (k)* | *gcmc* |
| *gemc* | *gjf* | *gld* | *gle* |
| *graphics/arrows* | *graphics/chunk* | *graphics/isosurface* | *graphics/labels* |
| *graphics/lines* | *graphics/objects* | *graphics/periodic* | *graphics/replica* |
| *gravity (ko)* | *grem* | *halt* | *heat* |
| *heat/flow* | *hmc* | *hyper/global* | *hyper/local* |
| *ilves* | *imd* | *indent* | *ipi* |
| *lambda/apip* | *lambda/la/csp/apip* | *lambda_thermostat/apip* | *langevin (k)* |
| *langevin/drude* | *langevin/eff* | *langevin/spin* | *lb/fluid* |
| *lb/momentum* | *lb/viscous* | *lineforce (k)* | *manifoldforce* |
| *mdi/qm* | *mdi/qmmm* | *meso/move* | *mol/swap* |
| *momentum (k)* | *momentum/chunk* | *move* | *msst* |
| *mvv/dpd* | *mvv/edpd* | *mvv/tdpd* | *neb* |
| *neb/spin* | *neighbor/swap* | *nonaffine/displacement* | *nph (ko)* |
| *nph/asphere (o)* | *nph/body* | *nph/eff* | *nph/sphere (ko)* |
| *nphug* | *npt (giko)* | *npt/asphere (o)* | *npt/body* |
| *npt/cauchy* | *npt/eff* | *npt/sphere (ko)* | *npt/uef* |
| *numdiff* | *numdiff/virial* | *nve (giko)* | *nve/asphere (gik)* |
| *nve/asphere/noforce* | *nve/body* | *nve/dot* | *nve/dotc/langevin* |
| *nve/eff* | *nve/limit (k)* | *nve/line* | *nve/manifold/rattle* |
| *nve/noforce (k)* | *nve/sphere (ko)* | *nve/bpm/sphere* | *nve/spin* |
| *nve/tri* | *nvk* | *nvt (giko)* | *nvt/asphere (o)* |
| *nvt/body* | *nvt/eff* | *nvt/manifold/rattle* | *nvt/sllod (iko)* |
| *nvt/sllod/eff* | *nvt/sphere (ko)* | *nvt/uef* | *oneway (k)* |
| *orient/bcc (o)* | *orient/fcc (o)* | *orient/eco (o)* | *pafi* |
| *pair* | *phonon* | *pimd/langevin* | *pimd/nvt* |
| *pimd/langevin/bosonic* | *pimd/nvt/bosonic* | *planeforce (k)* | *plumed* |
| *polarize/bem/gmres* | *polarize/bem/icc* | *polarize/functional* | *pour* |
| *precession/spin* | *press/berendsen* | *press/langevin* | *print* |
| *propel/self* | *property/atom (k)* | *python/invoke* | *python/move* |
| *qbmsst* | *qeq/comb (o)* | *qeq/ctip (o)* | *qeq/dynamic (o)* |
| *qeq/fire (o)* | *qeq/point (o)* | *qeq/reaxff (ko)* | *qeq/rel/reaxff* |
| *qeq/shielded (o)* | *qeq/slater (o)* | *qmmm* | *qtb* |
| *qtpie/reaxff* | *rattle* | *reaxff/bonds (k)* | *reaxff/species (k)* |
| *recenter (k)* | *restrain* | *rheo* | *rheo/oxidation* |
| *rheo/pressure* | *rheo/thermal* | *rheo/viscosity* | *rhok* |
| *rigid (o)* | *rigid/meso* | *rigid/nph (o)* | *rigid/nph/small* |
| *rigid/npt (o)* | *rigid/npt/small* | *rigid/nve (o)* | *rigid/nve/small* |
| *rigid/nvt (o)* | *rigid/nvt/small* | *rigid/small (o)* | *rx (k)* |
| *saed/vtk* | *set* | *setforce (k)* | *settorque/atom* |
| *setforce/spin* | *sgcmc* | *shake (k)* | *shardlow (k)* |
| *smd* | *smd/adjust_dt* | *smd/integrate_tlsph* | *smd/integrate_ulsph* |
| *smd/move_tri_surf* | *smd/setvel* | *smd/wall_surface* | *sph* |
| *sph/stationary* | *spring (k)* | *spring/chunk* | *spring/rg* |
| *spring/self (k)* | *srd* | *store/force* | *store/state* |
| *surface/global* | *surface/local* | *tdpd/source* | *temp/berendsen (k)* |
| *temp/csld* | *temp/csvr* | *temp/rescale (k)* | *temp/rescale/eff* |
| *tfmc* | *tgnpt/drude* | *tgnvt/drude* | *thermal/conductivity* |
| *ti/spring* | *tmd* | *ttm* | *ttm/grid* |
| *ttm/mod* | *ttm/thermal* | *tune/kspace* | *vector* |
| *viscosity* | *viscous (k)* | *viscous/nonlinear* | *viscous/sphere (k)* |
| *wall/body/polygon* | *wall/body/polyhedron* | *wall/colloid* | *wall/ees* |

continues on next page

Table 1 – continued from previous page

| | | | |
|---|---|---|---|
| *wall/flow (k)* | *wall/gran (k)* | *wall/gran/region* | *wall/harmonic (k)* |
| *wall/harmonic/outside* | *wall/lj1043 (k)* | *wall/lj126 (k)* | *wall/lj93 (k)* |
| *wall/lepton* | *wall/morse (k)* | *wall/piston* | *wall/reflect (k)* |
| *wall/reflect/stochastic* | *wall/region (k)* | *wall/region/ees* | *wall/srd* |
| *wall/table* | *widom* | | |

## 6.7 Compute styles

An alphabetic list of all LAMMPS *compute* commands. Some styles have accelerated versions. This is indicated by additional letters in parenthesis: g = GPU, i = INTEL, k = KOKKOS, o = OPENMP, t = OPT.

| | | | |
|---|---|---|---|
| *ackland/atom (o)* | *adf* | *aggregate/atom* | *angle* |
| *angle/local* | *angmom/chunk* | *ave/sphere/atom (ko)* | *basal/atom* |
| *body/local* | *bond* | *bond/local* | *born/matrix* |
| *centro/atom (o)* | *centroid/stress/atom* | *chunk/atom* | *chunk/spread/atom* |
| *cluster/atom* | *cna/atom (o)* | *cnp/atom (o)* | *com* |
| *com/chunk* | *composition/atom (k)* | *contact/atom* | *coord/atom (ko)* |
| *count/type* | *damage/atom* | *dihedral* | *dihedral/local* |
| *dilatation/atom* | *dipole* | *dipole/chunk* | *dipole/tip4p* |
| *dipole/tip4p/chunk* | *displace/atom* | *dpd* | *dpd/atom* |
| *edpd/temp/atom* | *efield/atom* | *efield/wolf/atom* | *entropy/atom (o)* |
| *erotate/asphere (k)* | *erotate/rigid* | *erotate/sphere (k)* | *erotate/sphere/atom (o)* |
| *event/displace* | *fabric* | *fep* | *fep/ta* |
| *force/tally* | *fragment/atom* | *gaussian/grid/local (k)* | *global/atom* |
| *group/group* | *gyration* | *gyration/chunk* | *gyration/shape* |
| *gyration/shape/chunk* | *hbond/local* | *heat/flux* | *heat/flux/tally* |
| *heat/flux/virial/tally* | *hexorder/atom (o)* | *hma* | *improper* |
| *improper/local* | *inertia/chunk* | *ke* | *ke/atom (o)* |
| *ke/atom/eff* | *ke/eff* | *ke/rigid* | *mliap* |
| *momentum* | *msd* | *msd/chunk* | *msd/nongauss* |
| *nbond/atom* | *omega/chunk* | *orientorder/atom (ko)* | *pace* |
| *pair* | *pair/local* | *pe* | *pe/atom* |
| *pe/mol/tally* | *pe/tally* | *plasticity/atom* | *pod/atom* |
| *podd/atom* | *pod/local* | *pod/global* | *pressure* |
| *pressure/alchemy* | *pressure/uef* | *property/atom* | *property/chunk* |
| *property/grid* | *property/local* | *ptm/atom* | *rattlers/atom* |
| *rdf (o)* | *reaxff/atom (k)* | *reduce* | *reduce/chunk* |
| *reduce/region* | *rheo/property/atom* | *rigid/local* | *saed* |
| *slcsa/atom* | *slice* | *smd/contact/radius* | *smd/damage* |
| *smd/hourglass/error* | *smd/internal/energy* | *smd/plastic/strain* | *smd/plastic/strain/rate* |
| *smd/rho* | *smd/tlsph/defgrad* | *smd/tlsph/dt* | *smd/tlsph/num/neighs* |
| *smd/tlsph/shape* | *smd/tlsph/strain* | *smd/tlsph/strain/rate* | *smd/tlsph/stress* |
| *smd/triangle/vertices* | *smd/ulsph/effm* | *smd/ulsph/num/neighs* | *smd/ulsph/strain* |
| *smd/ulsph/strain/rate* | *smd/ulsph/stress* | *smd/vol* | *snap* |
| *sna/atom* | *sna/grid (k)* | *sna/grid/local (k)* | *snad/atom* |
| *snav/atom* | *sph/e/atom* | *sph/rho/atom* | *sph/t/atom* |
| *spin* | *stress/atom* | *stress/cartesian* | *stress/cylinder* |
| *stress/mop* | *stress/mop/profile* | *stress/spherical* | *stress/tally* |
| *tdpd/cc/atom* | *temp (k)* | *temp/asphere* | *temp/body* |
| *temp/chunk* | *temp/com (k)* | *temp/cs* | *temp/deform (k)* |

continues on next page

Table 2 – continued from previous page

| | | | |
|---|---|---|---|
| *temp/deform/eff* | *temp/drude* | *temp/eff* | *temp/partial* |
| *temp/profile* | *temp/ramp* | *temp/region* | *temp/region/eff* |
| *temp/rotate* | *temp/sphere (k)* | *temp/uef* | *ti* |
| *torque/chunk* | *vacf* | *vacf/chunk* | *vcm/chunk* |
| *viscosity/cos* | *voronoi/atom* | *xrd* | |

## 6.8 Pair styles

All LAMMPS *pair_style* commands. Some styles have accelerated versions. This is indicated by additional letters in parenthesis: g = GPU, i = INTEL, k = KOKKOS, o = OPENMP, t = OPT.

| | | |
|---|---|---|
| *none* | *zero* | *hybrid (ko)* |
| *hybrid/molecular (o)* | *hybrid/overlay (ko)* | *hybrid/scaled (ko)* |
| *kim* | *list* | *tracker* |
| | | |
| *adp (ko)* | *agni (o)* | *aip/water/2dm (t)* |
| *airebo (io)* | *airebo/bc (o)* | *airebo/morse (io)* |
| *amoeba (g)* | *atm (o)* | *beck (gko)* |
| *body/nparticle (o)* | *body/rounded/polygon (o)* | *body/rounded/polyhedron (o)* |
| *bondval (k)* | *bondval/vec (k)* | *bop* |
| *born (gko)* | *born/coul/dsf (o)* | *born/coul/dsf/cs (o)* |
| *born/coul/long (gko)* | *born/coul/long/cs (go)* | *born/coul/msm (o)* |
| *born/coul/wolf (gko)* | *born/coul/wolf/cs (go)* | *born/gauss (ko)* |
| *bpm/spring* | *brownian (ko)* | *brownian/poly (o)* |
| *buck (giko)* | *buck/coul/cut (giko)* | *buck/coul/long (giko)* |
| *buck/coul/long/cs (o)* | *buck/coul/msm (o)* | *buck/long/coul/long (o)* |
| *buck/mdf (ko)* | *buck6d/coul/gauss/dsf (ko)* | *buck6d/coul/gauss/long (ko)* |
| *colloid (gko)* | *comb (o)* | *comb3* |
| *cosine/squared (ko)* | *coul/ctip* | *coul/cut (gko)* |
| *coul/cut/dielectric* | *coul/cut/global (ko)* | *coul/cut/soft (o)* |
| *coul/cut/soft/gapsys (o)* | *coul/debye (gko)* | *coul/diel (ko)* |
| *coul/dsf (gko)* | *coul/esp* | *coul/exclude (o)* |
| *coul/long (gko)* | *coul/long/cs (go)* | *coul/long/dielectric* |
| *coul/long/soft (o)* | *coul/msm (o)* | *coul/slater/cut (o)* |
| *coul/slater/long (gko)* | *coul/shield (ko)* | *coul/streitz (o)* |
| *coul/tt* | *coul/wolf (ko)* | *coul/wolf/cs (o)* |
| *dispersion/d3* | *dpd (giko)* | *dpd/coul/slater/long (g)* |
| *dpd/ext (ko)* | *dpd/ext/tstat (ko)* | *dpd/fdt* |
| *dpd/fdt/energy (k)* | *dpd/tstat (gko)* | *dsmc* |
| *e3b (o)* | *drip (o)* | *eam (gikot)* |
| *eam/alloy (gikot)* | *eam/cd* | *eam/cd/old* |
| *eam/fs (gikot)* | *eam/fs/apip* | *eam/he* |
| *eam/apip* | *edip (o)* | *edip/multi* |
| *edpd (g)* | *eff/cut* | *eim (o)* |
| *exp6/rx (k)* | *extep (o)* | *gauss (gko)* |
| *gauss/cut (ko)* | *gayberne (gio)* | *gran/hertz/history (o)* |
| *gran/hooke (o)* | *gran/hooke/history (ko)* | *granular* |
| *granular/superellipsoid* | *gw (o)* | *gw/zbl (o)* |
| *harmonic/cut (ko)* | *hbond/dreiding/lj (o)* | *hbond/dreiding/lj/angleoffset (o)* |

continues on next page

Table 3 – continued from previous page

| | | |
|---|---|---|
| *hbond/dreiding/morse (o)* | *hbond/dreiding/morse/angleoffset (o)* | *hdnnp* |
| *hippo (g)* | *ilp/graphene/hbn (t)* | *ilp/tmd (t)* |
| *kolmogorov/crespi/full* | *kolmogorov/crespi/z* | *lambda/input/apip* |
| *lambda/input/csp/apip* | *lambda/zone/apip* | *lcbop (o)* |
| *ldd* | *lebedeva/z* | *lennard/mdf (ko)* |
| *lepton (o)* | *lepton/coul (o)* | *lepton/sphere (o)* |
| *line/lj (o)* | *lj/charmm/coul/charmm (giko)* | *lj/charmm/coul/charmm/implicit (ko)* |
| *lj/charmm/coul/long (gikot)* | *lj/charmm/coul/long/soft (o)* | *lj/charmm/coul/msm (o)* |
| *lj/charmmfsw/coul/charmmfsh (o)* | *lj/charmmfsw/coul/long (ko)* | *lj/class2 (gko)* |
| *lj/class2/coul/cut (ko)* | *lj/class2/coul/cut/soft (o)* | *lj/class2/coul/long (gko)* |
| *lj/class2/coul/long/cs (o)* | *lj/class2/coul/long/soft (o)* | *lj/class2/soft (o)* |
| *lj/cubic (gko)* | *lj/cut (gikot)* | *lj/cut/coul/cut (gko)* |
| *lj/cut/coul/cut/dielectric (o)* | *lj/cut/coul/cut/soft (go)* | *lj/cut/coul/debye (gko)* |
| *lj/cut/coul/debye/dielectric (o)* | *lj/cut/coul/dsf (gko)* | *lj/cut/coul/esp* |
| *lj/cut/coul/long (gikot)* | *lj/cut/coul/long/cs (o)* | *lj/cut/coul/long/dielectric (o)* |
| *lj/cut/coul/long/soft (go)* | *lj/cut/coul/msm (go)* | *lj/cut/coul/msm/dielectric* |
| *lj/cut/coul/wolf (ko)* | *lj/cut/dipole/cut (gko)* | *lj/cut/dipole/long (go)* |
| *lj/cut/dipole/sf (go)* | *lj/cut/soft (o)* | *lj/cut/soft/gapsys* |
| *lj/cut/sphere (ko)* | *lj/cut/thole/long (o)* | *lj/cut/tip4p/cut (o)* |
| *lj/cut/tip4p/long (got)* | *lj/cut/tip4p/long/soft (o)* | *lj/expand (gko)* |
| *lj/expand/coul/long (gko)* | *lj/expand/sphere (o)* | *lj/gromacs (gko)* |
| *lj/gromacs/coul/gromacs (ko)* | *lj/long/coul/long (iot)* | *lj/long/coul/long/dielectric* |
| *lj/long/dipole/long* | *lj/long/tip4p/long (o)* | *lj/mdf (ko)* |
| *lj/pirani (ko)* | *lj/relres (o)* | *lj/spica (gko)* |
| *lj/spica/coul/long (gko)* | *lj/spica/coul/msm (o)* | *lj/sf/dipole/sf (go)* |
| *lj/smooth (gko)* | *lj/smooth/linear (o)* | *lj/switch3/coulgauss/long (ko)* |
| *lj96/cut (gko)* | *local/density (o)* | *lubricate (o)* |
| *lubricate/poly (o)* | *lubricateU* | *lubricateU/poly* |
| *mbx* | *mdpd (g)* | *mdpd/rhosum* |
| *meam (k)* | *meam/ms (k)* | *meam/spline (o)* |
| *meam/sw/spline* | *mesocnt* | *mesocnt/viscous* |
| *mgpt* | *mie/cut (gko)* | *mliap (k)* |
| *mm3/switch3/coulgauss/long (ko)* | *momb (ko)* | *morse (gkot)* |
| *morse/smooth/linear (ko)* | *morse/soft (o)* | *multi/lucy* |
| *multi/lucy/rx (k)* | *nb3b/harmonic (o)* | *nb3b/screened* |
| *nm/cut (ko)* | *nm/cut/coul/cut (ko)* | *nm/cut/coul/long (ko)* |
| *nm/cut/split (o)* | *oxdna/coaxstk* | *oxdna/excv* |
| *oxdna/hbond* | *oxdna/stk* | *oxdna/xstk* |
| *oxdna2/coaxstk* | *oxdna2/dh* | *oxdna2/excv* |
| *oxdna2/hbond* | *oxdna2/stk* | *oxdna2/xstk* |
| *oxdna3/coaxstk* | *oxdna3/dh* | *oxdna3/excv* |
| *oxdna3/hbond* | *oxdna3/stk* | *oxdna3/xstk* |
| *oxrna2/excv* | *oxrna2/hbond* | *oxrna2/dh* |
| *oxrna2/stk* | *oxrna2/xstk* | *oxrna2/coaxstk* |
| *pace (k)* | *pace/extrapolation (k)* | *pace/apip* |
| *pace/fast/apip* | *pace/precise/apip* | *pedone (ko)* |
| *pod (k)* | *peri/eps* | *peri/lps (o)* |
| *peri/pmb (o)* | *peri/ves* | *polymorphic* |
| *python* | *quip* | *rann* |
| *reaxff (ko)* | *rebo (io)* | *rebomos (o)* |

continues on next page

Table 3 – continued from previous page

| | | |
|---|---|---|
| *resquared (go)* | *rheo* | *rheo/solid* |
| *runner* | *saip/metal (t)* | *saip/metal/tmd (t)* |
| *sdpd/taitwater/isothermal* | *smatb* | *smatb/single* |
| *smd/hertz* | *smd/tlsph* | *smd/tri_surface* |
| *smd/ulsph* | *smtbq* | *snap (ik)* |
| *soft (gko)* | *sph/heatconduction (g)* | *sph/idealgas* |
| *sph/lj (g)* | *sph/rhosum* | *sph/taitwater (g)* |
| *sph/taitwater/morris* | *spin/dipole/cut* | *spin/dipole/long* |
| *spin/dmi* | *spin/exchange* | *spin/exchange/biquadratic* |
| *spin/magelec* | *spin/neel* | *srp* |
| *srp/react* | *surf/granular* | *sw (giko)* |
| *sw/angle/table* | *sw/mod (o)* | *table (gko)* |
| *table/rx (k)* | *tdpd* | *tersoff (giko)* |
| *tersoff/mod (gko)* | *tersoff/mod/c (o)* | *tersoff/table (o)* |
| *tersoff/zbl (gko)* | *thole* | *threebody/table (o)* |
| *tip4p/cut (o)* | *tip4p/long (o)* | *tip4p/long/soft (o)* |
| *tri/lj (o)* | *ufm (gkot)* | *uf3 (k)* |
| *vashishta (gko)* | *vashishta/table (o)* | *wf/cut (ko)* |
| *ylz (ko)* | *yukawa (gko)* | *yukawa/colloid (gko)* |
| *zbl (gko)* | | |

## 6.9 Bond styles

All LAMMPS *bond_style* commands. Some styles have accelerated versions. This is indicated by additional letters in parenthesis: g = GPU, i = INTEL, k = KOKKOS, o = OPENMP, t = OPT.

| | | | | |
|---|---|---|---|---|
| *none* | *zero* | *hybrid (k)* | | |
| | | | | |
| *bpm/rotational* | *bpm/spring* | *bpm/spring/plastic* | *class2 (ko)* | *fene (iko)* |
| *fene/expand (ko)* | *fene/nm (ko)* | *gaussian (ko)* | *gromos (ko)* | *harmonic (iko)* |
| *harmonic/restrain (ko)* | *harmonic/shift (ko)* | *harmonic/shift/cut (ko)* | *lepton (o)* | *mesocnt* |
| *mm3 (ko)* | *morse (ko)* | *nonlinear (ko)* | *oxdna/fene* | *oxdna2/fene* |
| *oxdna3/fene* | *oxrna2/fene* | *quartic (ko)* | *quartic/exp (k)* | *rheo/shell* |
| *special* | *table (o)* | | | |

## 6.10 Angle styles

All LAMMPS *angle_style* commands. Some styles have accelerated versions. This is indicated by additional letters in parenthesis: g = GPU, i = INTEL, k = KOKKOS, o = OPENMP, t = OPT.

| | | | | |
|---|---|---|---|---|
| *none* | *zero* | *hybrid (k)* | | |
| *amoeba* | *charmm (iko)* | *class2 (ko)* | *class2/p6 (ko)* | *class2xe (ko)* |
| *cosine (ko)* | *cosine/buck6d (o)* | *cosine/delta (ko)* | *cosine/periodic (ko)* | *cosine/shift (ko)* |
| *cosine/shift/exp (ko)* | *cosine/squared (ko)* | *cosine/squared/restricted (ko)* | *cross (ko)* | *dipole (ko)* |
| *fourier (ko)* | *fourier/simple (ko)* | *gaussian (ko)* | *harmonic (iko)* | *lepton (o)* |
| *mesocnt* | *mm3 (ko)* | *mwlc (ko)* | *quartic (ko)* | *spica (ko)* |
| *table (o)* | | | | |

## 6.11 Dihedral styles

All LAMMPS *dihedral_style* commands. Some styles have accelerated versions. This is indicated by additional letters in parenthesis: g = GPU, i = INTEL, k = KOKKOS, o = OPENMP, t = OPT.

| | | | | |
|---|---|---|---|---|
| *none* | *zero* | *hybrid (k)* | | |
| *charmm (iko)* | *charmmfsw (ko)* | *class2 (ko)* | *class2xe (ko)* | *cosine/shift/exp (ko)* |
| *cosine/squared/restricted (ko)* | *fourier (iko)* | *harmonic (iko)* | *helix (ko)* | *lepton (o)* |
| *multi/harmonic (ko)* | *nharmonic (ko)* | *opls (iko)* | *quadratic (ko)* | *spherical (ko)* |
| *table (o)* | *table/cut (o)* | | | |

## 6.12 Improper styles

All LAMMPS *improper_style* commands. Some styles have accelerated versions. This is indicated by additional letters in parenthesis: g = GPU, i = INTEL, k = KOKKOS, o = OPENMP, t = OPT.

| | | | | |
|---|---|---|---|---|
| *none* | *zero* | *hybrid (k)* | | |
| *amoeba* | *class2 (ko)* | *cossq (ko)* | *cvff (iko)* | *distance (ko)* |
| *distharm (ko)* | *fourier (ko)* | *harmonic (iko)* | *inversion/harmonic (ko)* | *ring (ko)* |
| *sqdistharm (ko)* | *umbrella (ko)* | | | |

## 6.13 KSpace styles

All LAMMPS *kspace_style* solvers. Some styles have accelerated versions. This is indicated by additional letters in parenthesis: g = GPU, i = INTEL, k = KOKKOS, o = OPENMP, t = OPT.

| | | | | |
|---|---|---|---|---|
| *esp* | *ewald (go)* | *ewald/disp* | *ewald/disp/dipole* | *ewald/dipole* |
| *ewald/dipole/spin* | *ewald/electrode* | *msm (o)* | *msm/cg (o)* | *msm/dielectric* |
| *pppm (giko)* | *pppm/cg (o)* | *pppm/dipole* | *pppm/dipole/spin* | *pppm/dielectric* |
| *pppm/disp (io)* | *pppm/disp/tip4p (o)* | *pppm/disp/dielectric* | *pppm/rk* | *pppm/stagger* |
| *pppm/tip4p (o)* | *pppm/dielectric* | *pppm/electrode (i)* | *scafacos* | *zero* |

## 6.14 Dump styles

An alphabetic list of all LAMMPS *dump* commands.

| | | | | | |
|---|---|---|---|---|---|
| *atom* | *atom/adios* | *atom/gz* | *atom/zstd* | *cfg* | *cfg/gz* |
| *cfg/uef* | *cfg/zstd* | *custom* | *custom/adios* | *custom/gz* | *custom/zstd* |
| *dcd* | *extxyz* | *grid* | *grid/vtk* | *h5md* | *image* |
| *local* | *local/gz* | *local/zstd* | *molfile* | *movie* | *netcdf* |
| *netcdf/mpiio* | *vtk* | *xtc* | *xyz* | *xyz/gz* | *xyz/zstd* |
| *yaml* | | | | | |

## 6.15 Removed commands and packages


- *amber2lmp tools*
- *ATC, AWPMD, and POEMS packages*
- *Neighbor style and comm mode multi/old*
- *LAMMPS-GUI source code*
- *GJF formulation in fix langevin*
- *LAMMPS shell*
- *i-PI tool*
- *USER-REAXC package*
- *MPIIO package*
- *MSCG package*
- *LATTE package*
- *Minimize style fire/old*
- *Pair style mesont/tpm, compute style mesont, atom style mesont*
- *Box command*
- *Reset_ids, reset_atom_ids, reset_mol_ids commands*
- *MESSAGE package*
- *REAX package*
- *MEAM package*
- *USER-CUDA package*
- *Compute atom/molecule*
- *Fix ave/spatial and fix ave/spatial/sphere*
- *restart2data tool*


This page lists LAMMPS commands and packages that have been removed from the distribution and provides suggestions for alternatives or replacements. LAMMPS has special dummy styles implemented, that will stop LAMMPS and print a suitable error message in most cases, when a style/command is used that has been removed or will replace the command with the direct alternative (if available) and print a warning.

### 6.15.1 amber2lmp tools

Deprecated since version 11Feb2026.

The tools in the `tools/amber2lmp` folder have been removed because they were unmaintained for a long time and required Python 2 which has been obsolete for a long time. Instead the external AMBER2LAMMPS tool can be used for the same purpose. There is an *Howto_amber2lammps* included in this manual.

### 6.15.2 ATC, AWPMD, and POEMS packages

Deprecated since version 10Sep2025.

The ATC, AWPMD, and POEMS packages are removed because they were unmaintained for a long time and their legacy C++ programming style started to create problems with modern C++ compilers. LAMMPS version 22 July 2025 is the last version that contains them. You have to download and compile this version, if you want to use any of these packages.

#### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

#### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

#### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

#### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

#### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

#### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

#### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

#### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

#### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

#### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

#### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

#### fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

### 6.15.3 Neighbor style and comm mode multi/old

Deprecated since version 10Sep2025.

The original implementation of neighbor style multi and comm mode multi, most recently available under "multi/old" has been removed. The new implementation should be used instead.

### 6.15.4 LAMMPS-GUI source code

Deprecated since version 10Sep2025.

The LAMMPS-GUI sources used to be included in LAMMPS but they are now hosted in their own git repository at https://github.com/akohlmey/lammps-gui/ and the corresponding online documentation is at https://lammps-gui.lammps.org/

### 6.15.5 GJF formulation in fix langevin

Deprecated since version 22Jul2025.

The *gjf* keyword in fix langevin has been removed. The GJF functionality has been moved to its own fix style *fix gjf*.

### 6.15.6 LAMMPS shell

Deprecated since version 29Aug2024.

The LAMMPS shell has been removed from the LAMMPS distribution. Users are encouraged to use the *LAMMPS-GUI* tool instead.

### 6.15.7 i-PI tool

Deprecated since version 27Jun2024.

The i-PI tool has been removed from the LAMMPS distribution. Instead, instructions to install i-PI from PyPI via pip are provided.

### 6.15.8 USER-REAXC package

Deprecated since version 7Feb2024.

The USER-REAXC package has been renamed to *REAXFF*. In the process also the pair style and related fixes were renamed to use the "reaxff" string instead of "reax/c". For a while LAMMPS was maintaining backward compatibility by providing aliases for the styles. These have been removed, so using "reaxff" is now *required*.

### 6.15.9 MPIIO package

Deprecated since version 21Nov2023.

The MPIIO package has been removed from LAMMPS since it was unmaintained for many years and thus not updated to incorporate required changes that had been applied to the corresponding non-MPIIO commands. As a consequence the MPIIO commands had become unreliable and sometimes crashing LAMMPS or corrupting data. Similar functionality is available through the *ADIOS package* and the *NETCDF package*. Also, the *dump_modify nfile or dump_modify fileper* keywords may be used for an efficient way of writing out dump files when running on large numbers of processors. Similarly, the "nfile" and "fileper" keywords exist for restarts: see *restart*, *read_restart*, *write_restart*.

### 6.15.10 MSCG package

Deprecated since version 21Nov2023.

The MSCG package has been removed from LAMMPS since it was unmaintained for many years and instead superseded by the OpenMSCG software of the Voth group at the University of Chicago, which can be used independent from LAMMPS.

### 6.15.11 LATTE package

Deprecated since version 15Jun2023.

The LATTE package with the fix latte command was removed from LAMMPS. This functionality has been superseded by *fix mdi/qm* and *fix mdi/qmmm* from the *MDI package*. These fixes are compatible with several quantum software packages, including LATTE. See the `examples/QUANTUM` dir and the *MDI coupling HOWTO* page. MDI supports running LAMMPS with LATTE as a plugin library (similar to the way fix latte worked), as well as on a different set of MPI processors.

### 6.15.12 Minimize style fire/old

Deprecated since version 8Feb2023.

Minimize style *fire/old* has been removed. Its functionality can be reproduced with style *fire* with specific options. Please see the *min_modify command* documentation for details.

### 6.15.13 Pair style mesont/tpm, compute style mesont, atom style mesont

Deprecated since version 8Feb2023.

Pair style *mesont/tpm*, compute style *mesont*, and atom style *mesont* have been removed from the *MESONT package*. The same functionality is available through *pair style mesocnt*, *bond style mesocnt* and *angle style mesocnt*.

### 6.15.14 Box command

Deprecated since version 22Dec2022.

The *box* command has been removed and the LAMMPS code changed so it won't be needed. If present, LAMMPS will ignore the command and print a warning.

### 6.15.15 Reset_ids, reset_atom_ids, reset_mol_ids commands

Deprecated since version 22Dec2022.

The *reset_ids*, *reset_atom_ids*, and *reset_mol_ids* commands have been folded into the *reset_atoms* command. If present, LAMMPS will replace the commands accordingly and print a warning.

### 6.15.16 MESSAGE package

Deprecated since version 4May2022.

The MESSAGE package has been removed since it was superseded by the *MDI package*. MDI implements the same functionality and in a more general way with direct support for more applications.

### 6.15.17 REAX package

Deprecated since version 4Jan2019.

The REAX package has been removed since it was superseded by the *REAXFF package*. The REAXFF package has been tested to yield equivalent results to the REAX package, offers better performance, supports OpenMP multi-threading via OPENMP, and GPU and threading parallelization through KOKKOS. The new pair styles are not syntax compatible with the removed reax pair style, so input files will have to be adapted. The REAXFF package was originally called USER-REAXC.

### 6.15.18 MEAM package

Deprecated since version 4Jan2019.

The MEAM package in Fortran has been replaced by a C++ implementation. The code in the *MEAM package* is a translation of the Fortran code of MEAM into C++, which removes several restrictions (e.g. there can be multiple instances in hybrid pair styles) and allows for some optimizations leading to better performance. The pair style *meam* has the exact same syntax. For a transition period the C++ version of MEAM was called USER-MEAMC so it could coexist with the Fortran version.

### 6.15.19 USER-CUDA package

Deprecated since version 31May2016.

The USER-CUDA package had been removed, since it had been unmaintained for a long time and had known bugs and problems. Significant parts of the design were transferred to the *KOKKOS package*, which has similar performance characteristics on NVIDIA GPUs. Both, the KOKKOS and the *GPU package* are maintained and allow running LAMMPS with GPU acceleration.

### 6.15.20 Compute atom/molecule

Deprecated since version 11: Dec2015

The atom/molecule command has been removed from LAMMPS since it was superseded by the more general and extensible "chunk infrastructure". Here the system is partitioned in one of many possible ways - including using molecule IDs - through the *compute chunk/atom* command and then summing is done using *compute reduce/chunk* Please refer to the *chunk HOWTO* section for an overview.

### 6.15.21 Fix ave/spatial and fix ave/spatial/sphere

Deprecated since version 11Dec2015.

The fixes ave/spatial and ave/spatial/sphere have been removed from LAMMPS since they were superseded by the more general and extensible "chunk infrastructure". Here the system is partitioned in one of many possible ways through

the *compute chunk/atom* command and then averaging is done using *fix ave/chunk*. Please refer to the *chunk HOWTO* section for an overview.

### 6.15.22 restart2data tool

Deprecated since version 23Nov2013.

The functionality of the restart2data tool has been folded into the LAMMPS executable directly instead of having a separate tool. A combination of the commands *read_restart* and *write_data* can be used to the same effect. For added convenience this conversion can also be triggered by *command-line flags*

CHAPTER

SEVEN

# ACCELERATE PERFORMANCE

This section describes various methods for improving LAMMPS performance for different classes of problems running on different kinds of machines.

There are two thrusts to the discussion that follows. The first is using code options that implement alternate algorithms that can speed-up a simulation. The second is to use one of the several accelerator packages provided with LAMMPS that contain code optimized for certain kinds of hardware, including multicore CPUs and GPUs.

The Benchmark page of the LAMMPS website gives performance results for the various accelerator packages discussed on the *Accelerator packages* page, for several of the standard LAMMPS benchmark problems, as a function of problem size and number of compute nodes, on different hardware platforms.

## 7.1 Benchmarks

Current LAMMPS performance is discussed on the Benchmarks page of the LAMMPS website where timings and parallel efficiency are listed. The page has several sections, which are briefly described below:

- CPU performance on 5 standard problems, strong and weak scaling
- GPU performance on same and related problems
- Comparison of cost of interatomic potentials
- Performance of huge, billion-atom problems

The 5 standard problems are as follow:

1. LJ = atomic fluid, Lennard-Jones potential with 2.5 sigma cutoff (55 neighbors per atom), NVE integration
2. Chain = bead-spring polymer melt of 100-mer chains, FENE bonds and LJ pairwise interactions with a $2^{\frac{1}{6}}$ sigma cutoff (5 neighbors per atom), NVE integration
3. EAM = metallic solid, Cu EAM potential with 4.95 Angstrom cutoff (45 neighbors per atom), NVE integration
4. Chute = granular chute flow, frictional history potential with 1.1 sigma cutoff (7 neighbors per atom), NVE integration
5. Rhodo = rhodopsin protein in solvated lipid bilayer, CHARMM force field with a 10 Angstrom LJ cutoff (440 neighbors per atom), particle-particle particle-mesh (PPPM) for long-range Coulombics, NPT integration

Input files for these 5 problems are provided in the bench directory of the LAMMPS distribution. Each has 32,000 atoms and runs for 100 timesteps. The size of the problem (number of atoms) can be varied using command-line switches as described in the `bench/README` file. This is an easy way to test performance and either strong or weak scalability on your machine.

The bench directory includes a few `log.*` files that show performance of these 5 problems on 1 or 4 cores of Linux desktop. The `bench/FERMI` and `bench/KEPLER` directories have input files and scripts and instructions for running the same (or similar) problems using OpenMP or GPU or Xeon Phi acceleration options. See the `README` files in those

directories and the *Accelerator packages* pages for instructions on how to build LAMMPS and run on that kind of hardware.

The `bench/POTENTIALS` directory has input files which correspond to the table of results on the Potentials section of the Benchmarks web page. So you can also run those test problems on your machine.

The billion-atom section of the Benchmarks web page has performance data for very large benchmark runs of simple Lennard-Jones (LJ) models, which use the `bench/in.lj` input script.

---

For all the benchmarks, a useful metric is the CPU cost per atom per timestep. Since performance scales roughly linearly with problem size and timesteps for all LAMMPS models (i.e. interatomic or coarse-grained potentials), the run time of any problem using the same model (atom style, force field, cutoff, etc) can then be estimated.

Performance on a parallel machine can also be predicted from one-core or one-node timings if the parallel efficiency can be estimated. The communication bandwidth and latency of a particular parallel machine affects the efficiency. On most machines LAMMPS will give a parallel efficiency on these benchmarks above 50% so long as the number of atoms/core is a few 100 or greater, and closer to 100% for large numbers of atoms/core. This is for all-MPI mode with one MPI task per core. For nodes with accelerator options or hardware (OpenMP, GPU, Phi), you should first measure single node performance. Then you can estimate parallel performance for multi-node runs using the same logic as for all-MPI mode, except that now you will typically need many more atoms/node to achieve good scalability.

## 7.2 Measuring performance

### 7.2.1 Factors that influence performance

Before trying to make your simulation run faster, you should understand how it currently performs and where the bottlenecks are. We generally distinguish between serial performance (how fast can a single process do the calculations?) and parallel efficiency (how much faster does a calculation get by using more processes?). There are many factors affecting either and below are some lists discussing some commonly known but also some less known factors.

Factors affecting serial performance (in no specific order):

- CPU hardware: clock rate, cache sizes, CPU architecture (instructions per clock, vectorization support, fused multiply-add support and more)
- RAM speed and number of channels that the CPU can use to access RAM
- Cooling: CPUs can change the CPU clock based on thermal load, thus the degree of cooling can affect the speed of a CPU. Sometimes even the temperature of neighboring compute nodes in a cluster can make a difference.
- Compiler optimization: most of LAMMPS is written to be easy to modify and thus compiler optimization can speed up calculations. However, too aggressive compiler optimization can produce incorrect results or crashes (during compilation or at runtime).
- Source code improvements: styles in the OPT, OPENMP, and INTEL package can be faster than their base implementation due to improved data access patterns, cache efficiency, or vectorization. Compiler optimization is required to take full advantage of these.
- Number and kind of fixes, computes, or variables used during a simulation, especially if they result in collective communication operations
- Pair style cutoffs and system density: calculations get slower the more neighbors are in the neighbor list and thus for which interactions need to be computed. Force fields with pair styles that compute interactions between triples or quadruples of atoms or that use embedding energies or charge equilibration will need to walk the neighbor lists multiple times.
- Neighbor list settings: tradeoff between neighbor list skin (larger skin = more neighbors, more distances to compute before applying the cutoff) and frequency of neighbor list builds (larger skin = fewer neighbor list builds).

- Proximity of per-atom data in physical memory that for atoms that are close in space improves cache efficiency (thus LAMMPS will by default sort atoms in local storage accordingly)
- Using r-RESPA multi-timestepping or a SHAKE or RATTLE fix to constrain bonds with higher-frequency vibrations may allow a larger (outer) timestep and thus fewer force evaluations (usually the most time consuming step in MD) for the same simulated time (with some tradeoff in accuracy).

Factors affecting parallel efficiency (in no specific order):

- Bandwidth and latency of communication between processes. This can vary a lot between processes on the same CPU or physical node and processes on different physical nodes and there vary between different communication technologies (like Ethernet or InfiniBand or other high-speed interconnects)
- Frequency and complexity of communication patterns required
- Number of "work units" (usually correlated with the number of atoms and choice of force field) per MPI-process required for one time step (if this number becomes too small, the cost of communication becomes dominant).
- Choice of parallelization method (MPI-only, OpenMP-only, MPI+OpenMP, MPI+GPU, MPI+GPU+OpenMP)
- Algorithmic complexity of the chosen force field (pair-wise vs. many-body potential, Ewald vs. PPPM vs. (compensated or smoothed) cutoff-Coulomb)
- Communication cutoff: a larger cutoff results in more ghost atoms and thus more data that needs to be communicated
- Frequency of neighbor list builds: during a neighbor list build the domain decomposition is updated and the list of ghost atoms rebuilt which requires multiple global communication steps
- FFT-grid settings and number of MPI processes for kspace style PPPM: PPPM uses parallel 3d FFTs which will drop much faster in parallel efficiency with respect to the number of MPI processes than other parts of the force computation. Thus using MPI+OpenMP parallelization or *run style verlet/split* can improve parallel efficiency by limiting the number of MPI processes used for the FFTs.
- Load (im-)balance: LAMMPS' domain decomposition assumes that atoms are evenly distributed across the entire simulation box. If there are areas of vacuum, this may lead to different amounts of work for different MPI processes. Using the *processors command* to change the spatial decomposition, or MPI+OpenMP parallelization instead of only-MPI to have larger sub-domains, or the (fix) balance command (without or with switching to communication style tiled) to change the sub-domain volumes are all methods that can help to avoid load imbalances.

### 7.2.2 Examples comparing serial performance

Before looking at your own input deck(s), you should get some reference data from a known input so that you know what kind of performance you should expect from your input. For the following we therefore use the `in.rhodo.scaled` input file and `data.rhodo` data file from the `bench` folder. This is a system of 32000 atoms using the CHARMM force field and long-range electrostatics running for 100 MD steps. The performance data is printed at the end of a run and only measures the performance during propagation and excludes the setup phase.

Running with a single MPI process on an AMD Ryzen Threadripper PRO 9985WX CPU (64 cores, 128 threads, base clock: 3.2GHz, max. clock 5.4GHz, L1/L2/L3 cache 5MB/64MB/256MB, 8 DDR5-6400 memory channels) one gets the following performance report:

```
Performance: 1.232 ns/day, 19.476 hours/ns, 7.131 timesteps/s, 228.197 katom-step/s
99.2% CPU use with 1 MPI tasks x 1 OpenMP threads
```

The %CPU value should be at 100% or very close. Lower values would be an indication that there are *other* processes also using the same CPU core and thus invalidating the performance data. The katom-step/s value is best suited for comparisons, since it is fairly independent from the system size. The *in.rhodo.scaled* input can be easily made larger

through replication in the three dimensions by settings variables "x", "y", "z" to values other than 1 from the command line with the "-var" flag. Example:

- 32000 atoms: 228.8 katom-step/s
- 64000 atoms: 231.6 katom-step/s
- 128000 atoms: 231.1 katom-step/s
- 256000 atoms: 226.4 katom-step/s
- 864000 atoms: 229.6 katom-step/s

Comparing to an AMD Ryzen 7 7840HS CPU (8 cores, 16 threads, base clock 3.8GHz, max. clock 5.1GHz, L1/L2/L3 cache 512kB/8MB/16MB, 2 DDR5-5600 memory channels), we get similar single core performance (~220 katom-step/s vs. ~230 katom-step/s) due to the similar clock and architecture:

- 32000 atoms: 219.8 katom-step/s
- 64000 atoms: 222.5 katom-step/s
- 128000 atoms: 216.8 katom-step/s
- 256000 atoms: 221.0 katom-step/s
- 864000 atoms: 221.1 katom-step/s

Switching to an older Intel Xeon E5-2650 v4 CPU (12 cores, 12 threads, base clock 2.2GHz, max. clock 2.9GHz, L1/L2/L3 cache (64kB/256kB/30MB, 4 DDR4-2400 memory channels) leads to a lower performance of approximately 109 katom-step/s due to differences in architecture and clock. In all cases, when looking at multiple runs, the katom-step/s property fluctuates by approximately 1% around the average.

From here on we are looking at the performance for the 256000 atom system only and change several settings incrementally:

1. No compiler optimization GCC (-Og -g): 183.8 katom-step/s
2. Moderate optimization with debug info GCC (-O2 -g): 231.1 katom-step/s
3. Full compiler optimization GCC (-DNDEBUG -O3): 236.0 katom-step/s
4. Aggressive compiler optimization GCC (-O3 -ffast-math -march=native): 239.9 katom-step/s
5. Source code optimization in OPENMP package (1 thread): 266.7 katom-step/s
6. Use *fix nvt* instead of *fix npt* (compute virial only every 50 steps): 272.9 katom-step/s
7. Increase pair style cutoff by 2 Å: 181.2 katom-step/s
8. Use tight PPPM convergence (1.0e-6 instead of 1.0e-4): 161.9 katom-step/s
9. Use Ewald summation instead of PPPM (at 1.0e-4 convergence): 19.9 katom-step/s

The numbers show that gains from aggressive compiler optimizations are rather small in LAMMPS, the data access optimizations in the OPENMP (and OPT) packages are more prominent. On the other side, using more accurate force field settings causes, not unexpectedly, a significant slowdown (to about half the speed). Finally, using regular Ewald summation causes a massive slowdown due to the bad algorithmic scaling with system size.

### 7.2.3 Examples comparing parallel performance

The parallel performance usually goes on top of the serial performance. Using twice as many processors should increase the performance metric by up to a factor of two. With the number of processors $N$ and the serial performance $p_1$ and the performance for $N$ processors $p_N$ we can define a *parallel efficiency* in percent as follows:

$$P_{eff} = \frac{p_N}{p_1 \cdot N} \cdot 100\%$$

For the AMD Ryzen Threadripper PRO 9985WX CPU and the serial simulation settings of point 6. from above, we get the following parallel efficiency data for the 256000 atom system:

- 1 MPI task: 273.6 katom-step/s, $P_{eff}$ = 100%
- 2 MPI tasks: 530.6 katom-step/s, $P_{eff}$ = 97%
- 4 MPI tasks: 1.021 Matom-step/s, $P_{eff}$ = 93%
- 8 MPI tasks: 1.837 Matom-step/s, $P_{eff}$ = 84%
- 16 MPI tasks: 3.574 Matom-step/s, $P_{eff}$ = 82%
- 32 MPI tasks: 6.479 Matom-step/s, $P_{eff}$ = 74%
- 64 MPI tasks: 9.032 Matom-step/s, $P_{eff}$ = 52%
- 128 MPI tasks: 12.03 Matom-step/s, $P_{eff}$ = 34%

The 128 MPI tasks run uses CPU cores from hyper-threading.

For a small system with only 32000 atoms the parallel efficiency drops off earlier when the number of work units is too small relative to the communication overhead:

- 1 MPI task: 270.8 katom-step/s, $P_{eff}$ = 100%
- 2 MPI tasks: 529.3 katom-step/s, $P_{eff}$ = 98%
- 4 MPI tasks: 989.8 katom-step/s, $P_{eff}$ = 91%
- 8 MPI tasks: 1.832 Matom-step/s, $P_{eff}$ = 85%
- 16 MPI tasks: 3.463 Matom-step/s, $P_{eff}$ = 80%
- 32 MPI tasks: 5.970 Matom-step/s, $P_{eff}$ = 69%
- 64 MPI tasks: 7.477 Matom-step/s, $P_{eff}$ = 42%
- 128 MPI tasks: 8.069 Matom-step/s, $P_{eff}$ = 23%

### 7.2.4 Measuring performance of your input deck

The best way to do this is run the your system (actual number of atoms) for a modest number of timesteps (say 100 steps) on several different processor counts, including a single processor if possible. Do this for an equilibrium version of your system, so that the 100-step timings are representative of a much longer run. There is typically no need to run for 1000s of timesteps to get accurate timings; you can simply extrapolate from short runs.

For the set of runs, look at the timing data printed to the screen and log file at the end of each LAMMPS run. The *screen and logfile output* page gives an overview.

Running on one (or a few processors) should give a good estimate of the serial performance and what portions of the timestep are taking the most time. Running the same problem on a few different processor counts should give an estimate of parallel scalability. I.e. if the simulation runs 16x faster on 16 processors, its 100% parallel efficient; if it runs 8x faster on 16 processors, it's 50% efficient.

The most important data to look at in the timing info is the timing breakdown and relative percentages. For example, trying different options for speeding up the long-range solvers will have little impact if they only consume 10% of the run time. If the pairwise time is dominating, you may want to look at GPU or OMP versions of the pair style, as discussed below. Comparing how the percentages change as you increase the processor count gives you a sense of how different operations within the timestep are scaling. If you are using PPPM as Kspace solver, you can turn on an additional output with *kspace_modify fftbench yes* which measures the time spent during PPPM on the 3d FFTs, which can be communication intensive for larger processor counts. This provides an indication whether it is worth trying out alternatives to the default FFT settings for additional performance.

Another important detail in the timing info are the histograms of atoms counts and neighbor counts. If these vary widely across processors, you have a load-imbalance issue. This often results in inaccurate relative timing data, because processors have to wait when communication occurs for other processors to catch up. Thus the reported times for "Communication" or "Other" may be higher than they really are, due to load-imbalance. If this is an issue, you can use the *timer sync* command to obtain synchronized timings.

## 7.3 General tips

**Note**

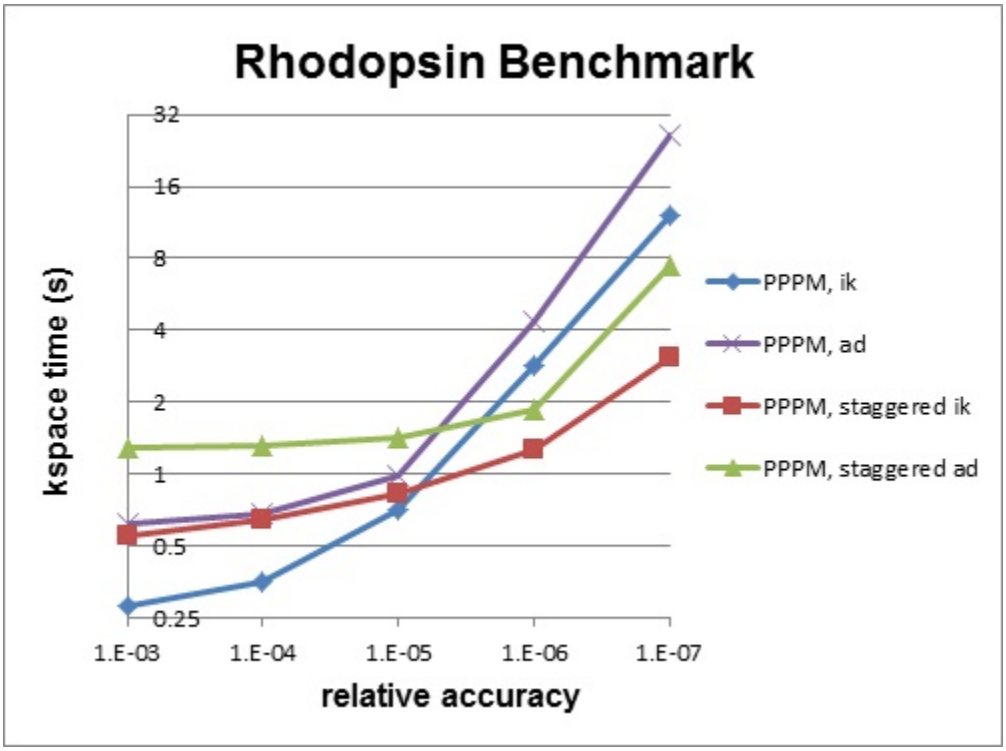
this page is still a work in progress

Here is a list of general ideas for improving simulation performance. Most of them are only applicable to certain models and certain bottlenecks in the current performance, so let the timing data you generate be your guide. It is hard, if not impossible, to predict how much difference these options will make, since it is a function of problem size, number of processors used, and your machine. There is no substitute for identifying performance bottlenecks, and trying out various options.

- rRESPA
- Two-FFT PPPM
- Staggered PPPM
- single vs double PPPM
- partial charge PPPM
- verlet/split run style
- processor command for proc layout and numa layout
- load-balancing: balance and fix balance

Two-FFT PPPM, also called *analytic differentiation* or *ad* PPPM, uses 2 FFTs instead of the 4 FFTs used by the default *ik differentiation* PPPM. However, 2-FFT PPPM also requires a slightly larger mesh size to achieve the same accuracy as 4-FFT PPPM. For problems where the FFT cost is the performance bottleneck (typically large problems running on many processors), 2-FFT PPPM may be faster than 4-FFT PPPM.

Staggered PPPM performs calculations using two different meshes, one shifted slightly with respect to the other. This can reduce force aliasing errors and increase the accuracy of the method, but also doubles the amount of work required. For high relative accuracy, using staggered PPPM allows one to half the mesh size in each dimension as compared to regular PPPM, which can give around a 4x speedup in the kspace time. However, for low relative accuracy, using staggered PPPM gives little benefit and can be up to 2x slower in the kspace time. For example, the rhodopsin benchmark was run on a single processor, and results for kspace time vs. relative accuracy for the different methods are shown in the figure below. For this system, staggered PPPM (using ik differentiation) becomes useful when using a relative accuracy of slightly greater than 1e-5 and above.




**Note**

Using staggered PPPM may not give the same increase in accuracy of energy and pressure as it does in forces, so some caution must be used if energy and/or pressure are quantities of interest, such as when using a barostat.

## 7.4 Accelerator packages

Accelerated versions of various *pair_style*, *fixes*, *computes*, and other commands have been added to LAMMPS, which will typically run faster than the standard non-accelerated versions. Some require appropriate hardware to be present on your system, e.g. GPUs.

All of these commands are in packages provided with LAMMPS. An overview of packages is give on the *Packages* doc pages.

These are the accelerator packages currently in LAMMPS:

| | |
|---|---|
| *GPU Package* | for GPUs via CUDA, OpenCL, or ROCm HIP |
| *INTEL Package* | for Intel CPUs |
| *KOKKOS Package* | for GPUs and threading via the Kokkos library |
| *OPENMP Package* | for OpenMP threading and generic CPU optimizations |
| *OPT Package* | generic CPU optimizations |

### 7.4.1 GPU package

The GPU package was developed by Mike Brown while at SNL and ORNL (now at Intel Corp.) and his collaborators, particularly Trung Nguyen (now at Northwestern). Support for AMD GPUs via HIP was added by Vsevolod Nikolskiy and coworkers at HSE University.

The GPU package provides GPU versions of many pair styles and for parts of the *kspace_style pppm* for long-range Coulombics. It has the following general features:

- It is designed to exploit common GPU hardware configurations where one or more GPUs are coupled to many cores of one or more multicore CPUs, e.g. within a node of a parallel machine.
- Atom-based data (e.g. coordinates, forces) are moved back-and-forth between the CPU(s) and GPU every timestep.
- Neighbor lists can be built on the CPU or on the GPU
- The charge assignment and force interpolation portions of PPPM can be run on the GPU. The FFT portion, which requires MPI communication between processors, runs on the CPU.
- Force computations of different style (pair vs. bond/angle/dihedral/improper) can be performed concurrently on the GPU and CPU(s), respectively.
- It allows for GPU computations to be performed in single or double precision, or in mixed-mode precision, where pairwise forces are computed in single precision, but accumulated into double-precision force vectors.
- LAMMPS-specific code is in the GPU package. It makes calls to a generic GPU library in the lib/gpu directory. This library provides either Nvidia support, AMD support, or more general OpenCL support (for Nvidia GPUs, AMD GPUs, Intel GPUs, and multicore CPUs). so that the same functionality is supported on a variety of hardware.

### Required hardware/software

To compile and use this package in CUDA mode, you currently need to have an NVIDIA GPU and install the corresponding NVIDIA CUDA toolkit software on your system (this is only tested on Linux and unsupported on Windows):

- Check if you have an NVIDIA GPU: `cat /proc/driver/nvidia/gpus/\*/information`
- Go to https://developer.nvidia.com/cuda-downloads
- Install a driver and toolkit appropriate for your system (SDK is not necessary)
- Run `lammps/lib/gpu/nvc_get_devices` (after building the GPU library, see below) to list supported devices and properties

To compile and use this package in OpenCL mode, you currently need to have the OpenCL headers and the (vendor neutral) OpenCL library installed. In OpenCL mode, the acceleration depends on having an OpenCL Installable Client Driver (ICD) installed. There can be multiple of them for the same or different hardware (GPUs, CPUs, Accelerators) installed at the same time. OpenCL refers to those as 'platforms'. The GPU library will try to auto-select the best suitable platform, but this can be overridden using the platform option of the *package* command. run `lammps/lib/gpu/ocl_get_devices` to get a list of available platforms and devices with a suitable ICD available.

To compile and use this package for Intel GPUs, OpenCL or the Intel oneAPI HPC Toolkit can be installed using linux package managers. The latter also provides optimized C++, MPI, and many other libraries and tools. See:

- https://software.intel.com/content/www/us/en/develop/tools/oneapi/hpc-toolkit/download.html

If you do not have a discrete GPU card installed, this package can still provide significant speedups on some CPUs that include integrated GPUs. Additionally, for many macs, OpenCL is already included with the OS and Makefiles are available in the `lib/gpu` directory.

To compile and use this package in HIP mode, you have to have the AMD ROCm software installed. Versions of ROCm older than 3.5 are currently deprecated by AMD.

### Building LAMMPS with the GPU package

See the *Build extras* page for instructions.

### Run with the GPU package from the command-line

The `mpirun` or `mpiexec` command sets the total number of MPI tasks used by LAMMPS (one or multiple per compute node) and the number of MPI tasks used per node. E.g. the `mpirun` command in MPICH does this via its `-np` and `-ppn` switches. Ditto for OpenMPI via `-np` and `-npernode`.

When using the GPU package, you cannot assign more than one GPU to a single MPI task. However multiple MPI tasks can share the same GPU, and in many cases it will be more efficient to run this way. Likewise it may be more efficient to use less MPI tasks/node than the available # of CPU cores. Assignment of multiple MPI tasks to a GPU will happen automatically if you create more MPI tasks/node than there are GPUs/mode. E.g. with 8 MPI tasks/node and 2 GPUs, each GPU will be shared by 4 MPI tasks.

The GPU package also has limited support for OpenMP for both multi-threading and vectorization of routines that are run on the CPUs. This requires that the GPU library and LAMMPS are built with flags to enable OpenMP support (e.g. `-fopenmp`). Some styles for time integration are also available in the GPU package. These run completely on the CPUs in full double precision, but exploit multi-threading and vectorization for faster performance.

Use the `-sf gpu` *command-line switch*, which will automatically append "gpu" to styles that support it. Use the `-pk gpu Ng` *command-line switch* to set `Ng` = # of GPUs/node to use. If `Ng` is 0, the number is selected automatically as the number of matching GPUs that have the highest number of compute cores.

```
# 1 MPI task uses 1 GPU
lmp_machine -sf gpu -pk gpu 1 -in in.script

# 12 MPI tasks share 2 GPUs on a single 16-core (or whatever) node
mpirun -np 12 lmp_machine -sf gpu -pk gpu 2 -in in.script

# ditto on 4 16-core nodes
mpirun -np 48 -ppn 12 lmp_machine -sf gpu -pk gpu 2 -in in.script
```

Note that if the `-sf gpu` switch is used, it also issues a default *package gpu 0* command, which will result in automatic selection of the number of GPUs to use.

Using the `-pk` switch explicitly allows for setting of the number of GPUs/node to use and additional options. Its syntax is the same as the `package gpu` command. See the *package* command page for details, including the default values used for all its options if it is not specified.

Note that the default for the *package gpu* command is to set the Newton flag to "off" pairwise interactions. It does not affect the setting for bonded interactions (LAMMPS default is "on"). The "off" setting for pairwise interaction is currently required for GPU package pair styles.

### Run with the GPU package by editing an input script

The discussion above for the `mpirun` or `mpiexec` command, MPI tasks/node, and use of multiple MPI tasks/GPU is the same.

Use the *suffix gpu* command, or you can explicitly add an "gpu" suffix to individual styles in your input script, e.g.

```
pair_style lj/cut/gpu 2.5
```

You must also use the *package gpu* command to enable the GPU package, unless the `-sf gpu` or `-pk gpu` *command-line switches* were used. It specifies the number of GPUs/node to use, as well as other options.

### Speed-up to expect

The performance of a GPU versus a multicore CPU is a function of your hardware, which pair style is used, the number of atoms/GPU, and the precision used on the GPU (double, single, mixed). Using the GPU package in OpenCL mode on CPUs (which uses vectorization and multithreading) is usually resulting in inferior performance compared to using LAMMPS' native threading and vectorization support in the OPENMP and INTEL packages.

See the Benchmark page of the LAMMPS website for performance of the GPU package on various hardware, including the Titan HPC platform at ORNL.

You should also experiment with how many MPI tasks per GPU to use to give the best performance for your problem and machine. This is also a function of the problem size and the pair style being using. Likewise, you should experiment with the precision setting for the GPU library to see if single or mixed precision will give accurate results, since they will typically be faster.

MPI parallelism typically outperforms OpenMP parallelism, but in some cases using fewer MPI tasks and multiple OpenMP threads with the GPU package can give better performance. 3-body potentials can often perform better with multiple OMP threads because the inter-process communication is higher for these styles with the GPU package in order to allow deterministic results.

### Guidelines for best performance

- Using multiple MPI tasks (2-10) per GPU will often give the best performance, as allowed by most multicore CPU/GPU configurations. Using too many MPI tasks will result in worse performance due to growing overhead with the growing number of MPI tasks.
- If the number of particles per MPI task is small (e.g. 100s of particles), it can be more efficient to run with fewer MPI tasks per GPU, even if you do not use all the cores on the compute node.
- The *package gpu* command has several options for tuning performance. Neighbor lists can be built on the GPU or CPU. Force calculations can be dynamically balanced across the CPU cores and GPUs. GPU-specific settings can be made which can be optimized for different hardware. See the *package* command page for details.
- As described by the *package gpu* command, GPU accelerated pair styles can perform computations asynchronously with CPU computations. The "Pair" time reported by LAMMPS will be the maximum of the time required to complete the CPU pair style computations and the time required to complete the GPU pair style computations. Any time spent for GPU-enabled pair styles for computations that run simultaneously with *bond*, *angle*, *dihedral*, *improper*, and *long-range* calculations will not be included in the "Pair" time.
- Since only part of the pppm kspace style is GPU accelerated, it may be faster to only use GPU acceleration for Pair styles with long-range electrostatics. See the "pair/only" keyword of the *package command* for a shortcut to do that. The distribution of work between kspace on the CPU and non-bonded interactions on the GPU can be balanced through adjusting the coulomb cutoff without loss of accuracy.
- When the *mode* setting for the package gpu command is force/neigh, the time for neighbor list calculations on the GPU will be added into the "Pair" time, not the "Neigh" time. An additional breakdown of the times required for various tasks on the GPU (data copy, neighbor calculations, force computations, etc) are output only with the LAMMPS screen output (not in the log file) at the end of each run. These timings represent total time spent on the GPU for each routine, regardless of asynchronous CPU calculations.
- The output section "GPU Time Info (average)" reports "Max Mem / Proc". This is the maximum memory used at one time on the GPU for data storage by a single MPI process.

### Restrictions

When using *hybrid pair styles*, the neighbor list must be generated on the host instead of the GPU and thus the potential GPU acceleration is reduced.

### 7.4.2 INTEL package

The INTEL package currently has no official maintainer. It accelerates simulations on Intel CPUs by running in single, mixed, or double precision with vectorization and multi-threading.

#### Currently Available INTEL Styles

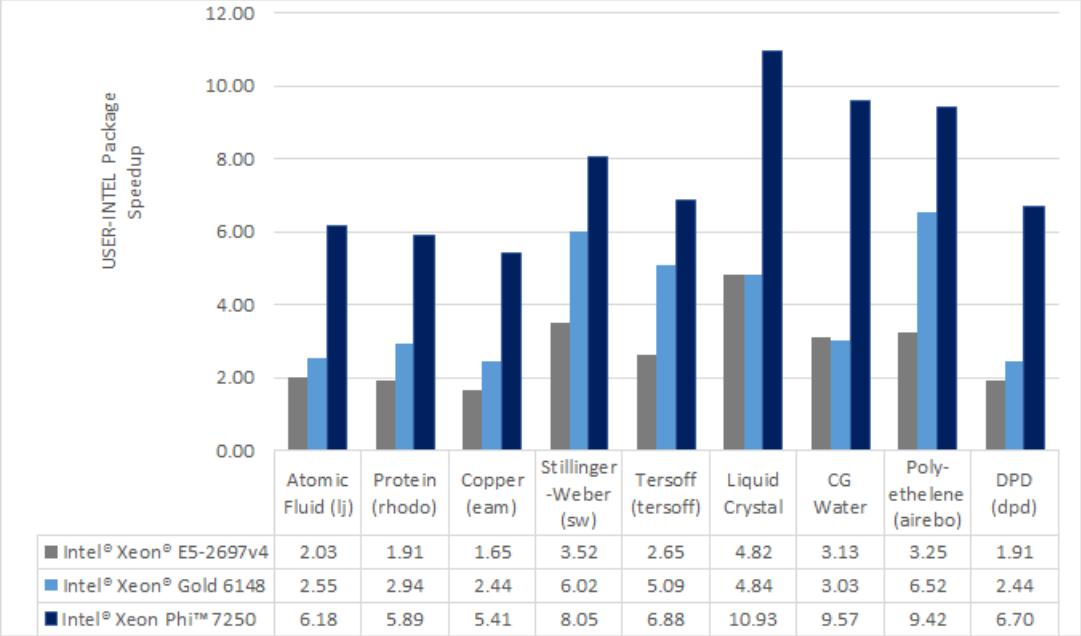
- Angle Styles: *charmm*, *harmonic*
- Bond Styles: *fene*, *harmonic*
- Dihedral Styles: *charmm*, *fourier*, *harmonic*, *opls*
- Fixes: *nve*, *npt*, *nvt*, *nvt/sllod*, *nve/asphere*, *electrode/conp*, *electrode/conq*, *electrode/thermo*
- Improper Styles: *cvff*, *harmonic*
- Pair Styles: *airebo*, *airebo/morse*, *buck/coul/cut*, *buck/coul/long*,:doc:*buck <pair_buck>*, *dpd*, *eam*, *eam/alloy*, *eam/fs*, *gayberne*, *lj/charmm/coul/charmm*, *lj/charmm/coul/long*, *lj/cut*, *lj/cut/coul/long*, *lj/long/coul/long*, *rebo*, *snap*, *sw*, *tersoff*
- K-Space Styles: *pppm*, *pppm/dispppppm/electrode*

**Warning**

None of the styles in the INTEL package currently support computing per-atom stress. If any compute or fix in your input requires it, LAMMPS will abort with an error message.

#### Speed-up to expect

The speedup will depend on your simulation, the hardware, which styles are used, the number of atoms, and the floating-point precision mode. Performance improvements are shown compared to LAMMPS *without using other acceleration packages* as these are under active development (and subject to performance changes). The measurements were performed using the input files available in the `src/INTEL/TEST` directory with the provided run script. These are scalable in size; the results given are with 512K particles (524K for Liquid Crystal). Most of the simulations are standard LAMMPS benchmarks (indicated by the filename extension in parenthesis) with modifications to the run length and to add a warm-up run.




| | Atomic Fluid (lj) | Protein (rhodo) | Copper (eam) | Stillinger-Weber (sw) | Tersoff (tersoff) | Liquid Crystal | CG Water | Poly-ethelene (airebo) | DPD (dpd) |
|---|---|---|---|---|---|---|---|---|---|
| Intel® Xeon® E5-2697v4 | 2.03 | 1.91 | 1.65 | 3.52 | 2.65 | 4.82 | 3.13 | 3.25 | 1.91 |
| Intel® Xeon® Gold 6148 | 2.55 | 2.94 | 2.44 | 6.02 | 5.09 | 4.84 | 3.03 | 6.52 | 2.44 |
| Intel® Xeon Phi™ 7250 | 6.18 | 5.89 | 5.41 | 8.05 | 6.88 | 10.93 | 9.57 | 9.42 | 6.70 |

Results are speedups obtained on Intel Xeon E5-2697v4 processors (code-named Broadwell) and Intel Xeon Gold 6148 processors (code-named Skylake) with "June 2017" LAMMPS built with Intel Parallel Studio 2017 update 2. Results are with 1 MPI task per physical core. See `src/INTEL/TEST/README` for the raw simulation rates and instructions to reproduce.

---

### Accuracy and order of operations

In most molecular dynamics software, parallelization parameters (# of MPI, OpenMP, and vectorization) can change the results due to changing the order of operations with finite-precision calculations. The INTEL package is deterministic. This means that the results should be reproducible from run to run with the *same* parallel configurations and when using deterministic libraries or library settings (MPI, OpenMP, FFT). However, there are differences in the INTEL package that can change the order of operations compared to LAMMPS without acceleration:

- Neighbor lists can be created in a different order
- Bins used for sorting atoms can be oriented differently
- The default stencil order for PPPM is 7. By default, LAMMPS will calculate other PPPM parameters to fit the desired accuracy with this order
- The *newton* setting applies to all atoms, not just atoms shared between MPI tasks
- Vectorization can change the order for adding pairwise forces
- When using the `-DLMP_USE_MKL_RNG` define (all included intel optimized makefiles do) at build time, the random number generator for dissipative particle dynamics (`pair style dpd/intel`) uses the Mersenne Twister generator included in the Intel MKL library (that should be more robust than the default Masaglia random number generator)

The precision mode (described below) used with the INTEL package can change the *accuracy* of the calculations. For the default *mixed* precision option, calculations between pairs or triplets of atoms are performed in single precision, intended to be within the inherent error of MD simulations. All accumulation is performed in double precision to prevent the error from growing with the number of atoms in the simulation. *Single* precision mode should not be used without appropriate validation.

---

### Quick Start for Experienced Users

LAMMPS should be built with the INTEL package installed. Simulations should be run with 1 MPI task per physical *core*, not *hardware thread*.

- Edit `src/MAKE/OPTIONS/Makefile.intel_cpu_intelmpi` as necessary.
- Set the environment variable `KMP_BLOCKTIME=0`
- `-pk intel 0 omp $t -sf intel` added to LAMMPS command-line
- `$t` should be 2 for Intel Xeon CPUs
- For some of the simple 2-body potentials without long-range electrostatics, performance and scalability can be better with the `newton off` setting added to the input script
- For simulations on higher node counts, add `processors * * * grid numa` to the beginning of the input script for better scalability
- If using `kspace_style pppm` in the input script, add `kspace_modify diff ad` for better performance

For simulations using `kspace_style pppm` on Intel CPUs supporting AVX-512:

- Add `kspace_modify diff ad` to the input script
- The command-line option should be changed to `-pk intel 0 omp $r lrt yes -sf intel` where `$r` is the number of threads minus 1.
- Do not use thread affinity (set `KMP_AFFINITY=none`)
- The `newton off` setting may provide better scalability

---

### Required hardware/software

When using Intel compilers version 16.0 or later is required.

Although any compiler can be used with the INTEL package, currently, vectorization directives are disabled by default when not using Intel compilers due to lack of standard support and observations of decreased performance. The OpenMP standard now supports directives for vectorization and we plan to transition the code to this standard once it is available in most compilers. We expect this to allow improved performance and support with other compilers.

### Notes about Simultaneous Multithreading

Modern CPUs often support Simultaneous Multithreading (SMT). On Intel processors, this is called Hyper-Threading (HT) technology. SMT is hardware support for running multiple threads efficiently on a single core. *Hardware threads* or *logical cores* are often used to refer to the number of threads that are supported in hardware. For example, the Intel Xeon E5-2697v4 processor is described as having 36 cores and 72 threads. This means that 36 MPI processes or OpenMP threads can run simultaneously on separate cores, but that up to 72 MPI processes or OpenMP threads can be running on the CPU without costly operating system context switches.

Molecular dynamics simulations will often run faster when making use of SMT. If a thread becomes stalled, for example because it is waiting on data that has not yet arrived from memory, another thread can start running so that the CPU pipeline is still being used efficiently. Although benefits can be seen by launching a MPI task for every hardware thread, for multinode simulations, we recommend that OpenMP threads are used for SMT instead, either with the INTEL package, *OPENMP package*, or *KOKKOS package*. In the example above, up to 36X speedups can be observed by using all 36 physical cores with LAMMPS. By using all 72 hardware threads, an additional 10-30% performance gain can be achieved.

The BIOS on many platforms allows SMT to be disabled, however, we do not recommend this on modern processors as there is little to no benefit for any software package in most cases. The operating system will report every hardware

thread as a separate core allowing one to determine the number of hardware threads available. On Linux systems, this information can normally be obtained with:

```
cat /proc/cpuinfo
```

### Building LAMMPS with the INTEL package

See the *Build extras* page for instructions. Some additional details are covered here.

For building with make, several example Makefiles for building with the Intel compiler are included with LAMMPS in the `src/MAKE/OPTIONS/` directory:

```
Makefile.intel_cpu_intelmpi  # Intel Compiler, Intel MPI
Makefile.intel_cpu_mpich     # Intel Compiler, MPICH
Makefile.intel_cpu_openmpi   # Intel Compiler, OpenMPI
```

For users with recent installations of Intel Parallel Studio, the process can be as simple as:

```
make yes-intel
source /opt/intel/parallel_studio_xe_2016.3.067/psxevars.sh
# or psxevars.csh for C-shell
make intel_cpu_intelmpi
```

The general requirements for Makefiles with the INTEL package are as follows. When using Intel compilers, `-restrict` is required and `-qopenmp` is highly recommended for `CCFLAGS` and `LINKFLAGS`. `CCFLAGS` should include `-DLMP_INTEL_USELRT` (unless POSIX Threads are not supported in the build environment) and `-DLMP_USE_MKL_RNG` (unless Intel Math Kernel Library (MKL) is not available in the build environment). For Intel compilers, `LIB` should include `-ltbbmalloc` or if the library is not available, `-DLMP_INTEL_NO_TBB` can be added to `CCFLAGS`. Other recommended `CCFLAG` options for best performance are `-O2 -fno-alias -ansi-alias -qoverride-limits fp-model fast=2 -no-prec-div`.

**Note**

See the `src/INTEL/README` file for additional flags that might be needed for best performance on Intel server processors code-named "Skylake".

**Note**

The vectorization and math capabilities can differ depending on the CPU. For Intel compilers, the `-x` flag specifies the type of processor for which to optimize. `-xHost` specifies that the compiler should build for the processor used for compiling. For fourth generation Intel Xeon (v4/Broadwell) processors, `-xCORE-AVX2` should be used. For older Intel Xeon processors, `-xAVX` will perform best in general for the different simulations in LAMMPS. The default in most of the example Makefiles is to use `-xHost`, however this should not be used when cross-compiling.

### Running LAMMPS with the INTEL package

Running LAMMPS with the INTEL package is similar to normal use with the exceptions that one should 1) specify that LAMMPS should use the INTEL package, 2) specify the number of OpenMP threads, and 3) optionally specify the specific LAMMPS styles that should use the INTEL package. 1) and 2) can be performed from the command-line or by editing the input script. 3) requires editing the input script. Advanced performance tuning options are also described below to get the best performance.

When running on a single node, best performance is normally obtained by using 1 MPI task per physical core and additional OpenMP threads with SMT. For Intel Xeon processors, 2 OpenMP threads should be used for SMT. In cases where the user specifies that LRT mode is used (described below), 1 or 3 OpenMP threads should be used. For multi-node runs, using 1 MPI task per physical core will often perform best, however, depending on the machine and scale, users might get better performance by decreasing the number of MPI tasks and using more OpenMP threads. For performance, the product of the number of MPI tasks and OpenMP threads should not exceed the number of available hardware threads in almost all cases.

### Run with the INTEL package from the command-line

To enable INTEL optimizations for all available styles used in the input script, the `-sf intel` *command-line switch* can be used without any requirement for editing the input script. This switch will automatically append "intel" to styles that support it. It also invokes a default command: *package intel 1*. This package command is used to set options for the INTEL package. The default package command will specify that INTEL calculations are performed in mixed precision and that the number of OpenMP threads is specified by the OMP_NUM_THREADS environment variable.

You can specify different options for the INTEL package by using the `-pk intel` *command-line switch* with keyword/value pairs as specified in the documentation. Common options to the INTEL package include *omp* to override any `OMP_NUM_THREADS` setting and specify the number of OpenMP threads, *mode* to set the floating-point precision mode, and *lrt* to enable Long-Range Thread mode as described below. See the *package intel* command for details, including the default values used for all its options if not specified, and how to set the number of OpenMP threads via the `OMP_NUM_THREADS` environment variable if desired.

Examples (see documentation for your MPI/Machine for differences in launching MPI applications):

```
# 2 nodes, 36 MPI tasks/node, $OMP_NUM_THREADS OpenMP Threads
mpirun -np 72 -ppn 36 lmp_machine -sf intel -in in.script

# use 2 OpenMP threads for each task, use double precision
mpirun -np 72 -ppn 36 lmp_machine -sf intel -in in.script \
       -pk intel 0 omp 2 mode double
```

### Or run with the INTEL package by editing an input script

As an alternative to adding command-line arguments, the input script can be edited to enable the INTEL package. This requires adding the *package intel* command to the top of the input script. For the second example above, this would be:

```
package intel 0 omp 2 mode double
```

To enable the INTEL package only for individual styles, you can add an "intel" suffix to the individual style, e.g.:

```
pair_style lj/cut/intel 2.5
```

Alternatively, the *suffix intel* command can be added to the input script to enable INTEL styles for the commands that follow in the input script.

### Tuning for Performance

**Note**

The INTEL package will perform better with modifications to the input script when *PPPM* is used: *kspace_modify diff ad* should be added to the input script.

Long-Range Thread (LRT) mode is an option to the *package intel* command that can improve performance when using *PPPM* for long-range electrostatics on processors with SMT. It generates an extra pthread for each MPI task. The

thread is dedicated to performing some of the PPPM calculations and MPI communications. This feature requires setting the pre-processor flag `-DLMP_INTEL_USELRT` in the makefile when compiling LAMMPS. It is unset in the default makefiles (`Makefile.mpi` and `Makefile.serial`) but it is set in all makefiles tuned for the INTEL package. On Intel Xeon processors, using this mode might result in better performance when using multiple nodes, depending on the specific machine configuration. To enable LRT mode, specify that the number of OpenMP threads is one less than would normally be used for the run and add the `lrt yes` option to the `-pk` command-line suffix or "package intel" command. For example, if a run would normally perform best with "-pk intel 0 omp 4", instead use `-pk intel 0 omp 3 lrt yes`. When using LRT, you should set the environment variable `KMP_AFFINITY=none`.

**Note**

Changing the *newton* setting to off can improve performance and/or scalability for simple 2-body potentials such as lj/cut or when using LRT mode on processors supporting AVX-512.

Not all styles are supported in the INTEL package. You can mix the INTEL package with styles from the *OPT* package or the *OPENMP package*. Of course, this requires that these packages were installed at build time. This can performed automatically by using `-sf hybrid intel opt` or `-sf hybrid intel omp` command-line options. Alternatively, the "opt" and "omp" suffixes can be appended manually in the input script. For the latter, the *package omp* command must be in the input script or the `-pk omp Nt` *command-line switch* must be used where `Nt` is the number of OpenMP threads. The number of OpenMP threads should not be set differently for the different packages. Note that the *suffix hybrid intel omp* command can also be used within the input script to automatically append the "omp" suffix to styles when INTEL styles are not available.

**Note**

For simulations on higher node counts, add *processors* * * * *grid numa* to the beginning of the input script for better scalability.

When running on many nodes, performance might be better when using fewer OpenMP threads and more MPI tasks. This will depend on the simulation and the machine. Using the *verlet/split* run style might also give better performance for simulations with *PPPM* electrostatics. Note that this is an alternative to LRT mode and the two cannot be used together.

### Restrictions

None of the pair styles in the INTEL package currently support the "inner", "middle", "outer" options for rRESPA integration via the *run_style respa* command; only the "pair" option is supported.

### References


- Brown, W.M., Carrillo, J.-M.Y., Mishra, B., Gavhane, N., Thakkar, F.M., De Kraker, A.R., Yamada, M., Ang, J.A., Plimpton, S.J., "Optimizing Classical Molecular Dynamics in LAMMPS", in Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition, J. Jeffers, J. Reinders, A. Sodani, Eds. Morgan Kaufmann.
- Brown, W. M., Semin, A., Hebenstreit, M., Khvostov, S., Raman, K., Plimpton, S.J. Increasing Molecular Dynamics Simulation Rates with an 8-Fold Increase in Electrical Power Efficiency. 2016 High Performance Computing, Networking, Storage and Analysis, SC16: International Conference (pp. 82-95).
- Brown, W.M., Carrillo, J.-M.Y., Gavhane, N., Thakkar, F.M., Plimpton, S.J. Optimizing Legacy Molecular Dynamics Software with Directive-Based Offload. Computer Physics Communications. 2015. 195: p. 95-101.

### 7.4.3 KOKKOS package

Kokkos is a templated C++ library that provides abstractions to allow a single implementation of an application kernel (e.g. a pair style) to run efficiently on different kinds of hardware, such as GPUs, Intel Xeon Phis, or many-core CPUs. Kokkos maps the C++ kernel onto different back end languages such as CUDA, OpenMP, or Pthreads. The Kokkos library also provides data abstractions to adjust (at compile time) the memory layout of data structures like 2d and 3d arrays to optimize performance on different hardware. For more information on Kokkos, see the Kokkos GitHub page.

The LAMMPS KOKKOS package contains versions of pair, fix, and atom styles that use data structures and macros provided by the Kokkos library, which is included with LAMMPS in /lib/kokkos. The KOKKOS package was developed primarily by Christian Trott (Sandia) and Stan Moore (Sandia) with contributions of various styles by others, including Sikandar Mashayak (UIUC), Ray Shan (Sandia), and Dan Ibanez (Sandia). For more information on developing using Kokkos abstractions see the Kokkos Wiki.

**Note**

The Kokkos library is under active development and tracking the availability of accelerator hardware, so is the KOKKOS package in LAMMPS. This means that only a certain range of versions of the Kokkos library are compatible with the KOKKOS package of a certain range of LAMMPS versions. For that reason LAMMPS comes with a bundled version of the Kokkos library that has been validated on multiple platforms and may contain selected back-ported bug fixes from upstream Kokkos versions. While it is possible to build LAMMPS with an external version of Kokkos, it is untested and may result in incorrect execution or crashes.

Kokkos currently provides full support for 4 modes of execution (per MPI task). These are Serial (MPI-only for CPUs and Intel Phi), OpenMP (threading for many-core CPUs and Intel Phi), CUDA (for NVIDIA GPUs) and HIP (for AMD GPUs). Additional modes (e.g. OpenMP target, Intel data center GPUs) are under development. You choose the mode at build time to produce an executable compatible with a specific hardware.

The following compatibility notes have been last updated for LAMMPS version 11 February 2026 and its bundled Kokkos library version 5.0.2.

**C++20 support**

Kokkos requires using a compiler that supports the C++20 standard. For some compilers, it may be necessary to add a flag to enable C++20 support. For example, the GNU compiler uses the `-std=c++20` flag. For a list of compilers that have been tested with the Kokkos library, see the requirements document of the Kokkos Wiki.

**NVIDIA CUDA support**

To build with Kokkos support for NVIDIA GPUs, the NVIDIA CUDA toolkit software version 12.2 or later must be installed on your system. See the discussion for the *GPU package* for details of how to check and do this.

**AMD ROCm (HIP) support**

To build with Kokkos support for AMD GPUs, the AMD ROCm toolkit software version 6.2.0 or later must be installed on your system.

**Intel Data Center GPU support**

Support for Kokkos with Intel Data Center GPU accelerators (formerly known under the code name "Ponte Vecchio") in LAMMPS is still a work in progress. Minimum required version is the Intel LLVM (not classic) compiler version 2024.1. The LAMMPS developers do not regularly test the status of this, so please contact the LAMMPS developers if you run into problems.

**CUDA and MPI library compatibility**

Kokkos with CUDA currently implicitly assumes that the MPI library is GPU-aware. This is not always the case, especially when using pre-compiled MPI libraries provided by a Linux distribution. This is not a problem when using only a single GPU with a single MPI rank. When running with multiple MPI ranks, you may see segmentation faults without GPU-aware MPI support. These can be avoided by adding the flags *-pk kokkos gpu/aware off* to the LAMMPS command-line or by using the command *package kokkos gpu/aware off* in the input file.

**Using multiple MPI ranks per GPU**

Unlike with the GPU package, there are limited benefits from using multiple MPI processes per GPU with KOKKOS. But when doing this it is **required** to enable CUDA MPS (Multi-Process Service :: GPU Deployment and Management Documentation ) to get acceptable performance.

### Building LAMMPS with the KOKKOS package

See the *Build extras* page for instructions.

### Running LAMMPS with the KOKKOS package

All Kokkos operations occur within the context of an individual MPI task running on a single node of the machine. The total number of MPI tasks used by LAMMPS (one or multiple per compute node) is set in the usual manner via the `mpirun` or `mpiexec` commands, and is independent of Kokkos. E.g. the mpirun command in OpenMPI does this via its `-np` and `-npernode` switches. Ditto for MPICH via `-np` and `-ppn`.

### Running on a multicore CPU

Here is a quick overview of how to use the KOKKOS package for CPU acceleration, assuming one or more 16-core nodes.

```
# 1 node, 16 MPI tasks/node, no multi-threading
mpirun -np 16 lmp_kokkos_mpi_only -k on -sf kk -in in.lj

# 2 nodes, 1 MPI task/node, 16 threads/task
mpirun -np 2 -ppn 1 lmp_kokkos_omp -k on t 16 -sf kk -in in.lj

# 1 node,  2 MPI tasks/node, 8 threads/task
mpirun -np 2 lmp_kokkos_omp -k on t 8 -sf kk -in in.lj

# 8 nodes, 4 MPI tasks/node, 4 threads/task
mpirun -np 32 -ppn 4 lmp_kokkos_omp -k on t 4 -sf kk -in in.lj
```

To run using the KOKKOS package, use the `-k on`, `-sf kk` and `-pk kokkos` *command-line switches* in your `mpirun`

command. You must use the `-k on` *command-line switch* to enable the KOKKOS package. It takes additional arguments for hardware settings appropriate to your system. For OpenMP use:

```
-k on t Nt
```

The `t Nt` option specifies how many OpenMP threads per MPI task to use with a node. The default is `Nt` = 1, which is MPI-only mode. Note that the product of MPI tasks * OpenMP threads/task should not exceed the physical number of cores (on a node), otherwise performance will suffer. If Hyper-Threading (HT) is enabled, then the product of MPI tasks * OpenMP threads/task should not exceed the physical number of cores * hardware threads. The `-k on` switch also issues a `package kokkos` command (with no additional arguments) which sets various KOKKOS options to default values, as discussed on the *package* command doc page.

The `-sf kk` *command-line switch* will automatically append the "/kk" suffix to styles that support it. In this manner no modification to the input script is needed. Alternatively, one can run with the KOKKOS package by editing the input script as described below.

**Note**

When using ONLY a single OpenMP thread, the Kokkos Serial back end (i.e. `-D Kokkos_ENABLE_SERIAL=yes`) will give better performance than the OpenMP back end (i.e. `-D Kokkos_ENABLE_OPENMP=yes`) because some of the overhead to make the code thread-safe is removed.

**Note**

Use the `-pk kokkos` *command-line switch* to change the default *package kokkos* options. See its doc page for details and default settings. Experimenting with its options can provide a speed-up for specific calculations. For example:

```
# Newton on, Half neighbor list, non-threaded comm
mpirun -np 16 lmp_kokkos_mpi_only -k on -sf kk \
       -pk kokkos newton on neigh half comm no -in in.lj
```

If the *newton* command is used in the input script, it can also override the Newton flag defaults.

For half neighbor lists and OpenMP, the KOKKOS package uses data duplication (i.e. thread-private arrays) by default to avoid thread-level write conflicts in the force arrays (and other data structures as necessary). Data duplication is typically fastest for small numbers of threads (i.e. 8 or less) but does increase memory footprint and is not scalable to large numbers of threads. An alternative to data duplication is to use thread-level atomic operations which do not require data duplication. The use of atomic operations can be enforced by compiling LAMMPS with the `-DLMP_KOKKOS_USE_ATOMICS` pre-processor flag. Most but not all Kokkos-enabled pair_styles support data duplication. Alternatively, full neighbor lists avoid the need for duplication or atomic operations but require more compute operations per atom. When using the Kokkos Serial back end or the OpenMP back end with a single thread, no duplication or atomic operations are used. For CUDA and half neighbor lists, the KOKKOS package always uses atomic operations.

### CPU Cores, Sockets and Thread Affinity

When using multi-threading, it is important for performance to bind both MPI tasks to physical cores, and threads to physical cores, so they do not migrate during a simulation.

If you are not certain MPI tasks are being bound (check the defaults for your MPI installation), binding can be forced with these flags:

```
# OpenMPI 1.8
mpirun -np 2 --bind-to socket --map-by socket ./lmp_openmpi ...

# Mvapich2 2.0
mpiexec -np 2 --bind-to socket --map-by socket ./lmp_mvapich ...
```

For binding threads with KOKKOS OpenMP, use thread affinity environment variables to force binding. With OpenMP 3.1 (gcc 4.7 or later, intel 12 or later) setting the environment variable `OMP_PROC_BIND=true` should be sufficient. In general, for best performance with OpenMP 4.0 or later set `OMP_PROC_BIND=spread` and `OMP_PLACES=threads`. For binding threads with the KOKKOS pthreads option, compile LAMMPS with the hwloc or libnuma support enabled as described in the *extra build options page*. Running on Knight's Landing (KNL) Intel Xeon Phi
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Here is a quick overview of how to use the KOKKOS package for the Intel Knight's Landing (KNL) Xeon Phi:

KNL Intel Phi chips have 68 physical cores. Typically 1 to 4 cores are reserved for the OS, and only 64 or 66 cores are used. Each core has 4 Hyper-Threads,so there are effectively N = 256 (4*64) or N = 264 (4*66) cores to run on. The product of MPI tasks * OpenMP threads/task should not exceed this limit, otherwise performance will suffer. Note that with the KOKKOS package you do not need to specify how many KNLs there are per node; each KNL is simply treated as running some number of MPI tasks.

Examples of mpirun commands that follow these rules are shown below.

```
# Running on an Intel KNL node with 68 cores
# (272 threads/node via 4x hardware threading):

# 1 node, 64 MPI tasks/node, 4 threads/task
mpirun -np 64 lmp_kokkos_phi -k on t 4 -sf kk -in in.lj

# 1 node, 66 MPI tasks/node, 4 threads/task
mpirun -np 66 lmp_kokkos_phi -k on t 4 -sf kk -in in.lj

# 1 node, 32 MPI tasks/node, 8 threads/task
mpirun -np 32 lmp_kokkos_phi -k on t 8 -sf kk -in in.lj

# 8 nodes, 64 MPI tasks/node, 4 threads/task
mpirun -np 512 -ppn 64 lmp_kokkos_phi -k on t 4 -sf kk -in in.lj
```

The `-np` setting of the mpirun command sets the number of MPI tasks/node. The `-k on t Nt` command-line switch sets the number of threads/task as `Nt`. The product of these two values should be N, i.e. 256 or 264.

**Note**

The default for the *package kokkos* command when running on KNL is to use "half" neighbor lists and set the Newton flag to "on" for both pairwise and bonded interactions. This will typically be best for many-body potentials. For simpler pairwise potentials, it may be faster to use a "full" neighbor list with Newton flag to "off". Use the `-pk kokkos` *command-line switch* to change the default *package kokkos* options. See its documentation page for details and default settings. Experimenting with its options can provide a speed-up for specific calculations. For example:

```
#  Newton on, half neighbor list, threaded comm
mpirun -np 64 lmp_kokkos_phi -k on t 4 -sf kk -pk kokkos comm host -in in.reax

# Newton off, full neighbor list, non-threaded comm
```

(continues on next page)

(continued from previous page)

```
mpirun -np 64 lmp_kokkos_phi -k on t 4 -sf kk \
       -pk kokkos newton off neigh full comm no -in in.lj
```

**Note**

MPI tasks and threads should be bound to cores as described above for CPUs.

**Note**

To build with Kokkos support for Intel Xeon Phi co-processors such as Knight's Corner (KNC), your system must be configured to use them in "native" mode, not "offload" mode like the INTEL package supports.

### Running on GPUs

Use the `-k` *command-line switch* to specify the number of GPUs per node. Typically the `-np` setting of the `mpirun` command should set the number of MPI tasks/node to be equal to the number of physical GPUs on the node. You can assign multiple MPI tasks to the same GPU with the KOKKOS package, but this is usually only faster if some portions of the input script have not been ported to use Kokkos. In this case, also packing/unpacking communication buffers on the host may give speedup (see the KOKKOS *package* command). Using CUDA MPS is recommended in this scenario.

Using a GPU-aware MPI library is highly recommended. GPU-aware MPI use can be avoided by using *-pk kokkos gpu/aware off*. As above for multicore CPUs (and no GPU), if N is the number of physical cores/node, then the number of MPI tasks/node should not exceed N.

```
-k on g Ng
```

Here are examples of how to use the KOKKOS package for GPUs, assuming one or more nodes, each with two GPUs:

```
# 1 node,   2 MPI tasks/node, 2 GPUs/node
mpirun -np 2 lmp_kokkos_cuda_openmpi -k on g 2 -sf kk -in in.lj

# 16 nodes, 2 MPI tasks/node, 2 GPUs/node (32 GPUs total)
mpirun -np 32 -ppn 2 lmp_kokkos_cuda_openmpi -k on g 2 -sf kk -in in.lj
```

**Note**

The default for the *package kokkos* command when running on GPUs is to use "full" neighbor lists and set the Newton flag to "off" for both pairwise and bonded interactions, along with threaded communication. When running on Maxwell or Kepler GPUs, this will typically be best. For Pascal GPUs and beyond, using "half" neighbor lists and setting the Newton flag to "on" may be faster. For many pair styles, setting the neighbor binsize equal to twice the CPU default value will give speedup, which is the default when running on GPUs. Use the `-pk kokkos` *command-line switch* to change the default *package kokkos* options. See its documentation page for details and default settings. Experimenting with its options can provide a speed-up for specific calculations. For example:

```
# Newton on, half neighbor list, set binsize = neighbor ghost cutoff
mpirun -np 2 lmp_kokkos_cuda_openmpi -k on g 2 -sf kk \
       -pk kokkos newton on neigh half binsize 2.8 -in in.lj
```

**Note**

The default binsize for *atom sorting* on GPUs is equal to the default CPU neighbor binsize (i.e. 2x smaller than the default GPU neighbor binsize). When running simple pair-wise potentials like Lennard Jones on GPUs, using a 2x larger binsize for atom sorting (equal to the default GPU neighbor binsize) and a more frequent sorting than default (e.g. sorting every 100 time steps instead of 1000) may improve performance.

**Note**

When running on GPUs with many MPI ranks (tens of thousands and more), the creation of the atom map (required for molecular systems) on the GPU can slow down significantly or run out of GPU memory and thus slow down the whole calculation or cause a crash. You can use the `-pk kokkos atom/map no` *command-line switch* of the *package kokkos atom/map no* command to create the atom map on the CPU instead.

**Note**

When using a GPU, you will achieve the best performance if your input script does not use fix or compute styles which are not yet Kokkos-enabled. This allows data to stay on the GPU for multiple timesteps, without being copied back to the host CPU. Invoking a non-Kokkos fix or compute, or performing I/O for *thermo* or *dump* output will cause data to be copied back to the CPU incurring a performance penalty.

**Note**

To get an accurate timing breakdown between time spend in pair, kspace, etc., you must set the environment variable `CUDA_LAUNCH_BLOCKING=1`. However, this will reduce performance and is not recommended for production runs.

### Troubleshooting segmentation faults on GPUs

As noted above, KOKKOS by default assumes that the MPI library is GPU-aware. This is not always the case and can lead to segmentation faults when using more than one MPI process. Normally, LAMMPS will print a warning like "*Turning off GPU-aware MPI since it is not detected*", or an error message like "*Kokkos with GPU-enabled back end assumes GPU-aware MPI is available*", OR a **segmentation fault**. To confirm that a segmentation fault is caused by this, you can turn off the GPU-aware assumption via the *package kokkos command* or the corresponding command-line flag.

If you still get a segmentation fault, despite running with only one MPI process or using the command-line flag to turn off expecting a GPU-aware MPI library, then using the CMake compile setting `-DKokkos_ENABLE_DEBUG=on` will generate useful output that can be passed to the LAMMPS developers for further debugging.

### Troubleshooting memory allocation on GPUs

Kokkos Tools provides a set of lightweight profiling and debugging utilities, which interface with instrumentation hooks (eg. space-time-stack) built directly into the Kokkos runtime. After compiling a dynamic LAMMPS library, you then have to set the environment variable `KOKKOS_TOOLS_LIBS` before executing your LAMMPS Kokkos run. Example:

```
export KOKKOS_TOOLS_LIBS=${HOME}/kokkos-tools/src/tools/memory-events/kp_memory_event.so
mpirun -np 4 lmp_kokkos_cuda_openmpi -in in.lj -k on g 4 -sf kk
```

Starting with the NVIDIA Pascal GPU architecture, CUDA supports "Unified Virtual Memory" (UVM) which enables

allocating more memory than a GPU possesses by also using memory on the host CPU and then CUDA will transparently move data between CPU and GPU as needed. The resulting LAMMPS performance depends on memory access pattern, data residency, and GPU memory oversubscription . The CMake option `-DKokkos_ENABLE_CUDA_UVM=on` enables using *UVM with Kokkos* when compiling LAMMPS.

### Run with the KOKKOS package by editing an input script

Alternatively the effect of the `-sf` or `-pk` switches can be duplicated by adding the *package kokkos* or *suffix kk* commands to your input script.

The discussion above for building LAMMPS with the KOKKOS package, the `mpirun` or `mpiexec` command, and setting appropriate thread properties are the same.

You must still use the `-k on` *command-line switch* to enable the KOKKOS package, and specify its additional arguments for hardware options appropriate to your system, as documented above.

You can use the *suffix kk* command, or you can explicitly add a "kk" suffix to individual styles in your input script, e.g.

```
pair_style lj/cut/kk 2.5
```

You only need to use the *package kokkos* command if you wish to change any of its option defaults, as set by the "-k on" *command-line switch*.

**Using OpenMP threading and CUDA together:**

With the KOKKOS package, both OpenMP multi-threading and GPUs can be compiled and used together in a few special cases. You need to enable both features as it is done in the `kokkos-cuda.cmake` CMake preset file.

```
cmake -DKokkos_ENABLE_CUDA=yes -DKokkos_ENABLE_OPENMP=yes ../cmake
```

The suffix "/kk" is equivalent to "/kk/device", and for Kokkos CUDA, using the `-sf kk` in the command-line gives the default CUDA version everywhere. However, if the "/kk/host" suffix is added to a specific style in the input script, the Kokkos OpenMP (CPU) version of that specific style will be used instead. Set the number of OpenMP threads as `t Nt` and the number of GPUs as `g Ng`

```
-k on t Nt g Ng
```

For example, the command to run with 1 GPU and 8 OpenMP threads is then:

```
mpiexec -np 1 lmp_kokkos_cuda_openmpi -in in.lj -k on g 1 t 8 -sf kk
```

Conversely, if the `-sf kk/host` is used in the command-line and then the "/kk" or "/kk/device" suffix is added to a specific style in your input script, then only that specific style will run on the GPU while everything else will run on the CPU in OpenMP mode. Note that the execution of the CPU and GPU styles will NOT overlap, except for a special case:

A kspace style and/or molecular topology (bonds, angles, etc.) running on the host CPU can overlap with a pair style running on the GPU. First compile with `--default-stream per-thread` added to `-D CMAKE_CXX_FLAGS` when configuring with CMake. Then explicitly use the "/kk/host" suffix for kspace and bonds, angles, etc. in the input file and the "kk" suffix (equal to "kk/device") on the command-line. Also make sure the environment variable `CUDA_LAUNCH_BLOCKING` is not set to "1" so CPU/GPU overlap can occur.

### Performance to expect

The performance of KOKKOS running in different modes is a function of your hardware, which KOKKOS-enable styles are used, and the problem size.

Generally speaking, the following rules of thumb apply:

- When running on CPUs only, with a single thread per MPI task, performance of a KOKKOS style is somewhere between the standard (un-accelerated) styles (MPI-only mode), and those provided by the OPENMP package. However the difference between all 3 is small (less than 20%).
- When running on CPUs only, with multiple threads per MPI task, performance of a KOKKOS style is a bit slower than the OPENMP package.
- When running large number of atoms per GPU, KOKKOS is typically faster than the GPU package when both are compiled for double precision.
- When running on Intel Phi hardware, KOKKOS is not as fast as the INTEL package, which is optimized for x86 hardware (not just from Intel) and compilation with the Intel compilers.
- The KOKKOS package by default assumes that you are using exactly one MPI rank per GPU. When trying to use multiple MPI ranks per GPU it is mandatory to enable CUDA Multi-Process Service (MPS) to get good performance. In this case it is better to not use all available MPI ranks in order to avoid competing with the MPS daemon for CPU resources.

See the Benchmark page of the LAMMPS website for performance of the KOKKOS package on different hardware.

#### Advanced Kokkos options

There are other allowed options when building with the KOKKOS package that can improve performance or assist in debugging or profiling. They are explained on the *KOKKOS section of the build extras* doc page,

#### References

**(Johansson)** A. Johansson, E. Weinberg, C. Trott, M. McCarthy, and S. Moore. LAMMPS-KOKKOS: Performance portable molecular dynamics across exascale architectures. In Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, page 1217-1232, 2025.

### 7.4.4 OPENMP package

The OPENMP package was developed by Axel Kohlmeyer at Temple University. It provides optimized and multi-threaded versions of many pair styles, nearly all bonded styles (bond, angle, dihedral, improper), several Kspace styles, and a few fix styles. It uses the OpenMP interface for multi-threading, but can also be compiled without OpenMP support, providing optimized serial styles in that case.

#### Required hardware/software

To enable multi-threading, your compiler must support the OpenMP interface. You should have one or more multicore CPUs, as multiple threads can only be launched by each MPI task on the local node (using shared memory).

#### Building LAMMPS with the OPENMP package

See the *Build extras* page for instructions.

#### Run with the OPENMP package from the command-line

These examples assume one or more 16-core nodes.

```
# 1 MPI task, 16 threads according to OMP_NUM_THREADS
env OMP_NUM_THREADS=16 lmp_omp -sf omp -in in.script

# 1 MPI task, no threads, optimized kernels
lmp_mpi -sf omp -in in.script

# 4 MPI tasks, 4 threads/task
```

(continues on next page)

(continued from previous page)

```
mpirun -np 4 lmp_omp -sf omp -pk omp 4 -in in.script

# 8 nodes, 4 MPI tasks/node, 4 threads/task
mpirun -np 32 -ppn 4 lmp_omp -sf omp -pk omp 4 -in in.script
```

The `mpirun` or `mpiexec` command sets the total number of MPI tasks used by LAMMPS (one or multiple per compute node) and the number of MPI tasks used per node. E.g. the mpirun command in MPICH does this via its `-np` and `-ppn` switches. Ditto for OpenMPI via `-np` and `-npernode`.

You need to choose how many OpenMP threads per MPI task will be used by the OPENMP package. Note that the product of MPI tasks * threads/task should not exceed the physical number of cores (on a node), otherwise performance will suffer.

As in the lines above, use the `-sf omp` *command-line switch*, which will automatically append "omp" to styles that support it. The `-sf omp` switch also issues a default *package omp 0* command, which will set the number of threads per MPI task via the `OMP_NUM_THREADS` environment variable.

You can also use the `-pk omp Nt` *command-line switch*, to explicitly set `Nt` = # of OpenMP threads per MPI task to use, as well as additional options. Its syntax is the same as the *package omp* command whose page gives details, including the default values used if it is not specified. It also gives more details on how to set the number of threads via the `OMP_NUM_THREADS` environment variable.

### Or run with the OPENMP package by editing an input script

The discussion above for the `mpirun` or `mpiexec` command, MPI tasks/node, and threads/MPI task is the same.

Use the *suffix omp* command, or you can explicitly add an "omp" suffix to individual styles in your input script, e.g.

```
pair_style lj/cut/omp 2.5
```

You must also use the *package omp* command to enable the OPENMP package. When you do this you also specify how many threads per MPI task to use. The command page explains other options and how to set the number of threads via the `OMP_NUM_THREADS` environment variable.

### Speed-up to expect

Depending on which styles are accelerated, you should look for a reduction in the "Pair time", "Bond time", "KSpace time", and "Loop time" values printed at the end of a run.

You may see a small performance advantage (5 to 20%) when running a OPENMP style (in serial or parallel) with a single thread per MPI task, versus running standard LAMMPS with its standard un-accelerated styles (in serial or all-MPI parallelization with 1 task/core). This is because many of the OPENMP styles contain similar optimizations to those used in the OPT package, described in *the OPT package* doc page.

With multiple threads/task, the optimal choice of number of MPI tasks/node and OpenMP threads/task can vary a lot and should always be tested via benchmark runs for a specific simulation running on a specific machine, paying attention to guidelines discussed in the next subsection.

A description of the multi-threading strategy used in the OPENMP package and some performance examples are presented here.

### Guidelines for best performance

For many problems on current generation CPUs, running the OPENMP package with a single thread/task is faster than running with multiple threads/task. This is because the MPI parallelization in LAMMPS is often more efficient than multi-threading as implemented in the OPENMP package. The parallel efficiency (in a threaded sense) also varies for different OPENMP styles.

Using multiple threads/task can be more effective under the following circumstances:

- Individual compute nodes have a significant number of CPU cores but the CPU itself has limited memory bandwidth, e.g. for Intel Xeon 53xx (Clovertown) and 54xx (Harpertown) quad-core processors. Running one MPI task per CPU core will result in significant performance degradation, so that running with 4 or even only 2 MPI tasks per node is faster. Running in hybrid MPI+OpenMP mode will reduce the inter-node communication bandwidth contention in the same way, but offers an additional speedup by utilizing the otherwise idle CPU cores.
- The interconnect used for MPI communication does not provide sufficient bandwidth for a large number of MPI tasks per node. For example, this applies to running over gigabit ethernet or on Cray XT4 or XT5 series supercomputers. As in the aforementioned case, this effect worsens when using an increasing number of nodes.
- The system has a spatially inhomogeneous particle density which does not map well to the *domain decomposition scheme* or *load-balancing* options that LAMMPS provides. This is because multi-threading achieves parallelism over the number of particles, not via their distribution in space.
- A machine is being used in "capability mode", i.e. near the point where MPI parallelism is maxed out. For example, this can happen when using the *PPPM solver* for long-range electrostatics on large numbers of nodes. The scaling of the KSpace calculation (see the *kspace_style* command) becomes the performance-limiting factor. Using multi-threading allows less MPI tasks to be invoked and can speed-up the long-range solver, while increasing overall performance by parallelizing the pairwise and bonded calculations via OpenMP. Likewise, additional speedup can sometimes be achieved by increasing the length of the Coulombic cutoff and thus reducing the work done by the long-range solver. Using the *run_style verlet/split* command, which is compatible with the OPENMP package, is an alternative way to reduce the number of MPI tasks assigned to the KSpace calculation.

Additional performance tips are as follows:

- The best parallel efficiency from *omp* styles is typically achieved when there is at least one MPI task per physical CPU chip, i.e. socket or die.
- It is usually most efficient to restrict threading to a single socket, i.e. use one or more MPI task per socket.
- NOTE: By default, several current MPI implementations use a processor affinity setting that restricts each MPI task to a single CPU core. Using multi-threading in this mode will force all threads to share the one core and thus is likely to be counterproductive. Instead, binding MPI tasks to a (multicore) socket, should solve this issue.

#### Restrictions

None.

### 7.4.5 OPT package

The OPT package was developed by James Fischer (High Performance Technologies), David Richie, and Vincent Natoli (Stone Ridge Technologies). It contains a handful of pair styles whose compute() methods were rewritten in C++ templated form to reduce the overhead due to if tests and other conditional code.

#### Required hardware/software

Any hardware. Any compiler.

#### Building LAMMPS with the OPT package

See the *Build extras* page for instructions.

### Run with the OPT package from the command-line

```
lmp_mpi -sf opt -in in.script                 # run in serial
mpirun -np 4 lmp_mpi -sf opt -in in.script    # run in parallel
```

Use the "-sf opt" *command-line switch*, which will automatically append "opt" to styles that support it.

### Or run with the OPT package by editing an input script

Use the *suffix opt* command, or you can explicitly add an "opt" suffix to individual styles in your input script, e.g.

```
pair_style lj/cut/opt 2.5
```

### Speed-up to expect

You should see a reduction in the "Pair time" value printed at the end of a run. On most machines for reasonable problem sizes, it will be a 5 to 20% savings.

### Guidelines for best performance

Just try out an OPT pair style to see how it performs.

### Restrictions

None.

Inverting this list, LAMMPS currently has acceleration support for three kinds of hardware, via the listed packages:

| | |
|---|---|
| Many-core CPUs | *INTEL*, *KOKKOS*, *OPENMP*, *OPT* packages |
| GPUs | *GPU*, *KOKKOS* packages |
| Intel AVX | *INTEL*, *KOKKOS* packages |

Which package is fastest for your hardware may depend on the size problem you are running and what commands (accelerated and non-accelerated) are invoked by your input script. While these doc pages include performance guidelines, there is no substitute for trying out the different packages appropriate to your hardware.

Any accelerated style has the same name as the corresponding standard style, except that a suffix is appended. Otherwise, the syntax for the command that uses the style is identical, their functionality is the same, and the numerical results it produces should also be the same, except for precision and round-off effects.

For example, all of these styles are accelerated variants of the Lennard-Jones *pair_style lj/cut*:

- *pair_style lj/cut/gpu*
- *pair_style lj/cut/intel*
- *pair_style lj/cut/kk*
- *pair_style lj/cut/omp*
- *pair_style lj/cut/opt*

To see what accelerate styles are currently available for a particular style, find the style name in the *Commands* style pages (fix,compute,pair,etc) and see what suffixes are listed (g,i,k,o,t) with it. The doc pages for individual commands (e.g. *pair lj/cut* or *fix nve*) also list any accelerated variants available for that style.

To use an accelerator package in LAMMPS, and one or more of the styles it provides, follow these general steps. Details vary from package to package and are explained in the individual accelerator doc pages, listed above.

| | |
|---|---|
| Configure LAMMPS to include the accelerator package | *Build LAMMPS with CMake* or *Build LAMMPS with make* |
| (Re-)Compile LAMMPS | *Build LAMMPS* |
| prepare and test a regular LAMMPS simulation | `lmp -in in.script; mpirun -np 32 lmp -in in.script` |
| enable specific accelerator support via `-k on` *command-line switch* | only needed for KOKKOS package |
| set any needed options for the package via `-pk` *command-line switch* or *package* command | only if defaults need to be changed |
| use accelerated styles in your input via `-sf` *command-line switch* or *suffix* command | `lmp_machine -in in.script -sf gpu` |

This is discussed on the *Packages* and *extra Build info* doc pages, and its use is illustrated in the individual accelerator sections. Typically these steps only need to be done once, to create an executable that uses one or more accelerator packages.

**Note**

With a few exceptions, you can build a single LAMMPS executable with all its accelerator packages installed. Note however that the KOKKOS package requires you to choose one of its hardware options when building for a specific platform, e.g. the OpenMP, CUDA, HIP, or SYCL option for the KOKKOS package. Or the OpenCL, HIP, or CUDA option for the GPU package.

As mentioned above, the Benchmark page of the LAMMPS website gives performance results for the various accelerator packages for several of the standard LAMMPS benchmark problems, as a function of problem size and number of compute nodes, on different hardware platforms.

Here is a brief summary of what the various packages provide. Details are in the individual accelerator sections.

- Styles with a "gpu" suffix are part of the GPU package and can be run on Intel, NVIDIA, or AMD GPUs. The speed-up on a GPU depends on a variety of factors, discussed in the accelerator sections.
- Styles with an "intel" suffix are part of the INTEL package. These styles support vectorized single and mixed precision calculations, in addition to full double precision. In extreme cases, this can provide speedups over 3.5x on CPUs.
- Styles with a "kk" suffix are part of the KOKKOS package, and can be run using OpenMP on multicore CPUs or on an NVIDIA or AMD GPU. The speed-up depends on a variety of factors, as discussed on the KOKKOS accelerator page.
- Styles with an "omp" suffix are part of the OPENMP package and allow a pair-style to be run in multi-threaded mode using OpenMP. This can be useful on nodes with high-core counts when using less MPI processes than cores is advantageous, e.g. when running with PPPM so that FFTs are run on fewer MPI processors or when the many MPI tasks would overload the available bandwidth for communication.
- Styles with an "opt" suffix are part of the OPT package and typically speed-up the pairwise calculations of your simulation by 5-25% on a CPU.

The individual accelerator package doc pages explain:

- what hardware and software the accelerated package requires
- how to build LAMMPS with the accelerated package
- how to run with the accelerated package either via command-line switches or modifying the input script
- speed-ups to expect

- guidelines for best performance
- restrictions

## 7.5 Comparison of various accelerator packages

The next section compares and contrasts the various accelerator options, since there are multiple ways to perform OpenMP threading, run on GPUs, and optimize for vector units on CPUs.

All of these packages can accelerate a LAMMPS calculation taking advantage of hardware features, but they do it in different ways and acceleration is not always guaranteed.

As a consequence, for a particular simulation on specific hardware, one package may be faster than the other. We give some guidelines below, but the best way to determine which package is faster for your input script is to try multiple of them on your machine and experiment with available performance tuning settings. See the benchmarking section below for examples where this has been done.

**Guidelines for using each package optimally:**

- Both, the GPU and the KOKKOS package allows you to assign multiple MPI ranks (= CPU cores) to the same GPU. For the GPU package, this can lead to a speedup through better utilization of the GPU (by overlapping computation and data transfer) and more efficient computation of the non-GPU accelerated parts of LAMMPS through MPI parallelization, as all system data is maintained and updated on the host. For KOKKOS, there is less to no benefit from this, due to its different memory management model, which tries to retain data on the GPU.
- The GPU package moves per-atom data (coordinates, forces, and (optionally) neighbor list data, if not computed on the GPU) between the CPU and GPU at every timestep. The KOKKOS/CUDA package only does this on timesteps when a CPU calculation is required (e.g. to invoke a fix or compute that is non-GPU-ized). Hence, if you can formulate your input script to only use GPU-ized fixes and computes, and avoid doing I/O too often (thermo output, dump file snapshots, restart files), then the data transfer cost of the KOKKOS/CUDA package can be very low, causing it to run faster than the GPU package.
- The GPU package is often faster than the KOKKOS/CUDA package, when the number of atoms per GPU is on the smaller side. The crossover point, in terms of atoms/GPU at which the KOKKOS/CUDA package becomes faster depends strongly on the pair style.
- When using LAMMPS with multiple MPI ranks assigned to the same GPU, its performance depends to some extent on the available bandwidth between the CPUs and the GPU. This can differ significantly based on the available bus technology, capability of the host CPU and mainboard, the wiring of the buses and whether switches are used to increase the number of available bus slots, or if GPUs are housed in an external enclosure. This can become quite complex.
- To achieve significant acceleration through GPUs, both KOKKOS and GPU package require capable GPUs with fast on-device memory and efficient data transfer rates. This requests capable upper mid-level to high-end (desktop) GPUs. Using lower performance GPUs (e.g. on laptops) may result in a slowdown instead.
- For the GPU package, specifically when running in parallel with MPI, it is often more efficient to exclude the PPPM kspace style from GPU acceleration and instead run it - concurrently with a GPU accelerated pair style - on the CPU. This can often be easily achieved with placing a *suffix off* command before and a *suffix on* command after the *kspace_style pppm* command.
- The KOKKOS/OpenMP and OPENMP package have different thread management strategies, which should result in OPENMP being more efficient for a small number of threads with increasing overhead as the number of threads per MPI rank grows. The KOKKOS/OpenMP kernels have less overhead in that case, but have lower performance with few threads.
- The INTEL package contains many options and settings for achieving additional performance on Intel hardware (CPU and accelerator cards), but to unlock this potential, an Intel compiler is required. The package code will

compile with GNU gcc, but it will not be as efficient.

**Differences between the GPU and KOKKOS packages:**

- The GPU package accelerates only pair force, neighbor list, and (parts of) PPPM calculations (and runs the remaining force computations on the CPU concurrently). The KOKKOS package attempts to run most of the calculation on the GPU, but can transparently support non-accelerated code (with a performance penalty due to having data transfers between host and GPU).
- The list of which styles are accelerated by the GPU or KOKKOS package differs with some overlap.
- The GPU package requires neighbor lists to be built on the CPU when using hybrid pair styles, exclusion lists, or a triclinic simulation box.
- The GPU package benefits from running multiple MPI processes (2-8) per GPU to parallelize the non-GPU accelerated styles. The KOKKOS package usually not, especially when all parts of the calculation have KOKKOS support.
- The GPU package can be compiled for CUDA, HIP, or OpenCL and thus supports NVIDIA, AMD, and Intel GPUs well. On NVIDIA or AMD hardware, using native CUDA or HIP compilation, respectively, with either GPU or KOKKOS results in equal or better performance over OpenCL.
- OpenCL in the GPU package supports NVIDIA, AMD, and Intel GPUs at the *same time* and with the *same executable*. KOKKOS currently does not support OpenCL.
- Both the GPU package and the KOKKOS package support single precision floating point, mixed precision floating point, and double precision floating point math. This must be chosen at compile time. Using single or mixed precision (recommended) results in significantly improved performance on consumer GPUs for some loss in accuracy (which is rather small with mixed precision).
- Some pair styles (for example *snap*, *mliap* or *reaxff* in the KOKKOS package have seen extensive optimizations and specializations for GPUs and CPUs.

CHAPTER

EIGHT

# OPTIONAL PACKAGES

This section gives an overview of the optional packages included in the LAMMPS distribution that extend LAMMPS' functionality. Packages are groups of files that enable a specific set of features. For example, force fields for molecular systems or rigid-body constraints are in packages. You can see the list of all packages and "make" commands to manage them by typing "make package" from within the src directory of the LAMMPS distribution. The list of packages that are included in a specific LAMMPS executable can be displayed by *running LAMMPS with the "-h" flag.*

The *Build package* page gives general info on how to install and uninstall packages as part of the LAMMPS build process.

**External packages**

In addition to packages included with LAMMPS there also are packages that are distributed independently and separate from LAMMPS. Some of them are part of a modified fork of the LAMMPS repository, others are add-on files that have to be integrated into the LAMMPS distribution first. The LAMMPS developers have in most cases no direct knowledge of those packages and which new commands and modifications to LAMMPS they contain. Since LAMMPS is *open source under the GPL-v2 license*, there is no requirement to tell us about it or integrated them back into LAMMPS. Thus you have to contact the developers of those individual packages in case of questions or problems.

As the LAMMPS internal code structure is sometimes changing, those external packages may need to be updated to remain compatible with the latest LAMMPS versions. The LAMMPS Programmer's Guide part contains a section with *notes for updating code written for older LAMMPS versions* which explains significant changes and how to update the corresponding code.

Below is the list of packages that are included in the LAMMPS distribution. The link for each package name gives more details.

Packages are supported by either the LAMMPS developers or the contributing authors. They should all be written in a syntax and style consistent with the rest of LAMMPS.

The "Examples" column is a subdirectory in the examples directory of the distribution which has one or more input scripts that use the package. E.g. `peptide` refers to the `examples/peptide` directory; `PACKAGES/atc` refers to the `examples/PACKAGES/atc` directory. The "Lib" column indicates`` whether an extra library is needed to build and use the package:

- no = no library
- sys = system library: you likely have it on your machine
- int = internal library: provided with LAMMPS, but you may need to build it
- ext = external library: you will need to download and install it on your machine

| Package | Description | Doc page | Examples | Lib |
|---|---|---|---|---|
| *ADIOS* | dump output via ADIOS | *dump adios* | PACKAGES/adios | ext |
| *AMOEBA* | AMOEBA and HIPPO force fields | *AMOEBA and HIPPO howto* | amoeba | no |
| *APIP* | adaptive-precision interatomic potentials | *Howto APIP* | PACKAGES/apip | ext |
| *ASPHERE* | aspherical particle models | *Howto spherical* | ellipse | no |
| *BOCS* | BOCS bottom up coarse graining and Local Density Dependent potentials | *fix bocs*, *pair_ldd*, *LDD Howto* | PACKAGES/bocs | no |
| *BODY* | body-style particles | *Howto body* | body | no |
| *BPM* | bonded particle models | *Howto bpm* | bpm | no |
| *BROWNIAN* | Brownian dynamics, self-propelled particles | *fix brownian*, *fix propel/self* | PACKAGES/brownian | no |
| *CG-DNA* | coarse-grained DNA force fields | src/CG-DNA/README | PACKAGES/cgdna | no |
| *CG-SPICA* | SPICA (SDK) coarse-graining model | *pair_style lj/spica* | PACKAGES/cgspica | no |
| *CLASS2* | class 2 force fields | *pair_style lj/class2* | n/a | no |
| *COLLOID* | colloidal particles | *atom_style colloid* | colloid | no |
| *COLVARS* | Colvars collective variables library | *fix colvars* | PACKAGES/colvars | int |
| *COMPRESS* | I/O compression | *dump */gz* | n/a | sys |
| *CORESHELL* | adiabatic core/shell model | *Howto coreshell* | coreshell | no |
| *DIELECTRIC* | dielectric boundary solvers and force styles | *compute efield/atom* | PACKAGES/dielectric | no |
| *DIFFRACTION* | virtual x-ray and electron diffraction | *compute xrd* | PACKAGES/diffraction | no |
| *DIPOLE* | point dipole particles | *pair_style lj/. . ./dipole* | dipole | no |
| *DPD-BASIC* | basic DPD models | *pair_styles dpd dpd/ext* | PACKAGES/dpd-basic | no |
| *DPD-MESO* | mesoscale DPD models | *pair_style edpd* | PACKAGES/dpd-meso | no |
| *DPD-REACT* | reactive dissipative particle dynamics | src/DPD-REACT/README | PACKAGES/dpd-react | no |
| *DPD-SMOOTH* | smoothed dissipative particle dynamics | src/DPD-SMOOTH/README | PACKAGES/dpd-smooth | no |
| *DRUDE* | Drude oscillators | *Howto drude* | PACKAGES/drude | no |
| *EFF* | electron force field | *pair_style eff/cut* | PACKAGES/eff | no |
| *ELECTRODE* | electrode charges to match potential | *fix electrode/conp* | PACKAGES/electrode | no |

continues on next page

Table 1 – continued from previous page

| Package | Description | Doc page | Examples | Lib |
|---|---|---|---|---|
| *EXTRA-COMMAND* | additional command styles | *general commands* | n/a | no |
| *EXTRA-COMPUTE* | additional compute styles | *compute* | n/a | no |
| *EXTRA-DUMP* | additional dump styles | *dump* | n/a | no |
| *EXTRA-FIX* | additional fix styles | *fix* | n/a | no |
| *EXTRA-MOLECULE* | additional molecular styles | *molecular styles* | n/a | no |
| *EXTRA-PAIR* | additional pair styles | *pair_style* | n/a | no |
| *FEP* | free energy perturbation | *compute fep* | `PACKAGES/fep` | no |
| *GPU* | GPU-enabled styles | *Section gpu* | Benchmarks | int |
| *GRANULAR* | granular systems | *Howto granular* | pour | no |
| *GRANSURF* | surfaces for granular systems | *Howto granular surfaces* | gransurf | no |
| *GRAPHICS* | generate graphics output | *Visualize LAMMPS snapshots* | n/a | no |
| *H5MD* | dump output via HDF5 | *dump h5md* | n/a | ext |
| *INTEL* | optimized Intel CPU styles | *Speed intel* | Benchmarks | no |
| *INTERLAYER* | Inter-layer pair potentials | *several pair styles* | `PACKAGES/interlayer` | no |
| *KIM* | OpenKIM wrapper | *pair_style kim* | kim | ext |
| *KOKKOS* | Kokkos-enabled styles | *Speed kokkos* | Benchmarks | no |
| *KSPACE* | long-range Coulombic solvers | *kspace_style* | peptide | no |
| *LATBOLTZ* | Lattice Boltzmann fluid | *fix lb/fluid* | `PACKAGES/latboltz` | no |
| *LEPTON* | evaluate strings as potential function | *pair_style lepton* | `PACKAGES/lepton` | int |
| *MACHDYN* | smoothed Mach dynamics | SMD User Guide | `PACKAGES/machdyn` | ext |
| *MANIFOLD* | motion on 2d surfaces | *fix manifoldforce* | `PACKAGES/manifold` | no |
| *MANYBODY* | many-body potentials | *pair_style tersoff* | shear | no |
| *MBX* | MBX MB-nrg potential library | *pair_mbx* | `PACKAGES/mbx` | no |
| *MC* | Monte Carlo options | *fix gcmc* | n/a | no |
| *MDI* | client-server code coupling | *MDI Howto* | `PACKAGES/mdi` | ext |
| *MEAM* | modified EAM potential (C++) | *pair_style meam* | meam | no |
| *MESONT* | mesoscopic tubular potential model | pair styles *mesocnt* | `PACKAGES/mesont` | no |
| *MGPT* | fast MGPT multi-ion potentials | *pair_style mgpt* | `PACKAGES/mgpt` | no |

continues on next page

Table 1 – continued from previous page

| Package | Description | Doc page | Examples | Lib |
|---|---|---|---|---|
| *MISC* | miscellaneous single-file commands | n/a | no | no |
| *ML-HDNNP* | High-dimensional neural network potentials | *pair_style hdnnp* | PACKAGES/hdnnp | ext |
| *ML-RUNNER* | Second-, third-, and fourth-generation high-dimensional neural network potentials | *pair_style runner* | PACKAGES/runner | ext |
| *ML-IAP* | multiple machine learning potentials | *pair_style mliap* | mliap | no |
| *ML-PACE* | Atomic Cluster Expansion potential | *pair pace* | PACKAGES/pace | ext |
| *ML-POD* | Proper orthogonal decomposition potentials | *pair pod* | pod | ext |
| *ML-QUIP* | QUIP/libatoms interface | *pair_style quip* | PACKAGES/quip | ext |
| *ML-RANN* | Pair style for RANN potentials | *pair rann* | PACKAGES/rann | no |
| *ML-SNAP* | quantum-fitted potential | *pair_style snap* | snap | no |
| *ML-UF3* | quantum-fitted ultra fast potentials | *pair_style uf3* | PACKAGES/uf3 | no |
| *MOFFF* | styles for MOF-FF force field | *pair_style buck6d/coul/gauss* | PACKAGES/mofff | no |
| *MOLECULE* | molecular system force fields | *Howto bioFF* | peptide | no |
| *MOLFILE* | VMD molfile plugins | *dump molfile* | n/a | ext |
| *NETCDF* | dump output via NetCDF | *dump netcdf* | n/a | ext |
| *OPENMP* | OpenMP-enabled styles | *Speed omp* | Benchmarks | no |
| *OPT* | optimized pair styles | *Speed opt* | Benchmarks | no |
| *ORIENT* | fixes for orientation depended forces | *fix orient/** | PACKAGES/orient_eco | no |
| *PERI* | Peridynamics models | *pair_style peri* | peri | no |
| *PHONON* | phonon dynamical matrix | *fix phonon* | PACKAGES/phonon | no |
| *PLUGIN* | Plugin loader command | *plugin* | plugins | no |
| *PLUMED* | PLUMED free energy library | *fix plumed* | PACKAGES/plumed | ext |
| *PTM* | Polyhedral Template Matching | *compute ptm/atom* | n/a | no |

continues on next page

Table 1 – continued from previous page

| Package | Description | Doc page | Examples | Lib |
|---|---|---|---|---|
| *PYTHON* | embed Python code in an input script | *python* | python | sys |
| *QEQ* | QEq charge equilibration | *fix qeq* | qeq | no |
| *QMMM* | QM/MM coupling | *fix qmmm* | PACKAGES/qmmm | ext |
| *QTB* | quantum nuclear effects | *fix qtb* *fix qbmsst* | qtb | no |
| *RHEO* | reproducing hydrodynamics and elastic objects | *Howto rheo* | rheo | no |
| *REACTION* | chemical reactions in classical MD | *fix bond/react* | PACKAGES/reaction | no |
| *REAXFF* | ReaxFF potential (C/C++) | *pair_style reaxff* | reax | no |
| *REPLICA* | multi-replica methods | *Howto replica* | tad | no |
| *RIGID* | rigid bodies and constraints | *fix rigid* | rigid | no |
| *SCAFACOS* | wrapper for ScaFaCoS Kspace solver | *kspace_style scafacos* | PACKAGES/scafacos | ext |
| *SHOCK* | shock loading methods | *fix msst* | n/a | no |
| *SMTBQ* | second moment tight binding potentials | pair styles *smtbq*, *smatb* | PACKAGES/smtbq | no |
| *SPH* | smoothed particle hydrodynamics | SPH User Guide | PACKAGES/sph | no |
| *SPIN* | magnetic atomic spin dynamics | *Howto spins* | SPIN | no |
| *SRD* | stochastic rotation dynamics | *fix srd* | srd | no |
| *TALLY* | pairwise tally computes | *compute XXX/tally* | PACKAGES/tally | no |
| *UEF* | extensional flow | *fix nvt/uef* | PACKAGES/uef | no |
| *VORONOI* | Voronoi tesselation | *compute voronoi/atom* | n/a | ext |
| *VTK* | dump output via VTK | *compute vtk* | n/a | ext |
| *YAFF* | additional styles implemented in YAFF | *angle_style cross* | PACKAGES/yaff | no |

## 8.1 Package details

Here is a brief description of all packages in LAMMPS. It lists authors (if applicable) and summarizes the package contents. It has specific instructions on how to install the package, including, if necessary, info on how to download or build any extra library it requires. It also gives links to documentation, example scripts, and pictures/movies (if available) that illustrate use of the package.

The majority of packages can be included in a LAMMPS build with a single setting (`-D PKG_<NAME>=on` for CMake) or command (`make yes-<name>` for make). See the *Build package* page for more info. A few packages may require additional steps; this is indicated in the descriptions below. The *Build extras* page gives those details.

**Note**

To see the complete list of commands a package adds to LAMMPS, you can examine the files in its src directory, e.g. `ls src/GRANULAR`. Files with names that start with fix, compute, atom, pair, bond, angle, etc correspond to commands with the same style name as contained in the file name.

| | | | | | |
|---|---|---|---|---|---|
| *ADIOS* | *AMOEBA* | *APIP* | *ASPHERE* | *BOCS* | *BODY* |
| *BPM* | *BROWNIAN* | *CG-DNA* | *CG-SPICA* | *CLASS2* | *COLLOID* |
| *COLVARS* | *COMPRESS* | *CORESHELL* | *DIELECTRIC* | *DIFFRAC-TION* | *DIPOLE* |
| *DPD-BASIC* | *DPD-MESO* | *DPD-REACT* | *DPD-SMOOTH* | *DRUDE* | *EFF* |
| *ELEC-TRODE* | *EXTRA-COMMAND* | *EXTRA-COMPUTE* | *EXTRA-DUMP* | *EXTRA-FIX* | *EXTRA-MOLECULE* |
| *EXTRA-PAIR* | *FEP* | *GPU* | *GRAPHICS* | *GRANULAR* | *GRANSURF* |
| *H5MD* | *INTEL* | *INTERLAYER* | *KIM* | *KOKKOS* | *KSPACE* |
| *LATBOLTZ* | *LEPTON* | *MACHDYN* | *MANIFOLD* | *MANYBODY* | *MBX* |
| *MC* | *MDI* | *MEAM* | *MESONT* | *MGPT* | *MISC* |
| *ML-HDNNP* | *ML-RUNNER* | *ML-IAP* | *ML-PACE* | *ML-POD* | *ML-QUIP* |
| *ML-RANN* | *ML-SNAP* | *ML-UF3* | *MOFFF* | *MOLECULE* | *MOLFILE* |
| *NETCDF* | *OPENMP* | *OPT* | *ORIENT* | *PERI* | *PHONON* |
| *PLUGIN* | *PLUMED* | *PTM* | *PYTHON* | *QEQ* | *QMMM* |
| *QTB* | *RHEO* | *REACTION* | *REAXFF* | *REPLICA* | *RIGID* |
| *SCAFACOS* | *SHOCK* | *SMTBQ* | *SPH* | *SPIN* | *SRD* |
| *TALLY* | *UEF* | *VORONOI* | *VTK* | *YAFF* | |

### 8.1.1 ADIOS package

**Contents:**

ADIOS is a high-performance I/O library. This package implements the *dump atom/adios*, *dump custom/adios* and *read_dump ... format adios* commands to write and read data using the ADIOS library.

**Authors:** Norbert Podhorszki (ORNL) from the ADIOS developer team.

Added in version 28Feb2019.

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- `src/ADIOS`: filenames -> commands
- `src/ADIOS/README`
- `examples/PACKAGES/adios`
- https://github.com/ornladios/ADIOS2
- *dump atom/adios*
- *dump custom/adios*

- *read_dump*

---

### 8.1.2 AMOEBA package

**Contents:**

Implementation of the AMOEBA and HIPPO polarized force fields originally developed by Jay Ponder's group at the U Washington at St Louis. The LAMMPS implementation is based on Fortran 90 code provided by the Ponder group in their Tinker MD software.

**Authors:** **Josh Rackers and Steve Plimpton (Sandia), Trung Nguyen (U** Chicago)

**Supporting info:**

- `src/AMOEBA`: filenames -> commands
- *AMOEBA and HIPPO howto*
- *pair_style amoeba*
- *pair_style hippo*
- *atom_style amoeba*
- *angle_style amoeba*
- *improper_style amoeba*
- *fix amoeba/bitorsion*
- *fix amoeba/pitorsion*
- tools/tinker/tinker2lmp.py
- `examples/amoeba`

---

### 8.1.3 APIP package

**Contents:**

This package provides adaptive-precision interatomic potentials (APIP) as described in:

D. Immel, R. Drautz and G. Sutmann, "Adaptive-precision potentials for large-scale atomistic simulations", J. Chem. Phys. 162, 114119 (2025) link

D. Immel, R. Drautz and G. Sutmann, "Conservative adaptive-precision interatomic potentials", arXiv:2512.07693 link

Adaptive-precision means, that a fast interatomic potential, such as EAM, is coupled to a precise interatomic potential, such as ACE. This package provides the required pair_styles and fixes to run an efficient, energy-conserving adaptive-precision simulation.

In the context of this package, precision refers to the accuracy of an interatomic potential.

**Authors:**

This package was written by David Immel^1, Ralf Drautz^2 and Godehard Sutmann^1^2.

^1: Forschungszentrum Juelich, Juelich, Germany

^2: Ruhr-University Bochum, Bochum, Germany

**Install:**

The APIP package requires also the installation of ML-PACE, which has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- `src/APIP`: filenames -> commands
- *Howto APIP*
- `examples/PACKAGES/apip`
- *fix atom_weight/apip*
- *fix lambda/apip*
- *fix lambda/la/csp/apip*
- *fix lambda_thermostat/apip*
- *pair_style eam/apip*
- *pair_style lambda/zone/apip*
- *pair_style lambda/input/apip*
- *pair_style pace/apip*

---

### 8.1.4 ASPHERE package

**Contents:**

Computes, time-integration fixes, and pair styles for aspherical particle models including ellipsoids, granular superellipsoids, 2d lines, and 3d triangles.

**Supporting info:**

- `src/ASPHERE`: filenames -> commands
- *Howto spherical*
- *pair_style gayberne*
- *pair_style resquared*
- *pair_style ylz*
- *pair_style line/lj*
- *pair_style tri/lj*
- *pair_style granular/superellipsoid*
- doc/PDF/pair_gayberne_extra.pdf
- doc/PDF/pair_resquared_extra.pdf
- `examples/ASPHERE`
- `examples/ellipse`
- https://www.lammps.org/movies.html#line
- https://www.lammps.org/movies.html#tri

---

### 8.1.5 BOCS package

**Contents:**

This package provides *fix bocs*, a modified version of *fix npt* which includes the pressure correction to the barostat as outlined in:

N. J. H. Dunn and W. G. Noid, "Bottom-up coarse-grained models that accurately describe the structure, pressure, and compressibility of molecular liquids", J. Chem. Phys. 143, 243148 (2015).

The package also includes a pair_style that flexibly defines interactions as a function of the local density and/or gradient of the local density around a central particle. Its per-atom local-density data can be written out with *fix pair* in a form that is compatible with the Bottom-up Open-source Coarse-graining Software (BOCS), an external coarse-graining package that can be used to parameterize such potentials from atomistic simulation data (see link below).

The package is compatible with molecular topologies, allows the user to specify the length-scale and weighting functions for multiple types of local densities, and distinguishes asymmetric local density potentials. (e.g. distinct potentials can govern 1.) solute density around solvent molecules and 2.) solvent density around solute molecules)

The local density potentials implemented define "local density" flexibly with different choices of indicator weighting functions and asymmetry in interactions as generically described in: Michael R. DeLyser and W. G. Noid (2019). "Analysis of local density potentials" The Journal of Chemical Physics 151, : 224106 DOI: 10.1063/1.5128665

The square gradient potential implementation is described in: Michael R. DeLyser and W. G. Noid (2021) "Coarse-grained models for local density gradients" The Journal of Chemical Physics, 156, 034106 DOI: 10.1063/5.0075291

**Authors:** - Nicholas J. H. Dunn and Michael R. DeLyser (The Pennsylvania State University) for *fix bocs* - Michael R. DeLyser, Maria Lesniewski and Will Noid (The Pennsylvania State University) for *pair_style ldd*

**Supporting info:**

- `src/BOCS`: filenames -> commands
- `src/BOCS/README`
- `examples/PACKAGES/bocs`
- *fix bocs*
- *Howto_ldd*
- *pair_style ldd*

The BOCS package in LAMMPS is part of the BOCS software package: https://github.com/noid-group/BOCS

See the following reference for information about the entire package:

Dunn, NJH; Lebold, KM; DeLyser, MR; Rudzinski, JF; Noid, WG. "BOCS: Bottom-Up Open-Source Coarse-Graining Software." J. Phys. Chem. B. 122, 13, 3363-3377 (2018).

Example inputs are in the `examples/PACKAGES/bocs` folder.

---

### 8.1.6 BODY package

**Contents:**

Body-style particles with internal structure. Computes, time-integration fixes, pair styles, as well as the body styles themselves. See the *Howto body* page for an overview.

**Supporting info:**

- `src/BODY` filenames -> commands
- *Howto_body*

- *atom_style body*
- *fix nve/body*
- *pair_style body/nparticle*
- `examples/body`

---

### 8.1.7 BPM package

**Contents:**

Pair styles, bond styles, fixes, and computes for bonded particle models for mesoscale simulations of solids and fracture. See the *Howto bpm* page for an overview.

**Authors:** Joel T. Clemmer (Sandia National Labs)

Added in version 4May2022.

**Supporting info:**

- `src/BPM`: filenames -> commands
- *Howto_bpm*
- *atom_style bpm/sphere*
- *bond_style bpm/rotational*
- *bond_style bpm/spring*
- *compute nbond/atom*
- *fix nve/bpm/sphere*
- *pair_style bpm/spring*
- https://www.lammps.org/movies.html#bpmpackage
- `examples/bpm`

---

### 8.1.8 BROWNIAN package

**Contents:**

This package provides *fix brownian, fix brownian/sphere, and fix brownian/asphere* as well as *fix propel/self* which allow performing Brownian Dynamics time integration of point, spherical and aspherical particles and also support self-propelled particles.

**Authors:** Sam Cameron (University of Bristol), Arthur Straube (Zuse Institute Berlin), Stefan Paquay (while at Brandeis University) (initial version of fix propel/self)

Added in version 14May2021.

Example inputs are in the `examples/PACKAGES/brownian` folder.

---

### 8.1.9 CG-DNA package

**Contents:**

Several pair styles, bond styles, and integration fixes for coarse-grained modelling of single- and double-stranded DNA and RNA based on the oxDNA and oxRNA model of Doye, Louis and Ouldridge. The package includes Langevin-type rigid-body integrators with improved stability.

**Author:** Oliver Henrich (University of Strathclyde, Glasgow).

**Install:**

The CG-DNA package requires that also the *MOLECULE* and *ASPHERE* packages are installed.

**Supporting info:**

- `src/CG-DNA`: filenames -> commands
- `src/CG-DNA/README`
- *pair_style oxdna/**
- *pair_style oxdna2/**
- *pair_style oxdna3/**
- *pair_style oxrna2/**
- *bond_style oxdna/**
- *bond_style oxdna2/**
- *bond_style oxdna3/**
- *bond_style oxrna2/**
- *fix nve/dotc/langevin*
- `examples/PACKAGES/cgdna`

---

### 8.1.10 CG-SPICA package

**Contents:**

Several pair styles and an angle style which implement the coarse-grained SPICA (formerly called SDK) model which enables simulation of biological or soft material systems.

**Original Author:** Axel Kohlmeyer (Temple U).

**Maintainers:** Yusuke Miyazaki and Wataru Shinoda (Okayama U).

**Supporting info:**

- `src/CG-SPICA`: filenames -> commands
- `src/CG-SPICA/README`
- *pair_style lj/spica/**
- *angle_style spica*
- `examples/PACKAGES/cgspica`
- https://www.lammps.org/pictures.html#cg
- https://www.spica-ff.org/

---

### 8.1.11 CLASS2 package

**Contents:**

Bond, angle, dihedral, improper, and pair styles for the COMPASS CLASS2 molecular force field.

**Supporting info:**

- `src/CLASS2`: filenames -> commands
- *bond_style class2*
- *angle_style class2*
- *dihedral_style class2*
- *improper_style class2*
- *pair_style lj/class2*

---

### 8.1.12 COLLOID package

**Contents:**

Coarse-grained finite-size colloidal particles. Pair styles and fix wall styles for colloidal interactions. Includes the Fast Lubrication Dynamics (FLD) method for hydrodynamic interactions, which is a simplified approximation to Stokesian dynamics.

**Authors:** This package includes Fast Lubrication Dynamics pair styles which were created by Amit Kumar and Michael Bybee from Jonathan Higdon's group at UIUC.

**Supporting info:**

- `src/COLLOID`: filenames -> commands
- *fix wall/colloid*
- *pair_style colloid*
- *pair_style yukawa/colloid*
- *pair_style brownian*
- *pair_style lubricate*
- *pair_style lubricateU*
- `examples/colloid`
- `examples/srd`

---

### 8.1.13 COLVARS package

**Contents:**

Colvars stands for collective variables, which can be used to implement various enhanced sampling methods, including Adaptive Biasing Force, Metadynamics, Steered MD, Umbrella Sampling and Restraints. A *fix colvars* command is implemented which wraps a COLVARS library, which implements these methods. simulations.

**Authors:** The COLVARS library is written and maintained by Giacomo Fiorin (NIH, Bethesda, MD, USA) and Jerome Henin (CNRS, Paris, France), originally for the NAMD MD code, but with portability in mind. Axel Kohlmeyer (Temple U) provided the interface to LAMMPS.

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- `src/COLVARS`: filenames -> commands
- https://colvars.github.io/master/colvars-refman-lammps.html
- doc/PDF/colvars-refman-lammps.pdf
- `src/COLVARS/README`
- `lib/colvars/README`
- *fix colvars*
- *group2ndx*
- *ndx2group*
- `examples/PACKAGES/colvars`

---

### 8.1.14 COMPRESS package

**Contents:**

Compressed output of dump files via the zlib compression library, using dump styles with a "gz" in their style name.

To use this package you must have the zlib compression library available on your system.

**Author:** Axel Kohlmeyer (Temple U).

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- `src/COMPRESS`: filenames -> commands
- `src/COMPRESS/README`
- `lib/compress/README`
- *dump atom/gz*
- *dump cfg/gz*
- *dump custom/gz*
- *dump xyz/gz*

---

### 8.1.15 CORESHELL package

**Contents:**

Compute and pair styles that implement the adiabatic core/shell model for polarizability. The pair styles augment Born, Buckingham, and Lennard-Jones styles with core/shell capabilities. The *compute temp/cs* command calculates the temperature of a system with core/shell particles. See the *Howto coreshell* page for an overview of how to use this package.

**Author:** Hendrik Heenen (Technical U of Munich).

**Supporting info:**

- `src/CORESHELL`: filenames -> commands
- *Howto coreshell*
- *Howto polarizable*
- *compute temp/cs*
- *pair_style born/coul/long/cs*
- *pair_style buck/coul/long/cs*
- *pair_style lj/cut/coul/long/cs*
- `examples/coreshell`

---

### 8.1.16 DIELECTRIC package

**Contents:**

An atom style, multiple pair styles, several fixes, Kspace styles and a compute for simulating systems using boundary element solvers for computing the induced charges at the interface between two media with different dielectric constants.

**Install:**

To use this package, also the *KSPACE* and *EXTRA-PAIR* packages need to be installed.

**Author:** Trung Nguyen and Monica Olvera de la Cruz (Northwestern U)

Added in version 2Jul2021.

**Supporting info:**

- `src/DIELECTRIC`: filenames -> commands
- *atom_style dielectric*
- *pair_style coul/cut/dielectric*
- *pair_style coul/long/dielectric*
- *pair_style lj/cut/coul/cut/dielectric*
- *pair_style lj/cut/coul/debye/dielectric*
- *pair_style lj/cut/coul/long/dielectric*
- *pair_style lj/cut/coul/msm/dielectric*
- *pair_style pppm/dielectric*
- *pair_style pppm/disp/dielectric*
- *pair_style msm/dielectric*
- *fix_style polarize/bem/icc*
- *fix_style polarize/bem/gmres*
- *fix_style polarize/functional*
- *compute efield/atom*
- `examples/PACKAGES/dielectric`

---

### 8.1.17 DIFFRACTION package

**Contents:**

Two computes and a fix for calculating x-ray and electron diffraction intensities based on kinematic diffraction theory.

**Author:** Shawn Coleman while at the U Arkansas.

**Supporting info:**

- `src/DIFFRACTION`: filenames -> commands
- *compute saed*
- *compute xrd*
- *fix saed/vtk*
- `examples/PACKAGES/diffraction`

---

### 8.1.18 DIPOLE package

**Contents:**

An atom style and several pair styles for point dipole models with short-range or long-range interactions.

**Supporting info:**

- `src/DIPOLE`: filenames -> commands
- *atom_style dipole*
- *pair_style lj/cut/dipole/cut*
- *pair_style lj/cut/dipole/long*
- *pair_style lj/long/dipole/long*
- *angle_style dipole*
- `examples/dipole`

---

### 8.1.19 DPD-BASIC package

**Contents:**

Pair styles for the basic dissipative particle dynamics (DPD) method and DPD thermostatting.

Pair style *dpd/coul/slater/long* also includes smeared charges for coulomb interactions and thus requires the *KSPACE* package to be installed to handle the long-range Coulomb part of the interactions.

**Authors:** Kurt Smith (U Pittsburgh), Martin Svoboda, Martin Lisal (ICPF and UJEP), Eddy Barraud (IFPEN)

**Supporting info:**

- `src/DPD-BASIC`: filenames -> commands
- *pair_style dpd*
- *pair_style dpd/tstat*
- *pair_style dpd/ext*
- *pair_style dpd/ext/tstat*

- *pair_style dpd/coul/slater/long*
- `examples/PACKAGES/dpd-basic`

---

### 8.1.20 DPD-MESO package

**Contents:**

Several extensions of the dissipative particle dynamics (DPD) method. Specifically, energy-conserving DPD (eDPD) that can model non-isothermal processes, many-body DPD (mDPD) for simulating vapor-liquid coexistence, and transport DPD (tDPD) for modeling advection-diffusion-reaction systems. The equations of motion of these DPD extensions are integrated through a modified velocity-Verlet (MVV) algorithm.

**Author:** Zhen Li (Department of Mechanical Engineering, Clemson University)

**Supporting info:**

- `src/DPD-MESO`: filenames -> commands
- `src/DPD-MESO/README`
- *atom_style edpd*
- *pair_style edpd*
- *pair_style mdpd*
- *pair_style tdpd*
- *fix mvv/dpd*
- `examples/PACKAGES/mesodpd`

---

### 8.1.21 DPD-REACT package

**Contents:**

DPD stands for dissipative particle dynamics. This package implements coarse-grained DPD-based models for energetic, reactive molecular crystalline materials. It includes many pair styles specific to these systems, including for reactive DPD, where each particle has internal state for multiple species and a coupled set of chemical reaction ODEs are integrated each timestep. Highly accurate time integrators for isothermal, isoenergetic, isobaric and isenthalpic conditions are included. These enable long timesteps via the Shardlow splitting algorithm.

**Authors:** Jim Larentzos (ARL), Tim Mattox (Engility Corp), and John Brennan (ARL).

**Supporting info:**

- `src/DPD-REACT`: filenames -> commands
- `src/DPD-REACT/README`
- *compute dpd*
- *compute dpd/atom*
- *fix eos/cv*
- *fix eos/table*
- *fix eos/table/rx*
- *fix shardlow*

- *fix rx*
- *pair_style table/rx*
- *pair_style dpd/fdt*
- *pair_style dpd/fdt/energy*
- *pair_style exp6/rx*
- *pair_style multi/lucy*
- *pair_style multi/lucy/rx*
- examples/PACKAGES/dpd-react

---

### 8.1.22 DPD-SMOOTH package

**Contents:**

A pair style for smoothed dissipative particle dynamics (SDPD), which is an extension of smoothed particle hydrodynamics (SPH) to mesoscale where thermal fluctuations are important (see the *SPH package*). Also two fixes for moving and rigid body integration of SPH/SDPD particles (particles of atom_style meso).

**Author:** Morteza Jalalvand (Institute for Advanced Studies in Basic Sciences, Iran).

**Supporting info:**

- src/DPD-SMOOTH: filenames -> commands
- src/DPD-SMOOTH/README
- *pair_style sdpd/taitwater/isothermal*
- *fix meso/move*
- *fix rigid/meso*
- examples/PACKAGES/dpd-smooth

---

### 8.1.23 DRUDE package

**Contents:**

Fixes, pair styles, and a compute to simulate thermalized Drude oscillators as a model of polarization. See the *Howto drude* and *Howto drude2* pages for an overview of how to use the package. There are auxiliary tools for using this package in tools/drude.

**Authors:** Alain Dequidt (U Clermont Auvergne), Julien Devemy (CNRS), and Agilio Padua (ENS de Lyon).

**Supporting info:**

- src/DRUDE: filenames -> commands
- *Howto drude*
- *Howto drude2*
- *Howto polarizable*
- src/DRUDE/README
- *fix drude*

- *fix drude/transform/**
- *compute temp/drude*
- *pair_style thole*
- *pair_style lj/cut/thole/long*
- `examples/PACKAGES/drude`
- tools/drude

---

### 8.1.24 EFF package

**Contents:**

EFF stands for electron force field which allows a classical MD code to model electrons as particles of variable radius. This package contains atom, pair, fix and compute styles which implement the eFF as described in A. Jaramillo-Botero, J. Su, Q. An, and W.A. Goddard III, JCC, 2010. The eFF potential was first introduced by Su and Goddard, in 2007. There are auxiliary tools for using this package in tools/eff; see its README file.

**Author:** Andres Jaramillo-Botero (CalTech).

**Supporting info:**

- `src/EFF`: filenames -> commands
- `src/EFF/README`
- *atom_style electron*
- *fix nve/eff*
- *fix nvt/eff*
- *fix npt/eff*
- *fix langevin/eff*
- *compute temp/eff*
- *pair_style eff/cut*
- *pair_style eff/inline*
- `examples/PACKAGES/eff`
- tools/eff/README
- tools/eff
- https://www.lammps.org/movies.html#eff

---

### 8.1.25 ELECTRODE package

**Contents:**

The ELECTRODE package allows the user to enforce a constant potential method for groups of atoms that interact with the remaining atoms as electrolyte.

**Authors:** The ELECTRODE package is written and maintained by Ludwig Ahrens-Iwers (TUHH, Hamburg, Germany), Shern Tee (UQ, Brisbane, Australia) and Robert Meissner (Helmholtz-Zentrum Hereon, Geesthacht and TUHH, Hamburg, Germany).

Added in version 4May2022.

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- *fix electrode/conp*
- *fix electrode/conq*
- *fix electrode/thermo*

---

### 8.1.26 EXTRA-COMMAND package

**Contents:**

Additional command styles that are less commonly used.

**Supporting info:**

- `src/EXTRA-COMMAND`: filenames -> commands
- *general commands*

---

### 8.1.27 EXTRA-COMPUTE package

**Contents:**

Additional compute styles that are less commonly used.

**Supporting info:**

- `src/EXTRA-COMPUTE`: filenames -> commands
- *compute*

---

### 8.1.28 EXTRA-DUMP package

**Contents:**

Additional dump styles that are less commonly used.

**Supporting info:**

- `src/EXTRA-DUMP`: filenames -> commands
- *dump*

---

### 8.1.29 EXTRA-FIX package

**Contents:**

Additional fix styles that are less commonly used.

**Supporting info:**

- `src/EXTRA-FIX`: filenames -> commands
- *fix*

---

### 8.1.30 EXTRA-MOLECULE package

**Contents:**

Additional bond, angle, dihedral, and improper styles that are less commonly used.

**Install:**

To use this package, also the *MOLECULE* package needs to be installed.

**Supporting info:**

- `src/EXTRA-MOLECULE`: filenames -> commands
- *molecular styles*

---

### 8.1.31 EXTRA-PAIR package

**Contents:**

Additional pair styles that are less commonly used.

**Supporting info:**

- `src/EXTRA-PAIR`: filenames -> commands
- *pair_style*
- `examples/PACKAGES/dispersion`

---

### 8.1.32 FEP package

**Contents:**

FEP stands for free energy perturbation. This package provides methods for performing FEP simulations by using a *fix adapt/fep* command with soft-core pair potentials, which have a "soft" in their style name. There are auxiliary tools for using this package in `tools/fep`; see its `README` file.

**Author:** Agilio Padua (ENS de Lyon)

**Supporting info:**

- `src/FEP`: filenames -> commands
- `src/FEP/README`
- *fix adapt/fep*
- *compute fep*

- *pair_style */soft*
- *pair_style coul/cut/soft/gapsys*
- *pair_style lj/cut/soft/gapsys*
- `examples/PACKAGES/fep`
- tools/fep/README
- tools/fep

---

### 8.1.33 GPU package

**Contents:**

Dozens of pair styles and a version of the PPPM long-range Coulombic solver optimized for GPUs. All such styles have a "gpu" as a suffix in their style name. The GPU code can be compiled with either CUDA or OpenCL, however the OpenCL variants are no longer actively maintained and only the CUDA versions are regularly tested. The *GPU package* page gives details of what hardware and GPU software is required on your system, and details on how to build and use this package. Its styles can be invoked at run time via the `-sf gpu` or `-suffix gpu` *command-line switches*. See also the *KOKKOS* package, which has GPU-enabled styles.

**Authors:** Mike Brown (Intel) while at Sandia and ORNL and Trung Nguyen (Northwestern U) while at ORNL and later. AMD HIP support by Evgeny Kuznetsov, Vladimir Stegailov, and Vsevolod Nikolskiy (HSE University).

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- `src/GPU`: filenames -> commands
- `src/GPU/README`
- `lib/gpu/README`
- *Accelerator packages*
- *GPU package*
- *Section 4.2 -sf gpu*
- *Section 4.2 -pk gpu*
- *package gpu*
- *Commands* pages (*pair*, *kspace*) for styles followed by (g)
- Benchmarks page of website

---

### 8.1.34 GRAPHICS package

**Contents:**

Dump styles *image and movie*, supporting classes for rendering images and fonts, several fixes for adding graphics objects to visualizations, and the region2vmd command for exporting visualizations of regions as scripted graphics in VMD.

**Supporting info:**

- `src/GRAPHICS`: filenames -> commands
- *Visualize LAMMPS snapshots*
- *dump image*
- *dump movie*
- *fix graphics/arrows*
- *fix graphics/isosurface*
- *fix graphics/labels*,
- *fix graphics/lines*,
- *fix graphics/objects*,
- *fix graphics/periodic*,
- *region2vmd*
- https://www.youtube.com/watch?v=9HEsGaOsdik
- https://www.youtube.com/watch?v=f4hfPs7aCmI
- https://www.youtube.com/shorts/1QEjIITapwQ
- https://www.youtube.com/shorts/OYn_VVodnIg
- https://www.youtube.com/shorts/4Cm5p0SfgNU

---

### 8.1.35 GRANULAR package

**Contents:**

Pair styles and fixes for finite-size granular particles, which interact with each other and boundaries via frictional and dissipative potentials.

**Supporting info:**

- `src/GRANULAR`: filenames -> commands
- *Howto granular*
- *fix pour*
- *fix wall/gran*
- *pair_style gran/hooke*
- *pair_style gran/hertz/history*
- `examples/granregion`
- `examples/pour`
- bench/in.chute
- https://www.lammps.org/pictures.html#jamming
- https://www.lammps.org/movies.html#hopper
- https://www.lammps.org/movies.html#dem
- https://www.lammps.org/movies.html#brazil
- https://www.lammps.org/movies.html#granregion

### 8.1.36 GRANSURF package

**Contents:**

Granular surfaces consisting of triangles (3d) or line segments (2d). These interact with finite-size granular particles as static or moving boundary conditions and support the same kind of interaction models as granular pair styles do for particle/particle interactions. The collection of triangles or lines can be "global" with each processor storing all of them. Or it can be "local" where the triangles/lines are distributed across processors.

**Supporting info:**

- src/GRANSURF: filenames -> commands
- *Howto granular surfaces*
- *fix surface/global*
- *fix surface/local*
- *pair_style surf/granular*
- examples/gransurf

---

### 8.1.37 H5MD package

**Contents:**

H5MD stands for HDF5 for MD. HDF5 is a portable, binary, self-describing file format, used by many scientific simulations. H5MD is a format for molecular simulations, built on top of HDF5. This package implements a *dump h5md* command to output LAMMPS snapshots in this format.

To use this package you must have the HDF5 library available on your system.

**Author:** Pierre de Buyl (KU Leuven) created both the package and the H5MD format.

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- `src/H5MD`: filenames -> commands
- `src/H5MD/README`
- `lib/h5md/README`
- *dump h5md*

---

### 8.1.38 INTEL package

**Contents:**

Dozens of pair, fix, bond, angle, dihedral, improper, and kspace styles which are optimized for Intel CPUs. All of them have an "intel" in their style name. The *INTEL package* page gives details of what hardware and compilers are required on your system, and how to build and use this package. Its styles can be invoked at run time via the `-sf intel` or `-suffix intel` *command-line switches*. Also see the *KOKKOS*, *OPT*, and *OPENMP* packages, which have styles optimized for CPUs and KNLs.

You need to have an Intel compiler, version 14 or higher to take full advantage of this package. While compilation with GNU compilers is supported, performance will be sub-optimal.

> **Note**
>
> the INTEL package contains styles that require using the -restrict flag, when compiling with Intel compilers.

**Author:** Mike Brown (Intel).

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- `src/INTEL`: filenames -> commands
- `src/INTEL/README`
- *Accelerator packages*
- *INTEL package*
- *Section 4.2 -sf intel*
- *Section 4.2 -pk intel*
- *package intel*
- Search the *commands* pages (*fix*, *compute*, *pair*, *bond, angle, dihedral, improper*, *kspace*) for styles followed by (i)
- `src/INTEL/TEST`
- Benchmarks page of website

---

### 8.1.39 INTERLAYER package

**Contents:**

A collection of pair styles specifically to be used for modeling layered materials, most commonly graphene sheets (or equivalents).

**Supporting info:**

- `src/INTERLAYER`: filenames -> commands
- *Pair style* page
- `examples/PACKAGES/interlayer`

---

### 8.1.40 KIM package

**Contents:**

This package contains a command with a set of sub-commands that serve as a wrapper on the Open Knowledgebase of Interatomic Models (OpenKIM) repository of interatomic models (IMs) enabling compatible ones to be used in LAMMPS simulations.

This includes *kim init*, and *kim interactions* commands to select, initialize and instantiate the IM, a *kim query* command to perform web queries for material property predictions of OpenKIM IMs, a *kim param* command to access KIM Model Parameters from LAMMPS, and a *kim property* command to write material properties computed in LAMMPS to standard KIM property instance format.

Support for KIM IMs that conform to the KIM Application Programming Interface (API) is provided by the *pair_style kim* command.

**Note**

The command *pair_style kim* is called by *kim interactions* and is not recommended to be directly used in input scripts.

To use this package you must have the KIM API library available on your system. The KIM API is available for download on the OpenKIM website. When installing LAMMPS from binary, the kim-api package is a dependency that is automatically downloaded and installed.

Information about the KIM project can be found at its website: https://openkim.org. The KIM project is led by Ellad Tadmor and Ryan Elliott (U Minnesota) and is funded by the National Science Foundation.

**Authors:** Ryan Elliott (U Minnesota) is the main developer for the KIM API and the *pair_style kim* command. Yaser Afshar (U Minnesota), Axel Kohlmeyer (Temple U), Ellad Tadmor (U Minnesota), and Daniel Karls (U Minnesota) contributed to the *kim command* interface in close collaboration with Ryan Elliott.

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- *kim command*
- *pair_style kim*
- `src/KIM`: filenames -> commands
- `src/KIM/README`
- `lib/kim/README`
- `examples/kim`

---

### 8.1.41 KOKKOS package

**Contents:**

Dozens of atom, pair, bond, angle, dihedral, improper, fix, compute styles adapted to compile using the Kokkos library which can convert them to OpenMP or CUDA code so that they run efficiently on multicore CPUs, KNLs, or GPUs. All the styles have a "kk" as a suffix in their style name. The *KOKKOS package* page gives details of what hardware and software is required on your system, and how to build and use this package. Its styles can be invoked at run time via the `-sf kk` or `-suffix kk` *command-line switches*. Also see the *GPU*, *OPT*, *INTEL*, and *OPENMP* packages, which have styles optimized for CPUs, KNLs, and GPUs.

You must have a C++20 compatible compiler to use this package. KOKKOS makes extensive use of advanced C++ features, which can expose compiler bugs, especially when compiling for maximum performance at high optimization levels. Please see the file `lib/kokkos/README` for a list of compilers and their respective platforms, that are known to work.

**Authors:** The KOKKOS package was created primarily by Christian Trott and Stan Moore (Sandia), with contributions from other folks as well. It uses the open-source Kokkos library which was developed by Carter Edwards, Christian Trott, and others at Sandia, and which is included in the LAMMPS distribution in `lib/kokkos`.

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- `src/KOKKOS`: filenames -> commands
- `src/KOKKOS/README`
- `lib/kokkos/README`
- *Accelerator packages*
- *KOKKOS package*
- *Section 4.2 -k on ...*
- *Section 4.2 -sf kk*
- *Section 4.2 -pk kokkos*
- *package kokkos*
- Search the *commands* pages (*fix*, *compute*, *pair*, *bond, angle, dihedral, improper*, *kspace*) for styles followed by (k)
- Benchmarks page of website

---

### 8.1.42 KSPACE package

**Contents:**

A variety of long-range Coulombic solvers, as well as pair styles which compute the corresponding short-range pairwise Coulombic interactions. These include Ewald, particle-particle particle-mesh (PPPM), and multilevel summation method (MSM) solvers.

**Install:**

Building with this package requires a 1d FFT library be present on your system for use by the PPPM solvers. This can be the KISS FFT library provided with LAMMPS, third party libraries like FFTW, or a vendor-supplied FFT library. See the *Build settings* page for details on how to select different FFT options for your LAMMPS build.

**Supporting info:**

- `src/KSPACE`: filenames -> commands
- *kspace_style*
- doc/PDF/kspace.pdf
- *Howto tip3p*
- *Howto tip4p*
- *Howto spc*
- *pair_style coul*
- Search the *pair style* page for styles with "long" or "msm" in name
- `examples/peptide`

- bench/in.rhodo

---

### 8.1.43 LATBOLTZ package

**Contents:**

Fixes which implement a background Lattice-Boltzmann (LB) fluid, which can be used to model MD particles influenced by hydrodynamic forces.

**Authors:** Frances Mackay and Colin Denniston (University of Western Ontario).

**Install:**

The LATBOLTZ package requires that LAMMPS is built in *MPI parallel mode*.

**Supporting info:**

- `src/LATBOLTZ`: filenames -> commands
- `src/LATBOLTZ/README`
- *fix lb/fluid*
- *fix lb/momentum*
- *fix lb/viscous*
- `examples/PACKAGES/latboltz`

---

### 8.1.44 LEPTON package

**Contents:**

Styles for pair, bond, and angle forces that evaluate the potential function from a string using the Lepton mathematical expression parser. Lepton is a C++ library that is bundled with OpenMM and can be used for parsing, evaluating, differentiating, and analyzing mathematical expressions. This is a more lightweight and efficient alternative for evaluating custom potential function to an embedded Python interpreter as used in the *PYTHON package*. On the other hand, since the potentials are evaluated form analytical expressions, they are more precise than what can be done with *tabulated potentials*.

**Authors:** Axel Kohlmeyer (Temple U). Lepton itself is developed by Peter Eastman at Stanford University.

Added in version 8Feb2023.

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- `src/LEPTON`: filenames -> commands
- `lib/lepton/README`.md
- *pair_style lepton*
- *bond_style lepton*
- *angle_style lepton*
- *dihedral_style lepton*

---

### 8.1.45 MACHDYN package

**Contents:**

An atom style, fixes, computes, and several pair styles which implements smoothed Mach dynamics (SMD) for solids, which is a model related to smoothed particle hydrodynamics (SPH) for liquids (see the *SPH package*).

This package solves solids mechanics problems via a state of the art stabilized meshless method with hourglass control. It can specify hydrostatic interactions independently from material strength models, i.e. pressure and deviatoric stresses are separated. It provides many material models (Johnson-Cook, plasticity with hardening, Mie-Grueneisen, Polynomial EOS) and allows new material models to be added. It implements rigid boundary conditions (walls) which can be specified as surface geometries from *.STL files.

**Author:** Georg Ganzenmuller (Fraunhofer-Institute for High-Speed Dynamics, Ernst Mach Institute, Germany).

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- `src/MACHDYN`: filenames -> commands
- `src/MACHDYN/README`
- doc/PDF/MACHDYN_LAMMPS_userguide.pdf
- `examples/PACKAGES/machdyn`

---

### 8.1.46 MANIFOLD package

**Contents:**

Several fixes and a "manifold" class which enable simulations of particles constrained to a manifold (a 2D surface within the 3D simulation box). This is done by applying the RATTLE constraint algorithm to formulate single-particle constraint functions g(xi,yi,zi) = 0 and their derivative (i.e. the normal of the manifold) n = grad(g).

**Author:** Stefan Paquay (until 2017: Eindhoven University of Technology (TU/e), The Netherlands; since 2017: Brandeis University, Waltham, MA, USA)

**Supporting info:**

- `src/MANIFOLD`: filenames -> commands
- `src/MANIFOLD/README`
- *Howto manifold*
- *fix manifoldforce*
- *fix nve/manifold/rattle*
- *fix nvt/manifold/rattle*
- `examples/PACKAGES/manifold`
- https://www.lammps.org/movies.html#manifold

---

### 8.1.47 MANYBODY package

**Contents:**

A variety of many-body and bond-order potentials. These include (AI)REBO, BOP, EAM, EIM, Stillinger-Weber, and Tersoff potentials.

**Supporting info:**

- `src/MANYBODY`: filenames -> commands
- *Pair style* page
- `examples/comb`
- `examples/eim`
- `examples/nb3d`
- `examples/shear`
- `examples/streitz`
- `examples/vashishta`
- bench/in.eam

---

### 8.1.48 MBX Package

**Contents**

The pair_style mbx command implements the MBX library for MB-pol and MB-nrg data-driven many-body potential energy functions. MBX is called using *pair_style mbx* command, which allows for MB-nrg potentials such as MB-pol to be used in LAMMPS. For more information on MBX, see the MBX library website.

**Authors:** The MBX library is developed by the Paesani group at the University of California, San Diego. Major contributors include: Marc Riera, Christopher Knight, Ethan Bull-Vulpe, and Henry Agnew.

Added in version 11Feb2026.

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- `src/MBX`: filenames -> commands
- *pair_style mbx*
- https://mbxsimulations.com/

---

### 8.1.49 MC package

**Contents:**

Several fixes and a pair style that have Monte Carlo (MC) or MC-like attributes. These include fixes for creating, breaking, and swapping bonds, for performing atomic swaps, and performing grand canonical MC (GCMC), semi-grand canonical MC (SGCMC), Gibbs ensemble MC (GEMC) or similar processes in conjunction with molecular dynamics (MD).

**Supporting info:**

- `src/MC`: filenames -> commands
- *fix atom/swap*
- *fix bond/break*
- *fix bond/create*
- *fix bond/create/angle*
- *fix bond/swap*
- *fix charge/regulation*
- *fix gcmc*
- *fix gemc*
- *fix hmc*
- *fix mol/swap*
- *fix neighbor/swap*
- *fix sgcmc*
- *fix tfmc*
- *fix widom*
- *pair_style dsmc*
- https://www.lammps.org/movies.html#gcmc

---

### 8.1.50 MDI package

**Contents:**

A LAMMPS command and fixes to allow client-server coupling of LAMMPS to other atomic or molecular simulation codes or materials modeling workflows via the MolSSI Driver Interface (MDI) library.

**Author:** Taylor Barnes - MolSSI, taylor.a.barnes at gmail.com

Added in version 14May2021.

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- `src/MDI/README`
- `lib/mdi/README`
- *Howto MDI*
- *mdi*
- *fix mdi/qm*
- `examples/PACKAGES/mdi`

---

### 8.1.51 MEAM package

**Contents:**

A pair style for the modified embedded atom (MEAM) potential translated from the Fortran version in the (obsolete) MEAM package to plain C++. The MEAM fully replaces the MEAM package, which has been removed from LAMMPS after the 12 December 2018 version.

**Author:** Sebastian Huetter, (Otto-von-Guericke University Magdeburg) based on the Fortran version of Greg Wagner (Northwestern U) while at Sandia.

**Supporting info:**

- `src/MEAM`: filenames -> commands
- `src/MEAM/README`
- *pair_style meam*
- `examples/meam`

---

### 8.1.52 MESONT package

**Contents:**

MESONT is a LAMMPS package for simulation of nanomechanics of nanotubes (NTs). The model is based on a coarse-grained representation of NTs as "flexible cylinders" consisting of a variable number of segments. Internal interactions within a NT and the van der Waals interaction between the tubes are described by a mesoscopic force field designed and parameterized based on the results of atomic-level molecular dynamics simulations. The description of the force field is provided in the papers listed in `src/MESONT/README`.

This package used to have two independent implementations of this model: the original implementation using a Fortran library written by the developers of the model and a second implementation written in C++ by Philipp Kloza (U Cambridge). Since the C++ implementation offers the same features as the original implementation with the addition of friction, is typically faster, and easier to compile/install, the Fortran library based implementation has since been obsoleted and removed from the distribution. You have to download and compile an older version of LAMMPS if you want to use those.

**Download of potential files:**

The potential files for these pair styles are *very* large and thus are not included in the regular downloaded packages of LAMMPS or the git repositories. Instead, they will be automatically downloaded from a web server when the package is installed for the first time.

**Authors of the obsoleted *mesont* styles:**

Maxim V. Shugaev (University of Virginia), Alexey N. Volkov (University of Alabama), Leonid V. Zhigilei (University of Virginia)

Deprecated since version 8Feb2023.

**Author of the C++ styles:** Philipp Kloza (U Cambridge)

Added in version 15Jun2020.

**Supporting info:**

- `src/MESONT`: filenames -> commands
- `src/MESONT/README`
- *bond_style mesocnt*

- *angle_style mesocnt*
- *pair_style mesocnt*
- `examples/PACKAGES/mesont`

---

### 8.1.53 MGPT package

**Contents:**

A pair style which provides a fast implementation of the quantum-based MGPT multi-ion potentials. The MGPT or model GPT method derives from first-principles DFT-based generalized pseudopotential theory (GPT) through a series of systematic approximations valid for mid-period transition metals with nearly half-filled d bands. The MGPT method was originally developed by John Moriarty at LLNL. The pair style in this package calculates forces and energies using an optimized matrix-MGPT algorithm due to Tomas Oppelstrup at LLNL.

**Authors:** Tomas Oppelstrup and John Moriarty (LLNL).

**Supporting info:**

- `src/MGPT`: filenames -> commands
- `src/MGPT/README`
- *pair_style mgpt*
- `examples/PACKAGES/mgpt`

---

### 8.1.54 MISC package

**Contents:**

A variety of compute, fix, pair, bond styles with specialized capabilities that don't align with other packages. Do a directory listing, `ls src/MISC`, to see the list of commands.

> **Note**
>
> the MISC package contains styles that require using the -restrict flag, when compiling with Intel compilers.

**Supporting info:**

- `src/MISC`: filenames -> commands
- *bond_style special*
- *compute viscosity/cos*
- *fix accelerate/cos*
- *fix imd*
- *fix ipi*
- *pair_style agni*
- *pair_style list*
- *pair_style srp*
- *pair_style tracker*

### 8.1.55 ML-HDNNP package

**Contents:**

A *pair_style hdnnp* command which allows the use of high-dimensional neural network potentials (HDNNPs), a form of machine learning potentials. HDNNPs must be carefully trained prior to their application in a molecular dynamics simulation.

To use this package you must have the n2p2 library installed and compiled on your system.

**Author:** Andreas Singraber

Added in version 27May2021.

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- `src/ML-HDNNP`: filenames -> commands
- `src/ML-HDNNP/README`
- `lib/hdnnp/README`
- *pair_style hdnnp*
- `examples/PACKAGES/hdnnp`

### 8.1.56 ML-RUNNER package

**Contents:** A *pair_style runner* command for the efficient evaluation of second-, third-, and fourth-generation high-dimensional neural network potentials (HDNNPs).

To use this package you must have the RuNNer library compiled on your system.

**Authors: K. Nikolas Lausch, Alexander L. M. Knoll, Moritz R. Schaefer,**
Gunnar Schmitz, Joerg Behler (Ruhr-University Bochum)

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

Added in version 4Jul2026.

**Supporting info:**

- `src/ML-RUNNER`: filenames -> commands
- *pair_style runner*
- `examples/PACKAGES/ml-runner` (see README.txt)

### 8.1.57 ML-IAP package

**Contents:**

A general interface for machine-learning interatomic potentials, including PyTorch.

**Install:**

To use this package, also the *ML-SNAP* package needs to be installed. To make the *mliappy* model available, also the *PYTHON* package needs to be installed, the version of Python must be 3.6 or later, and the cython software must be installed.

**Author:** Aidan Thompson (Sandia), Nicholas Lubbers (LANL).

Added in version 30Jun2020.

**Supporting info:**

- `src/ML-IAP`: filenames -> commands
- `src/ML-IAP/README.md`
- *pair_style mliap*
- *compute_style mliap*
- `examples/mliap` (see README)

When built with the *mliappy* model this package includes an extension for coupling with Python models, including PyTorch. In this case, the Python interpreter linked to LAMMPS will need the `cython` and `numpy` modules installed. The provided examples build models with PyTorch, which would therefore also needs to be installed to run those examples.

---

### 8.1.58 ML-PACE package

**Contents:**

A pair style for the Atomic Cluster Expansion potential (ACE). ACE is a methodology for deriving a highly accurate classical potential fit to a large archive of quantum mechanical (DFT) data. The ML-PACE package provides an efficient implementation for running simulations with ACE potentials.

**Authors:**

This package was written by Yury Lysogorskiy^1, Cas van der Oord^2, Anton Bochkarev^1, Sarath Menon^1, Matteo Rinaldi^1, Thomas Hammerschmidt^1, Matous Mrovec^1, Aidan Thompson^3, Gabor Csanyi^2, Christoph Ortner^4, Ralf Drautz^1.

^1: Ruhr-University Bochum, Bochum, Germany

^2: University of Cambridge, Cambridge, United Kingdom

^3: Sandia National Laboratories, Albuquerque, New Mexico, USA

^4: University of British Columbia, Vancouver, BC, Canada

Added in version 14May2021.

**Install:**

This package has *specific installation instructions* on the *Build extras* page. This package may also be compiled as a plugin to avoid licensing conflicts when distributing binaries.

**Supporting info:**

- `src/ML-PACE`: filenames -> commands

- *pair_style pace*
- `examples/PACKAGES/pace`

---

### 8.1.59 ML-POD package

**Contents:**

A pair style and fitpod style for Proper Orthogonal Descriptors (POD). POD is a methodology for deriving descriptors based on the proper orthogonal decomposition. The ML-POD package provides an efficient implementation for running simulations with POD potentials, along with fitting the potentials natively in LAMMPS.

**Authors:**

Ngoc Cuong Nguyen (MIT), Andrew Rohskopf (Sandia)

Added in version 22Dec2022.

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- `src/ML-POD`: filenames -> commands
- *pair_style pod*
- *command_style fitpod*
- `examples/PACKAGES/pod`

---

### 8.1.60 ML-QUIP package

**Contents:**

A *pair_style quip* command which wraps the QUIP libAtoms library, which includes a variety of interatomic potentials, including Gaussian Approximation Potential (GAP) models developed by the Cambridge University group.

To use this package you must have the QUIP libAtoms library available on your system.

**Author:** Albert Bartok (Cambridge University)

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- `src/ML-QUIP`: filenames -> commands
- `src/ML-QUIP/README`
- *pair_style quip*
- `examples/PACKAGES/quip`

---

### 8.1.61 ML-RANN package

**Contents:**

A pair style for using rapid atomistic neural network (RANN) potentials. These neural network potentials work by first generating a series of symmetry functions from the neighbor list and then using these values as the input layer of a neural network.

**Authors:**

This package was written by Christopher Barrett with contributions by Doyl Dickel, Mississippi State University.

Added in version 27May2021.

**Supporting info:**

- `src/ML-RANN`: filenames -> commands
- *pair_style rann*
- `examples/PACKAGES/rann`

---

### 8.1.62 ML-SNAP package

**Contents:**

A pair style for the spectral neighbor analysis potential (SNAP). SNAP is methodology for deriving a highly accurate classical potential fit to a large archive of quantum mechanical (DFT) data. Also several computes which analyze attributes of the potential.

**Author:** Aidan Thompson (Sandia).

**Supporting info:**

- `src/ML-SNAP`: filenames -> commands
- *pair_style snap*
- *compute sna/atom*
- *compute sna/grid*
- *compute sna/grid/local*
- *compute snad/atom*
- *compute snav/atom*
- `examples/snap`

---

### 8.1.63 ML-UF3 package

**Contents:**

A pair style for the ultra-fast force field potentials (UF3). UF3 is a methodology for deriving a highly accurate classical potential which is fast to evaluate and is fitted to a large archives of quantum mechanical (DFT) data. The use of b-spline basis set in UF3 enables the rapid evaluation of 2-body and 3-body interactions.

**Authors:** Ajinkya C Hire (University of Florida), Hendrik Krass (University of Constance), Matthias Rupp (Luxembourg Institute of Science and Technology), Richard Hennig (University of Florida)

**Supporting info:**

- src/ML-UF3: filenames -> commands
- *pair_style uf3*
- examples/uf3
- https://github.com/uf3/uf3

### 8.1.64 MOFFF package

**Contents:**

Pair, angle and improper styles needed to employ the MOF-FF force field by Schmid and coworkers with LAMMPS. MOF-FF is a first principles derived force field with the primary aim to simulate MOFs and related porous framework materials, using spherical Gaussian charges. It is described in S. Bureekaew et al., Phys. Stat. Sol. B 2013, 250, 1128-1141. For the usage of MOF-FF see the example in the example directory as well as the MOF+ website.

**Author:** Hendrik Heenen (Technical U of Munich), Rochus Schmid (Ruhr-University Bochum).

**Supporting info:**

- src/MOFFF: filenames -> commands
- src/MOFFF/README
- *pair_style buck6d/coul/gauss*
- *angle_style class2*
- *angle_style cosine/buck6d*
- *improper_style inversion/harmonic*
- examples/PACKAGES/mofff

---

### 8.1.65 MOLECULE package

**Contents:**

A large number of atom, pair, bond, angle, dihedral, improper styles that are used to model molecular systems with fixed covalent bonds. The pair styles include the Dreiding (hydrogen-bonding) and CHARMM force fields, and a TIP4P water model.

**Supporting info:**

- src/MOLECULE: filenames -> commands
- *atom_style*
- *bond_style*
- *angle_style*
- *dihedral_style*
- *improper_style*
- *pair_style hbond/dreiding/lj*
- *pair_style lj/charmm/coul/charmm*
- *Howto bioFF*
- examples/cmap
- examples/dreiding

- examples/micelle,
- examples/peptide
- bench/in.chain
- bench/in.rhodo

---

### 8.1.66 MOLFILE package

**Contents:**

A *dump molfile* command which uses molfile plugins that are bundled with the VMD molecular visualization and analysis program, to enable LAMMPS to dump snapshots in formats compatible with various molecular simulation tools.

To use this package you must have the desired VMD plugins available on your system.

Note that this package only provides the interface code, not the plugins themselves, which will be accessed when requesting a specific plugin via the *dump molfile* command. Plugins can be obtained from a VMD installation which has to match the platform that you are using to compile LAMMPS for. By adding plugins to VMD, support for new file formats can be added to LAMMPS (or VMD or other programs that use them) without having to re-compile the application itself. More information about the VMD molfile plugins can be found at https://www.ks.uiuc.edu/Research/vmd/plugins/molfile.

**Author:** Axel Kohlmeyer (Temple U).

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- src/MOLFILE: filenames -> commands
- src/MOLFILE/README
- lib/molfile/README
- *dump molfile*

---

### 8.1.67 NETCDF package

**Contents:**

Dump styles for writing NetCDF formatted dump files. NetCDF is a portable, binary, self-describing file format developed on top of HDF5. The file contents follow the AMBER NetCDF trajectory conventions (https://ambermd.org/netcdf/nctraj.xhtml), but include extensions.

To use this package you must have the NetCDF library available on your system.

Note that NetCDF files can be directly visualized with the following tools:

- Ovito (Ovito supports the AMBER convention and the extensions mentioned above)
- VMD

**Author:** Lars Pastewka (Karlsruhe Institute of Technology).

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- `src/NETCDF`: filenames -> commands
- `src/NETCDF/README`
- `lib/netcdf/README`
- *dump netcdf*

---

### 8.1.68 OPENMP package

**Contents:**

Hundreds of pair, fix, compute, bond, angle, dihedral, improper, and kspace styles which are altered to enable threading on many-core CPUs via OpenMP directives. All of them have an "omp" in their style name. The *OPENMP package* page gives details of what hardware and compilers are required on your system, and how to build and use this package. Its styles can be invoked at run time via the `-sf omp` or `-suffix omp` *command-line switches*. Also see the *KOKKOS*, *OPT*, and *INTEL* packages, which have styles optimized for CPUs.

**Author:** Axel Kohlmeyer (Temple U).

> **Note**
>
> To enable multi-threading support the compile flag `-fopenmp` and the link flag `-fopenmp` (for GNU compilers, you have to look up the equivalent flags for other compilers) must be used to build LAMMPS. When using Intel compilers, also the `-restrict` flag is required. The OPENMP package can be compiled without enabling OpenMP; then all code will be compiled as serial and the only improvement over the regular styles are some data access optimization. These flags should be added to the CCFLAGS and LINKFLAGS lines of your Makefile.machine. See `src/MAKE/OPTIONS/Makefile`.omp for an example.

Once you have an appropriate Makefile.machine, you can install/uninstall the package and build LAMMPS in the usual manner:

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- `src/OPENMP`: filenames -> commands
- `src/OPENMP/README`
- *Accelerator packages*
- *OPENMP package*
- *Command-line option -suffix/-sf omp*
- *Command-line option -package/-pk omp*
- *package omp*
- Search the *commands* pages (*fix*, *compute*, *pair*, *bond, angle, dihedral, improper*, *kspace*) for styles followed by (o)
- Benchmarks page of website

---

### 8.1.69 OPT package

**Contents:**

A handful of pair styles which are optimized for improved CPU performance on single or multiple cores. These include EAM, LJ, CHARMM, and Morse potentials. The styles have an "opt" suffix in their style name. The *OPT package* page gives details of how to build and use this package. Its styles can be invoked at run time via the `-sf opt` or `-suffix opt` *command-line switches*. See also the *KOKKOS*, *INTEL*, and *OPENMP* packages, which have styles optimized for CPU performance.

**Authors:** James Fischer (High Performance Technologies), David Richie, and Vincent Natoli (Stone Ridge Technology).

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- `src/OPT`: filenames -> commands
- *Accelerator packages*
- *OPT package*
- *Section 4.2 -sf opt*
- Search the *pair style* page for styles followed by (t)
- Benchmarks page of website

### 8.1.70 ORIENT package

**Contents:**

A few fixes that apply orientation dependent forces for studying grain boundary migration.

**Supporting info:**

- `src/ORIENT`: filenames -> commands
- *fix orient/bcc*
- *fix orient/fcc*
- *fix orient/eco*

---

### 8.1.71 PERI package

**Contents:**

An atom style, several pair styles which implement different Peridynamics materials models, and several computes which calculate diagnostics. Peridynamics is a particle-based meshless continuum model.

**Authors:** The original package was created by Mike Parks (Sandia). Additional Peridynamics models were added by Rezwanur Rahman and John Foster (UTSA).

**Supporting info:**

- `src/PERI`: filenames -> commands
- *Peridynamics Howto*
- doc/PDF/PDLammps_overview.pdf

- doc/PDF/PDLammps_EPS.pdf
- doc/PDF/PDLammps_VES.pdf
- *atom_style peri*
- *pair_style peri/**
- *compute damage/atom*
- *compute plasticity/atom*
- `examples/peri`
- https://www.lammps.org/movies.html#impact

---

### 8.1.72 PHONON package

**Contents:**

A *fix phonon* command that calculates dynamical matrices, which can then be used to compute phonon dispersion relations, directly from molecular dynamics simulations. And a *dynamical_matrix* as well as a *third_order* command to compute the dynamical matrix and third order tensor from finite differences.

**Install:**

The fix phonon command also requires that the *KSPACE* package is installed.

**Authors:** Ling-Ti Kong (Shanghai Jiao Tong University) for "fix phonon" and Charlie Sievers (UC Davis) for "dynamical_matrix" and "third_order"

**Supporting info:**

- `src/PHONON`: filenames -> commands
- `src/PHONON/README`
- *fix phonon*
- *dynamical_matrix*
- *third_order*
- `examples/PACKAGES/phonon`

---

### 8.1.73 PLUGIN package

**Contents:**

A *plugin* command that can load and unload several kind of styles in LAMMPS from shared object files at runtime without having to recompile and relink LAMMPS.

When the environment variable `LAMMPS_PLUGIN_PATH` is set, then LAMMPS will search the directory (or directories) listed in this path for files with names that end in `plugin.so` (e.g. `helloplugin.so`) and will try to load the contained plugins automatically at start-up.

**Authors:** Axel Kohlmeyer (Temple U)

Added in version 8Apr2021.

**Supporting info:**

- `src/PLUGIN`: filenames -> commands

- *plugin command*
- *Information on writing plugins*
- `examples/plugin`

---

### 8.1.74 PLUMED package

**Contents:**

The fix plumed command allows you to use the PLUMED free energy plugin for molecular dynamics to analyze and bias your LAMMPS trajectory on the fly. The PLUMED library is called from within the LAMMPS input script by using the *fix plumed* command.

**Authors:** The PLUMED library is written and maintained by Massimilliano Bonomi, Giovanni Bussi, Carlo Camiloni, and Gareth Tribello.

**Install:**

This package has *specific installation instructions* on the *Build extras* page. This package may also be compiled as a plugin to avoid licensing conflicts when distributing binaries.

**Supporting info:**

- `src/PLUMED/README`
- `lib/plumed/README`
- *fix plumed*
- `examples/PACKAGES/plumed`

---

### 8.1.75 PTM package

**Contents:**

A *compute ptm/atom* command that calculates local structure characterization using the Polyhedral Template Matching methodology.

**Author:** Peter Mahler Larsen (MIT).

**Supporting info:**

- `src/PTM`: filenames not starting with ptm_ -> commands
- `src/PTM`: filenames starting with ptm_ -> supporting code
- `src/PTM/LICENSE`
- *compute ptm/atom*

---

### 8.1.76 PYTHON package

**Contents:**

A *python* command which allow you to execute Python code from a LAMMPS input script. The code can be in a separate file or embedded in the input script itself. See the *Python call* page for an overview of using Python from LAMMPS in this manner and all the *Python* manual pages for other ways to use LAMMPS and Python together.

> **Note**
>
> Building with the PYTHON package assumes you have a Python development environment (headers and libraries) available on your system, which needs to be Python version 3.6 or later.

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- `src/PYTHON`: filenames -> commands
- *Python call*
- `lib/python/README`
- `examples/python`

---

### 8.1.77 QEQ package

**Contents:**

Several fixes for performing charge equilibration (QEq) via different algorithms. These can be used with pair styles that perform QEq as part of their formulation.

**Supporting info:**

- `src/QEQ`: filenames -> commands
- *fix qeq/** 
- `examples/qeq`
- `examples/streitz`

---

### 8.1.78 QMMM package

**Contents:**

A *fix qmmm* command which allows LAMMPS to be used as the MM code in a QM/MM simulation. This is currently only available in combination with the Quantum ESPRESSO package.

To use this package you must have Quantum ESPRESSO (QE) available on your system and include its coupling library in the compilation and then compile LAMMPS as a library. For QM/MM calculations you then build a custom binary with MPI support, that sets up 3 partitions with MPI sub-communicators (for inter- and intra-partition communication) and then calls the corresponding library interfaces on each partition (2x LAMMPS and 1x QE).

The current implementation supports an ONIOM style mechanical coupling and a multi-pole based electrostatic coupling to the Quantum ESPRESSO plane wave DFT package. The QM/MM interface has been written in a manner that coupling to other QM codes should be possible without changes to LAMMPS itself.

**Authors:** Axel Kohlmeyer (Temple U). Mariella Ippolito and Carlo Cavazzoni (CINECA, Italy)

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- src/QMMM: filenames -> commands
- src/QMMM/README
- lib/qmmm/README
- *fix phonon*
- lib/qmmm/example-ec/README
- lib/qmmm/example-mc/README

---

### 8.1.79 QTB package

**Contents:**

Two fixes which provide a self-consistent quantum treatment of vibrational modes in a classical molecular dynamics simulation. By coupling the MD simulation to a colored thermostat, it introduces zero point energy into the system, altering the energy power spectrum and the heat capacity to account for their quantum nature. This is useful when modeling systems at temperatures lower than their classical limits or when temperatures ramp across the classical limits in a simulation.

**Author:** Yuan Shen (Stanford U).

**Supporting info:**

- src/QTB: filenames -> commands
- src/QTB/README
- *fix qtb*
- *fix qbmsst*
- examples/PACKAGES/qtb

---

### 8.1.80 REACTION package

**Contents:**

This package implements the REACTER protocol, which allows for complex bond topology changes (reactions) during a running MD simulation when using classical force fields. Topology changes are defined in pre- and post-reaction molecule templates and can include creation and deletion of bonds, angles, dihedrals, impropers, atom types, bond types, angle types, dihedral types, improper types, and/or atomic charges. Other options currently available include reaction constraints (e.g., angle and Arrhenius constraints), deletion of reaction byproducts or other small molecules, creation of new atoms or molecules bonded to existing atoms, and using LAMMPS variables for input parameters.

**Author:** Jacob R. Gissinger (NASA Langley Research Center).

**Supporting info:**

- src/REACTION: filenames -> commands
- src/REACTION/README
- *fix bond/react*
- examples/PACKAGES/reaction
- 2017 LAMMPS Workshop
- 2019 LAMMPS Workshop

- 2021 LAMMPS Workshop
- REACTER website (reacter.org)

---

### 8.1.81 REAXFF package

**Contents:**

A pair style which implements the ReaxFF potential in C/C++. ReaxFF is a universal reactive force field. See the `src/REAXFF/README` file for more info on differences between the two packages. Also two fixes for monitoring molecules as bonds are created and destroyed.

**Author:** Hasan Metin Aktulga (MSU) while at Purdue University.

**Supporting info:**

- `src/REAXFF`: filenames -> commands
- `src/REAXFF/README`
- *pair_style reaxff*
- *fix reaxff/bonds*
- *fix reaxff/species*
- `examples/reaxff`

---

### 8.1.82 REPLICA package

**Contents:**

A collection of multi-replica methods which can be used when running multiple LAMMPS simulations (replicas). See the *Howto replica* page for an overview of how to run multi-replica simulations in LAMMPS. Methods in the package include nudged elastic band (NEB), parallel replica dynamics (PRD), temperature accelerated dynamics (TAD), parallel tempering, and a verlet/split algorithm for performing long-range Coulombics on one set of processors, and the remainder of the force field calculation on another set.

**Supporting info:**

- `src/REPLICA`: filenames -> commands
- *Howto replica*
- *neb*
- *prd*
- *tad*
- *temper*,
- *temper/npt*,
- *temper/grem*,
- *run_style verlet/split*
- `examples/neb`
- `examples/prd`
- `examples/tad`

- examples/PACKAGES/grem

---

### 8.1.83 RHEO package

**Contents:**

Pair styles, bond styles, fixes, and computes for reproducing hydrodynamics and elastic objects. See the *Howto rheo* page for an overview.

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Authors:** Joel T. Clemmer (Sandia National Labs), Thomas C. O'Connor (Carnegie Mellon University)

Added in version 29Aug2024.

**Supporting info:**

- src/RHEO filenames -> commands
- *Howto_rheo*
- *atom_style rheo*
- *atom_style rheo/thermal*
- *bond_style rheo/shell*
- *compute rheo/property/atom*
- *fix rheo*
- *fix rheo/oxidation*
- *fix rheo/pressure*
- *fix rheo/thermal*
- *fix rheo/viscosity*
- *pair_style rheo*
- *pair_style rheo/solid*
- https://www.lammps.org/movies.html#rheopackage
- examples/rheo

---

### 8.1.84 RIGID package

**Contents:**

Fixes which enforce rigid constraints on collections of atoms or particles. This includes SHAKE and RATTLE, as well as various rigid-body integrators for a few large bodies or many small bodies. Also several computes which calculate properties of rigid bodies.

**Supporting info:**

- src/RIGID: filenames -> commands
- *compute erotate/rigid*
- *fix shake*

- *fix rattle*
- *fix rigid/**
- `examples/ASPHERE`
- `examples/rigid`
- bench/in.rhodo
- https://www.lammps.org/movies.html#box
- https://www.lammps.org/movies.html#star

---

### 8.1.85 SCAFACOS package

**Contents:**

A KSpace style which wraps the ScaFaCoS Coulomb solver library to compute long-range Coulombic interactions.

To use this package you must have the ScaFaCoS library available on your system.

**Author:** Rene Halver (JSC) wrote the scafacos LAMMPS command.

ScaFaCoS itself was developed by a consortium of German research facilities with a BMBF (German Ministry of Science and Education) funded project in 2009-2012. Participants of the consortium were the Universities of Bonn, Chemnitz, Stuttgart, and Wuppertal as well as the Forschungszentrum Juelich.

**Install:**

This package has *specific installation instructions* on the *Build extras* page. The SCAFACOS package requires that LAMMPS is built in *MPI parallel mode*.

**Supporting info:**

- `src/SCAFACOS`: filenames -> commands
- `src/SCAFACOS/README`
- *kspace_style scafacos*
- *kspace_modify*
- `examples/PACKAGES/scafacos`

---

### 8.1.86 SHOCK package

**Contents:**

Fixes for running impact simulations where a shock-wave passes through a material.

**Supporting info:**

- `src/SHOCK`: filenames -> commands
- *fix append/atoms*
- *fix msst*
- *fix nphug*
- *fix wall/piston*
- `examples/hugoniostat`

- examples/msst

---

### 8.1.87 SMTBQ package

**Contents:**

Pair styles which implement Second Moment Tight Binding models. One with QEq charge equilibration (SMTBQ) for the description of ionocovalent bonds in oxides, and two more as plain SMATB models.

**Authors:** SMTBQ: Nicolas Salles, Emile Maras, Olivier Politano, and Robert Tetot (LAAS-CNRS, France); SMATB: Daniele Rapetti (Politecnico di Torino)

**Supporting info:**

- src/SMTBQ: filenames -> commands
- src/SMTBQ/README
- *pair_style smtbq*
- *pair_style smatb*, *pair_style smatb/single*
- examples/PACKAGES/smtbq

---

### 8.1.88 SPH package

**Contents:**

An atom style, fixes, computes, and several pair styles which implements smoothed particle hydrodynamics (SPH) for liquids. See the related *MACHDYN package* package for smooth Mach dynamics (SMD) for solids.

This package contains ideal gas, Lennard-Jones equation of states, Tait, and full support for complete (i.e. internal-energy dependent) equations of state. It allows for plain or Monaghans XSPH integration of the equations of motion. It has options for density continuity or density summation to propagate the density field. It has *set* command options to set the internal energy and density of particles from the input script and allows the same quantities to be output with thermodynamic output or to dump files via the *compute property/atom* command.

**Author:** Georg Ganzenmuller (Fraunhofer-Institute for High-Speed Dynamics, Ernst Mach Institute, Germany).

**Supporting info:**

- src/SPH: filenames -> commands
- src/SPH/README
- doc/PDF/SPH_LAMMPS_userguide.pdf
- examples/PACKAGES/sph
- https://www.lammps.org/movies.html#sph

> **Note**
>
> Please note that the SPH PDF guide file has not been updated for many years and thus does not reflect the current *syntax* of the SPH package commands. For that please refer to the LAMMPS manual.

> **Note**
>
> Please also note, that the *RHEO package* offers similar functionality in a more modern and flexible implementation.

---

### 8.1.89 SPIN package

**Contents:**

Model atomic magnetic spins classically, coupled to atoms moving in the usual manner via MD. Various pair, fix, and compute styles.

**Author:** Julien Tranchida (Sandia).

**Supporting info:**

- `src/SPIN`: filenames -> commands
- *Howto spins*
- *pair_style spin/dipole/cut*
- *pair_style spin/dipole/long*
- *pair_style spin/dmi*
- *pair_style spin/exchange*
- *pair_style spin/exchange/biquadratic*
- *pair_style spin/magelec*
- *pair_style spin/neel*
- *fix nve/spin*
- *fix langevin/spin*
- *fix precession/spin*
- *compute spin*
- *neb/spin*
- `examples/SPIN`

---

### 8.1.90 SRD package

**Contents:**

A pair of fixes which implement the Stochastic Rotation Dynamics (SRD) method for coarse-graining of a solvent, typically around large colloidal particles.

**Supporting info:**

- `src/SRD`: filenames -> commands
- *fix srd*
- *fix wall/srd*
- `examples/srd`

- `examples/ASPHERE`
- https://www.lammps.org/movies.html#tri
- https://www.lammps.org/movies.html#line
- https://www.lammps.org/movies.html#poly

---

### 8.1.91 TALLY package

**Contents:**

Several compute styles that can be called when pairwise interactions are calculated to tally information (forces, heat flux, energy, stress, etc) about individual interactions.

**Author:** Axel Kohlmeyer (Temple U).

**Supporting info:**

- `src/TALLY`: filenames -> commands
- `src/TALLY/README`
- *compute */tally*
- `examples/PACKAGES/tally`

---

### 8.1.92 UEF package

**Contents:**

A fix style for the integration of the equations of motion under extensional flow with proper boundary conditions, as well as several supporting compute styles and an output option.

**Author:** David Nicholson (MIT).

**Supporting info:**

- `src/UEF`: filenames -> commands
- `src/UEF/README`
- *fix nvt/uef*
- *fix npt/uef*
- *compute pressure/uef*
- *compute temp/uef*
- *dump cfg/uef*
- `examples/uef`

---

### 8.1.93 VORONOI package

**Contents:**

A compute command which calculates the Voronoi tesselation of a collection of atoms by wrapping the Voro++ library. This can be used to calculate the local volume of atoms or their near neighbors.

To use this package you must have the Voro++ library available on your system.

**Author:** Daniel Schwen (INL) while at LANL. The open-source Voro++ library was written by Chris Rycroft (Harvard U) while at UC Berkeley and LBNL.

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- `src/VORONOI`: filenames -> commands
- `src/VORONOI/README`
- `lib/voronoi/README`
- *compute voronoi/atom*
- `examples/voronoi`

---

### 8.1.94 VTK package

**Contents:**

A *dump vtk* command which outputs snapshot info in the VTK format, enabling visualization by Paraview or other visualization packages.

To use this package you must have VTK library available on your system.

**Authors:** Richard Berger (JKU) and Daniel Queteschiner (DCS Computing).

**Install:**

This package has *specific installation instructions* on the *Build extras* page.

**Supporting info:**

- `src/VTK`: filenames -> commands
- `src/VTK/README`
- `lib/vtk/README`
- *dump vtk*

---

### 8.1.95 YAFF package

**Contents:**

Some potentials that are also implemented in the Yet Another Force Field (YAFF) code. The expressions and their use are discussed in the following papers

- Vanduyfhuys et al., J. Comput. Chem., 36 (13), 1015-1027 (2015) link
- Vanduyfhuys et al., J. Comput. Chem., 39 (16), 999-1011 (2018) link

which discuss the QuickFF methodology.

**Author:** Steven Vandenbrande.

Added in version 1Feb2019.

**Supporting info:**

- `src/YAFF/README`
- *angle_style cross*
- *angle_style mm3*
- *bond_style mm3*
- *improper_style distharm*
- *improper_style sqdistharm*
- *pair_style mm3/switch3/coulgauss/long*
- *pair_style lj/switch3/coulgauss/long*
- `examples/PACKAGES/yaff`

CHAPTER

NINE

# AUXILIARY TOOLS

LAMMPS is designed to be a computational kernel for performing molecular dynamics computations. Additional pre- and post-processing steps are often necessary to setup and analyze a simulation. A list of such tools can be found on the LAMMPS webpage at these links:

- Pre/Post processing
- External LAMMPS packages & tools
- Pizza.py toolkit

The last link for Pizza.py is a Python-based tool developed at Sandia which provides tools for doing setup, analysis, plotting, and visualization for LAMMPS simulations.

Additional tools included in the LAMMPS distribution are described on this page.

Note that many users write their own setup or analysis tools or use other existing codes and convert their output to a LAMMPS input format or vice versa. The tools listed here are included in the LAMMPS distribution as examples of auxiliary tools. Some of them are not actively supported by the LAMMPS developers, as they were contributed by LAMMPS users. If you have problems using them, we can direct you to the authors.

The source code for each of these codes is in the tools subdirectory of the LAMMPS distribution. There is a Makefile (which you may need to edit for your platform) which will build several of the tools which reside in that directory. Most of them are larger packages in their own subdirectories with their own Makefiles and/or README files.

## 9.1 Pre-processing tools

| | | | | | |
|---|---|---|---|---|---|
| *ch2lmp* | *chain* | *createatoms* | *drude* | *eam database* | *eam generate* |
| *eff* | *ipp* | *micelle2d* | *moltemplate* | *msi2lmp* | *polybond* |
| *stl_bin2txt* | *tabulate* | *tinker* | *AMBER2LAMMPS* | | |

## 9.2 Post-processing tools

| | | | | | |
|---|---|---|---|---|---|
| *binary2txt* | *ch2lmp* | *colvars* | *eff* | *fep* | *lmp2arc* |
| *lmp2cfg* | *matlab* | *phonon* | *pymol_asphere* | *python* | *replica* |
| *smd* | *spin* | *xmgrace* | | | |

## 9.3 Miscellaneous tools

| | | | | | |
|---|---|---|---|---|---|
| *LAMMPS coding standards* | *emacs* | *i-PI* | *JSON support* | *kate* | *LAMMPS-GUI* |
| *LAMMPS magic patterns for file(1)* | *Offline build tool* | *Regression tester* | *singularity/apptainer* | *SWIG interface* | *valgrind* |
| *vim* | | | | | |

## 9.4 Tool descriptions

### 9.4.1 binary2txt tool

The file binary2txt.cpp converts one or more binary LAMMPS dump file into ASCII text files. The syntax for running the tool is

```
binary2txt file1 file2 ...
```

which creates file1.txt, file2.txt, etc. This tool must be compiled on a platform that can read the binary file created by a LAMMPS run, since binary files are not compatible across all platforms.

### 9.4.2 ch2lmp tool

The ch2lmp subdirectory contains tools for converting files back-and-forth between the CHARMM MD code and LAMMPS.

They are intended to make it easy to use CHARMM as a builder and as a post-processor for LAMMPS. Using charmm2lammps.pl, you can convert a PDB file with associated CHARMM info, including CHARMM force field data, into its LAMMPS equivalent. Support for the CMAP correction of CHARMM22 and later is available as an option. This tool can also add solvent water molecules and Na+ or Cl- ions to the system. Using lammps2pdb.pl you can convert LAMMPS atom dumps into PDB files.

See the README file in the ch2lmp subdirectory for more information.

These tools were created by Pieter in't Veld (pjintve at sandia.gov) and Paul Crozier (pscrozi at sandia.gov) at Sandia.

CMAP support added and tested by Xiaohu Hu (hux2 at ornl.gov) and Robert A. Latour (latourr at clemson.edu), David Hyde-Volpe, and Tigran Abramyan, (Clemson University) and Chris Lorenz (chris.lorenz at kcl.ac.uk), King's College London.

### 9.4.3 chain tool

The file chain.f90 creates a LAMMPS data file containing bead-spring polymer chains and/or monomer solvent atoms. It uses a text file containing chain definition parameters as an input. The created chains and solvent atoms can strongly overlap, so LAMMPS needs to run the system initially with a "soft" pair potential to un-overlap it. The syntax for running the tool is

```
chain < def.chain > data.file
```

See the def.chain or def.chain.ab files in the tools directory for examples of definition files. This tool was used to create the system for the *chain benchmark*.

### 9.4.4 LAMMPS coding standard

The `coding_standard` folder contains multiple python scripts to check for and apply some LAMMPS coding conventions. The following scripts are available:

```
permissions.py   # detects if sources have executable permissions and scripts have not
whitespace.py    # detects TAB characters and trailing whitespace
homepage.py      # detects outdated LAMMPS homepage URLs (pointing to sandia.gov instead␣
↪of lammps.org)
errordocs.py     # detects deprecated error docs in header files
versiontags.py   # detects .. versionadded:: or .. versionchanged:: with pending version␣
↪date
```

The tools need to be given the main folder of the LAMMPS distribution or individual file names as argument and will by default check them and report any non-compliance. With the optional `-f` argument the corresponding script will try to change the non-compliant file(s) to match the conventions.

For convenience this scripts can also be invoked by the make file in the `src` folder with, *make check-whitespace* or *make fix-whitespace* to either detect or edit the files. Correspondingly for the other python scripts. *make check* will run all checks.

---

### 9.4.5 colvars tools

The colvars directory contains a collection of tools for post-processing data produced by the colvars collective variable library. To compile the tools, edit the makefile for your system and run "make".

Please report problems and issues the colvars library and its tools at: https://github.com/colvars/colvars/issues

abf_integrate:

MC-based integration of multidimensional free energy gradient Version 20110511

```
./abf_integrate < filename > [-n < nsteps >] [-t < temp >] [-m [0|1] (metadynamics)] [-h
↪< hill_height >] [-f < variable_hill_factor >]
```

The LAMMPS interface to the colvars collective variable library, as well as these tools, were created by Axel Kohlmeyer (akohlmey at gmail.com) while at ICTP, Italy.

---

### 9.4.6 createatoms tool

The tools/createatoms directory contains a Fortran program called createAtoms.f which can generate a variety of interesting crystal structures and geometries and output the resulting list of atom coordinates in LAMMPS or other formats.

See the included Manual.pdf for details.

The tool is authored by Xiaowang Zhou (Sandia), xzhou at sandia.gov.

---

### 9.4.7 drude tool

The tools/drude directory contains a Python script called polarizer.py which can add Drude oscillators to a LAMMPS data file in the required format.

See the header of the polarizer.py file for details.

The tool is authored by Agilio Padua and Alain Dequidt: agilio.padua at ens-lyon.fr, alain.dequidt at uca.fr

---

### 9.4.8 eam database tool

The tools/eam_database directory contains a Fortran and a Python program that will generate EAM alloy setfl potential files for any combination of the 17 elements: Cu, Ag, Au, Ni, Pd, Pt, Al, Pb, Fe, Mo, Ta, W, Mg, Co, Ti, Zr, Cr. The files can then be used with the *pair_style eam/alloy* command.

The Fortran version of the tool was authored by Xiaowang Zhou (Sandia), xzhou at sandia.gov, with updates from Lucas Hale (NIST) lucas.hale at nist.gov and is based on his paper:

X. W. Zhou, R. A. Johnson, and H. N. G. Wadley, Phys. Rev. B, 69, 144113 (2004).

The parameters for Cr were taken from:

Lin Z B, Johnson R A and Zhigilei L V, Phys. Rev. B 77 214108 (2008).

The Python version of the tool was authored by Germain Clavier (Unicaen) germain.clavier at unicaen.fr

**Note**

The parameters in the database are only optimized for individual elements. The mixed parameters for interactions between different elements generated by this tool are derived from simple mixing rules and are thus inferior to parameterizations that are specifically optimized for specific mixtures and combinations of elements.

---

### 9.4.9 eam generate tool

The tools/eam_generate directory contains several one-file C programs that convert an analytic formula into a tabulated *embedded atom method (EAM)* setfl potential file. The potentials they produce are in the potentials directory, and can be used with the *pair_style eam/alloy* command.

The source files and potentials were provided by Gerolf Ziegenhain (gerolf at ziegenhain.com).

---

### 9.4.10 eff tool

The tools/eff directory contains various scripts for generating structures and post-processing output for simulations using the electron force field (eFF).

These tools were provided by Andres Jaramillo-Botero at CalTech (ajaramil at wag.caltech.edu).

---

### 9.4.11 emacs tool

The tools/emacs directory contains an Emacs Lisp add-on file for GNU Emacs that enables a lammps-mode for editing input scripts when using GNU Emacs, with various highlighting options set up.

These tools were provided by Aidan Thompson at Sandia (athomps at sandia.gov).

---

### 9.4.12 fep tool

The tools/fep directory contains Python scripts useful for post-processing results from performing free-energy perturbation simulations using the FEP package.

The scripts were contributed by Agilio Padua (ENS de Lyon), agilio.padua at ens-lyon.fr.

See README file in the tools/fep directory.

---

### 9.4.13 i-PI tool

Changed in version 27June2024.

The tools/i-pi directory used to contain a bundled version of the i-PI software package for use with LAMMPS. This version, however, was removed in 06/2024.

The i-PI package was created and is maintained by Michele Ceriotti, michele.ceriotti at gmail.com, to interface to a variety of molecular dynamics codes.

i-PI is now available via PyPI using the pip package manager at: https://pypi.org/project/ipi/

Here are the commands to set up a virtual environment and install i-PI into it with all its dependencies.

```
python -m venv ipienv
source ipienv/bin/activate
pip install --upgrade pip
pip install ipi
```

To install the development version from GitHub, please use:

```
pip install git+https://github.com/i-pi/i-pi.git
```

For further information, please consult the [i-PI home page](https://ipi-code.org).

---

### 9.4.14 ipp tool

The tools/ipp directory contains a Perl script ipp which can be used to facilitate the creation of a complicated file (say, a LAMMPS input script or tools/createatoms input file) using a template file.

ipp was created and is maintained by Reese Jones (Sandia), rjones at sandia.gov.

See two examples in the tools/ipp directory. One of them is for the tools/createatoms tool's input file.

---

### 9.4.15 JSON support files

Added in version 12June2025.

The `tools/json` directory contains files and tools to support using JSON format files in LAMMPS. Currently only the *molecule command* supports files in JSON format directly, but this is planned to be expanded in the future.

### JSON file validation

The JSON syntax is independent of its content, and thus the data in the file must follow suitable conventions to be correctly parsed during input. This can be done in a portable fashion using a JSON schema file (which is in JSON format as well) to define those conventions. A suitable JSON validator software can then validate JSON files against the requirements. Validating a particular JSON file against a schema ensures that both, the syntax *and* the conventions are followed. This is useful when writing or editing JSON files in a text editor or when writing a pre-processing script or tool to create JSON files for a specific purpose in LAMMPS. It **cannot** check whether the file contents are physically meaningful, though.

One such validator tool is check-jsonschema which is written in Python and can be installed using the pip Python package manager, best in a virtual environment as shown below (for a Bourne Shell command line):

```
python -m venv validate-json
source validate-json/bin/activate
pip install --upgrade pip
pip install check-jsonschema
```

To validate a specific JSON file against a provided schema (here for a *molecule command file* you would then run for example:

```
check-jsonschema --schemafile molecule-schema.json tip3p.json
```

The latest schema files are also maintained and available for download at https://download.lammps.org/json/ . This enables validation of JSON files even if the LAMMPS sources are not locally available. Example:

```
check-jsonschema --schemafile https://download.lammps.org/json/molecule-schema.json␣
↪tip3p.json
```

Because YAML is a superset of JSON, the same tool and mechanism can also validate YAML files against a JSON schema. This is used for the force-style regression test reference files in the `unittest/force-styles` folder, which are described by `force-style-test-schema.json`. For example, to validate all of them:

```
check-jsonschema --schemafile force-style-test-schema.json \
    unittest/force-styles/tests/*.yaml
```

See the *unit test developer documentation* for a description of the available keys in these reference files.

### JSON file format normalization

There are extensions to the strict JSON format that allow for comments or ignore additional (dangling) commas. The `reformat-json.cpp` tool will read JSON files in relaxed format, but write it out in strict format. It is also possible to change the level of indentation from -1 (all data one long line) to any positive integer value. The original file will be backed up (.bak added to file name) and then overwritten.

Manual compilation (it will be automatically included in the CMake build if building tools is requested during CMake configuration):

```
g++ -I <path/to/lammps/src> -o reformat-json reformat-json.cpp
```

Usage:

```
reformat-json <indent-width> <json-file-1> [<json-file-2> ...]
```

### 9.4.16 kate tool

The file in the tools/kate directory is an add-on to the Kate editor in the KDE suite that allow syntax highlighting of LAMMPS input scripts. See the README.txt file for details.

The file was provided by Alessandro Luigi Sellerio (alessandro.sellerio at ieni.cnr.it).

---

### 9.4.17 LAMMPS-GUI

Changed in version 10Sep2025.

LAMMPS-GUI is a graphical text editor customized for editing LAMMPS input files that is linked to the *LAMMPS C-library*. It used to be included with LAMMPS in the `tools/lammps-gui` folder, but it is now hosted in its own git repository at https://github.com/akohlmey/lammps-gui/ and the online documentation is at https://lammps-gui.lammps.org/

It is still possible to compile *LAMMPS-GUI together with LAMMPS*.

---

### 9.4.18 lmp2arc tool

The lmp2arc subdirectory contains a tool for converting LAMMPS output files to the format for Accelrys' Insight MD code (formerly MSI/Biosym and its Discover MD code). See the README file for more information.

This tool was written by John Carpenter (Cray), Michael Peachey (Cray), and Steve Lustig (Dupont). John is now at the Mayo Clinic (jec at mayo.edu), but still fields questions about the tool.

This tool was updated for the current LAMMPS C++ version by Jeff Greathouse at Sandia (jagreat at sandia.gov).

---

### 9.4.19 lmp2cfg tool

The lmp2cfg subdirectory contains a tool for converting LAMMPS output files into a series of *.cfg files which can be read into the AtomEye visualizer. See the README file for more information.

This tool was written by Ara Kooser at Sandia (askoose at sandia.gov).

---

### 9.4.20 Magic patterns for the "file" command

Added in version 10Mar2021.

The file `magic` contains patterns that are used by the file program available on most Unix-like operating systems which enables it to detect various LAMMPS files and print some useful information about them. To enable these patterns, append or copy the contents of the file to either the file `.magic` in your home directory or (as administrator) to `/etc/magic` (for a system-wide installation). Afterwards the `file` command should be able to detect most LAMMPS restarts, dump, data and log files. Examples:

```
$ file *.*
dihedral-quadratic.restart:   LAMMPS binary restart file (rev 2), Version 10 Mar 2021,␣
↪Little Endian
mol-pair-wf_cut.restart:      LAMMPS binary restart file (rev 2), Version 24 Dec 2020,␣
↪Little Endian
atom.bin:                     LAMMPS atom style binary dump (rev 2), Little Endian,␣
```

(continues on next page)

(continued from previous page)

```
↪First time step: 445570
custom.bin:                   LAMMPS custom style binary dump (rev 2), Little Endian,␣
↪First time step: 100
bn1.lammpstrj:                LAMMPS text mode dump, First time step: 5000
data.fourmol:                 LAMMPS data file written by LAMMPS
pnc.data:                     LAMMPS data file written by msi2lmp
data.spce:                    LAMMPS data file written by TopoTools
B.data:                       LAMMPS data file written by OVITO
log.lammps:                   LAMMPS log file written by version 10 Feb 2021
```

### 9.4.21 matlab tool

The matlab subdirectory contains several MATLAB scripts for post-processing LAMMPS output. The scripts include readers for log and dump files, a reader for EAM potential files, and a converter that reads LAMMPS dump files and produces CFG files that can be visualized with the AtomEye visualizer.

See the README.pdf file for more information.

These scripts were written by Arun Subramaniyan at Purdue Univ (asubrama at purdue.edu).

### 9.4.22 micelle2d tool

The file micelle2d.f creates a LAMMPS data file containing short lipid chains in a monomer solution. It uses a text file containing lipid definition parameters as an input. The created molecules and solvent atoms can strongly overlap, so LAMMPS needs to run the system initially with a "soft" pair potential to un-overlap it. The syntax for running the tool is

```
micelle2d < def.micelle2d > data.file
```

See the def.micelle2d file in the tools directory for an example of a definition file. This tool was used to create the system for the *micelle example*.

### 9.4.23 moltemplate tool

The moltemplate subdirectory contains instructions for installing moltemplate, a Python-based tool for building molecular systems based on a text-file description, and creating LAMMPS data files that encode their molecular topology as lists of bonds, angles, dihedrals, etc. See the README.txt file for more information.

This tool was written by Andrew Jewett (jewett.aij at gmail.com), who supports it. It has its own WWW page at https://moltemplate.org. The latest sources can be found on its GitHub page

### 9.4.24 msi2lmp tool

The msi2lmp subdirectory contains a tool for creating LAMMPS template input and data files from BIOVIA's Materias Studio files (formerly Accelrys' Insight MD code, formerly MSI/Biosym and its Discover MD code).

This tool was written by John Carpenter (Cray), Michael Peachey (Cray), and Steve Lustig (Dupont). Several people contributed changes to remove bugs and adapt its output to changes in LAMMPS.

This tool has several known limitations and is no longer under active development, so there are no changes except for the occasional bug fix.

See the README file in the tools/msi2lmp folder for more information.

---

### 9.4.25 Scripts for building LAMMPS when offline

In some situations it might be necessary to build LAMMPS on a system without direct internet access. The scripts in `tools/offline` folder allow you to pre-load external dependencies for both the documentation build and for building LAMMPS with CMake.

It does so by

1. downloading necessary `pip` packages,
2. cloning `git` repositories
3. downloading tarballs

to a designated cache folder.

As of April 2021, all of these downloads make up around 600MB. By default, the offline scripts will download everything into the `$HOME/.cache/lammps` folder, but this can be changed by setting the `LAMMPS_CACHING_DIR` environment variable.

Once the caches have been initialized, they can be used for building the LAMMPS documentation or compiling LAMMPS using CMake on an offline system.

The `use_caches.sh` script must be sourced into the current shell to initialize the offline build environment. Note that it must use the same `LAMMPS_CACHING_DIR`. This script does the following:

1. Set up environment variables that modify the behavior of both, `pip` and `git`
2. Start a simple local HTTP server using Python to host files for CMake

Afterwards, it will print out instructions on how to modify the CMake commands to make sure it uses the local HTTP server.

To undo the environment changes and shutdown the local HTTP server, run the `deactivate_caches` command.

#### Examples

For all of the examples below, you first need to create the cache, which requires an internet connection.

```
./tools/offline/init_caches.sh
```

Afterwards, you can disconnect or copy the contents of the `LAMMPS_CACHING_DIR` folder to an offline system.

#### Documentation Build

The documentation build will create a new virtual environment that typically first installs dependencies from `pip`. With the offline environment loaded, these installations will instead grab the necessary packages from your local cache.

```
# if LAMMPS_CACHING_DIR is different from default, make sure to set it first
# export LAMMPS_CACHING_DIR=path/to/folder
source tools/offline/use_caches.sh
cd doc/
make html

deactivate_caches
```

#### CMake Build

When compiling certain packages with external dependencies, the CMake build system will download necessary files or sources from the web. For more flexibility the CMake configuration allows users to specify the URL of each of these dependencies. What the `init_caches.sh` script does is create a CMake "preset" file, which sets the URLs for all of the known dependencies and redirects the download to the local cache.

```
# if LAMMPS_CACHING_DIR is different from default, make sure to set it first
# export LAMMPS_CACHING_DIR=path/to/folder
source tools/offline/use_caches.sh

mkdir build
cd build
cmake -D LAMMPS_DOWNLOADS_URL=${HTTP_CACHE_URL} -C "${LAMMPS_HTTP_CACHE_CONFIG}" -C ../
↪cmake/presets/most.cmake -D DOWNLOAD_POTENTIALS=off ../cmake
make -j 8

deactivate_caches
```

---

### 9.4.26 phonon tool

The phonon subdirectory contains a post-processing tool, *phana*, useful for analyzing the output of the *fix phonon* command in the PHONON package.

See the README file for instructions on building the tool and what library it needs. And see the examples/PACKAGES/phonon directory for example problems that can be post-processed with this tool.

This tool was written by Ling-Ti Kong at Shanghai Jiao Tong University.

---

### 9.4.27 polybond tool

The polybond subdirectory contains a Python-based tool useful for performing "programmable polymer bonding". The Python file lmpsdata.py provides a "Lmpsdata" class with various methods which can be invoked by a user-written Python script to create data files with complex bonding topologies.

See the Manual.pdf for details and example scripts.

This tool was written by Zachary Kraus at Georgia Tech.

---

### 9.4.28 pymol_asphere tool

The pymol_asphere subdirectory contains a tool for converting a LAMMPS dump file that contains orientation info for ellipsoidal particles into an input file for the PyMol visualization package or its open source variant.

Specifically, the tool triangulates the ellipsoids so they can be viewed as true ellipsoidal particles within PyMol. See the README and examples directory within pymol_asphere for more information.

This tool was written by Mike Brown at Sandia.

---

### 9.4.29 python tool

The python subdirectory contains several Python scripts that perform common LAMMPS post-processing tasks, such as:

- extract thermodynamic info from a log file as columns of numbers
- plot two columns of thermodynamic info from a log file using GnuPlot
- sort the snapshots in a dump file by atom ID
- convert multiple *NEB* dump files into one dump file for viz
- convert dump files into XYZ, CFG, or PDB format for viz by other packages

These are simple scripts built on Pizza.py modules. See the README for more info on Pizza.py and how to use these scripts.

---

### 9.4.30 Regression tester tool

The regression-tests subdirectory contains a tool for performing regression tests with a given LAMMPS binary. The tool launches the LAMMPS binary with any given input script under one of the *examples* subdirectories, and compares the thermo output in the generated log file with those in the provided log file with the same number of processors in the same subdirectory. If the differences between the actual and reference values are within specified tolerances, the test is considered passed. For each test batch, that is, a set of example input scripts, the `mpirun` command, the LAMMPS command-line arguments, and the tolerances for individual thermo quantities can be specified in a configuration file in YAML format.

The tool also reports if and how the run fails, and if a reference log file is missing. See the README file for more information.

This tool was written by Trung Nguyen at U of Chicago (ndactrung at gmail.com).

---

### 9.4.31 replica tool

The tools/replica directory contains the reorder_remd_traj python script which can be used to reorder the replica trajectories (resulting from the use of the temper command) according to temperature. This will produce discontinuous trajectories with all frames at the same temperature in each trajectory. Additional options can be used to calculate the canonical configurational log-weight for each frame at each temperature using the pymbar package. See the README.md file for further details. Try out the peptide example provided.

This tool was written by (and is maintained by) Tanmoy Sanyal, while at the Shell lab at UC Santa Barbara. (tanmoy dot 7989 at gmail.com)

---

### 9.4.32 smd tool

The smd subdirectory contains a C++ file dump2vtk_tris.cpp and Makefile which can be compiled and used to convert triangle output files created by the Smooth-Mach Dynamics (MACHDYN) package into a VTK-compatible unstructured grid file. It could then be read in and visualized by VTK.

See the header of dump2vtk.cpp for more details.

This tool was written by the MACHDYN package author, Georg Ganzenmuller at the Fraunhofer-Institute for High-Speed Dynamics, Ernst Mach Institute in Germany (georg.ganzenmueller at emi.fhg.de).

---

### 9.4.33 spin tool

The spin subdirectory contains a C file interpolate.c which can be compiled and used to perform a cubic polynomial interpolation of the MEP following a GNEB calculation.

See the README file in tools/spin/interpolate_gneb for more details.

This tool was written by the SPIN package author, Julien Tranchida at Sandia National Labs (jtranch at sandia.gov, and by Aleksei Ivanov, at University of Iceland (ali5 at hi.is).

---

### 9.4.34 singularity/apptainer tool

The singularity subdirectory contains container definitions files that can be used to build container images for building and testing LAMMPS on specific OS variants using the Apptainer or Singularity container software. Contributions for additional variants are welcome. For more details please see the README.md file in that folder.

---

### 9.4.35 stl_bin2txt tool

The file stl_bin2txt.cpp converts binary STL files - like they are frequently offered for download on the web - into ASCII format STL files that LAMMPS can read with the *create_atoms mesh* or the *fix smd/wall_surface* commands. The syntax for running the tool is

```
stl_bin2txt infile.stl outfile.stl
```

which creates outfile.stl from infile.stl. This tool must be compiled on a platform compatible with the byte-ordering that was used to create the binary file. This usually is a so-called little endian hardware (like x86).

---

### 9.4.36 SWIG interface

The SWIG tool offers a mostly automated way to incorporate compiled code modules into scripting languages. It processes the function prototypes in C and generates wrappers for a wide variety of scripting languages from it. Thus it can also be applied to the *C language library interface* of LAMMPS so that build a wrapper that allows to call LAMMPS from programming languages like: C#/Mono, Lua, Java, JavaScript, Perl, Python, R, Ruby, Tcl, and more.

#### What is included

We provide here an "interface file", `lammps.i`, that has the content of the `library.h` file adapted so SWIG can process it. That will create wrappers for all the functions that are present in the LAMMPS C library interface. Please note that not all kinds of C functions can be automatically translated, so you would have to add custom functions to be able to utilize those where the automatic translation does not work. A few functions for converting pointers and accessing arrays are predefined. We provide the file here on an "as is" basis to help people getting started, but not as a fully tested and supported feature of the LAMMPS distribution. Any contributions to complete this are, of course, welcome. Please also note, that for the case of creating a Python wrapper, a fully supported *Ctypes based lammps module* already exists. That module is designed to be object-oriented while SWIG will generate a 1:1 translation of the functions in the interface file.

#### Building the wrapper

When using CMake, the build steps for building a wrapper module are integrated for the languages: Java, Lua, Perl5, Python, Ruby, and Tcl. These require that the LAMMPS library is build as a shared library and all necessary development headers and libraries are present.

```
-D WITH_SWIG=on           # to enable building any SWIG wrapper
-D BUILD_SWIG_JAVA=on     # to enable building the Java wrapper
-D BUILD_SWIG_LUA=on      # to enable building the Lua wrapper
-D BUILD_SWIG_PERL5=on    # to enable building the Perl 5.x wrapper
-D BUILD_SWIG_PYTHON=on   # to enable building the Python wrapper
-D BUILD_SWIG_RUBY=on     # to enable building the Ruby wrapper
-D BUILD_SWIG_TCL=on      # to enable building the Tcl wrapper
```

Manual building allows a little more flexibility. E.g. one can choose the name of the module and build and use a dynamically loaded object for Tcl with:

```
swig -tcl -module tcllammps lammps.i
gcc -fPIC -shared $(pkg-config tcl --cflags) -o tcllammps.so \
            lammps_wrap.c -L ../src/ -llammps
tclsh
```

Or one can build an extended Tcl shell command with the wrapped functions included with:

```
swig -tcl -module tcllmps lammps_shell.i
gcc -o tcllmpsh lammps_wrap.c -Xlinker -export-dynamic \
         -DHAVE_CONFIG_H $(pkg-config tcl --cflags) \
         $(pkg-config tcl --libs) -L ../src -llammps
```

In both cases it is assumed that the LAMMPS library was compiled as a shared library in the `src` folder. Otherwise the last part of the commands needs to be adjusted.

### Utility functions

Definitions for several utility functions required to manage and access data passed or returned as pointers are included in the `lammps.i` file. So most of the functionality of the library interface should be accessible. What works and what does not depends a bit on the individual language for which the wrappers are built and how well SWIG supports those. The SWIG documentation has very detailed instructions and recommendations.

### Usage examples

The `tools/swig` folder has multiple shell scripts, `run_<name>_example.sh` that will create a small example script and demonstrate how to load the wrapper and run LAMMPS through it in the corresponding programming language.

For illustration purposes below is a part of the Tcl example script.

```
load ./tcllammps.so
set lmp [lammps_open_no_mpi 0 NULL NULL]
lammps_command $lmp "units real"
lammps_command $lmp "lattice fcc 2.5"
lammps_command $lmp "region box block -5 5 -5 5 -5 5"
lammps_command $lmp "create_box 1 box"
lammps_command $lmp "create_atoms 1 box"

set dt [doublep_value [voidp_to_doublep [lammps_extract_global $lmp dt]]]
puts "LAMMPS version $ver"
puts [format "Number of created atoms: %g" [lammps_get_natoms $lmp]]
puts "Current size of timestep: $dt"
puts "LAMMPS version: [lammps_version $lmp]"
lammps_close $lmp
```

### 9.4.37 tabulate tool

Added in version 22Dec2022.

The `tabulate` folder contains Python scripts to generate and visualize tabulated potential files for LAMMPS. The bulk of the code is in the `tabulate` module in the `tabulate.py` file. Some example files demonstrating its use are included. See the README file for more information.

### 9.4.38 tinker tool

The `tinker` folder contains Python scripts to convert Tinker input files to LAMMPS.

See the README file for more information.

Those scripts were written by Steve Plimpton sjplimp at gmail.com

### 9.4.39 valgrind tool

The `valgrind` folder contains additional suppressions for LAMMPS when using valgrind's ` memcheck tool to search for memory access violation and memory leaks. These suppressions are automatically invoked when running tests through CMake "ctest -T memcheck". See the instructions in the `README` file to add these suppressions when using valgrind with LAMMPS or other programs.

### 9.4.40 vim tool

The files in the `tools/vim` directory are add-ons to the VIM editor that allow easier editing of LAMMPS input scripts. See the `README.txt` file for details.

These files were provided by Gerolf Ziegenhain (gerolf at ziegenhain.com)

### 9.4.41 xmgrace tool

The files in the tools/xmgrace directory can be used to plot the thermodynamic data in LAMMPS log files via the xmgrace plotting package. There are several tools in the directory that can be used in post-processing mode. The lammpsplot.cpp file can be compiled and used to create plots from the current state of a running LAMMPS simulation.

See the README file for details.

These files were provided by Vikas Varshney (vv0210 at gmail.com)

CHAPTER

# TEN

# HOWTO DISCUSSIONS

These doc pages describe how to perform various tasks with LAMMPS, both for users and developers. The glossary website page also lists MD terminology, with links to corresponding LAMMPS manual pages. The example input scripts included in the `examples` directory of the LAMMPS source code distribution and highlighted on the *Example scripts* page also show how to set up and run various kinds of simulations.

## 10.1 General howto

### 10.1.1 Restart a simulation

There are 3 ways to continue a long LAMMPS simulation. Multiple *run* commands can be used in the same input script. Each run will continue from where the previous run left off. Or binary restart files can be saved to disk using the *restart* command. At a later time, these binary files can be read via a *read_restart* command in a new script. Or they can be converted to text data files using the *-r command-line switch* and read by a *read_data* command in a new script.

Here we give examples of 2 scripts that read either a binary restart file or a converted data file and then issue a new run command to continue where the previous run left off. They illustrate what settings must be made in the new script. Details are discussed in the documentation for the *read_restart* and *read_data* commands.

Look at the *in.chain* input script provided in the *bench* directory of the LAMMPS distribution to see the original script that these 2 scripts are based on. If that script had the line

```
restart         50 tmp.restart
```

added to it, it would produce two binary restart files (`tmp.restart.50` and `tmp.restart.100`) as it ran.

This script could be used to read the first restart file and re-run the last 50 timesteps:

```
read_restart    tmp.restart.50

neighbor        0.4 bin
neigh_modify    every 1 delay 1

fix             1 all nve
fix             2 all langevin 1.0 1.0 10.0 904297

timestep        0.012

run             50
```

Note that the following commands do not need to be repeated because their settings are included in the restart file: **units**, **atom_style**, **special_bonds**, **pair_style**, **bond_style**. However, these commands do need to be used, since their settings are not in the restart file: **neighbor**, **fix**, **timestep**.

If you actually use this script to perform a restarted run, you will notice that the thermodynamic data match at step 50 (if you also put a **thermo** 50 command in the original script), but do not match at step 100. This is because the *fix langevin* command uses random numbers in a way that does not allow for perfect restarts.

As an alternate approach, the restart file could be converted to a data file as follows:

```
lmp_g++ -r tmp.restart.50 tmp.restart.data
```

Then, this script could be used to re-run the last 50 steps:

```
units           lj
atom_style      bond
pair_style      lj/cut 1.12
pair_modify     shift yes
bond_style      fene
special_bonds   0.0 1.0 1.0

read_data       tmp.restart.data

neighbor        0.4 bin
neigh_modify    every 1 delay 1

fix             1 all nve
fix             2 all langevin 1.0 1.0 10.0 904297

timestep        0.012

reset_timestep  50
run             50
```

Note that nearly all the settings specified in the original `in.chain` script must be repeated, except the **pair_coeff** and **bond_coeff** commands, since the new data file lists the force field coefficients. Also, the *reset_timestep* command is used to tell LAMMPS the current timestep. This value is stored in restart files, but not in data files.

### 10.1.2 Visualize LAMMPS snapshots

Snapshots from LAMMPS simulations can be viewed, visualized, and analyzed in a variety of ways.

LAMMPS snapshots are created by the *dump* command, which can create files in several formats. The native LAMMPS dump format is a text file (see **dump atom** or **dump custom**) which can be visualized by several visualization tools for MD simulation trajectories. OVITO and VMD seem to be the most popular choices among them.

The *dump image* and *dump movie* styles can output internally rendered images or convert them to a movie during the MD run. It is also possible to create visualizations from LAMMPS inputs or restart file with LAMMPS-GUI, which uses the *dump image* command internally. If the LAMMPS input already contains a *dump image* command, the resulting images will be noted by LAMMPS-GUI and can be viewed and animated directly in the `Slide Show` dialog. The images can be transformed (i.e. scaled, mirrored, or rotated) and exported into a video, too. The `Image Viewer` dialog in LAMMPS-GUI can be used to visualize the *current* system, adjust a variety of visualization settings quasi-interactively from the GUI.

Programs included with LAMMPS as auxiliary tools can convert between LAMMPS format files and other formats so they can be read by certain programs. See the *Tools* page for details. These are rarely needed these days.

### 10.1.3 Basic workflow for loading LAMMPS trajectories in VMD

VMD can read native LAMMPS dump files (in text format not binary) and several other dump styles. The native LAMMPS format is preferred since it contains simulation box information. VMD does not support reading data files directly, but the TopoTools plugin does. This step is usually necessary, since the LAMMPS dump files do not contain information about the molecular topology and elements and that information can be obtained from the data file. There also is the pbctools plugin that can be used for "repairing" bonds that are broken because one of its atom passed through a periodic boundary. Because of using the plugins, the following commands need to be typed into the VMD console to read a trajectory. The following commands are based on the `peptide` example to which the command `dump 1 all atom 10 dump.peptide` was added. Note the use of the group "all" in contrast to the commented out dump commands that use the group "peptide". This is to match the number of atoms in the data file. VMD does not support cases where the number of atoms change or you are trying to read in a subset of a system unless *all* files contain the same subset.

```
# load data file including its coordinates
topo readlammpsdata data.peptide full
# atom style full is the default ^^^^ and can be omitted

# try to infer missing atom properties from available information
topo guessatom lammps data

# now add one of more dump files
mol addfile dump.peptide type lammpstrj waitfor all
# we tell VMD which file type ^^^^^^^^^ if the filename does not end in .lammpstrj

# jump to the first frame with the data file coordinates
animate goto 0
# if needed  use the following command to try and repair broken bonds
# pbc join fragment -now

# unwrap the entire trajectory to repair all broken bonds
# this is faster and more accurate than running `pbc join` for all frames
pbc unwrap -all
pbc wrap -all -compound fragment -orthorhombic
#       remove for triclinic box ^^^^^^^^^^^^

# delete the coordinate frame from the data file
animate delete beg 0 end 0 top
# write out the topology information without coordinates to a PSF file
animate write psf peptide.psf top
# and the processed trajectory data to a DCD file for future use
animate write dcd peptide.dcd top waitfor all

##############################################################
# the steps up to this line only need to be run once
##############################################################
# set visualization defaults
display perspective orthographic
mol default style Licorice
mol default material Diffuse
display backgroundgradient on

# delete molecule and load psf and dcd file
mol delete top
```

(continues on next page)

(continued from previous page)

```
mol new peptide.psf
mol addfile peptide.dcd waitfor all
pbc box
```

To automate this process, you can also save these commands to a file, e.g. `peptide.vmd` and then load this file from the VMD "File" menu with "Load Visualization State…" or type in the command console `play peptide.vmd`.

### 10.1.4 Advanced graphics features in the *dump image* command

Added in version 11Feb2026.

The following paragraphs discuss some of the more advanced features in the *dump image* command in LAMMPS with the help of some input file examples. For exact details of keywords and arguments, please refer to the detailed documentation of the respective commands.

Please note that many of these features were added or significantly updated after LAMMPS version 10 Dec 2025 and well into the 2026 stable version development cycle. If you are using an older version of LAMMPS, these examples may likely cause errors or look differently.

**Available topics**


- *Complete example inputs*
- *Image quality and resolution*
- *Color selection and color management*
- *Transparency*
- *Creating and viewing animated GIFs and movie files*
- *Prototyping dump image visualizations with LAMMPS-GUI*
- *Visualizing systems using potentials with implicit bonds*
- *Visualizing body particles*
- *Visualizing ellipsoid and superellipsoid particles*
- *Visualizing regions*
- *Visualizing graphics provided by compute or fix commands*
  - *Fix graphics/objects*
  - *Fix graphics/labels*
  - *Fix graphics/arrows*
  - *Fix graphics/isosurface*
  - *Compute hbond/local*
  - *Fix reaxff/bonds*
  - *Fix smd/wall_surface*
  - *MC package fixes*

### Complete example inputs

The discussions below only quote relevant sections of input files to show specifically the commands used in the visualizations. There are complete example files in the `examples/GRAPHICS` folder of the LAMMPS source code distribution.

---

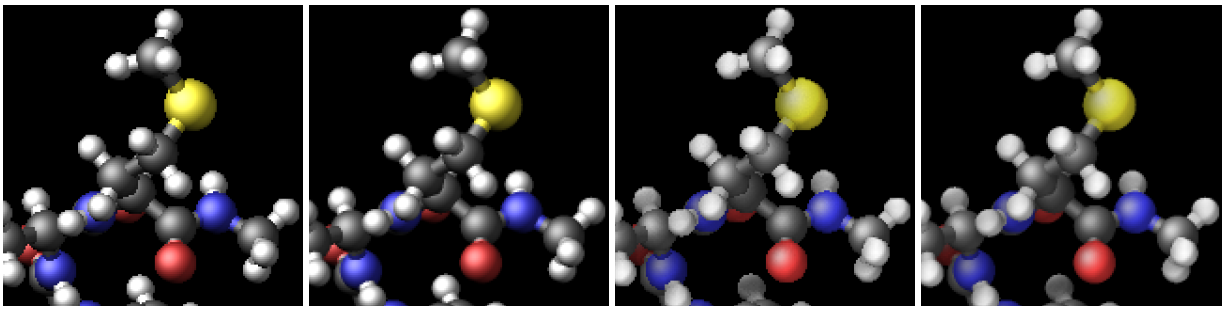
### Image quality and resolution

The image resolution is determined by the *size* keyword. The default setting is to create images with 512x512 pixels. This is rather low resolution. A typical (non-"retina") computer screen at the time of writing this howto has about 100 dpi (= dots per inch) so that this image covers an area of a bit over 5x5 inches (or about 135x135 millimeters). Images for use in presentations, posters, or publications should generally be generated at larger sizes. When the image is meant for printing, one has to consider that a printer may have a resolution of 300 dpi or even 600 dpi which means that images should be created 3x3 or 6x6 times larger to utilize the full resolution of the printer and show as much details and be as clean as possible. Otherwise images have to be scaled up which can make them look blurry and with ragged edges.

The keywords *fsaa* and *ssao* can be used to further improve the image quality at the expense of significant additional computational cost to render the images:

- FSAA stands for Full Scene Anti-Aliasing and means in the case of LAMMPS that the image is rendered at four times the size (double the width and double the height) and then the pixels in the final image are computed by taking the average of 2x2 pixel groups. This will result in smoother, less ragged edges of objects in the image.
- SSAO stands for Screen Space Ambient Occlusion and is an image enhancement technique that uses intermediate image data (like the information of how close or far away a pixel is to the camera and its neighboring pixels) and brightens or darkens randomly selected pixels in its neighborhood based on that information. This enhances the depth perception of objects in an image.

Both methods are complementary and thus can be combined for additional improvement of the image quality. The images below show from left to right the same excerpt of a dump image output with the default settings, with *fsaa* enabled, with *ssao* enabled, and with both features enabled.



The computational cost to create the images with *dump image* depends on the image size, the number of objects to be rendered (this number can grow quickly when using fine triangle meshes), and the choice of the *fsaa* and *ssao* settings. For high resolution images, a correspondingly large image size has to be chosen.

Since the simulation has to wait for the dump image command to complete its image rendering, creating high resolution, high quality images can slow down a simulation significantly with frequent output of images. On the other hand, the image rasterizer in LAMMPS is fairly simple and thus fast compared to more advanced image generation tools like ray tracers. The method it uses to generate the images allows to have each MPI process create image data for the atoms and objects they "own" and then this image data is merged for the final output with a procedure that has O(log(N)) complexity with the number of MPI processes. At the moment there is no GPU acceleration and only limited multi-threading parallelization available (e.g. for SSAO post-processing of image data).

---

### Color selection and color management

The *dump image* command in LAMMPS has a variety of options to assign colors to the rendered graphics. In most cases the color is assigned to atom (or bond) types and uses a default map with six colors as follows:

- type 1 = red
- type 2 = forestgreen
- type 3 = blue
- type 4 = gold
- type 5 = cyan
- type 6 = magenta
- type 7 = silver
- type 8 = orange
- type 9 = lime
- type 10 = gray
- type 11 = darkred
- type 12 = darkgreen
- type 13 = darkblue
- type 14 = darkcyan
- type 15 = darkmagenta
- type 16 = darkgray

and repeats itself for types > 16. The default color sequence is thus:

This mapping can be changed by the "dump_modify acolor" command, though. If you want to change the color of a specific atom type, you can use *dump modify acolor*. For example to color atoms of type 1 in gray and type 2 in white, you would use:

```
dump_modify img  acolor 1 gray acolor 2 white
```

There are 140 predefined colors, but you can add new colors or modify existing ones, too, with the *dump_modify color* keyword. The *color* keyword is followed by the name of the color and the intensity of the red, green, and blue components (R/G/B) in a range from 0.0 to 1.0. Here is an example to create eight new color names followed by the *acolor* keyword with a wildcard to replace the default map of six atom colors with a new map of the eight newly defined colors.

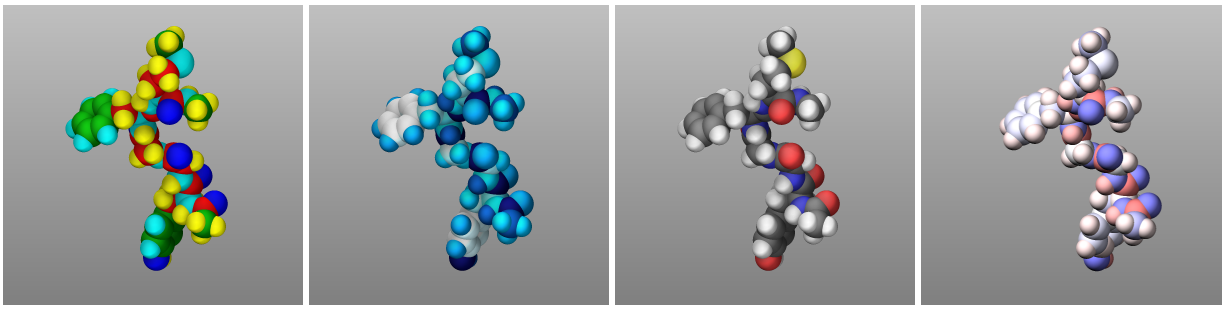
```
dump_modify viz color map1 0.012 0.016 0.322 color map2 0.008 0.243 0.541 &
    color map3 0 0.467 0.714 color map4 0 0.588 0.780 color map5 0 0.706 0.847 &
    color map6 0.282 0.792 0.894 color map7 0.565 0.878 0.937 color map8 0.792 0.941 0.
↪973 &
    acolor * map1/map2/map3/map4/map5/map6/map7/map8
```

Yet another option is to color the atoms by element. The per-element colors are predefined but LAMMPS does not know which element an atom type corresponds to and by default uses carbon for all atom types. The correct element information needs to be provided with a *dump_modify element* command followed by an element name for each atom type. In case of the `peptide` example bundled with LAMMPS this would be:

```
dump viz peptide image 1000 image-*.png element type size 600 600 zoom 2.0
dump_modify viz element C C O H N C C C O H H S O H
```

Finally, atoms can be colored by the value of a per-atom property using a color map. There are several variants of color maps. Here is a simple example coloring the atoms in the `peptide` example by their charge with a continuous color map where white is neutral, red positive, blue negative, and the color intensity corresponds to the magnitude of the charge:

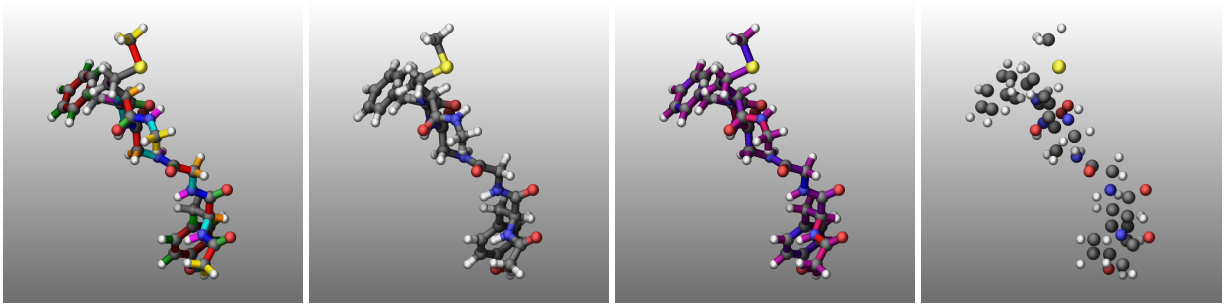
```
dump viz peptide image 1000 image-*.png q type size 600 600 zoom 2.0
dump_modify viz amap -1.0 1.0 ca 0 3 min blue 0.0 white max red
```



Changed in version 4Jul2026.

Similar color selections are available for coloring bonds. The available options are: *type*, *atom*, *c_ID* (or *c_ID[I]*), and *none*. With *type* the bonds are colored by having a color assigned to each bond type which follows the same color sequence as for atoms but can be set for each bond type independent from atom types. When using the *atom* selection the bond color follows the color of the atoms. Bonds are drawn in two pieces as a cylinder from the center of the bond to each of the atoms. Thus if two atoms have different color, the bond also as two parts with different colors with this setting. If the a compute reference is used (e.g. *c_ID* or *c_ID[I]*) the bond color is taken from a colormap and the color depends on the value of the compute for the given bond. An input example for coloring bonds by the force magnitude is given below. When the bond color argument is *none*, no bonds are drawn.

```
compute  bforce peptide bond/local force
dump     viz    peptide image 100 myimage-*.png element type bond c_bforce type
```

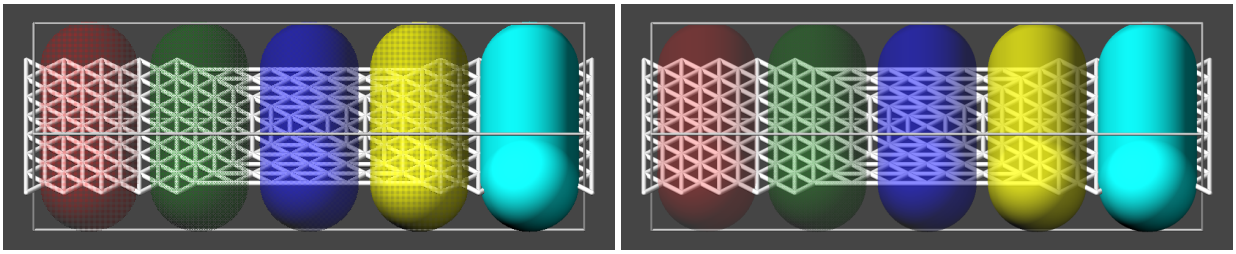


### Transparency

It is now possible to create approximately transparent graphics objects using an ordered dithering algorithm which results in a so-called *screen-door transparency* effect. In essence, for a transparent object only a part of the pixels are drawn and thus exposing any object behind the transparent object where drawing the pixels is skipped. LAMMPS employs a 16x16 Bayer matrix pattern that leads to rather regular patterns. A benefit of this approach is that it does not add extra cost to the rendering and for a 25%, 50%, and 75% transparency setting, there are no visible pixel patterns

when also FSAA is enabled. In this case each pixel is the average of a 2x2 block of pixels in an image of double the width and height, and thus the transparent object will contribute 3, 2, or 1 pixels to the 2x2 block which is averaged.

Transparency is typically - like the color of objects - associated with an atom type and can be modified through the *dump_modify atrans* command and specified as an opacity ratio, i.e. a number between 0 (fully transparent) and 1 (fully opaque). Other choices are available and described in the documentation page.



### Creating and viewing animated GIFs and movie files

A series of JPEG, PNG, or PPM images can be converted into a movie file and then played as a movie using commonly available tools. Using dump style *movie* automates this step *and* avoids the intermediate step of writing (many) image snapshot file. But LAMMPS has to be compiled with `-DLAMMPS_FFMPEG` and a compatible FFmpeg executable has to be installed. When using LAMMPS-GUI to run LAMMPS, you can run the simulation and LAMMPS-GUI will automatically show the created images in its `Slideshow Viewer` dialog. From there you can animate or single step through them and also export them to a movie file via FFMpeg.

To manually convert JPEG, PNG or PPM files into an animated GIF or MPEG or other movie file you can use:

1. Use the ImageMagick `convert` program (called `magick` in recent versions).

   ```
   convert *.jpg foo.gif
   convert -loop 1 *.ppm foo.mpg
   ```

   Animated GIF files from ImageMagick are not optimized. You can use a program like gifsicle to optimize and thus massively shrink them. MPEG files created by ImageMagick are in MPEG-1 format with a rather inefficient compression and low quality compared to more modern compression styles like MPEG-4, H.264, VP8, VP9, H.265 and so on.

2. Use QuickTime.

   Select "Open Image Sequence" under the File menu Load the images into QuickTime to animate them Select "Export" under the File menu Save the movie as a QuickTime movie (*.mov) or in another format. QuickTime can generate very high quality and efficiently compressed movie files. Some of the supported formats require to buy a license and some are not readable on all platforms until specific runtime libraries are installed.

3. Use FFmpeg

   FFMpeg is a command-line tool that is available on many platforms and allows extremely flexible encoding and decoding of movies.

   ```
   cat snap.*.jpg | ffmpeg -y -f image2pipe -c:v mjpeg -i - -b:v 2000k movie.m4v
   cat snap.*.ppm | ffmpeg -y -f image2pipe -c:v ppm -i - -b:v 2400k movie.mp4
   ```

   Front ends for FFmpeg exist for multiple platforms. For more information see the FFmpeg homepage

Play the movie:

1. Use your web browser to view an animated GIF or MP4 movie format movie.

   Select "Open File" under the File menu Load the animated GIF or MP4 movie file

2. Use the freely available VideoLAN media player (vlc) or FFMpeg player tool (ffplay) to view a movie.

   Both are available for multiple operating systems and support a large variety of file formats and decoders. There are plenty more media player packages available on the different operating systems.

   ```
   vlc foo.mpg
   ffplay bar.avi
   ```

3. Use the Pizza.py animate tool, which works directly on a series of image files.

   ```
   a = animate("foo*.jpg")
   ```

4. QuickTime and other Windows- or macOS-based media players can obviously play movie files directly. Similarly for corresponding tools bundled with Linux desktop environments. However, due to licensing issues with some file formats, the formats may require installing additional libraries, purchasing a license, or may not be supported.

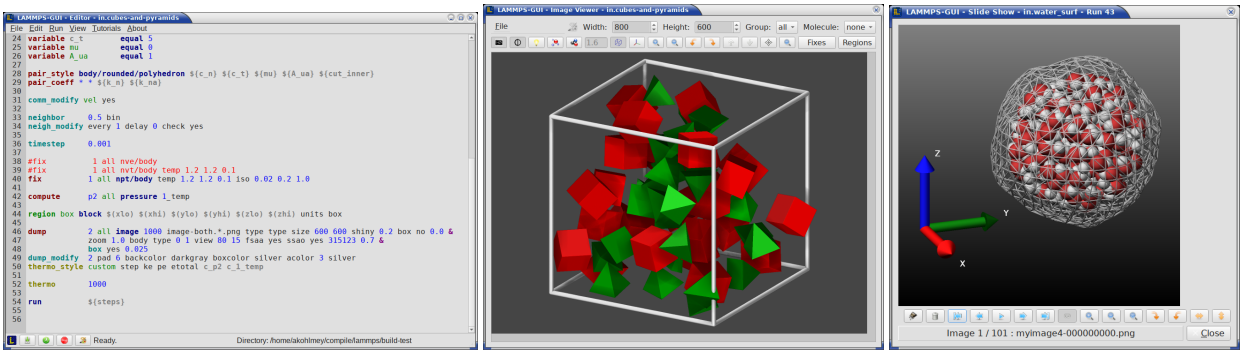
---

### Prototyping dump image visualizations with LAMMPS-GUI

One of the challenges when using *dump image* for creating visualizations compared to the likes of OVITO and VMD is that it is non-interactive and it can be tedious and time consuming to find suitable settings for camera view or zoom factor and others for a specific system. You would have to run LAMMPS to create an image, then view it in an image viewer program, edit the input file and repeat the process until you have found settings that you like.

This process can be streamlined with LAMMPS-GUI, which does not contain its own visualization code, but rather uses *dump image* through the *LAMMPS C-language library interface* and displays the resulting images for its Image Viewer Dialog. Through the GUI elements and dialogs many settings can be adjusted and then the image will be recreated with the updated settings and thus allowing to refine a visualization in a quasi-interactive fashion. The resulting command lines can be transferred to the cut-n-paste buffer of the windowing system and pasted into the input file and then further adjusted.

Once the input contains a *dump image* command, LAMMPS-GUI notices when a new image has been created and loads it into the "Slide Show Dialog". This streamlines the process of building more complex visualizations once you have copied an initial draft created with the "Image Viewer Dialog" into the input since you have editor and image viewer as part of the same program and can quickly start and stop LAMMPS with a mouse click or keystroke. A large part of the visualization examples shown in this Howto page have been created this way.



### Visualizing systems using potentials with implicit bonds

There are several pair styles available in LAMMPS where the bond information is not taken from the bond topology in a data file but the potentials first determine a "bond-order" parameter for pairs of atoms and - depending on the value of that parameter - apply forces for bonded interactions. This applies to *ReaxFF*, *REBO and AIREBO*, *BOP*, and several others pair styles. These implicit bonds will not be shown by *dump image* since its mechanism for displaying bonds relies on explicit bonds being present in the bond topology.

One can hide the fact that there are no bonds by setting the atom radii to the covalent radii of the corresponding elements (see leftmost example image below). This will result in a representation often labeled as "VDW" in popular visualization tools. Otherwise, there are currently three approaches to make those bonds visible.

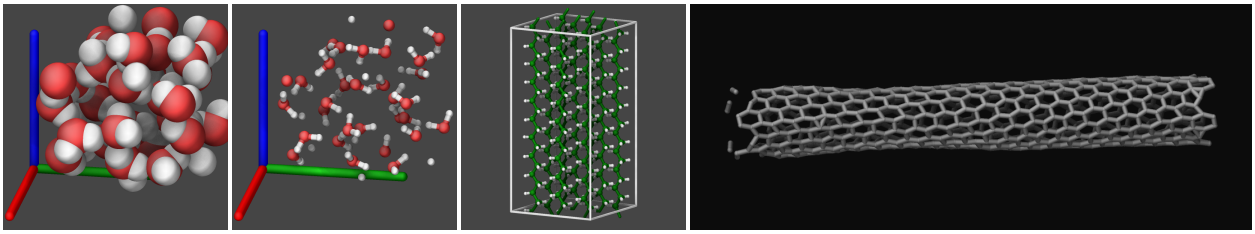
1. Access the (internal) bond order information from the pair style through a custom fix and then use the *fix* keyword of the *dump image* command to use the graphics objects information provided by the fix to visualize the bonds (see below for more information). That includes bonds that are broken and formed. This is the most accurate option since it uses the data that is used by the computation of the model. This is currently only available for ReaxFF by using *fix reaxff/bonds*.

2. Use the *autobond* keyword of *dump image* to approximate the bonds based on a simple distance heuristic. This is similar to the *Dynamic Bonds* representation in VMD. How accurate this option will be depends on the complexity of the system and how many different bond lengths there are. For the simple water system shown below, there are no significant differences, since there is only one type of bond (between oxygen and hydrogen). The *autobond* keyword uses on top of the distance cutoff between atoms, the heuristic that no bonds will be drawn between two hydrogen atoms and thus bogus bonds are avoided when using a larger bond cutoff (e.g. suitable for carbon-carbon bonds) which is larger than the typical hydrogen-hydrogen distance for hydrogen atoms bound to the same atom (e.g. in water, methane or hydrocarbon chains).

3. Use a combination of *fix bond/break* and *fix bond/create/angle* with *bond style zero* to dynamically create and remove bonds that do not add any forces. This also requires to tell the neighbor list code to not treat any pairs of atoms as special neighbors (otherwise the corresponding pairs of atoms could be excluded from the neighbor list and thus the forces computed by the pair style incorrect) through using the *special_bonds* command. Unlike the two other options, which were added more recently, this method also works with older versions of LAMMPS. Here is an example of the necessary commands for a carbon nanotube modeled with the AIREBO potential:

```
bond_style zero
bond_coeff 1 1.4
special_bonds lj/coul 1.0 1.0 1.0
fix break all bond/break 1000 1 2.5
fix form all bond/create/angle 1000 1 1 2.0 1 aconstrain 90.0 180
```

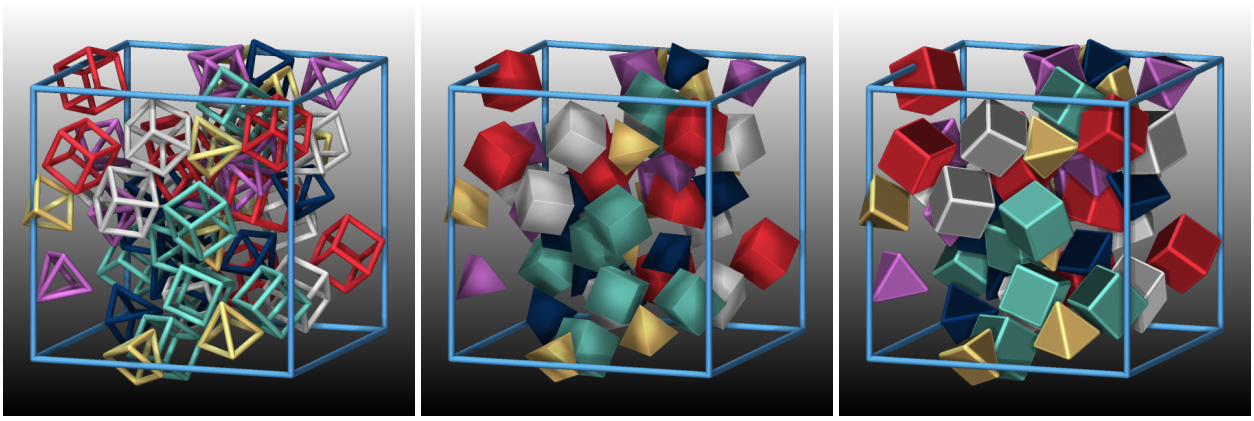
This "graphics hack" was originally posted as part of the LAMMPS tutorial at https://lammpstutorials.github.io/sphinx/build/html/tutorial2/breaking-a-carbon-nanotube.html



### Visualizing body particles

Body particles are objects formed from either a collection of spherical particles, polygons (in 2d), or polyhedra (in 3d) formed from triangular or quadrilateral surfaces. The regular *dump* command can only output the center of those bodies (and their orientation), which complicates the visualization with external tools. In addition, the position of the constituent particles of *nparticles* bodies or the positions of the vertices of *rounded/polygon* or *rounded/polyhedron* bodies, which can be computed with *compute body/local* and output with *dump local*.

As an alternative, the bodies can be visualized directly with *dump image* using the *body* keyword. Without the *body* keyword the body particles would be visualized like atoms as single spheres. The color and transparency settings can be changed by settings those properties for the corresponding atom types. It is also possible to represent the bodies as either wireframes (*bflag1* value 2), planar faces (*bflag1* value 1), or both (*bflag1* value 3).



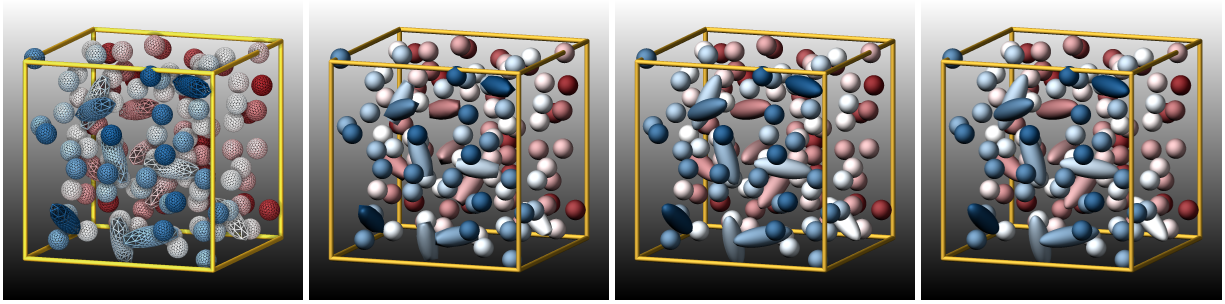
### Visualizing ellipsoid and superellipsoid particles

Added in version 11Feb2026.

Ellipsoidal particles are a generalization of spheres that may have three different radii to define the shape. Superellipsoids are in turn a generalization of ellipsoids. They can be modeled using pair styles like *gayberne* or *resquared*. The regular *dump custom* command can output the center of those bodies, the shape parameters and the orientation as quaternions. If one follows the required conventions and follows the documented steps, those trajectory dump files can be imported and visualized in OVITO

Changed in version 30Mar2026: Now uses curved triangles instead of flat ones; "both" option is removed; support for superellipsoids was added

As an alternative, the ellipsoid and superellipsoid particles can be visualized directly with *dump image* using the *ellipsoid* keyword. The color and transparency settings can be changed by setting those properties for the corresponding atom types. It is also possible to represent the ellipsoids via generating a triangle mesh and visualizing it as either wireframes (*eflag* value 2) or rounded triangle faces (*eflag* value 1). The use of a triangle mesh is currently required since the rasterizer built into LAMMPS does not offer suitable graphics primitives for ellipsoids or superellipsoids. The mesh is constructed by iteratively refining a triangle mesh representing an icosahedron, where each triangle is replaced by four triangles in each iteration. For a smooth representation a refinement level of 4 seems sufficient, but high resolution images may benefit from a higher level (maximum is 6, see example images below). A high refinement level can cause a significant slowdown of the rendering of the image due to the large number of triangles that need to be computed and drawn. This slowdown will be more pronounced when enabling FSAA or SSAO or both.

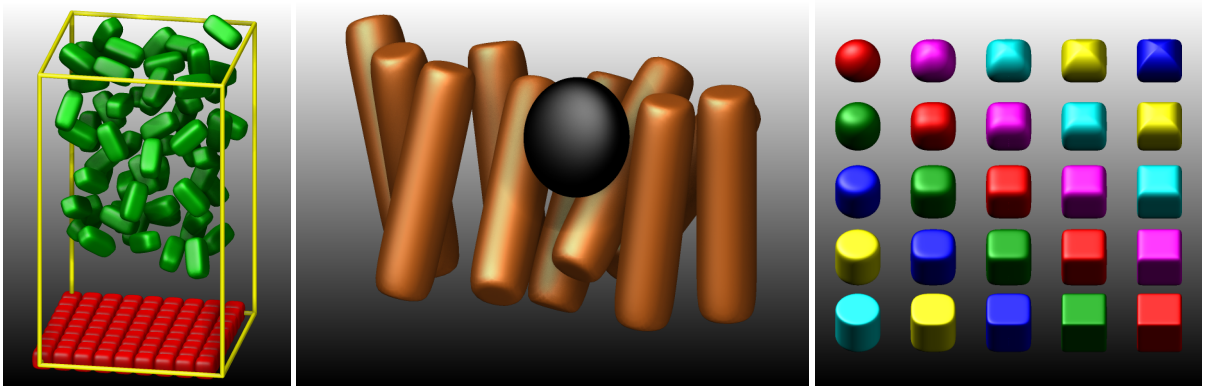


These images were created by adding the following *dump image and dump_modify* commands to the `in.ellipse.resquared` input example:

```
#                                                  change /V\ this
dump viz all image 1000 image-*.png x type ellipsoid atom 1 4 0.2 &
     size 600 600 zoom 1.331 view 80 20 box yes 0.025 shiny 0.2 fsaa yes
dump_modify viz pad 6 boxcolor goldenrod backcolor black backcolor2 white &
     color map1 0.459 0.055 0.075 color map2 0.000 0.227 0.427 &
     amap min max cf 0.0 5 min map1 0.1 map1 0.5 white 0.9 map2 max map2
```

Added in version 30Mar2026.

The visualization of superellipsoids works exactly the same way as for ellipsoids by creating a triangle mesh of an icosahedron and refining and deforming it. The difference is merely internally the applied deformation function and the corresponding computation of the surface normals. LAMMPS will auto-detect which function to use. Some visualizations of the `in.drop_test`, the `in.bowling`, and the `in.super_table` examples from the `examples/ASPHERE/superellipsoid_gran` folder are shown below.



### Visualizing regions

Since there are several commands that operate on atoms within a specific *region* , it can be helpful to visualize the extent of those regions, either for debugging the command settings or for visualizing the production simulation. This is even more useful when regions are (dynamically) translated or rotated or both with the *rotate* or *move* keyword of the *region* command. The *dump image region* keyword can be used to enable such visualizations. There are currently four different draw styles available:

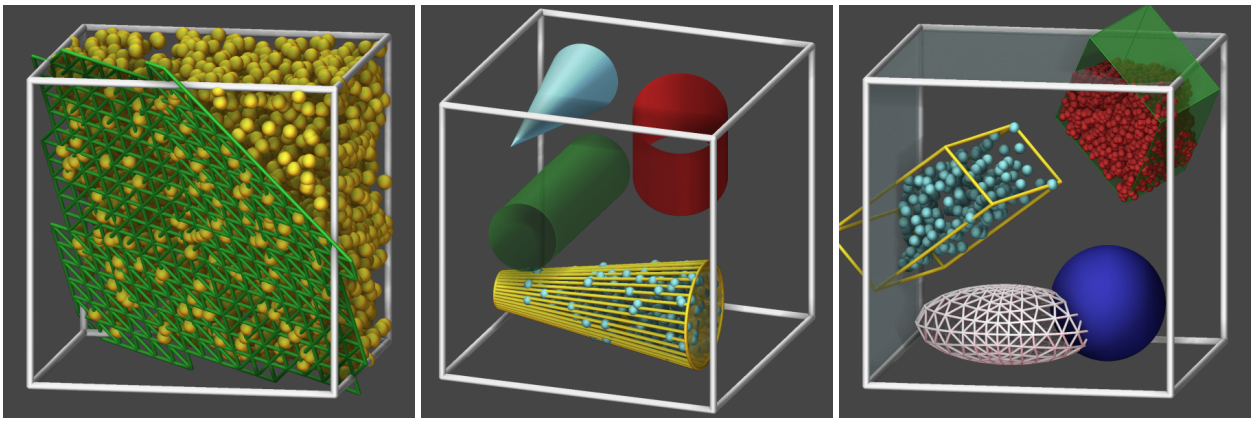
- *filled*: a mesh of triangles is constructed for the surface of the region and then drawn in the selected color. This draw style accounts for the *open* keyword of the *region* command and only draws the faces of a region that are closed.
- *transparent*: this is the same visualization as draw style *filled*, but in addition one can set the transparency of that visualization through an opacity parameter in the range of 0.0 (invisible) to 1.0 (fully opaque).

- *frame*: uses the same mesh of triangles as the two styles above above, but renders the region's surface as a wireframe mesh with a given cylinder diameter.
- *points*: generates a cloud of random points inside the simulation box and then only draws a point as a sphere with the given color and radius if it is located inside the region. This is useful to check the impact of the *side in/out* setting of a region, and complementary to the other three draw styles, which only show the region surfaces. For visualization purposes any open faces of a region are ignored, since a region with an open face matches *all* particles.

It can sometimes be convenient to draw the same region with multiple draw styles as can be seen from the example visualization images below.

Notes on the visualization of individual region styles:

- *plane*: since in theory planes extend infinitely, some form of boundary has to be established for the visualization. Therefore a mesh of triangles is constructed and transformed according to the region settings and then only those triangles are drawn, where at least one corner is inside the simulation box. For draw style *points* the *side in/out* setting determines on which side of the plane the points are drawn.
- *cone* and *cylinder*: use triangle meshes for draw styles *filled* and *transparent* since there are no equivalent primitives (the cylinder primitive only draws half of the cylinder, since it is optimized for being capped). For draw style *frame*, only the rim of the top or bottom is drawn and the side is represented by lines from the top to the bottom.
- *box* and *prism*: for draw style *frame* only the edges of the region are shown; for draw styles *filled* and *transparent*, the faces are drawn instead, but only if they are not set as "open" in the *region* command.



Below is an example input deck for visualizing *cone* and *cylinder* regions:

```
region      box block -2 2 -2 2 -2 2
create_box 0 box

variable rot equal PI*step/1000.0

region c1 cylinder x -1.0 0.0 0.5 -1.5 1.5 open 1 open 2
region c2 cone y  1.0 -1.0 0.25 0.75 -1.5 1.5 open 3
region c3 cylinder z  0.0 1.0 0.66 -0.1 1.5 open 1 open 2 rotate v_rot 0.3 0.0 0.3 0.0 1.
↪0 1.0
region c4 cone x -1.0 1.5 0.5 0.0 -1.0 2.0

dump viz all image 100 image-*.png type type size 600 600 zoom 1.4 shiny 0.1 view 70 20 &
             box yes 0.025 axes no 0.0 0.0 fsaa yes ssao yes 314123 0.7 &
             region c1 forestgreen transparent 0.5 &
```

(continues on next page)

(continued from previous page)

```
             region c2 goldenrod frame 0.05 &
             region c2 cadetblue points 10000 0.15 &
             region c2 goldenrod transparent 0.5 &
             region c3 firebrick filled &
             region c4 skyblue filled
dump_modify viz pad 4 boxcolor silver backcolor darkgray

run 500
```

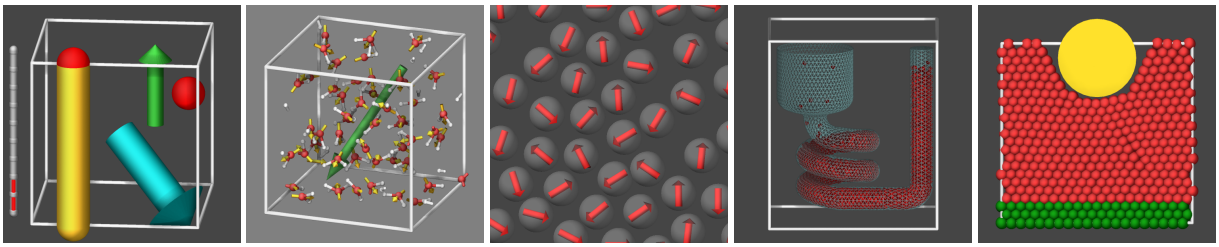
### Visualizing graphics provided by compute or fix commands

LAMMPS can display additional graphics objects in the *dump image* output that are added by compute or fix styles. These fall in two categories: fixes that were written with the specific purpose of adding graphics to the visualization and computes or fixes that make objects or data visible that they maintain internally. Examples for the latter case are visualizing the indenter object from *fix indent* or the wall position from one of the wall fixes. The details of what kind of graphics is added and how it can be configured is described in a section titled **Dump image info** in the documentation of the individual fix commands.

Below is a table with links to the documentation of supported compute and fix styles:

| | | | |
|---|---|---|---|
| *compute chunk/atom* | *compute hbond/local* | *fix graphics/arrows* | *fix graphics/chunk* |
| *fix graphics/isosurface* | *fix graphics/labels* | *fix graphics/lines* | *fix graphics/objects* |
| *fix graphics/periodic* | *fix graphics/replica* | *fix atom/swap* | *fix bond/break* |
| *fix bond/create* | *fix bond/create/angle* | *fix bond/react* | *fix mol/swap* |
| *fix neighbor/swap* | *fix indent* | *fix reaxff/bonds* | *fix smd/wall_surface* |
| *fix surface/global* | *fix wall/body/polygon* | *fix wall/body/polyhedron* | *fix wall/ees* |
| *fix wall/lj93* | *fix wall/lj126* | *fix wall/lj1043* | *fix wall/colloid* |
| *fix wall/gran* | *fix wall/harmonic* | *fix wall/harmonic/outside* | *fix wall/lepton* |
| *fix wall/morse* | *fix wall/reflect* | *fix wall/reflect/stochastic* | *fix wall/srd* |
| *fix wall/table* | | | |

There is no support for *fix wall/region* and *fix wall/gran/region*, since regions can be visualized with the *region* keyword of *dump image* (see discussion above).



Below are discussions about some aspects of specific fix commands and some input examples.

### Fix graphics/objects

Fix *graphics/objects* adds some graphics primitives and more complex objects like a progress bar to the visualization where properties of the object(s) are controlled by *equal-style or compatible variables*.

### Fix graphics/labels

Fix *graphics/labels* adds graphics from pixmaps to the visualization. These can be either images or text that is rendered internally into a pixmap with a background and a frame. Those pixmaps can be scaled, moved, made transparent, and updated during the simulation and then integrated into the *dump image* output. In both cases a "transparency" color can be chosen to skip copying any pixels of that color to make parts of the pixmap transparent. The text labels can contain *variables* that will be expanded in the same was as with *fix print* so they can report values of computed properties. Label positions are provided in the frame of reference of the simulation box and thus they can obscure other objects in then visualization or be obscured by objects.

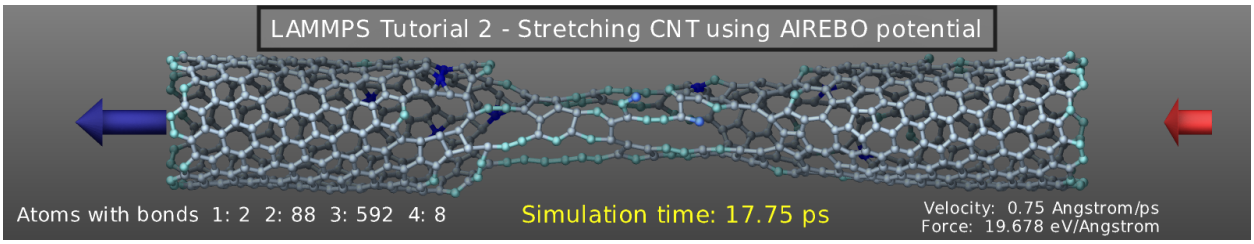



Fig. 1: Example image using graphics objects and labels that are updated during the simulation.

### Fix graphics/arrows

Fix *graphics/arrows* adds per-atom or per-chunk arrows to the visualization. The arrows represent some per-atom property or per-chunk 3-vector property. For atoms, there are the pre-defined properties *force*, *velocity*, and *dipole* (requires atom style *dipole*). Everything else would have to be computed in *atom-style or compatible variables* and use the *variable* keyword. For per-chunk properties, one needs to use the *chunk* keyword and provide the IDs of three computes: a *chunk/atom compute* and two per-chunk computes where the first defines the position of the (middle of the) arrow and the second the direction and length of the arrow. Popular choices would create per-molecule or per-bin chunks. Below is an example input section that computes and displays both, the total dipole moment (using *fix graphics/objects* and the per-molecule dipole moment as arrows in addition to the per-atom velocities:

```
compute molchunk all chunk/atom molecule
compute cpos all com/chunk molchunk wrap yes
compute cdip all dipole/chunk molchunk
fix vel all graphics/arrows 10 velocity 50.0 0.066 autoscale 0.25
fix vec all graphics/arrows 10 chunk molchunk cpos cdip 3 0.1

compute dip all dipole
variable scale equal 0.75
variable dip1x equal -v_scale*c_dip[1]
variable dip1y equal -v_scale*c_dip[2]
variable dip1z equal -v_scale*c_dip[3]
variable dip2x equal v_scale*c_dip[1]
variable dip2y equal v_scale*c_dip[2]
variable dip2z equal v_scale*c_dip[3]
fix dipole all graphics/objects 1 arrow 1  v_dip1x v_dip1y v_dip1z v_dip2x v_dip2y v_
↪dip2z 0.3 0.2

dump viz all image 100 image-*.png element type size 600 600 zoom 1.3 view 70 20 shiny 0.
↪1 &
            bond atom 0.2 box yes 0.025 axes no 0.0 0.0 center s 0.5 0.5 0.5 fsaa yes &
```

(continues on next page)

(continued from previous page)

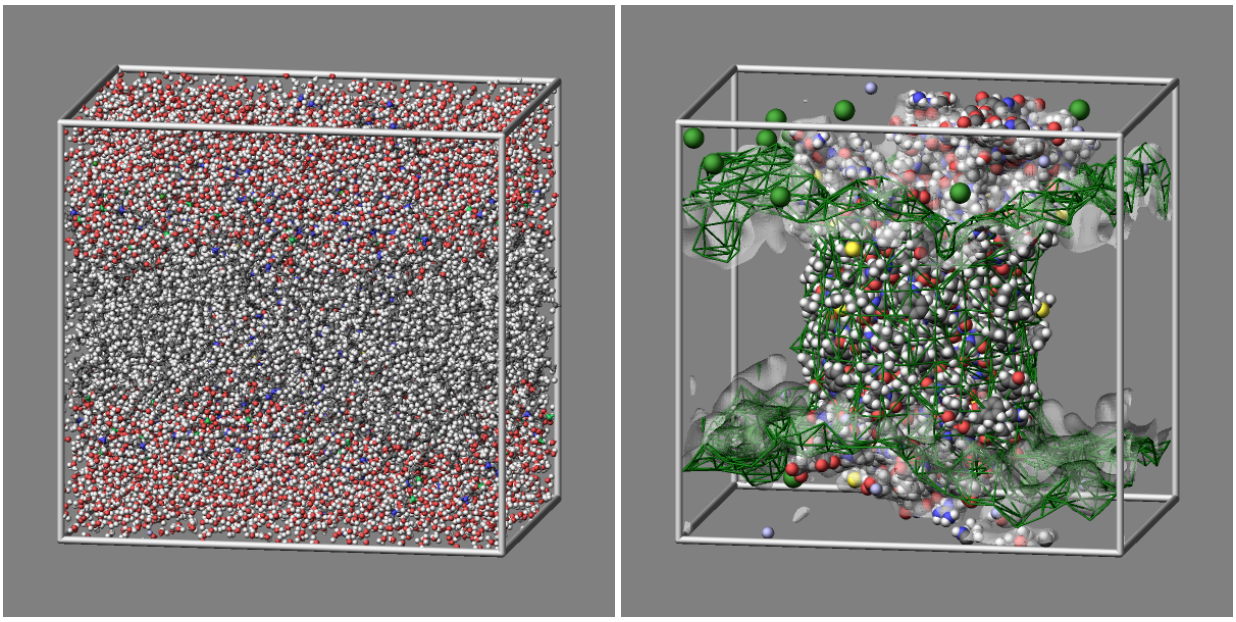
```
             fix dipole const 0 0 fix vec const 0 0 fix vel const 0 0 ssao yes 315465 0.8
dump_modify viz pad 6 boxcolor white backcolor gray element O H  bdiam 1 0.2 &
             adiam 1 0.5 adiam 2 0.3 acolor 1 silver acolor 2 red fcolor vec goldenrod &
             fcolor dipole forestgreen ftrans dipole 0.75 fcolor vel cyan ftrans vel 0.5
```

### Fix graphics/isosurface

Fix *graphics/isosurface* adds a triangulated surface following a given isovalue through a 3d-grid of data of some per-atom property. The data is spread out using a Gaussian distribution with a given width and can be just a value of 1 (leading to a grid representing the number density) or mass or some other computed per-atom property from a *compute*, *fix*, or atom-style *variable*. The isosurface can be represented by either a wireframe or a mesh of triangles and there are five choices for the resolution of the grid and thus the smoothness of the surface.

The commands below provide an example for how to use the *graphics/isosurface* fix to visualize the water and lipid bilayer of the "rhodo" benchmark example as isosurfaces by using a green wireframe and a transparent white triangle surface to represent those molecules.

```
group water type 4 33
group membrane type 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51
group other subtract all water membrane

fix water water graphics/isosurface 100 30.0 3.0 quality high property mass
fix membrane membrane graphics/isosurface 100 25.0 4.0 quality low property mass

dump viz other image 100 image-*.png element type size 600 600 zoom 1.4641 view 80 10 &
              shiny 0.2 fsaa yes bond atom type box yes 0.025  &
              fix membrane const 2 0.33 fix water const 1 0.2
dump_modify viz pad 9 boxcolor silver backcolor gray &
       fcolor membrane darkgreen ftrans membrane 1.0 ftrans water 0.5 &
       element H H H H H H H H C C C C C C C C C C C C N N N N N N O O O O S S &
               H H H H H C C C C C C N O O O P Cl Na H H H N C C C C C C C C C C &
       adiam 1*9 1.92 adiam 10*22 2.72 adiam 23*29 2.48 adiam 30*33 2.432 adiam 34*35 2.
↪88 &
       adiam 36*40 1.92 adiam 41*46 2.72 adiam 47 2.48 adiam 48*50 2.432 adiam 51 2.88 &
       adiam 52 3.632 adiam 53 2.176 adiam 54*56 1.92 adiam 57 2.48 adiam 58*68 2.72
```



### Compute hbond/local

Compute *hbond/local* of the *EXTRA-COMPUTE package* provides access to the list of hydrogen bonds as they are dynamically computed by the compute style. These can be added to the *dump image output* as arrows pointing from the hydrogen bond hydrogen atom to the hydrogen bond acceptor atom by using the *compute* keyword. The compute provides a lot of flexibility in which atoms are considered for the hydrogen bonds so that the visualization can be rather simple and straightforward but also more complex to visualize only selected aspects.

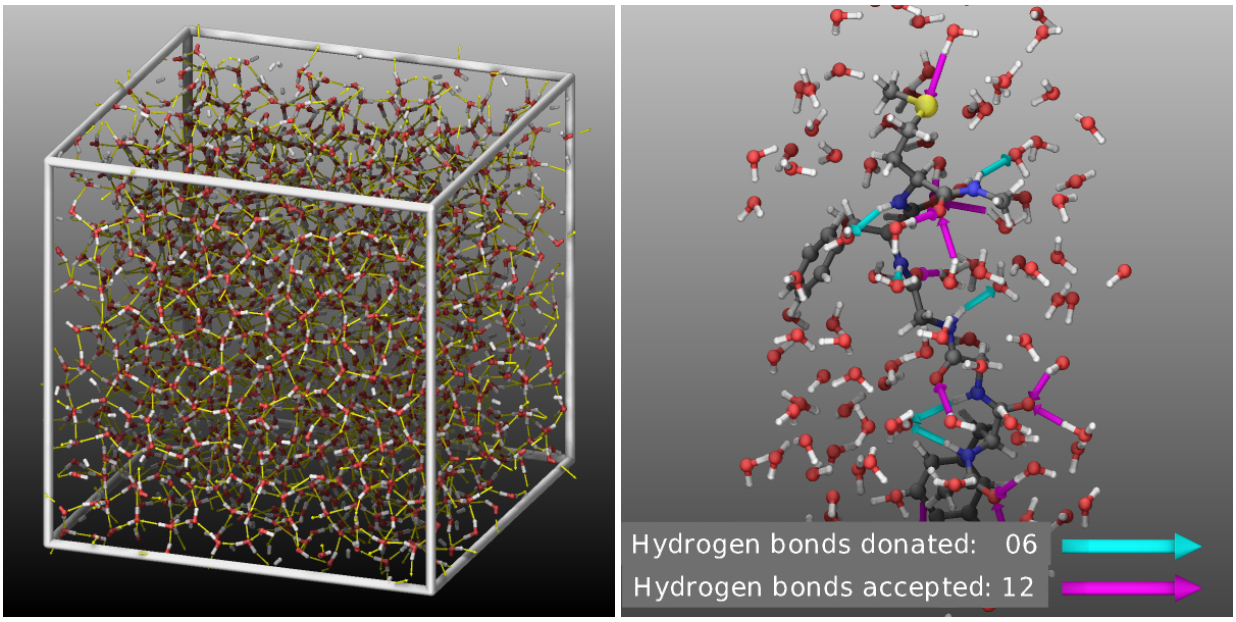
For a simple bulk water system, one could just show the entire hydrogen bond network and consider that the water oxygen atoms function as both, hydrogen bond donors and hydrogen bond acceptors. Here is an example input segment that could be added to the `examples/rdf-adf/in.spce` input file:

```
group       ogroup type 1
group       hgroup type 2
compute     hb     all hbond/local 3.5 30.0 ogroup ogroup hgroup

dump        viz    all image 100 water-*.png element type size 600 600 zoom 1.331 view␣
↪70 20 &
                                            shiny 0.2 ssao yes 348276 0.6 fsaa yes  box␣
↪yes 0.025 &
                                            bond atom 0.35 compute hb const -0.5 0.15
dump_modify viz    pad 6  boxcolor cadetblue backcolor darkgray backcolor2 silver␣
↪element O H &
                         adiam 1 0.5 adiam 2 0.4 ccolor hb yellow
```

Note how the *cflag1* parameter is used to shrink the arrows so that their tips just touch the hydrogen bond acceptor atoms.

If one wants to distinguish between donated and accepted hydrogen bonds for a subsystem, the input could become more complex and multiple computes may be needed. Below is an input that could be added to `examples/peptide/in.peptide` input file. This selects a group of atoms centered on the peptide and containing the peptide and a shell of water molecules and uses different groups for hydrogen bond donors and acceptors to determine selectively hydrogen bonds between the peptide and the surrounding water molecules in both directions.

```
# select atoms for visualization: peptide and water molecules within 3.5 angstrom
group     viz dynamic peptide within 3.5 include molecule every 100

# define groups of donor, acceptor, and hydrogen atoms for peptide and water
group           pdonor    type  5  9        # peptide donors : nitrogens and phenol␣
↪oxygen
group           woxygen   type 13           # water oxygens are donor and acceptor
group           pacceptor type  3  5  9 12  # peptide acceptors: oxygens, nitrogens, and␣
↪sulfur
group           hydrogen  type  4 10 14     # hydrogens bonded to oxygens and nitrogens

# peptide-water hydrogen bonds where the peptide is the donor
compute hb1 all hbond/local 3.5 30.0 pdonor woxygen hydrogen
# peptide-water hydrogen bonds where the peptide is the acceptor
compute hb2 all hbond/local 3.7 30.0 woxygen pacceptor hydrogen

# create donor/acceptor hydrogen bond info text
fix label all graphics/labels 100 text "Hydrogen bonds donated:   $(c_hb1:%02.0f)" 207␣
↪72 0.0 &
                                      size 24 backcolor darkgray &
                                   text "Hydrogen bonds accepted: $(c_hb2:%02.0f)" 210 30␣
↪0.0 &
                                      size 24 backcolor darkgray
# create colored arrows to go with text labels
fix obj   all graphics/objects 100 arrow 5 80.0 61.0 39 80.0 67.0 39 0.3 0.2 &
                                   arrow 6 80.0 61.0 37.2 80.0 67.0 37.2 0.3 0.2

# combine the graphics into visualization using only a subset of atom
dump viz viz image 100 hbonds-*.png element element size 600 600 zoom 2.1 view 80 0␣
↪center s 0.5 0.52 0.52 &
                       bond atom 0.3 fsaa yes ssao yes 12384 0.6 shiny 0.1 box no 0.1 &
                       compute hb1 const -0.4 0.3 compute hb2 const -0.4 0.3 fix label␣
↪const 1 0 fix obj type 0.0 0.0
dump_modify viz pad 5 boxcolor white backcolor darkgray backcolor2 silver &
                      element C C O H N C C C O H H S O H ccolor hb1 cyan ccolor hb2␣
↪magenta
```




### Fix reaxff/bonds

Fix *reaxff/bonds* of the *REAXFF package* provides access to the list of bonds as they are dynamically computed by the *ReaxFF pair style*. As discussed above, this can be used to visualize bonds for a system where there is no explicit bond topology defined.

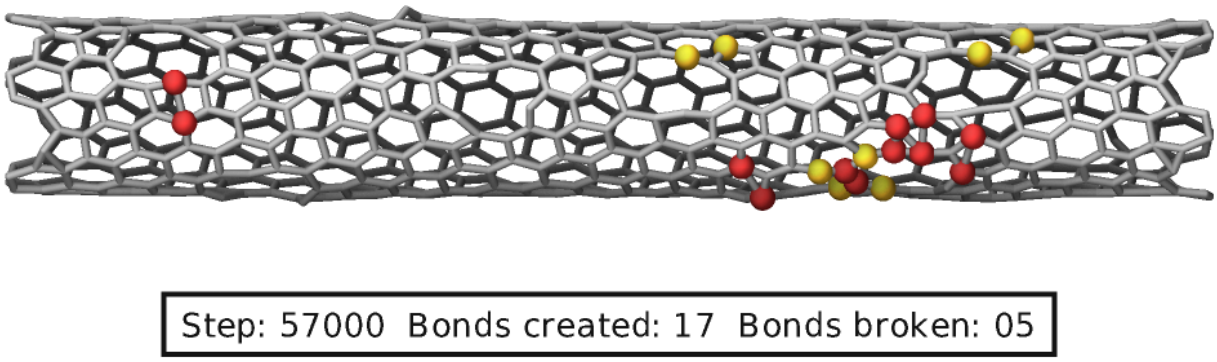
### Fix smd/wall_surface

Fix *smd/wall_surface* of the *MACHDYN package* creates a custom wall from a mesh of triangles that is read from an STL format file.

### MC package fixes

Several fixes from the *MC package* have support for adding graphics to a visualization. These are typically added spheres of the atoms that were swapped or involved in a bond that was created or broken. Below is an example for input commands that use both, *fix bond/break* and *fix bond/create/angle*. Atoms involved in a created bond are highlighted in red while atoms involved in a broken bond in yellow.

```
fix break all bond/break 500 1 2.5
fix form all bond/create/angle 500 1 1 2.2 1 aconstrain 90.0 180
variable nsteps index 500
fix label all graphics/labels ${nsteps}  &
      text "Step: $(step:%05.0f)  Bonds created: $(f_form[2]:%02.0f)  Bonds broken: $(f_
↪break[2]:%02.0f)" 500 32 0 &
      fontcolor black framecolor black

dump viz all image ${nsteps} breakable-*.png type type size 1000 400 zoom 8 shiny 0.1␣
↪fsaa yes &
            bond atom 0.5 view 160 90 box no 0.0 ssao yes 238174 0.6 &
            fix break const 0 1.5 fix form const 0 1.5 fix label const 1 0
dump_modify viz pad 6 backcolor white element C acolor 1 gray adiam 1 0.5 &
            fcolor break goldenrod fcolor form firebrick
```




### 10.1.5 Run multiple simulations from one input script

This can be done in several ways. See the documentation for individual commands for more details on how these examples work.

If "multiple simulations" means to continue a previous simulation for more timesteps, then you simply use the *run* command multiple times. For example, this script

```
units lj
atom_style atomic
read_data data.lj
run 10000
run 10000
run 10000
run 10000
run 10000
```

would run 5 successive simulations of the same system for a total of 50,000 timesteps.

If you wish to run totally different simulations, one after the other, the *clear* command can be used in between them to re-initialize LAMMPS. For example, this script

```
units lj
atom_style atomic
read_data data.lj
run 10000
clear
units lj
atom_style atomic
read_data data.lj.new
run 10000
```

would run 2 independent simulations, one after the other.

For large numbers of independent simulations, you can use *variables* and the *next* and *jump* commands to loop over the same input script multiple times with different settings. For example, this script, named `in.polymer`

```
variable d index run1 run2 run3 run4 run5 run6 run7 run8
shell cd $d
read_data data.polymer
run 10000
shell cd ..
clear
```

(continues on next page)

(continued from previous page)

```
next d
jump in.polymer
```

would run 8 simulations in different directories, using a `data.polymer` file in each directory. The same concept could be used to run the same system at 8 different temperatures, using a temperature variable and storing the output in different log and dump files, for example

```
variable a loop 8
variable t index 0.8 0.85 0.9 0.95 1.0 1.05 1.1 1.15
log log.$a
read data.polymer
velocity all create $t 352839
fix 1 all nvt $t $t 100.0
dump 1 all atom 1000 dump.$a
run 100000
clear
next t
next a
jump in.polymer
```

All of the above examples work whether you are running on 1 or multiple processors, but assumed you are running LAMMPS on a single partition of processors. LAMMPS can be run on multiple partitions via the *-partition command-line switch*.

In the last 2 examples, if LAMMPS were run on 3 partitions, the same scripts could be used if the `index` and `loop` variables were replaced with *universe*-style variables, as described in the *variable* command. Also, the **`next t`** and **`next a`** commands would need to be replaced with a single **`next a t`** command. With these modifications, the 8 simulations of each script would run on the 3 partitions one after the other until all were finished. Initially, 3 simulations would be started simultaneously, one on each partition. When one finished, that partition would then start the fourth simulation, and so forth, until all 8 were completed.

### 10.1.6 Multi-replica simulations

Several commands in LAMMPS run multi-replica simulations, meaning that multiple instances (replicas) of your simulation are run simultaneously, with small amounts of data exchanged between replicas periodically.

These are the relevant commands:

- *hyper* for bond boost hyperdynamics (HD)
- *neb* for nudged elastic band calculations (NEB)
- *neb_spin* for magnetic nudged elastic band calculations
- *prd* for parallel replica dynamics (PRD)
- *tad* for temperature accelerated dynamics (TAD)
- *temper* for parallel tempering with fixed volume
- *temper/npt* for parallel tempering extended for NPT
- *temper/grem* for parallel tempering with generalized replica exchange (gREM)
- *fix pimd* for path-integral molecular dynamics (PIMD)

NEB is a method for finding transition states and barrier potential energies. HD, PRD, and TAD are methods for performing accelerated dynamics to find and perform infrequent events. Parallel tempering or replica exchange runs

different replicas at a series of temperature to facilitate rare-event sampling. PIMD runs different replicas whose individual particles in different replicas are coupled together by springs to model a system of ring-polymers which can represent the quantum nature of atom cores.

These commands can only be used if LAMMPS was built with the REPLICA package. See the *Build package* page for more info.

In all these cases, you must run with one or more processors per replica. The processors assigned to each replica are determined at run-time by using the *-partition command-line switch* to launch LAMMPS on multiple partitions, which in this context are the same as replicas. E.g. these commands:

```
mpirun -np 16 lmp_linux -partition 8x2 -in in.temper
mpirun -np 8 lmp_linux -partition 8x1 -in in.neb
```

would each run 8 replicas, on either 16 or 8 processors. Note the use of the *-in command-line switch* to specify the input script which is required when running in multi-replica mode.

Also note that with MPI installed on a machine (e.g. your desktop), you can run on more (virtual) processors than you have physical processors. Thus, the above commands could be run on a single-processor (or few-processor) desktop so that you can run a multi-replica simulation on more replicas than you have physical processors. This is useful for testing and debugging, since with most modern processors and MPI libraries, the efficiency of a calculation can severely diminish when oversubscribing processors.

### 10.1.7 Library interface to LAMMPS

As described on the *Build basics* doc page, LAMMPS can be built as a static or shared library, so that it can be called by another code, used in a *coupled manner* with other codes, or driven through a *Python interface*.

At the core of LAMMPS is the `LAMMPS` class, which encapsulates the state of the simulation program through the state of the various class instances that it is composed of. So a calculation using LAMMPS requires creating an instance of the `LAMMPS` class and then send it (text) commands, either individually or from a file, or perform other operations that modify the state stored inside that instance or drive simulations. This is essentially what the `src/main.cpp` file does as well for the standalone LAMMPS executable, reading commands either from an input file or the standard input.

Creating a LAMMPS instance can be done by using C++ code directly or through a C-style interface library to LAMMPS that is provided in the files `src/library.cpp` and `src/library.h`. This *C language API*, can be used from C and C++, and is also the basis for the *Python* and *Fortran* interfaces or the *SWIG based wrappers* included in the LAMMPS source code.

The `examples/COUPLE` and `python/examples` directories contain some example programs written in C++, C, Fortran, and Python, which show how a driver code can link to LAMMPS as a library, run LAMMPS on a subset of processors (so the others are available to run some other code concurrently), grab data from LAMMPS, change it, and send it back into LAMMPS.

A detailed documentation of the available APIs and examples of how to use them can be found in the *Programmer Guide* section of this manual.

### 10.1.8 Coupling LAMMPS to other codes

LAMMPS is designed to support being coupled to other codes. For example, a quantum mechanics code might compute forces on a subset of atoms and pass those forces to LAMMPS. Or a continuum finite element (FE) simulation might use atom positions as boundary conditions on FE nodal points, compute a FE solution, and return interpolated forces on MD atoms.

LAMMPS can be coupled to other codes in at least 4 different ways. Each has advantages and disadvantages, which you will have to think about in the context of your application.

1. Define a new *fix* or *compute* command that calls the other code. In this scenario, LAMMPS is the driver code. During timestepping, the fix or compute is invoked, and can make library calls to the other code, which has been

linked to LAMMPS as a library. This is the way the *VORONOI* package, which computes Voronoi tesselations using the Voro++ library, is interfaced to LAMMPS. See the *compute voronoi* command for more details. Also see the *Modify* pages for information on how to add a new fix or compute to LAMMPS.

2. Define a new LAMMPS command that calls the other code. This is conceptually similar to method (1), but in this case LAMMPS and the other code are on a more equal footing. Note that now the other code is not called during the timestepping of a LAMMPS run, but between runs. The LAMMPS input script can be used to alternate LAMMPS runs with calls to the other code, invoked via the new command. The *run* command facilitates this with its *every* option, which makes it easy to run a few steps, invoke the command, run a few steps, invoke the command, etc.

   In this scenario, the other code can be called as a library, as in 1., or it could be a stand-alone code, invoked by a `system()` call made by the command (assuming your parallel machine allows one or more processors to start up another program). In the latter case the stand-alone code could communicate with LAMMPS through files that the command writes and reads.

   See the *Modify command* page for information on how to add a new command to LAMMPS.

3. Use LAMMPS as a library called by another code. In this case, the other code is the driver and calls LAMMPS as needed. Alternately, a wrapper code could link and call both LAMMPS and another code as libraries. Again, the *run* command has options that allow it to be invoked with minimal overhead (no setup or clean-up) if you wish to do multiple short runs, driven by another program. Details about using the library interface are given in the *library API* documentation.

4. Couple LAMMPS with another code in a client/server fashion, using the MDI Library developed by the Molecular Sciences Software Institute (MolSSI) to run LAMMPS as either an MDI driver (client) or an MDI engine (server). The MDI driver issues commands to the MDI server to exchange data between them. See the *Using LAMMPS with the MDI library for code coupling* page for more information about how LAMMPS can operate in either of these modes.

### 10.1.9 Using LAMMPS with the MDI library for code coupling

Client/server coupling of two (or more) codes is where one code is the "client" and sends request messages (data) to one (or more) "server" code(s). A server responds to each request with a reply message (data). This enables two (or more) codes to work in tandem to perform a simulation. In this context, LAMMPS can act as either a client or server code. It does this by using the MolSSI Driver Interface (MDI) library, developed by the Molecular Sciences Software Institute (MolSSI), which is supported by the *MDI* package.

Alternate methods for coupling codes with LAMMPS are described on the *Coupling LAMMPS to other codes* page.

Some advantages of client/server coupling are that the codes can run as stand-alone executables; they need not be linked together. Thus, neither code needs to have a library interface. This also makes it easy to run the two codes on different numbers of processors. If a message protocol (format and content) is defined for a particular kind of simulation, then in principle any code which implements the client-side protocol can be used in tandem with any code which implements the server-side protocol. Neither code needs to know what specific other code it is working with.

In MDI nomenclature, a client code is the "driver", and a server code is an "engine". One driver code can communicate with one or more instances of one or more engine codes. Driver and engine codes can be written in any language: C, C++, Fortran, Python, etc.

In addition to allowing driver and engine(s) to run as stand-alone executables, MDI also enables an engine to be a *plugin* to the client code. In this scenario, server code(s) are compiled as shared libraries, and one (or more) instances of the server are instantiated by the driver code. If the driver code runs in parallel, it can split its MPI communicator into multiple sub-communicators, and launch each plugin engine instance on a sub-communicator. Driver processors within that sub-communicator exchange messages with the corresponding engine instance, and can also send MPI messages to other processors in the driver. The driver code can also destroy engine instances and re-instantiate them. LAMMPS can operate as either a stand-alone or plugin MDI engine. When it operates as a driver, it can use either stand-alone or plugin MDI engines.

The way in which an MDI driver communicates with an MDI engine is by making MDI_Send() and MDI_Recv() calls, which are conceptually similar to MPI_Send() and MPI_Recv() calls. Each send or receive operation uses a string to identify the command name, and optionally some data, which can be a single value or vector of values of any data type. Inside the MDI library, data is exchanged between the driver and engine via MPI calls or sockets. This is a run-time choice by the user.

---

The *MDI* package provides a *mdi engine* command, which enables LAMMPS to operate as an MDI engine. Its doc page explains the variety of standard and custom MDI commands which the LAMMPS engine recognizes and can respond to.

The package also provides a *mdi plugin* command, which enables LAMMPS to operate as an MDI driver and load an MDI engine as a plugin library.

The package furthermore includes a *fix mdi/qm* command, in which LAMMPS operates as an MDI driver in conjunction with a quantum mechanics code as an MDI engine. The post_force() method of the `fix_mdi_qm.cpp` file shows how a driver issues MDI commands to another code. This command can be used to couple to an MDI engine, which is either a stand-alone code or a plugin library.

As explained in the *fix mdi/qm* command documentation, it can be used to perform *ab initio* MD simulations or energy minimizations, or to evaluate the quantum energy and forces for a series of independent systems. The `examples/mdi` directory has example input scripts for all of these use cases.

The package also has a *fix mdi/qmmm* command in which LAMMPS operates as an MDI driver in conjunction with a quantum mechanics code as an MDI engine to perform QM/MM simulations. The LAMMPS input script partitions the system into QM and MM (molecular mechanics) atoms. As described below the `examples/QUANTUM` directory has examples for coupling to 3 different quantum codes in this manner.

---

The `examples/mdi` directory contains Python scripts and LAMMPS input script which use LAMMPS as either an MDI driver or engine, or both. Currently, 5 example use cases are provided:

- Run ab initio MD (AIMD) using 2 instances of LAMMPS. As a driver, LAMMPS performs the timestepping in either NVE or NPT mode. As an engine, LAMMPS computes forces and is a surrogate for a quantum code.
- LAMMPS runs an MD simulation as a driver. Every N steps it passes the current snapshot to an MDI engine to evaluate the energy, virial, and peratom forces. As the engine, LAMMPS is a surrogate for a quantum code.
- LAMMPS loops over a series of data files and passes the configuration to an MDI engine to evaluate the energy, virial, and peratom forces and thus acts as a simulation driver. As the engine, LAMMPS is used as a surrogate for a quantum code.
- A Python script driver invokes a sequence of unrelated LAMMPS calculations. Calculations can be single-point energy/force evaluations, MD runs, or energy minimizations.
- Run AIMD with a Python driver code and 2 LAMMPS instances as engines. The first LAMMPS instance performs MD timestepping. The second LAMMPS instance acts as a surrogate QM code to compute forces.

**Note**

In any of these examples where LAMMPS is used as an engine, an actual QM code (provided it has support for MDI) could be used in its place, without modifying the input scripts or launch commands, except to specify the name of the QM code.

The `examples/mdi/Run.sh` file illustrates how to launch both driver and engine codes so that they communicate using the MDI library via either MPI or sockets, or using the engine as a stand-alone code, or as a plugin library.

---

As of March 2023, these are quantum codes with MDI support provided via Python wrapper scripts included in the LAMMPS distribution. These can be used with the fix mdi/qm and fix mdi/qmmm commands to perform QM calculations of an entire system (e.g. AIMD) or QM/MM simulations. See the `examples/QUANTUM` sub-directories for more details:

- LATTE - AIMD only
- PySCF - QM/MM only
- NWChem - AIMD or QM/MM

There are also at least two quantum codes which have direct MDI support, Quantum ESPRESSO (QE) and INQ. There are also several QM codes which have indirect support through QCEngine or i-PI. The former means they require a wrapper program (QCEngine) with MDI support which writes/read files to pass data to the quantum code itself. The list of QCEngine-supported and i-PI-supported quantum codes is on the MDI webpage.

These direct- and indirect-support codes should be usable for full system calculations (e.g. AIMD). Whether they support QM/MM models depends on the individual QM code.

## 10.2 Settings howto

### 10.2.1 2d simulations

You must use the *dimension* command to specify a 2d simulation. The default is 3d.

A 2d simulation box must be periodic in z as set by the *boundary* command. This is the default.

Simulation boxes in LAMMPS can be either orthogonal or triclinic in shape. Orthogonal boxes in 2d are a rectangle with 4 edges that are each perpendicular to either the x or y coordinate axes. Triclinic boxes in 2d are a parallelogram with opposite pairs of faces parallel to each other. LAMMPS supports two forms of triclinic boxes, restricted and general, which for 2d differ in how the box is oriented with respect to the xy coordinate axes. See the *Howto triclinic* for a detailed description of all 3 kinds of simulation boxes.

Here are examples of using the *create_box* command to define the simulation box for a 2d system.

```
# 2d orthogonal box using a block-style region
region mybox block -10 10 0 10 -0.5 0.5
create_box 1 mybox

# 2d restricted triclinic box using a prism-style region with only xy tilt
region mybox prism 0 10 0 10 -0.5 0.5 2.0 0.0 0.0
create_box 1 mybox

# 2d general triclinic box using a primitive cell for a 2d hex lattice
lattice       custom 1.0 a1 1.0 0.0 0.0 a2 0.5 0.86602540378 0.0 &
              a3 0.0 0.0 1.0 basis 0.0 0.0 0.0 triclinic/general
create_box    1 NULL 0 5 0 5 -0.5 0.5
```

Note that for 2d orthogonal or restricted triclinic boxes, the box has a 3rd dimension which must straddle z = 0.0 in the z dimension. Typically the width of box in the z dimension should be narrow, e.g. -0.5 to 0.5, but that is not required. For a 2d general triclinic box, the *a3* vector defined by the *lattice* command must be (0.0,0.0,1.0), which is its default value. Also the *clo* and *chi* arguments of the *create_box* command must be -0.5 and 0.5.

Here are examples of using the *read_data* command to define the simulation box for a 2d system via keywords in the header section of the data file. These are the same boxes as the examples for the *create_box* command

```
# 2d orthogonal box
-10 10   xlo xhi
```

(continues on next page)

(continued from previous page)

```
0 10     ylo yhi
-0.5 0.5 zlo zhi       # this is the default, so no need to specify

# 2d restricted triclinic box with only xy tilt
-10 10   xlo xhi
0 10     ylo yhi
-0.5 0.5 zlo zhi       # this is the default, so no need to specify
2.0 0.0 0.0 xy xz yz

# 3d general triclinic box using a primitive cell for a 2d hex lattice
5 0 0              avec
2.5 4.3301270189 0 bvec
0 0 1              cvec          # this is the default, so no need to specify
0 0 -0.5           abc origin    # this is the default for 2d, so no need to specify
```

Note that for 2d orthogonal or restricted triclinic boxes, the box has a 3rd dimension specified by the *zlo zhi* values, which must straddle z = 0.0. Typically the width of box in the z dimension should be narrow, e.g. -0.5 to 0.5, but that is not required. For a 2d general triclinic box, the z component of *avec* and *bvec* must be zero, and *cvec* must be (0,0,1), which is the default. The z component of *abc origin* must also be -0.5, which is the default.

If using the *create_atoms* command to create atoms in the 2d simulation box, all the z coordinates of created atoms will be zero.

If using the *read_data* command to read in a data file of atom coordinates for a 2d system, the z coordinates of all atoms should be zero. A value within epsilon of zero is also allowed in case the data file was generated by another program with finite numeric precision, in which case the z coord for the atom will be set to zero.

Use the *fix enforce2d* command as the last fix defined in the input script. It ensures that the z-components of velocities and forces are zeroed out every timestep. The reason to make it the last fix is so that any forces added by other fixes will also be zeroed out.

Many of the example input scripts included in the examples directory are for 2d models.

**Note**

Some models in LAMMPS treat particles as finite-size spheres, as opposed to point particles. See the *atom_style sphere* and *fix nve/sphere* commands for details. By default, for 2d simulations, such particles will still be modeled as 3d spheres, not 2d discs (circles), meaning their moment of inertia will be that of a sphere. If you wish to model them as 2d discs, see the *set density/disc* command and the *disc* option for the *fix nve/sphere*, *fix nvt/sphere*, *fix nph/sphere*, *fix npt/sphere* commands.

### 10.2.2 Type labels

Added in version 15Sep2022.

Each atom in LAMMPS has an associated numeric atom type. Similarly, each bond, angle, dihedral, and improper is assigned a bond type, angle type, and so on. The primary use of these types is to map potential (force field) parameters to the interactions of the atom, bond, angle, dihedral, and improper.

By default, type values are entered as integers from 1 to Ntypes wherever they appear in LAMMPS input or output files. The total number Ntypes for each interaction is "locked in" when the simulation box is created.

A recent addition to LAMMPS is the option to use strings - referred to as type labels - as an alternative. Using type labels instead of numeric types can be advantageous in various scenarios. For example, type labels can make inputs more readable and generic (i.e. usable through the *include command* for different systems with different numerical values

assigned to types. This generality also applies to other inputs like data files read by *read_data* or molecule template files read by the *molecule* command. A discussion of the current type label support can be found in *(Gissinger)*. See below for a list of other commands that can use type labels in different ways.

LAMMPS will *internally* continue to use numeric types, which means that many previous restrictions still apply. For example, the total number of types is locked in when creating the simulation box, and potential parameters for each type must be provided even if not used by any interactions.

A collection of type labels for all type-kinds (atom types, bond types, etc.) is stored as a "label map" which is simply a list of numeric types and their associated type labels. Within a type-kind, each type label must be unique. It can be assigned to only one numeric type. To read and write type labels to data files for a given type-kind, *all* associated numeric types need have a type label assigned. Partial maps can be saved with the *labelmap write* command and read back with the *include* command.

Valid type labels can contain most ASCII characters, but cannot start with a number, a '#', or a '*'. Also, labels must not contain whitespace characters. When using the *labelmap command* in the LAMMPS input, if certain characters appear in the type label, such as the single (') or double (") quote or the '#' character, the label must be put in either double, single, or triple ("""") quotes. Triple quotes allow for the most generic type label strings, but they require to have a leading and trailing blank space. When defining type labels the blanks will be ignored. Example:

```
labelmap angle 1 """ C1'-C2"-C3# """
```

This command will map the string `` `C1'-C2"-C3#` `` to the angle type 1.

There are two ways to define label maps. One is via the *labelmap* command. The other is via the *read_data* command. A data file can have sections such as *Atom Type Labels*, *Bond Type Labels*, etc., which assign type labels to numeric types. The label map can be written out to data files by the *write_data* command. This map is also written to and read from restart files, by the *write_restart* and *read_restart* commands.

---

### Use of type labels in LAMMPS input or output

Many LAMMPS input script commands that take a numeric type as an argument can use the associated type label instead. If a type label is not defined for a particular numeric type, only its numeric type can be used.

This example assigns labels to the atom types, and then uses the type labels to redefine the pair coefficients.

```
pair_coeff 1 2 1.0 1.0              # numeric types
labelmap atom 1 C 2 H
pair_coeff C H 1.0 1.0              # type labels
```

Adding support for type labels to various commands is an ongoing project. If an input script command (or a section in a file read by a command) allows substituting a type label for a numeric type argument, it will be explicitly mentioned in that command's documentation page.

As a temporary measure, input script commands can take advantage of variables and how they can be expanded during processing of the input. The variables can use functions that will translate type label strings to their respective number as defined in the current label map. See the *variable* command for details.

For example, here is how the pair_coeff command could be used with type labels if it did not yet support them, either with an explicit variable command or an implicit variable used in the pair_coeff command.

```
labelmap atom 1 C 2 H
variable atom1 equal label2type(atom,C)
variable atom2 equal label2type(atom,H)
pair_coeff ${atom1} ${atom2} 1.0 1.0
```

```
labelmap atom 1 C 2 H
pair_coeff $(label2type(atom,C)) $(label2type(atom,H)) 80.0 1.2
```

#### Commands that can use label types

Any workflow that involves reading multiple data files, molecule templates or a combination of the two can be streamlined by using type labels instead of numeric types, because types are automatically synced between the files. The creation of simulation-ready reaction templates for *fix bond/react* is much simpler when using type labels, and results in templates that can be used without modification in multiple simulations or different systems.

**(Gissinger)** J. R. Gissinger, I. Nikiforov, Y. Afshar, B. Waters, M. Choi, D. S. Karls, A. Stukowski, W. Im, H. Heinz, A. Kohlmeyer, and E. B. Tadmor, J Phys Chem B, 128, 3282-3297 (2024).

### 10.2.3 Triclinic (non-orthogonal) simulation boxes

By default, LAMMPS uses an orthogonal simulation box to encompass the particles. The orthogonal box has its "origin" at (xlo,ylo,zlo) and extends to (xhi,yhi,zhi). Conceptually it is defined by 3 edge vectors starting from the origin given by **A** = (xhi-xlo,0,0); **B** = (0,yhi-ylo,0); **C** = (0,0,zhi-zlo). The *boundary* command sets the boundary conditions for the 6 faces of the box (periodic, non-periodic, etc). The 6 parameters (xlo,xhi,ylo,yhi,zlo,zhi) are defined at the time the simulation box is created by one of these commands:

- *create_box*
- *read_data*
- *read_restart*
- *read_dump*

Internally, LAMMPS defines box size parameters lx,ly,lz where lx = xhi-xlo, and similarly in the y and z dimensions. The 6 parameters, as well as lx,ly,lz, can be output via the *thermo_style custom* command. See the *Howto 2d* doc page for info on how zlo and zhi are defined for 2d simulations.

#### Triclinic simulation boxes

LAMMPS also allows simulations to be performed using triclinic (non-orthogonal) simulation boxes shaped as a 3d parallelepiped with triclinic symmetry. For 2d simulations a triclinic simulation box is effectively a parallelogram; see the *Howto 2d* doc page for details.

One use of triclinic simulation boxes is to model solid-state crystals with triclinic symmetry. The *lattice* command can be used with non-orthogonal basis vectors to define a lattice that will tile a triclinic simulation box via the *create_atoms* command.

A second use is to run Parrinello-Rahman dynamics via the *fix npt* command, which will adjust the xy, xz, yz tilt factors to compensate for off-diagonal components of the pressure tensor. The analog for an *energy minimization* is the *fix box/relax* command.

A third use is to shear a bulk solid to study the response of the material. The *fix deform* command can be used for this purpose. It allows dynamic control of the xy, xz, yz tilt factors as a simulation runs. This is discussed in the *Howto NEMD* doc page on non-equilibrium MD (NEMD) simulations.

Conceptually, a triclinic parallelepiped is defined with an "origin" at (xlo,ylo,zhi) and 3 edge vectors **A** = (ax,ay,az), **B** = (bx,by,bz), **C** = (cx,cy,cz) which can be arbitrary vectors, so long as they are non-zero, distinct, and not co-planar. In addition, they must define a right-handed system, such that (**A** cross **B**) points in the direction of **C**. Note that a

left-handed system can be converted to a right-handed system by simply swapping the order of any pair of the **A**, **B**, **C** vectors.

The 4 commands listed above for defining orthogonal simulation boxes have triclinic options which allow for specification of the origin and edge vectors **A**, **B**, **C**. For each command, this can be done in one of two ways, for what LAMMPS calls a *general* triclinic box or a *restricted* triclinic box.

A *general* triclinic box is specified by an origin (xlo, ylo, zlo) and arbitrary edge vectors **A** = (ax,ay,az), **B** = (bx,by,bz), and **C** = (cx,cy,cz). So there are 12 parameters in total.

A *restricted* triclinic box also has an origin (xlo,ylo,zlo), but its edge vectors are of the following restricted form: **A** = (xhi-xlo,0,0), **B** = (xy,yhi-ylo,0), **C** = (xz,yz,zhi-zlo). So there are 9 parameters in total. Note that the restricted form requires **A** to be along the x-axis, **B** to be in the xy plane with a y-component in the +y direction, and **C** to have its z-component in the +z direction. Note that a restricted triclinic box is *right-handed* by construction since (**A** cross **B**) points in the direction of **C**.

The *xy,xz,yz* values can be zero or positive or negative. They are called "tilt factors" because they are the amount of displacement applied to edges of faces of an orthogonal box to change it into a restricted triclinic parallelepiped.

**Note**

Any right-handed general triclinic box (i.e. solid-state crystal basis vectors) can be rotated in 3d around its origin in order to conform to the LAMMPS definition of a restricted triclinic box. See the discussion in the next sub-section about general triclinic simulation boxes in LAMMPS.

Note that the *thermo_style custom* command has keywords for outputting the various parameters that define the size and shape of orthogonal, restricted triclinic, and general triclinic simulation boxes.

For orthogonal boxes there are 6 thermo keywords (xlo,ylo,zlo) and (xhi,yhi,zhi).

For restricted triclinic boxes there are 9 thermo keywords for (xlo,ylo,zlo), (xhi,yhi,zhi), and the (xy,xz,yz) tilt factors.

For general triclinic boxes there are 12 thermo keywords for (xlo,ylo,zhi) and the components of the **A**, **B**, **C** edge vectors, namely (avecx,avecy,avecz), (bvecx,bvecy,bvecz), and (cvecx,cvecy,cvecz),

The remainder of this doc page explains (a) how LAMMPS operates with general triclinic simulation boxes, (b) mathematical transformations between general and restricted triclinic boxes which may be useful when creating LAMMPS inputs or interpreting outputs for triclinic simulations, and (c) how LAMMPS uses tilt factors for restricted triclinic simulation boxes.

---

### General triclinic simulation boxes in LAMMPS

LAMMPS allows specification of general triclinic simulation boxes with their atoms as a convenience for users who may be converting data from solid-state crystallographic representations or from DFT codes for input to LAMMPS. Likewise it allows output of dump files, data files, and thermodynamic data (e.g. pressure tensor) in a general triclinic format.

However internally, LAMMPS only uses restricted triclinic simulation boxes. This is for parallel efficiency and to formulate partitioning of the simulation box across processors, neighbor list building, and inter-processor communication of per-atom data with methods similar to those used for orthogonal boxes.

This means 4 things which are important to understand:

- Input of a general triclinic system is immediately converted to a restricted triclinic system.
- If output of per-atom data for a general triclinic system is requested (e.g. for atom coordinates in a dump file), conversion from a restricted to general triclinic system is done at the time of output.

- The conversion of the simulation box and per-atom data from general triclinic to restricted triclinic (and vice versa) is a 3d rotation operation around an origin, which is the lower left corner of the simulation box. This means an input data file for a general triclinic system should specify all per-atom quantities consistent with the general triclinic box and its orientation relative to the standard x,y,z coordinate axes. For example, atom coordinates should be inside the general triclinic simulation box defined by the edge vectors **A**, **B**, **C** and its origin. Likewise per-atom velocities should be in directions consistent with the general triclinic box orientation. E.g. a velocity vector which will be in the +x direction once LAMMPS converts from a general to restricted triclinic box, should be specified in the data file in the direction of the **A** edge vector. See the *read_data* doc page for info on all the per-atom vector quantities to which this rule applies when a data file for a general triclinic box is input.
- If commands such as *write_data* or *dump custom* are used to output general triclinic information, it is effectively the inverse of the operation described in the preceding bullet.
- Other LAMMPS commands such as *region* or *velocity* or *set*, operate on a restricted triclinic system even if a general triclinic system was defined initially.

This is the list of commands which have general triclinic options:

- *create_box* - define a general triclinic box
- *create_atoms* - add atoms to a general triclinic box
- *lattice* - define a custom lattice consistent with the **A**, **B**, **C** edge vectors of a general triclinic box
- *read_data* - read a data file for a general triclinic system
- *write_data* - write a data file for a general triclinic system
- *dump atom, dump custom* - output dump snapshots in general triclinic format
- *dump_modify triclinic/general* - select general triclinic format for dump output
- *thermo_style* - output the pressure tensor in general triclinic format
- *thermo_modify triclinic/general* - select general triclinic format for thermo output
- *read_restart* - read a restart file for a general triclinic system
- *write_restart* - write a restart file for a general triclinic system

---

### Transformation from general to restricted triclinic boxes

Let **A**,**B**,**C** be the right-handed edge vectors of a general triclinic simulation box. The equivalent LAMMPS **a**,**b**,**c** for a restricted triclinic box are a 3d rotation of **A**, **B**, and **C** and can be computed as follows:

$$\begin{pmatrix}\mathbf{a} & \mathbf{b} & \mathbf{c}\end{pmatrix} = \begin{pmatrix} a_x & b_x & c_x \\ 0 & b_y & c_y \\ 0 & 0 & c_z \end{pmatrix}$$

$$a_x = A$$

$$b_x = \mathbf{B} \cdot \hat{\mathbf{A}} = B\cos\gamma$$

$$b_y = |\hat{\mathbf{A}} \times \mathbf{B}| = B\sin\gamma = \sqrt{B^2 - {b_x}^2}$$

$$c_x = \mathbf{C} \cdot \hat{\mathbf{A}} = C\cos\beta$$

$$c_y = \mathbf{C} \cdot (\widehat{\mathbf{A} \times \mathbf{B}}) \times \hat{\mathbf{A}} = \frac{\mathbf{B} \cdot \mathbf{C} - b_x c_x}{b_y}$$

$$c_z = |\mathbf{C} \cdot (\widehat{\mathbf{A} \times \mathbf{B}})| = \sqrt{C^2 - {c_x}^2 - {c_y}^2}$$

where A = | **A** | indicates the scalar length of **A**. The hat symbol (^) indicates the corresponding unit vector. $\beta$ and $\gamma$ are angles between the **A**, **B**, **C** vectors as described below.

For consistency, the same rotation applied to the triclinic box edge vectors can also be applied to atom positions, velocities, and other vector quantities. This can be conveniently achieved by first converting to fractional coordinates in the general triclinic coordinates and then converting to coordinates in the restricted triclinic basis. The transformation is given by the following equation:

$$\mathbf{x} = \begin{pmatrix} \mathbf{a} & \mathbf{b} & \mathbf{c} \end{pmatrix} \cdot \frac{1}{V} \begin{pmatrix} \mathbf{B} \times \mathbf{C} \\ \mathbf{C} \times \mathbf{A} \\ \mathbf{A} \times \mathbf{B} \end{pmatrix} \cdot \mathbf{X}$$

where $V$ is the volume of the box (same in either basis), **X** is the fractional vector in the general triclinic basis and **x** is the resulting vector in the restricted triclinic basis.

---

### Crystallographic general triclinic representation of a simulation box

General triclinic crystal structures are often defined using three lattice constants $a$, $b$, and $c$, and three angles $\alpha$, $\beta$, and $\gamma$. Note that in this nomenclature, the a, b, and c lattice constants are the scalar lengths of the edge vectors **a**, **b**, and **c** defined above. The relationship between these 6 quantities (a, b, c, $\alpha$, $\beta$, $\gamma$) and the LAMMPS restricted triclinic box sizes (lx,ly,lz) = (xhi-xlo,yhi-ylo,zhi-zlo) and tilt factors (xy,xz,yz) is as follows:

$$\begin{aligned} a &= \mathrm{lx} \\ b^2 &= \mathrm{ly}^2 + \mathrm{xy}^2 \\ c^2 &= \mathrm{lz}^2 + \mathrm{xz}^2 + \mathrm{yz}^2 \\ \cos\alpha &= \frac{\mathrm{xy} * \mathrm{xz} + \mathrm{ly} * \mathrm{yz}}{b * c} \\ \cos\beta &= \frac{\mathrm{xz}}{c} \\ \cos\gamma &= \frac{\mathrm{xy}}{b} \end{aligned}$$

The inverse relationship can be written as follows:

$$\begin{aligned} \mathrm{lx} &= a \\ \mathrm{xy} &= b\cos\gamma \\ \mathrm{xz} &= c\cos\beta \\ \mathrm{ly}^2 &= b^2 - \mathrm{xy}^2 \\ \mathrm{yz} &= \frac{b * c\cos\alpha - \mathrm{xy} * \mathrm{xz}}{\mathrm{ly}} \\ \mathrm{lz}^2 &= c^2 - \mathrm{xz}^2 - \mathrm{yz}^2 \end{aligned}$$

The values of $a$, $b$, $c$, $\alpha$ , $\beta$, and $\gamma$ can be printed out or accessed by computes using the *thermo_style custom* keywords *cella*, *cellb*, *cellc*, *cellalpha*, *cellbeta*, *cellgamma*, respectively.

---

### Output of restricted and general triclinic boxes in a dump file

As discussed on the *dump* command doc page, when the BOX BOUNDS for a snapshot is written to a dump file for a restricted triclinic box, an orthogonal bounding box which encloses the triclinic simulation box is output, along with the 3 tilt factors (xy, xz, yz) of the restricted triclinic box, formatted as follows:

```
ITEM: BOX BOUNDS xy xz yz
xlo_bound xhi_bound xy
ylo_bound yhi_bound xz
zlo_bound zhi_bound yz
```

This bounding box is convenient for many visualization programs and is calculated from the 9 restricted triclinic box parameters (xlo,xhi,ylo,yhi,zlo,zhi,xy,xz,yz) as follows:

```
xlo_bound = xlo + MIN(0.0,xy,xz,xy+xz)
xhi_bound = xhi + MAX(0.0,xy,xz,xy+xz)
ylo_bound = ylo + MIN(0.0,yz)
yhi_bound = yhi + MAX(0.0,yz)
zlo_bound = zlo
zhi_bound = zhi
```

These formulas can be inverted if you need to convert the bounding box back into the restricted triclinic box parameters, e.g. xlo = xlo_bound - MIN(0.0,xy,xz,xy+xz).

---

### Periodicity and tilt factors for triclinic simulation boxes

There is no requirement that a triclinic box be periodic in any dimension, though it typically should be in y or z if you wish to enforce a shift in coordinates due to periodic boundary conditions across the y or z boundaries. See the doc page for the *boundary* command for an explanation of shifted coordinates for restricted triclinic boxes which are periodic.

Some commands that work with triclinic boxes, e.g. the *fix deform* and *fix npt* commands, require periodicity or non-shrink-wrap boundary conditions in specific dimensions. See the command doc pages for details.

A restricted triclinic box can be defined with all 3 tilt factors = 0.0, so that it is initially orthogonal. This is necessary if the box will become non-orthogonal, e.g. due to use of the *fix npt* or *fix deform* commands. Alternatively, you can use the *change_box* command to convert a simulation box from orthogonal to restricted triclinic and vice versa.

**Note**

Highly tilted restricted triclinic simulation boxes can be computationally inefficient. This is due to the large volume of communication needed to acquire ghost atoms around a processor's irregular-shaped subdomain. For extreme values of tilt, LAMMPS may also lose atoms and generate an error.

LAMMPS will issue a warning if you define a restricted triclinic box with a tilt factor which skews the box more than half the distance of the parallel box length, which is the first dimension in the tilt factor (e.g. x for xz).

For example, if xlo = 2 and xhi = 12, then the x box length is 10 and the xy tilt factor should be between -5 and 5 to avoid the warning. Similarly, both xz and yz should be between -(xhi-xlo)/2 and +(yhi-ylo)/2. Note that these are not limitations, since if the maximum tilt factor is 5 (as in this example), then simulations boxes and atom configurations with tilt = ..., -15, -5, 5, 15, 25, ... are all geometrically equivalent.

If the box tilt exceeds this limit during a dynamics run (e.g. due to the *fix deform* command), then by default the box is "flipped" to an equivalent shape with a tilt factor within the warning bounds, and the run continues. See the *fix deform* page for further details. Box flips that would normally occur using the *fix deform* or *fix npt* commands can be suppressed using the *flip no* option with either of the commands.

One exception to box flipping is if the first dimension in the tilt factor (e.g. x for xy) is non-periodic. In that case, the limits on the tilt factor are not enforced, since flipping the box in that dimension would not change the atom positions

due to non-periodicity. In this mode, if the system tilts to large angles, the simulation will simply become inefficient, due to the highly skewed simulation box.

### 10.2.4 Thermostats

Thermostatting means controlling the temperature of particles in an MD simulation. *Barostatting* means controlling the pressure. Since the pressure includes a kinetic component due to particle velocities, both these operations require calculation of the temperature. Typically a target temperature (T) and/or pressure (P) is specified by the user, and the thermostat or barostat attempts to equilibrate the system to the requested T and/or P.

Thermostatting in LAMMPS is performed by *fixes*, or in one case by a pair style. Several thermostatting fixes are available: Nose-Hoover (nvt), Berendsen, CSVR, Langevin, and direct rescaling (temp/rescale). Dissipative particle dynamics (DPD) thermostatting can be invoked via the *dpd/tstat* pair style:

- *fix nvt*
- *fix nvt/sphere*
- *fix nvt/asphere*
- *fix nvt/sllod*
- *fix temp/berendsen*
- *fix temp/csvr*
- *fix ffl*
- *fix gjf*
- *fix gld*
- *fix gle*
- *fix langevin*
- *fix temp/rescale*
- *pair_style dpd/tstat*
- *pair_style dpd/ext/tstat*

*Fix nvt* only thermostats the translational velocity of particles. *Fix nvt/sllod* also does this, except that it subtracts out a velocity bias due to a deforming box and integrates the SLLOD equations of motion. See the *Howto nemd* page for further details. *Fix nvt/sphere* and *fix nvt/asphere* thermostat not only translation velocities but also rotational velocities for spherical and aspherical particles.

**Note**

A recent (2017) book by *(Daivis and Todd)* discusses use of the SLLOD method and non-equilibrium MD (NEMD) thermostatting generally, for both simple and complex fluids, e.g. molecular systems. The latter can be tricky to do correctly.

DPD thermostatting alters pairwise interactions in a manner analogous to the per-particle thermostatting of *fix langevin*.

Any of the thermostatting fixes can be instructed to use custom temperature computes that remove bias which has two effects: first, the current calculated temperature, which is compared to the requested target temperature, is calculated with the velocity bias removed; second, the thermostat adjusts only the thermal temperature component of the particle's velocities, which are the velocities with the bias removed. The removed bias is then added back to the adjusted velocities. See the doc pages for the individual fixes and for the *fix_modify* command for instructions on how to assign a temperature compute to a thermostatting fix.

For example, you can apply a thermostat only to atoms in a spatial region by using it in conjunction with *compute temp/region*. Or you can apply a thermostat to only the x and z components of velocity by using it with *compute temp/partial*. Of you could thermostat only the thermal temperature of a streaming flow of particles without affecting the streaming velocity, by using *compute temp/profile*.

Below is a list of custom temperature computes that can be used like that:

- *compute temp/asphere command*
- *compute temp/body command*
- *compute temp/chunk command*
- *compute temp/com command*
- *compute temp/deform command*
- *compute temp/partial command*
- *compute temp/profile command*
- *compute temp/ramp command*
- *compute temp/region command*
- *compute temp/rotate command*
- *compute temp/sphere command*

**Note**

Not all thermostat fixes perform time integration, meaning they update the velocities and positions of particles due to forces and velocities respectively. The other thermostat fixes only adjust velocities; they do NOT perform time integration updates. Thus, they should be used in conjunction with a constant NVE integration fix such as these:

- *fix nve*
- *fix nve/sphere*
- *fix nve/asphere*

Thermodynamic output, which can be setup via the *thermo_style* command, often includes temperature values. As explained on the page for the *thermo_style* command, the default temperature is setup by the thermo command itself. It is NOT the temperature associated with any thermostatting fix you have defined or with any compute you have defined that calculates a temperature. The doc pages for the thermostatting fixes explain the ID of the temperature compute they create. Thus if you want to view these temperatures, you need to specify them explicitly via the *thermo_style custom* command. Or you can use the *thermo_modify* command to re-define what temperature compute is used for default thermodynamic output.

---

**(Daivis and Todd)** Daivis and Todd, Nonequilibrium Molecular Dynamics (book), Cambridge University Press, https://doi.org/10.1017/9781139017848, (2017).

### 10.2.5 Barostats

Barostatting means controlling the pressure in an MD simulation. *Thermostatting* means controlling the temperature of the particles. Since the pressure includes a kinetic component due to particle velocities, both these operations require calculation of the temperature. Typically a target temperature (T) and/or pressure (P) is specified by the user, and the thermostat or barostat attempts to equilibrate the system to the requested T and/or P.

Barostatting in LAMMPS is performed by *fixes*. Three barostatting methods are currently available: Nose-Hoover (npt and nph), Berendsen, and various linear controllers in deform/pressure:

- *fix npt*
- *fix npt/sphere*
- *fix npt/asphere*
- *fix nph*
- *fix press/berendsen*
- *fix deform/pressure*

The *fix npt* commands include a Nose-Hoover thermostat and barostat. *Fix nph* is just a Nose/Hoover barostat; it does no thermostatting. The fixes *nph*, *press/berendsen*, and *deform/pressure* can be used in conjunction with any of the thermostatting fixes.

As with the *thermostats*, *fix npt* and *fix nph* only use translational motion of the particles in computing T and P and performing thermo/barostatting. *Fix npt/sphere* and *fix npt/asphere* thermo/barostat using not only translation velocities but also rotational velocities for spherical and aspherical particles.

All of the barostatting fixes use the *compute pressure* compute to calculate a current pressure. By default, this compute is created with a simple *compute temp* (see the last argument of the *compute pressure* command), which is used to calculated the kinetic component of the pressure. The barostatting fixes can also use temperature computes that remove bias for the purpose of computing the kinetic component which contributes to the current pressure. See the doc pages for the individual fixes and for the *fix_modify* command for instructions on how to assign a temperature or pressure compute to a barostatting fix.

**Note**

As with the thermostats, the Nose/Hoover methods (*fix npt* and *fix nph*) perform time integration. *Fix press/berendsen* and *fix deform/pressure* do NOT, so they should be used with one of the constant NVE fixes or with one of the NVT fixes.

Thermodynamic output, which can be setup via the *thermo_style* command, often includes pressure values. As explained on the page for the *thermo_style* command, the default pressure is setup by the thermo command itself. It is NOT the pressure associated with any barostatting fix you have defined or with any compute you have defined that calculates a pressure. The doc pages for the barostatting fixes explain the ID of the pressure compute they create. Thus if you want to view these pressures, you need to specify them explicitly via the *thermo_style custom* command. Or you can use the *thermo_modify* command to re-define what pressure compute is used for default thermodynamic output.

### 10.2.6 Walls

Walls in an MD simulation are typically used to bound particle motion, i.e. to serve as a boundary condition.

Walls in LAMMPS can be of rough (made of particles) or idealized surfaces. Ideal walls can be smooth, generating forces only in the normal direction, or frictional, generating forces also in the tangential direction.

Rough walls, built of particles, can be created in various ways. The particles themselves can be generated like any other particle, via the *lattice* and *create_atoms* commands, or read in via the *read_data* command.

Their motion can be constrained by many different commands, so that they do not move at all, move together as a group at constant velocity or in response to a net force acting on them, move in a prescribed fashion (e.g. rotate around a point), etc. Note that if a time integration fix like *fix nve* or *fix nvt* is not used with the group that contains wall particles, their positions and velocities will not be updated.

- *fix aveforce* - set force on particles to average value, so they move together
- *fix setforce* - set force on particles to a value, e.g. 0.0
- *fix freeze* - freeze particles for use as granular walls

- *fix nve/noforce* - advect particles by their velocity, but without force
- *fix move* - prescribe motion of particles by a linear velocity, oscillation, rotation, variable

The *fix move* command offers the most generality, since the motion of individual particles can be specified with *variable* formula which depends on time and/or the particle position.

For rough walls, it may be useful to turn off pairwise interactions between wall particles via the *neigh_modify exclude* command.

Rough walls can also be created by specifying frozen particles that do not move and do not interact with mobile particles, and then tethering other particles to the fixed particles, via a *bond*. The bonded particles do interact with other mobile particles.

Idealized walls can be specified via several fix commands. *Fix wall/gran* creates frictional walls for use with granular particles; all the other commands create smooth walls.

- *fix wall/reflect* - reflective flat walls
- *fix wall/lj93* - flat walls, with Lennard-Jones 9/3 potential
- *fix wall/lj126* - flat walls, with Lennard-Jones 12/6 potential
- *fix wall/colloid* - flat walls, with *pair_style colloid* potential
- *fix wall/harmonic* - flat walls, with repulsive harmonic spring potential
- *fix wall/morse* - flat walls, with Morse potential
- *fix wall/region* - use region surface as wall
- *fix wall/gran* - flat or curved walls with *pair_style granular* potential

The *lj93*, *lj126*, *colloid*, *harmonic*, and *morse* styles all allow the flat walls to move with a constant velocity, or oscillate in time. The *fix wall/region* command offers the most generality, since the region surface is treated as a wall, and the geometry of the region can be a simple primitive volume (e.g. a sphere, or cube, or plane), or a complex volume made from the union and intersection of primitive volumes. *Regions* can also specify a volume "interior" or "exterior" to the specified primitive shape or *union* or *intersection*. *Regions* can also be "dynamic" meaning they move with constant velocity, oscillate, or rotate.

The only frictional idealized walls currently in LAMMPS are flat or curved surfaces specified by the *fix wall/gran* command. At some point we plan to allow region surfaces to be used as frictional walls, as well as triangulated surfaces.

### 10.2.7 NEMD simulations

Non-equilibrium molecular dynamics or NEMD simulations are typically used to measure a fluid's rheological properties such as viscosity. In LAMMPS, such simulations can be performed by first setting up a non-orthogonal simulation box (see the preceding Howto section).

A shear strain can be applied to the simulation box at a desired strain rate by using the *fix deform* command. The *fix nvt/sllod* command can be used to thermostat the sheared fluid and integrate the SLLOD equations of motion for the system. Fix nvt/sllod uses *compute temp/deform* to compute a thermal temperature by subtracting out the streaming velocity of the shearing atoms. The velocity profile or other properties of the fluid can be monitored via the *fix ave/chunk* command.

**Note**

A recent (2017) book by *(Daivis and Todd)* discusses use of the SLLOD method and non-equilibrium MD (NEMD) thermostatting generally, for both simple and complex fluids, e.g. molecular systems. The latter can be tricky to do correctly.

As discussed in the previous section on non-orthogonal simulation boxes, the amount of tilt or skew that can be applied is limited by LAMMPS for computational efficiency to be 1/2 of the parallel box length. However, *fix deform* can continuously strain a box by an arbitrary amount. As discussed in the *fix deform* command, when the tilt value reaches a limit, the box is flipped to the opposite limit which is an equivalent tiling of periodic space. The strain rate can then continue to change as before. In a long NEMD simulation these box re-shaping events may occur many times.

In a NEMD simulation, the "remap" option of *fix deform* should be set to "remap v", since that is what *fix nvt/sllod* assumes to generate a velocity profile consistent with the applied shear strain rate. Alternatively, if the laboratory-frame velocity is not required (e.g. if only calculating viscosity), atom velocities can be stored in the peculiar frame using the "peculiar yes" flag of *fix nvt/sllod*, which will typically give better performance. In this case, the velocity reported by LAMMPS is the velocity relative to the flow, the "remap" option of *fix deform* should be set to "remap none", and *fix nvt/sllod* calculates temperature using *compute temp*.

An alternative method for calculating viscosities is provided via the *fix viscosity* command.

NEMD simulations can also be used to measure transport properties of a fluid through a pore or channel. Simulations of steady-state flow can be performed using the *fix flow/gauss* command.

---

**(Daivis and Todd)** Daivis and Todd, Nonequilibrium Molecular Dynamics (book), Cambridge University Press, https://doi.org/10.1017/9781139017848, (2017).

### 10.2.8 Long-range dispersion settings

The PPPM method computes interactions by splitting the pair potential into two parts, one of which is computed in a normal pairwise fashion, the so-called real-space part, and one of which is computed using the Fourier transform, the so called reciprocal-space or kspace part. For both parts, the potential is not computed exactly but is approximated. Thus, there is an error in both parts of the computation, the real-space and the kspace error. The just mentioned facts are true both for the PPPM for Coulomb as well as dispersion interactions. The deciding difference - and also the reason why the parameters for pppm/disp have to be selected with more care - is the impact of the errors on the results: The kspace error of the PPPM for Coulomb and dispersion interaction and the real-space error of the PPPM for Coulomb interaction have the character of noise. In contrast, the real-space error of the PPPM for dispersion has a clear physical interpretation: the underprediction of cohesion. As a consequence, the real-space error has a much stronger effect than the kspace error on simulation results for pppm/disp. Parameters must thus be chosen in a way that this error is much smaller than the kspace error.

When using pppm/disp and not making any specifications on the PPPM parameters via the kspace modify command, parameters will be tuned such that the real-space error and the kspace error are equal. This will result in simulations that are either inaccurate or slow, both of which is not desirable. For selecting parameters for the pppm/disp that provide fast and accurate simulations, there are two approaches, which both have their up- and downsides.

The first approach is to set desired real-space an kspace accuracies via the *kspace_modify force/disp/real* and *kspace_modify force/disp/kspace* commands. Note that the accuracies have to be specified in force units and are thus dependent on the chosen unit settings. For real units, 0.0001 and 0.002 seem to provide reasonable accurate and efficient computations for the real-space and kspace accuracies. 0.002 and 0.05 work well for most systems using lj units. PPPM parameters will be generated based on the desired accuracies. The upside of this approach is that it usually provides a good set of parameters and will work for both the *kspace_modify diff ad* and *kspace_modify diff ik* options. The downside of the method is that setting the PPPM parameters will take some time during the initialization of the simulation.

The second approach is to set the parameters for the pppm/disp explicitly using the *kspace_modify mesh/disp*, *kspace_modify order/disp*, and *kspace_modify gewald/disp* commands. This approach requires a more experienced user who understands well the impact of the choice of parameters on the simulation accuracy and performance. This approach provides a fast initialization of the simulation. However, it is sensitive to errors: A combination of parameters that will perform well for one system might result in far-from-optimal conditions for other simulations. For example, parameters that provide accurate and fast computations for all-atomistic force fields can provide insufficient accuracy or united-atomistic force fields (which is related to that the latter typically have larger dispersion coefficients).

To avoid inaccurate or inefficient simulations, the pppm/disp stops simulations with an error message if no action is taken to control the PPPM parameters. If the automatic parameter generation is desired and real-space and kspace accuracies are desired to be equal, this error message can be suppressed using the *kspace_modify disp/auto yes* command.

A reasonable approach that combines the upsides of both methods is to make the first run using the *kspace_modify force/disp/real* and *kspace_modify force/disp/kspace* commands, write down the PPPM parameters from the output, and specify these parameters using the second approach in subsequent runs (which have the same composition, force field, and approximately the same volume).

Concerning the performance of the pppm/disp there are two more things to consider. The first is that when using the pppm/disp, the cutoff parameter does no longer affect the accuracy of the simulation (subject to that gewald/disp is adjusted when changing the cutoff). The performance can thus be increased by examining different values for the cutoff parameter. A lower bound for the cutoff is only set by the truncation error of the repulsive term of pair potentials.

The second is that the mixing rule of the pair style has an impact on the computation time when using the pppm/disp. Fastest computations are achieved when using the geometric mixing rule. Using the arithmetic mixing rule substantially increases the computational cost. The computational overhead can be reduced using the *kspace_modify mix/disp geom* and *kspace_modify splittol* commands. The first command simply enforces geometric mixing of the dispersion coefficients in kspace computations. This introduces some error in the computations but will also significantly speed-up the simulations. The second keyword sets the accuracy with which the dispersion coefficients are approximated using a matrix factorization approach. This may result in better accuracy then using the first command, but will usually also not provide an equally good increase of efficiency.

Finally, pppm/disp can also be used when no mixing rules apply. This can be achieved using the *kspace_modify mix/disp none* command. Note that the code does not check automatically whether any mixing rule is fulfilled. If mixing rules do not apply, the user will have to specify this command explicitly.

### 10.2.9 Convert bulk system to slab

A regularly encountered simulation problem is how to convert a bulk system that has been run for a while to equilibrate into a slab system with some vacuum space and free surfaces. The challenge here is that one cannot just change the box dimensions with the *change_box command* or edit the box boundaries in a data file because some atoms will have non-zero image flags from diffusing around.

Changing the box dimensions results in an undesired displacement of those atoms, since the image flags indicate how many times the box length in x-, y-, or z-direction needs to be added or subtracted to get the "unwrapped" coordinates. By changing the box dimension this distance is changed and thus those atoms move unphysically relative to their neighbors with zero image flags. Setting image flags forcibly to zero creates problems because that could break apart molecules by having one atom of a bond on the top of the system and the other at the bottom.

**Disclaimer**

The following workflow will work for many bulk systems, but not all. Some systems cannot be converted (e.g. polymers with bonds to the same molecule across periodic boundaries, sometimes called "infinite polymers"). The amount of vacuum that needs to be added depends on the length of the molecules where the system is split (the example here splits where there is water with short molecules). In some cases, the system may need to be re-centered in the box first using the *displace_atoms command*. Also, the time spent on strong thermalization and equilibration will depend on the specific system and its thermodynamic conditions.

Below is a suggested workflow using the *Rhodopsin benchmark input* for demonstration. The figure shows the state *before* the procedure on the left (with unwrapped atoms that have diffused out of the box) and *after* on the right (with the vacuum added above and below). The procedure is implemented by modifying a copy of the `in.rhodo` input file. The first lines up to and including the *read_data command* remain unchanged. Then we insert the following lines to add vacuum to the z direction above and below the system:

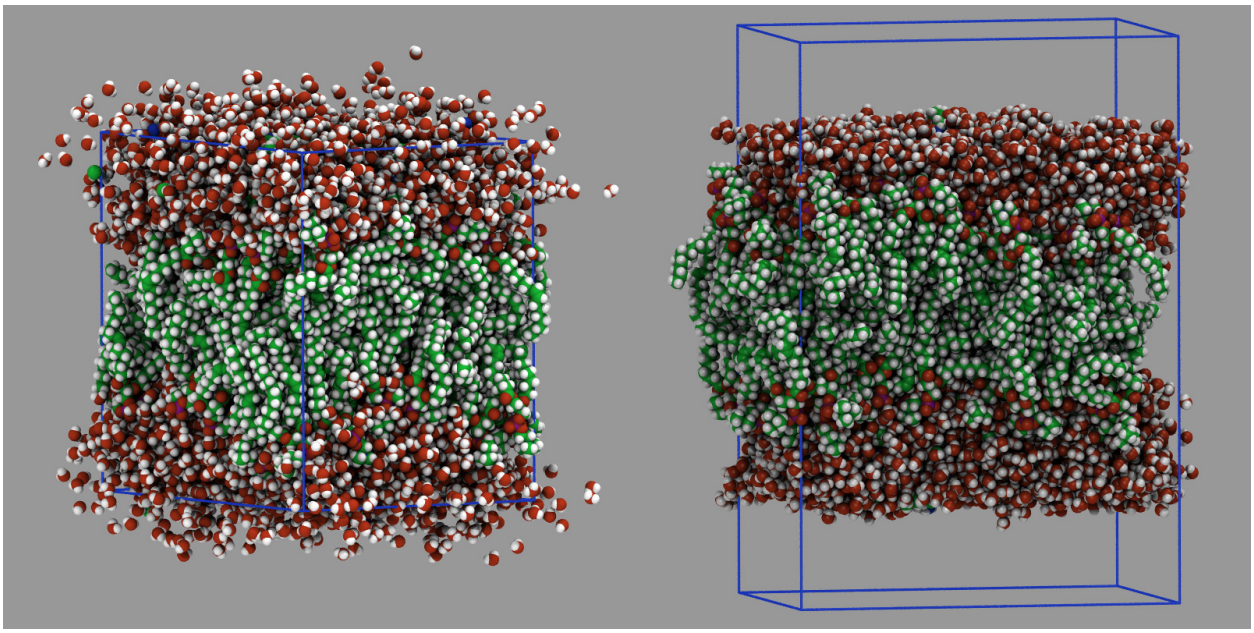


Fig. 2: Snapshots of the bulk Rhodopsin in lipid layer and water system (right) and the generated slab geometry (left)

```
variable        delta index 10.0
reset_atoms     image all
write_dump      all custom rhodo-unwrap.lammpstrj id xu yu zu
change_box      all z final $(zlo-2.0*v_delta) $(zhi+2.0*v_delta) &
                    boundary p p f
read_dump       rhodo-unwrap.lammpstrj 0 x y z box no replace yes
kspace_modify   slab auto
```

Specifically, the *variable delta* (set to 10.0) represents a distance that determines the amount of vacuum added: we add twice its value in each direction to the z-dimension; thus in total 40Å get added. The *reset_atoms image all* command shall reset any image flags to become either 0 or $\pm 1$ and thus have the minimum distance from the center of the simulation box, but the correct relative distance for bonded atoms.

Using `kspace_modify slab auto` lets the solver choose the near-optimal virtual vacuum used by the reciprocal-space slab correction from the current long-range accuracy target and box geometry. A fixed value such as `slab 3.0` can still be used if a manual override is preferred.

The *write_dump command* then writes out the resulting *unwrapped* coordinates of the system. After expanding the box, coordinates that were outside the box should now be inside and the unwrapped coordinates will become "wrapped", while atoms outside the periodic boundaries will be wrapped back into the box and their image flags in those directions restored.

The *change_box command* adds the desired distance to the low and high box boundary in z-direction and then changes the *boundary to "p p f"* which will force the image flags in z-direction to zero and create an undesired displacement for the atoms with non-zero image flags.

With the *read_dump command* we read back and replace partially incorrect coordinates with the previously saved, unwrapped coordinates. It is important to ignore the box dimensions stored in the dump file. We want to preserve the expanded box. Finally, we turn on the slab correction for the PPPM long-range solver with the *kspace_modify command* as required when using a long range Coulomb solver for non-periodic z-dimension.

Next we replace the *fix npt command* with:

```
fix             2 nvt temp 300.0 300.0 10.0
```

We now have an open system and thus the adjustment of the cell in z-direction is no longer required. Since splitting the bulk water region where the vacuum is inserted, creates surface atoms with high potential energy, we reduce the thermostat time constant from 100.0 to 10.0 to remove excess kinetic energy resulting from that change faster.

Also the high potential energy of the surface atoms can cause that some of them are ejected from the slab. In order to suppress that, we add soft harmonic walls to push back any atoms that want to leave the slab. To determine the position of the wall, we first need to determine the extent of the atoms in z-direction and then place the harmonic walls based on that information:

```
compute         zmin all reduce min z
compute         zmax all reduce max z
thermo_style    custom zlo c_zmin zhi c_zmax
run             0 post no
fix             3 all wall/harmonic zhi $(c_zmax+v_delta) 10.0 0.0 ${delta} &
                                    zlo $(c_zmin-v_delta) 10.0 0.0 ${delta}
```

The two *compute reduce* command determine the minimum and maximum z-coordinate across all atoms. In order to trigger the execution of the compute commands we need to "consume" them. This is done with the *thermo_style custom* command followed by the *run 0* command. This avoids and error accessing the min/max values determined by the compute commands to compute the location of the wall in lower and upper direction. This uses the previously defined *delta* variable to determine the distance of the wall from the extent of the system and the cutoff for the wall interaction. This way only atoms that move beyond the min/max values in z-direction will experience a restoring force, nudging them back to the slab. The force constant of $10.0 \frac{\mathrm{kcal/mol}}{\mathrm{\AA}}$ was determined empirically.

Adding these "restoring" soft walls assists in making the free surfaces above and below the slab flat, instead of having rugged or undulated surfaces. The impact of the walls can be changed by adjusting the force constant, cutoff, and position of the wall.

Finally, we replace the *run 100* of the original input with:

```
run             1000 post no

unfix           3
fix             2 all nvt temp 300.0 300.0 100.0
run             1000 post no

write_data      data.rhodo-slab
```

This runs the system converted to a slab first for 1000 MD steps using the walls and stronger Nose-Hoover thermostat. Then the walls are removed with *unfix 3* and the thermostat time constant reset to 100.0 and the system run for another 1000 steps. Finally the resulting slab geometry is written to a new data file `data.rhodo-slab` with a *write_data command*. The number of MD steps required to reach a proper equilibrium state is very likely larger. The number of 1000 steps (corresponding to 2 picoseconds) was chosen for demonstration purposes, so that the procedure can be easily and quickly tested.

### 10.2.10 Broken Bonds

Typically, molecular bond interactions persist for the duration of a simulation in LAMMPS. However, some commands break bonds dynamically, including the following:

- *bond_style quartic*
- *fix bond/break*
- *fix bond/react*

- *BPM package* bond styles

A bond can break if it is stretched beyond a user-defined threshold or more generally if other criteria are met.

For the quartic bond style, when a bond is broken its bond type is set to 0 to effectively break it and pairwise forces between the two atoms in the broken bond are "turned on". Angles, dihedrals, etc cannot be defined for a system when *bond_style quartic* is used.

Similarly, bond styles in the BPM package are also incompatible with angles, dihedrals, etc. and when a bond breaks its type is set to zero. However, in the BPM package one can either turn off all pair interactions between bonded particles or leave them on, overlaying pair forces on top of bond forces. To remove pair forces, the special bond list is dynamically updated. More details can be found on the *Howto BPM* page.

The *fix bond/break* and *fix bond/react* commands allow breaking of bonds within a molecular topology with may also define angles, dihedrals, etc. These commands update internal topology data structures to remove broken bonds, as well as the appropriate angle, dihedral, etc interactions which include the bond. They also trigger a rebuild of the neighbor list when this occurs, to turn on the appropriate pairwise forces.

Note that when bonds are dumped to a file via the *dump local* command, bonds with type 0 are not included.

The *delete_bonds* command can be used to query the status of broken bonds with type = 0 or permanently delete them, e.g.:

```
delete_bonds all stats
delete_bonds all bond 0 remove
```

The compute *count/type* command tallies the current number of bonds (or angles, etc) for each bond (angle, etc) type. It also tallies broken bonds with type = 0.

The compute *nbond/atom* command tallies the current number of bonds each atom is part of, excluding broken bonds with type = 0.

## 10.3 Analysis howto

### 10.3.1 Output from LAMMPS (thermo, dumps, computes, fixes, variables)

There are four basic forms of LAMMPS output:

- *Thermodynamic output*, which is a list of quantities printed every few timesteps to the screen and logfile.
- *Dump files*, which contain snapshots of atoms and various per-atom values and are written at a specified frequency.
- Certain fixes can output user-specified quantities to files: *fix ave/time* for time averaging, *fix ave/chunk* for spatial or other averaging, and *fix print* for single-line output of *variables*. Fix print can also output to the screen.
- *Restart files*.

A simulation prints one set of thermodynamic output and (optionally) restart files. It can generate any number of dump files and fix output files, depending on what *dump* and *fix* commands you specify.

As discussed below, LAMMPS gives you a variety of ways to determine what quantities are calculated and printed when the thermodynamics, dump, or fix commands listed above perform output. Throughout this discussion, note that users can also *add their own computes and fixes to LAMMPS* which can generate values that can then be output with these commands.

The following subsections discuss different LAMMPS commands related to output and the kind of data they operate on and produce:

- *Global/per-atom/local/per-grid data*
- *Scalar/vector/array data*

- *Disambiguation*
- *Thermodynamic output*
- *Dump file output*
- *Fixes that write output files*
- *Computes that process output quantities*
- *Fixes that process output quantities*
- *Computes that generate values to output*
- *Fixes that generate values to output*
- *Variables that generate values to output*
- *Summary table of output options and data flow between commands*

### Global/per-atom/local/per-grid data

Various output-related commands work with four different "styles" of data: global, per-atom, local, and per-grid. A global datum is one or more system-wide values, e.g. the temperature of the system. A per-atom datum is one or more values per atom, e.g. the kinetic energy of each atom. Local datums are calculated by each processor based on the atoms it owns, and there may be zero or more per atom, e.g. a list of bond distances.

A per-grid datum is one or more values per grid cell, for a grid which overlays the simulation domain. Similar to atoms and per-atom data, the grid cells and the data they store are distributed across processors; each processor owns the grid cells whose center points fall within its subdomain.

### Scalar/vector/array data

Global, per-atom, local, and per-grid datums can come in three "kinds": a single scalar value, a vector of values, or a 2d array of values. More specifically these are the valid kinds for each style:

- global scalar
- global vector
- global array
- per-atom vector
- per-atom array
- local vector
- local array
- per-grid vector
- per-grid array

A per-atom vector means a single value per atom; the "vector" is the length of the number of atoms. A per-atom array means multiple values per atom. Similarly a local vector or array means one or multiple values per entity (e.g. per bond in the system). And a per-grid vector or array means one or multiple values per grid cell.

The doc page for a compute or fix or variable that generates data will specify both the styles and kinds of data it produces, e.g. a per-atom vector. Note that a compute or fix may generate multiple styles and kinds of output. However, for per-atom data only a vector or array is output, never both. Likewise for per-local and per-grid data. An example of a fix which generates multiple styles and kinds of data is the *fix mdi/qm* command. It outputs a global scalar, global vector, and per-atom array for the quantum mechanical energy and virial of the system and forces on each atom.

By contrast, different variable styles generate only a single kind of data: a global scalar for an equal-style variable, global vector for a vector-style variable, and a per-atom vector for an atom-style variable.

When data is accessed by another command, as in many of the output commands discussed below, it can be referenced via the following bracket notation, where ID in this case is the ID of a compute. The leading "c_" would be replaced by "f_" for a fix, or "v_" for a variable (and ID would be the name of the variable):

| | |
|---|---|
| c_ID | entire scalar, vector, or array |
| c_ID[I] | one element of vector, one column of array |
| c_ID[I][J] | one element of array |

Note that using one bracket reduces the dimension of the data once (vector -> scalar, array -> vector). Using two brackets reduces the dimension twice (array -> scalar). Thus a command that uses scalar values as input can also conceptually operate on an element of a vector or array.

Per-grid vectors or arrays are accessed similarly, except that the ID for the compute or fix includes a grid name and a data name. This is because a fix or compute can create multiple grids (of different sizes) and multiple sets of data (for each grid). The fix or compute defines names for each grid and for each data set, so that all of them can be accessed by other commands. See the *Howto grid* doc page for more details.

### Disambiguation

When a compute or fix produces data in multiple styles, e.g. global and per-atom, a reference to the data can sometimes be ambiguous. Usually the context in which the input script references the data determines which style is meant.

For example, if a compute outputs a global vector and a per-atom array, an element of the global vector will be accessed by using `c_ID[I]` in *thermodynamic output*, while a column of the per-atom array will be accessed by using `c_ID[I]` in a *dump custom* command.

However, if a *atom-style variable* references `c_ID[I]`, then it could be intended to refer to a single element of the global vector or a column of the per-atom array. The doc page for any command that has a potential ambiguity (variables are the most common) will explain how to resolve the ambiguity.

In this case, an atom-style variables references per-atom data if it exists. If access to an element of a global vector is needed (as in this example), an equal-style variable which references the value can be defined and used in the atom-style variable formula instead.

Similarly, *thermodynamic output* can only reference global data from a compute or fix. But you can indirectly access per-atom data as follows. The reference `c_ID[245][2]` for the ID of a *compute displace/atom* command, refers to the y-component of displacement for the atom with ID 245. While you cannot use that reference directly in the *thermo_style* command, you can use it an equal-style variable formula, and then reference the variable in thermodynamic output.

### Thermodynamic output

The frequency and format of thermodynamic output is set by the *thermo*, *thermo_style*, and *thermo_modify* commands. The *thermo_style* command also specifies what values are calculated and written out. Pre-defined keywords can be specified (e.g. press, etotal, etc). Three additional kinds of keywords can also be specified (c_ID, f_ID, v_name), where a *compute* or *fix* or *variable* provides the value to be output. In each case, the compute, fix, or variable must generate global values for input to the *thermo_style custom* command.

Note that thermodynamic output values can be "extensive" or "intensive". The former scale with the number of atoms in the system (e.g. total energy), the latter do not (e.g. temperature). The setting for *thermo_modify norm* determines whether extensive quantities are normalized or not. Computes and fixes produce either extensive or intensive values; see their individual doc pages for details. *Equal-style variables* produce only intensive values; you can include a division by "natoms" in the formula if desired, to make an extensive calculation produce an intensive result.

### Dump file output

Dump file output is specified by the *dump* and *dump_modify* commands. There are several pre-defined formats (dump atom, dump xtc, etc).

There is also a *dump custom* format where the user specifies what values are output with each atom. Pre-defined atom attributes can be specified (id, x, fx, etc). Three additional kinds of keywords can also be specified (c_ID, f_ID, v_name), where a *compute* or *fix* or *variable* provides the values to be output. In each case, the compute, fix, or variable must generate per-atom values for input to the *dump custom* command.

There is also a *dump local* format where the user specifies what local values to output. A pre-defined index keyword can be specified to enumerate the local values. Two additional kinds of keywords can also be specified (c_ID, f_ID), where a *compute* or *fix* or *variable* provides the values to be output. In each case, the compute or fix must generate local values for input to the *dump local* command.

There is also a *dump grid* format where the user specifies what per-grid values to output from computes or fixes that generate per-grid data.

### Fixes that write output files

Several fixes take various quantities as input and can write output files: *fix ave/time*, *fix ave/chunk*, *fix ave/histo*, *fix ave/correlate*, and *fix print*.

The *fix ave/time* command enables direct output to a file and/or time-averaging of global scalars or vectors. The user specifies one or more quantities as input. These can be global *compute* values, global *fix* values, or *variables* of any style except the atom style which produces per-atom values. Since a variable can refer to keywords used by the *thermo_style custom* command (like temp or press) and individual per-atom values, a wide variety of quantities can be time averaged and/or output in this way. If the inputs are one or more scalar values, then the fix generate a global scalar or vector of output. If the inputs are one or more vector values, then the fix generates a global vector or array of output. The time-averaged output of this fix can also be used as input to other output commands.

The *fix ave/chunk* command enables direct output to a file of chunk-averaged per-atom quantities like those output in dump files. Chunks can represent spatial bins or other collections of atoms, e.g. individual molecules. The per-atom quantities can be atom density (mass or number) or atom attributes such as position, velocity, force. They can also be per-atom quantities calculated by a *compute*, by a *fix*, or by an atom-style *variable*. The chunk-averaged output of this fix is global and can also be used as input to other output commands.

Note that the *fix ave/grid* command can also average the same per-atom quantities within spatial bins, but it does this for a distributed grid whose grid cells are owned by different processors. It outputs per-grid data, not global data, so it is more efficient for large numbers of averaging bins.

The *fix ave/histo* command enables direct output to a file of histogrammed quantities, which can be global or per-atom or local quantities. The histogram output of this fix can also be used as input to other output commands.

The *fix ave/correlate* command enables direct output to a file of time-correlated quantities, which can be global values. The correlation matrix output of this fix can also be used as input to other output commands.

The *fix print* command can generate a line of output written to the screen and log file or to a separate file, periodically during a running simulation. The line can contain one or more *variable* values for any style variable except the vector or atom styles). As explained above, variables themselves can contain references to global values generated by *thermodynamic keywords*, *computes*, *fixes*, or other *variables*, or to per-atom values for a specific atom. Thus the *fix print* command is a means to output a wide variety of quantities separate from normal thermodynamic or dump file output.

### Computes that process output quantities

The *compute reduce* and *compute reduce/region* commands take one or more per-atom or local vector quantities as inputs and "reduce" them (sum, min, max, ave) to scalar quantities. These are produced as output values which can be used as input to other output commands.

The *compute slice* command take one or more global vector or array quantities as inputs and extracts a subset of their values to create a new vector or array. These are produced as output values which can be used as input to other output commands.

The *compute property/atom* command takes a list of one or more pre-defined atom attributes (id, x, fx, etc) and stores the values in a per-atom vector or array. These are produced as output values which can be used as input to other output commands. The list of atom attributes is the same as for the *dump custom* command.

The *compute property/local* command takes a list of one or more pre-defined local attributes (bond info, angle info, etc) and stores the values in a local vector or array. These are produced as output values which can be used as input to other output commands.

The *compute property/grid* command takes a list of one or more pre-defined per-grid attributes (id, grid cell coords, etc) and stores the values in a per-grid vector or array. These are produced as output values which can be used as input to the *dump grid* command.

The *compute property/chunk* command takes a list of one or more pre-defined chunk attributes (id, count, coords for spatial bins) and stores the values in a global vector or array. These are produced as output values which can be used as input to other output commands.

### Fixes that process output quantities

The *fix vector* command can create global vectors as output from global scalars as input, accumulating them one element at a time.

The *fix ave/atom* command performs time-averaging of per-atom vectors. The per-atom quantities can be atom attributes such as position, velocity, force. They can also be per-atom quantities calculated by a *compute*, by a *fix*, or by an atom-style *variable*. The time-averaged per-atom output of this fix can be used as input to other output commands.

The *fix store/state* command can archive one or more per-atom attributes at a particular time, so that the old values can be used in a future calculation or output. The list of atom attributes is the same as for the *dump custom* command, including per-atom quantities calculated by a *compute*, by a *fix*, or by an atom-style *variable*. The output of this fix can be used as input to other output commands.

The *fix ave/grid* command performs time-averaging of either per-atom or per-grid data.

For per-atom data it performs averaging for the atoms within each grid cell, similar to the *fix ave/chunk* command when its chunks are defined as regular 2d or 3d bins. The per-atom quantities can be atom density (mass or number) or atom attributes such as position, velocity, force. They can also be per-atom quantities calculated by a *compute*, by a *fix*, or by an atom-style *variable*.

The chief difference between the *fix ave/grid* and *fix ave/chunk* commands when used in this context is that the former uses a distributed grid, while the latter uses a global grid. Distributed means that each processor owns the subset of grid cells within its subdomain. Global means that each processor owns a copy of the entire grid. The *fix ave/grid* command is thus more efficient for large grids.

For per-grid data, the *fix ave/grid* command takes inputs for grid data produced by other computes or fixes and averages the values for each grid point over time.

### Computes that generate values to output

Every *compute* in LAMMPS produces either global or per-atom or local or per-grid values. The values can be scalars or vectors or arrays of data. These values can be output using the other commands described in this section. The page for each compute command describes what it produces. Computes that produce per-atom or local or per-grid values have the word “atom” or “local” or “grid” as the last word in their style name. Computes without the word “atom” or “local” or “grid” produce global values.

### Fixes that generate values to output

Some *fixes* in LAMMPS produces either global or per-atom or local or per-grid values which can be accessed by other commands. The values can be scalars or vectors or arrays of data. These values can be output using the other commands described in this section. The page for each fix command tells whether it produces any output quantities and describes them.

### Variables that generate values to output

*Variables* defined in an input script can store one or more strings. But equal-style, vector-style, and atom-style or atomfile-style variables generate a global scalar value, global vector or values, or a per-atom vector, respectively, when accessed. The formulas used to define these variables can contain references to the thermodynamic keywords and to global and per-atom data generated by computes, fixes, and other variables. The values generated by variables can be used as input to and thus output by the other commands described in this section.

Per-grid variables have not (yet) been implemented.

### Summary table of output options and data flow between commands

This table summarizes the various commands that can be used for generating output from LAMMPS. Each command produces output data of some kind and/or writes data to a file. Most of the commands can take data from other commands as input. Thus you can link many of these commands together in pipeline form, where data produced by one command is used as input to another command and eventually written to the screen or to a file. Note that to hook two commands together the output and input data types must match, e.g. global/per-atom/local data and scalar/vector/array data.

Also note that, as described above, when a command takes a scalar as input, that could also be an element of a vector or array. Likewise a vector input could be a column of an array.

| Command | Input | Output |
|---|---|---|
| *thermo_style custom* | global scalars | screen, log file |
| *dump custom* | per-atom vectors | dump file |
| *dump local* | local vectors | dump file |
| *dump grid* | per-grid vectors | dump file |
| *fix print* | global scalar from variable | screen, file |
| *print* | global scalar from variable | screen |
| *computes* | N/A | global/per-atom/local/per-grid scalar/vector/array |
| *fixes* | N/A | global/per-atom/local/per-grid scalar/vector/array |
| *variables* | global scalars and vectors, per-atom vectors | global scalar and vector, per-atom vector |
| *compute reduce* | per-atom/local vectors | global scalar/vector |
| *compute slice* | global vectors/arrays | global vector/array |
| *compute property/atom* | N/A | per-atom vector/array |
| *compute property/local* | N/A | local vector/array |
| *compute property/grid* | N/A | per-grid vector/array |
| *compute property/chunk* | N/A | global vector/array |
| *fix vector* | global scalars | global vector |
| *fix ave/atom* | per-atom vectors | per-atom vector/array |
| *fix ave/time* | global scalars/vectors | global scalar/vector/array, file |
| *fix ave/chunk* | per-atom vectors | global array, file |
| *fix ave/grid* | per-atom vectors or per-grid vectors | per-grid vector/array |
| *fix ave/histo* | global/per-atom/local scalars and vectors | global array, file |
| *fix ave/correlate* | global scalars | global array, file |
| *fix store/state* | per-atom vectors | per-atom vector/array |

### 10.3.2 Output structured data from LAMMPS

LAMMPS can output structured data with the *print* and *fix print* command. This gives you flexibility since you can build custom data formats that contain system properties, thermo data, and variables values. This output can be directed to the screen and/or to a file for post processing.

#### Writing the current system state, thermo data, variable values

Use the *print* command to output the current system state, which can include system properties, thermo data and variable values.

##### YAML

```
print """---
timestep: $(step)
pe: $(pe)
ke: $(ke)
..."""" file current_state.yaml screen no
```

Listing 1: current_state.yaml

```
---
timestep: 250
pe: -4.7774327356321810711
ke: 2.4962152903997174569
```

### JSON

```
print """{
  "timestep": $(step),
  "pe": $(pe),
  "ke": $(ke)
}""" file current_state.json screen no
```

Listing 2: current_state.json

```
{
  "timestep": 250,
  "pe": -4.7774327356321810711,
  "ke": 2.4962152903997174569
}
```

### YAML format thermo_style or dump_style output

### Extracting data from log file

Added in version 24Mar2022.

LAMMPS supports the thermo style "yaml" and for "custom" style thermodynamic output the format can be changed to YAML with *thermo_modify line yaml*. This will produce a block of output in a compact YAML format - one "document" per run - of the following style:

```
---
keywords: ['Step', 'Temp', 'E_pair', 'E_mol', 'TotEng', 'Press', ]
data:
  - [100, 0.757453103239935, -5.7585054860159, 0, -4.62236133677021, 0.207261053624721, ]
  - [110, 0.759322359337036, -5.7614668389562, 0, -4.62251889318624, 0.194314975399602, ]
  - [120, 0.759372342462676, -5.76149365656489, 0, -4.62247073844943, 0.191600048851267,␣
↪]
  - [130, 0.756833027516501, -5.75777334823494, 0, -4.62255928350835, 0.208792327853067,␣
↪]
...
```

This data can be extracted and parsed from a log file using python with:

```
import re, yaml
try:
    from yaml import CSafeLoader as Loader
except ImportError:
    from yaml import SafeLoader as Loader
```

(continues on next page)

(continued from previous page)

```
docs = ""
with open("log.lammps") as f:
    for line in f:
        m = re.search(r"^(keywords:.*$|data:$|---$|\.\.\.$|  - \[.*\]$)", line)
        if m: docs += m.group(0) + '\n'

thermo = list(yaml.load_all(docs, Loader=Loader))

print("Number of runs: ", len(thermo))
print(thermo[1]['keywords'][4], ' = ', thermo[1]['data'][2][4])
```

After loading the YAML data, *thermo* is a list containing a dictionary for each "run" where the tag "keywords" maps to the list of thermo header strings and the tag "data" has a list of lists where the outer list represents the lines of output and the inner list the values of the columns matching the header keywords for that step. The second print() command for example will print the header string for the fifth keyword of the second run and the corresponding value for the third output line of that run:

```
Number of runs:  2
TotEng  =  -4.62140097780047
```

### Extracting data from dump file

Added in version 4May2022.

YAML format output has been added to multiple commands in LAMMPS, for example *dump yaml* or *fix ave/time* Depending on the kind of data being written, organization of the data or the specific syntax used may change, but the principles are very similar and all files should be readable with a suitable YAML parser. A simple example for this is given below:

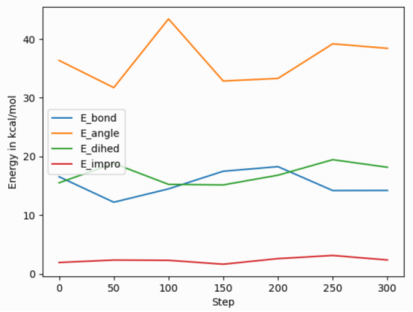
```
import yaml
try:
    from yaml import CSafeLoader as YamlLoader
except ImportError:
    from yaml import SafeLoader as YamlLoader

timesteps = []
with open("dump.yaml", "r") as f:
    data = yaml.load_all(f, Loader=YamlLoader)

    for d in data:
        print('Processing timestep %d' % d['timestep'])
        timesteps.append(d)

print('Read %d timesteps from yaml dump' % len(timesteps))
print('Second timestep: ', timesteps[1]['timestep'])
print('Box info: x: ' , timesteps[1]['box'][0], ' y:', timesteps[1]['box'][1], ' z:',
 ↪timesteps[1]['box'][2])
print('First 5 per-atom columns: ', timesteps[1]['keywords'][0:5])
print('Corresponding 10th atom data: ', timesteps[1]['data'][9][0:5])
```

The corresponding output for a YAML dump command added to the "melt" example is:

```
Processing timestep 0
Processing timestep 50
Processing timestep 100
Processing timestep 150
Processing timestep 200
Processing timestep 250
Read 6 timesteps from yaml dump
Second timestep:  50
Box info: x:  [0, 16.795961913825074]  y: [0, 16.795961913825074]  z: [0, 16.
↪795961913825074]
First 5 per-atom columns:  ['id', 'type', 'x', 'y', 'z']
Corresponding 10th atom data:  [10, 1, 4.43828, 0.968481, 0.108555]
```

### Processing scalar data with Python

After reading and parsing the YAML format data, it can be easily imported for further processing and visualization with the pandas and matplotlib Python modules. Because of the organization of the data in the YAML format thermo output, it needs to be told to process only the 'data' part of the imported data to create a pandas data frame, and one needs to set the column names from the 'keywords' entry. The following Python script code example demonstrates this, and creates the image shown on the right of a simple plot of various bonded energy contributions versus the timestep from a run of the 'peptide' example input after changing the *thermo style* to 'yaml'. The properties to be used for x and y values can be conveniently selected through the keywords. Please note that those keywords can be changed to custom strings with the *thermo_modify colname* command.




```
import re, yaml
import pandas as pd
import matplotlib.pyplot as plt

try:
    from yaml import CSafeLoader as Loader
except ImportError:
    from yaml import SafeLoader as Loader

docs = ""
with open("log.lammps") as f:
    for line in f:
        m = re.search(r"^(keywords:.*$|data:$|---$|\.\.\.$|  - \[.*\]$)", line)
        if m: docs += m.group(0) + '\n'

thermo = list(yaml.load_all(docs, Loader=Loader))

df = pd.DataFrame(data=thermo[0]['data'], columns=thermo[0]['keywords'])
fig = df.plot(x='Step', y=['E_bond', 'E_angle', 'E_dihed', 'E_impro'], ylabel='Energy in␣
↪kcal/mol')
plt.savefig('thermo_bondeng.png')
```

### Processing vector data with Python

Global *vector* data as produced by *fix ave/time* uses a slightly different organization of the data. You still have the dictionary keys 'keywords' and 'data' for the column headers and the data. But the data is a dictionary indexed by the time step and for each step there are multiple rows of values each with a list of the averaged properties. This requires a slightly different processing, since the entire data cannot be directly imported into a single pandas DataFrame class instance. The following Python script example demonstrates how to read such data. The result will combine the data for the different steps into one large "multi-index" table. The pandas IndexSlice class can then be used to select data from this combined data frame.

```
import yaml
import pandas as pd

try:
    from yaml import CSafeLoader as Loader
except ImportError:
    from yaml import SafeLoader as Loader

with open("ave.yaml") as f:
    ave = yaml.load(f, Loader=Loader)

keys = ave['keywords']
df = {}
for k in ave['data'].keys():
    df[k] = pd.DataFrame(data=ave['data'][k], columns=keys)

# create multi-index data frame
df = pd.concat(df)

# output only the first 3 value for steps 200 to 300 of the column Pressure
idx = pd.IndexSlice
print(df['Pressure'].loc[idx[200:300, 0:2]])
```

### Processing scalar data with Perl

The ease of processing YAML data is not limited to Python. Here is an example for extracting and processing a LAMMPS log file with Perl instead.

```
use YAML::XS;

open(LOG, "log.lammps") or die("could not open log.lammps: $!");
my $file = "";
while(my $line = <LOG>) {
    if ($line =~ /^(keywords:.*$|data:$|---$|\.\.\.$|  - \[.*\]$)/) {
        $file .= $line;
    }
}
close(LOG);

# convert YAML to perl as nested hash and array references
my $thermo = Load $file;

# convert references to real arrays
```

(continues on next page)

(continued from previous page)

```
my @keywords = @{$thermo->{'keywords'}};
my @data = @{$thermo->{'data'}};

# print first two columns
print("$keywords[0] $keywords[1]\n");
foreach (@data) {
    print("${$_}[0]  ${$_}[1]\n");
}
```

### Writing continuous data during a simulation

The *fix print* command allows you to output an arbitrary string at defined times during a simulation run.

#### YAML

```
fix extra all print 50 """
- timestep: $(step)
  pe: $(pe)
  ke: $(ke)""" file output.yaml screen no
```

Listing 3: output.yaml

```
# Fix print output for fix extra
- timestep: 0
  pe: -6.77336805325924729
  ke: 4.4988750000000026219

- timestep: 50
  pe: -4.8082494418323200591
  ke: 2.5257981827119797558

- timestep: 100
  pe: -4.7875608875581505686
  ke: 2.5062598821985102582

- timestep: 150
  pe: -4.7471033686005483787
  ke: 2.466095925545450207

- timestep: 200
  pe: -4.7509052858544134068
  ke: 2.4701136792591693592

- timestep: 250
  pe: -4.7774327356321810711
  ke: 2.4962152903997174569
```

Post-processing of YAML files can be easily be done with Python and other scripting languages. In case of Python the *yaml* package allows you to load the data files and obtain a list of dictionaries.

```
import yaml
```

(continues on next page)

(continued from previous page)

```
with open("output.yaml") as f:
   data = yaml.load(f, Loader=yaml.FullLoader)

print(data)
```

```
[{'timestep': 0, 'pe': -6.773368053259247, 'ke': 4.498875000000003},
 {'timestep': 50, 'pe': -4.80824944183232, 'ke': 2.5257981827119798},
 {'timestep': 100, 'pe': -4.787560887558151, 'ke': 2.5062598821985103},
 {'timestep': 150, 'pe': -4.747103368600548, 'ke': 2.46609592554545},
 {'timestep': 200, 'pe': -4.750905285854413, 'ke': 2.4701136792591694},
 {'timestep': 250, 'pe': -4.777432735632181, 'ke': 2.4962152903997175}]
```

### Line Delimited JSON (LD-JSON)

The JSON format itself is very strict when it comes to delimiters. For continuous output/streaming data it is beneficial use the *line delimited JSON* format. Each line represents one JSON object.

```
fix extra all print 50 """{"timestep": $(step), "pe": $(pe), "ke": $(ke)}""" &
    title "" file output.json screen no
```

Listing 4: output.json

```
{"timestep": 0, "pe": -6.7733680532592473, "ke": 4.4988750000000026}
{"timestep": 50, "pe": -4.8082494418323201, "ke": 2.5257981827119798}
{"timestep": 100, "pe": -4.7875608875581506, "ke": 2.5062598821985103}
{"timestep": 150, "pe": -4.7471033686005484, "ke": 2.4660959255454502}
{"timestep": 200, "pe": -4.7509052858544134, "ke": 2.4701136792591694}
{"timestep": 250, "pe": -4.7774327356321811, "ke": 2.4962152903997175}
```

One simple way to load this data into a Python script is to use the *pandas* package. It can directly load these files into a data frame:

```
import pandas as pd

data = pd.read_json('output.json', lines=True)
print(data)
```

```
   timestep        pe        ke
0         0 -6.773368  4.498875
1        50 -4.808249  2.525798
2       100 -4.787561  2.506260
3       150 -4.747103  2.466096
4       200 -4.750905  2.470114
5       250 -4.777433  2.496215
```

## 10.3.3 Use chunks to calculate system properties

In LAMMPS, "chunks" are collections of atoms, as defined by the *compute chunk/atom* command, which assigns each atom to a chunk ID (or to no chunk at all). The number of chunks and the assignment of chunk IDs to atoms can be static or change over time. Examples of "chunks" are molecules or spatial bins or atoms with similar values (e.g. coordination number or potential energy).

The per-atom chunk IDs can be used as input to two other kinds of commands, to calculate various properties of a system:

- *fix ave/chunk*
- any of the *compute */chunk* commands

Here a brief overview for each of the 4 kinds of chunk-related commands is provided. Then some examples are given of how to compute different properties with chunk commands.

### Compute chunk/atom command:

This compute can assign atoms to chunks of various styles. Only atoms in the specified group and optional specified region are assigned to a chunk. Here are some possible chunk definitions:

| | |
|---|---|
| atoms in same molecule | chunk ID = molecule ID |
| atoms of same atom type | chunk ID = atom type |
| all atoms with same atom property (charge, radius, etc) | chunk ID = output of compute property/atom |
| atoms in same cluster | chunk ID = output of *compute cluster/atom* command |
| atoms in same spatial bin | chunk ID = bin ID |
| atoms in same rigid body | chunk ID = molecule ID used to define rigid bodies |
| atoms with similar potential energy | chunk ID = output of *compute pe/atom* |
| atoms with same local defect structure | chunk ID = output of *compute centro/atom* or *compute coord/atom* command |

Note that chunk IDs are integer values, so for atom properties or computes that produce a floating point value, they will be truncated to an integer. You could also use the compute in a variable that scales the floating point value to spread it across multiple integers.

Spatial bins can be of various kinds, e.g. 1d bins = slabs, 2d bins = pencils, 3d bins = boxes, spherical bins, cylindrical bins.

This compute also calculates the number of chunks *Nchunk*, which is used by other commands to tally per-chunk data. *Nchunk* can be a static value or change over time (e.g. the number of clusters). The chunk ID for an individual atom can also be static (e.g. a molecule ID), or dynamic (e.g. what spatial bin an atom is in as it moves).

Note that this compute allows the per-atom output of other *computes*, *fixes*, and *variables* to be used to define chunk IDs for each atom. This means you can write your own compute or fix to output a per-atom quantity to use as chunk ID. See the *Modify* doc pages for info on how to do this. You can also define a *per-atom variable* in the input script that uses a formula to generate a chunk ID for each atom.

### Fix ave/chunk command:

This fix takes the ID of a *compute chunk/atom* command as input. For each chunk, it then sums one or more specified per-atom values over the atoms in each chunk. The per-atom values can be any atom property, such as velocity, force, charge, potential energy, kinetic energy, stress, etc. Additional keywords are defined for per-chunk properties like density and temperature. More generally any per-atom value generated by other *computes*, *fixes*, and *per-atom variables*, can be summed over atoms in each chunk.

Similar to other averaging fixes, this fix allows the summed per-chunk values to be time-averaged in various ways, and output to a file. The fix produces a global array as output with one row of values per chunk.

### Compute */chunk commands:

The following computes operate on chunks of atoms to produce per-chunk values. Any compute whose style name ends in "/chunk" is in this category:

- *compute com/chunk*
- *compute gyration/chunk*
- *compute inertia/chunk*
- *compute msd/chunk*
- *compute property/chunk*
- *compute temp/chunk*
- *compute torque/chunk*
- *compute vcm/chunk*

They each take the ID of a *compute chunk/atom* command as input. As their names indicate, they calculate the center-of-mass, radius of gyration, moments of inertia, mean-squared displacement, temperature, torque, and velocity of center-of-mass for each chunk of atoms. The *compute property/chunk* command can tally the count of atoms in each chunk and extract other per-chunk properties.

The reason these various calculations are not part of the *fix ave/chunk command*, is that each requires a more complicated operation than simply summing and averaging over per-atom values in each chunk. For example, many of them require calculation of a center of mass, which requires summing mass*position over the atoms and then dividing by summed mass.

All of these computes produce a global vector or global array as output, with one or more values per chunk. The output can be used in various ways:

- As input to the *fix ave/time* command, which can write the values to a file and optionally time average them.
- As input to the *fix ave/histo* command to histogram values across chunks. E.g. a histogram of cluster sizes or molecule diffusion rates.
- As input to special functions of *equal-style variables*, like sum() and max() and ave(). E.g. to find the largest cluster or fastest diffusing molecule or average radius-of-gyration of a set of molecules (chunks).

### Other chunk commands:

- *compute chunk/spread/atom*
- *compute reduce/chunk*

The *compute chunk/spread/atom* command spreads per-chunk values to each atom in the chunk, producing per-atom values as its output. This can be useful for outputting per-chunk values to a per-atom *dump file*. Or for using an atom's associated chunk value in an *atom-style variable*. Or as input to the *fix ave/chunk* command to spatially average per-chunk values calculated by a per-chunk compute.

The *compute reduce/chunk* command reduces a peratom value across the atoms in each chunk to produce a value per chunk. When used with the *compute chunk/spread/atom* command it can create peratom values that induce a new set of chunks with a second *compute chunk/atom* command.

### Example calculations with chunks

Here are examples using chunk commands to calculate various properties:

1. Average velocity in each of 1000 2d spatial bins:

```
compute cc1 all chunk/atom bin/2d x 0.0 0.1 y lower 0.01 units reduced
fix 1 all ave/chunk 100 10 1000 cc1 vx vy file tmp.out
```

2. Temperature in each spatial bin, after subtracting a flow velocity:

```
compute cc1 all chunk/atom bin/2d x 0.0 0.1 y lower 0.1 units reduced
compute vbias all temp/profile 1 0 0 y 10
fix 1 all ave/chunk 100 10 1000 cc1 temp bias vbias file tmp.out
```

3. Center of mass of each molecule:

```
compute cc1 all chunk/atom molecule
compute myChunk all com/chunk cc1
fix 1 all ave/time 100 1 100 c_myChunk[*] file tmp.out mode vector
```

4. Total force on each molecule and ave/max across all molecules:

```
compute cc1 all chunk/atom molecule
fix 1 all ave/chunk 1000 1 1000 cc1 fx fy fz file tmp.out
variable xave equal ave(f_1[2])
variable xmax equal max(f_1[2])
thermo 1000
thermo_style custom step temp v_xave v_xmax
```

5. Histogram of cluster sizes:

```
compute cluster all cluster/atom 1.0
compute cc1 all chunk/atom c_cluster compress yes
compute size all property/chunk cc1 count
fix 1 all ave/histo 100 1 100 0 20 20 c_size mode vector ave running beyond ignore file␣
↪tmp.histo
```

6. An example for using a per-chunk value to apply per-atom forces to compress individual polymer chains (molecules) in a mixture, is explained on the *compute chunk/spread/atom* command doc page.

7. An example for using one set of per-chunk values for molecule chunks, to create a second set of micelle-scale chunks (clustered molecules, due to hydrophobicity), is explained on the *compute reduce/chunk* command doc page.

8. An example for using one set of per-chunk values (dipole moment vectors) for molecule chunks, spreading the values to each atom in each chunk, then defining a second set of chunks as spatial bins, and using the *fix ave/chunk* command to calculate an average dipole moment vector for each bin. This example is explained on the *compute chunk/spread/atom* command doc page.

### 10.3.4 Using distributed grids

Added in version 22Dec2022.

LAMMPS has internal capabilities to create uniformly spaced grids which overlay the simulation domain. For 2d and 3d simulations these are 2d and 3d grids respectively. Conceptually a grid can be thought of as a collection of grid cells. Each grid cell can store one or more values (data).

The grid cells and data they store are distributed across processors. Each processor owns the grid cells (and data) whose center points lie within the spatial subdomain of the processor. If needed for its computations, a processor may also store ghost grid cells with their data.

Distributed grids can overlay orthogonal or triclinic simulation boxes; see the *Howto triclinic* doc page for an explanation of the latter. For a triclinic box, the grid cell shape conforms to the shape of the simulation domain, e.g.

parallelograms instead of rectangles in 2d.

If the box size or shape changes during a simulation, the grid changes with it, so that it always overlays the entire simulation domain. For non-periodic dimensions, the grid size in that dimension matches the box size, as set by the *boundary* command for fixed or shrink-wrapped boundaries.

If load-balancing is invoked by the *balance* or *fix balance* commands, then the subdomain owned by a processor can change which may also change which grid cells they own.

Post-processing and visualization of grid cell data can be enabled by the *dump grid*, *dump grid/vtk*, and *dump image* commands. The latter has an optional *grid* keyword. The OVITO visualization tool also plans (as of Nov 2022) to add support for visualizing grid cell data (along with atoms) using *dump grid* output files as input.

**Note**

For developers, distributed grids are implemented within the code via two classes: Grid2d and Grid3d. These partition the grid across processors and have methods which allow forward and reverse communication of ghost grid data as well as load balancing. If you write a new compute or fix which needs a distributed grid, these are the classes to look at. A new pair style could use a distributed grid by having a fix define it. Please see the section on *using distributed grids within style classes* for a detailed description.

---

These are the commands which currently define or use distributed grids:

- *fix ttm/grid* - store electron temperature on grid
- *fix ave/grid* - time average per-atom or per-grid values
- *compute property/grid* - generate grid IDs and coords
- *dump grid* - output per-grid values in LAMMPS format
- *dump grid/vtk* - output per-grid values in VTK format
- *dump image grid* - include colored grid in output images
- *pair_style amoeba* - FFT grids
- *kspace_style pppm* (and variants) - FFT grids
- *kspace_style msm* (and variants) - MSM grids

The grids used by the *kspace_style* can not be referenced by an input script. However the grids and data created and used by the other commands can be.

A compute or fix command may create one or more grids (of different sizes). Each grid can store one or more data fields. A data field can be a single value per grid point (per-grid vector) or multiple values per grid point (per-grid array). See the *Howto output* doc page for an explanation of how per-grid data can be generated by some commands and used by other commands.

A command accesses grid data from a compute or fix using a *grid reference* with the following syntax:

- c_ID:gname:dname
- c_ID:gname:dname[I]
- f_ID:gname:dname
- f_ID:gname:dname[I]

The prefix "c_" or "f_" refers to the ID of the compute or fix; gname is the name of the grid, which is assigned by the compute or fix; dname is the name of the data field, which is also assigned by the compute or fix.

If the data field is a per-grid vector (one value per grid point), then no brackets are used to access the values. If the data field is a per-grid array (multiple values per grid point), then brackets are used to specify the column I of the array. I ranges from 1 to Ncol inclusive, where Ncol is the number of columns in the array and is defined by the compute or fix.

Currently, there are no per-grid variables implemented in LAMMPS. We may add this feature at some point.

### 10.3.5 Calculate temperature

Temperature is computed as kinetic energy divided by some number of degrees of freedom (and the Boltzmann constant). Since kinetic energy is a function of particle velocity, there is often a need to distinguish between a particle's advection velocity (due to some aggregate motion of particles) and its thermal velocity. The sum of the two is the particle's total velocity, but the latter is often what is wanted to compute a temperature.

LAMMPS has several options for computing temperatures, any of which can be used in *thermostatting* and *barostatting*. These *compute commands* calculate temperature:

- *compute temp*
- *compute temp/sphere*
- *compute temp/asphere*
- *compute temp/com*
- *compute temp/deform*
- *compute temp/partial*
- *compute temp/profile*
- *compute temp/ramp*
- *compute temp/region*

All but the first 3 calculate velocity biases directly (e.g. advection velocities) that are removed when computing the thermal temperature. *Compute temp/sphere* and *compute temp/asphere* compute kinetic energy for finite-size particles that includes rotational degrees of freedom. They both allow for velocity biases indirectly, via an optional extra argument which is another temperature compute that subtracts a velocity bias. This allows the translational velocity of spherical or aspherical particles to be adjusted in prescribed ways.

### 10.3.6 Calculate elastic constants

Elastic constants characterize the stiffness of a material. The formal definition is provided by the linear relation that holds between the stress and strain tensors in the limit of infinitesimal deformation. In tensor notation, this is expressed as

$$s_{ij} = C_{ijkl} e_{kl}$$

where the repeated indices imply summation. $s_{ij}$ are the elements of the symmetric stress tensor. $e_{kl}$ are the elements of the symmetric strain tensor. $C_{ijkl}$ are the elements of the fourth rank tensor of elastic constants. In three dimensions, this tensor has $3^4 = 81$ elements. Using Voigt notation, the tensor can be written as a 6x6 matrix, where $C_{ij}$ is now the derivative of $s_i$ w.r.t. $e_j$. Because $s_i$ is itself a derivative w.r.t. $e_i$, it follows that $C_{ij}$ is also symmetric, with at most $\frac{7 \times 6}{2} = 21$ distinct elements.

At zero temperature, it is easy to estimate these derivatives by deforming the simulation box in one of the six directions using the *change_box* command and measuring the change in the stress tensor. A general-purpose script that does this is given in the `examples/ELASTIC` directory described on the *Examples* doc page.

Calculating elastic constants at finite temperature is more challenging, because it is necessary to run a simulation that performs time averages of differential properties. There are at least 3 ways to do this in LAMMPS. The most reliable way to do this is by exploiting the relationship between elastic constants, stress fluctuations, and the Born matrix, the second derivatives of energy w.r.t. strain *(Ray)*. The Born matrix calculation has been enabled by the *compute born/matrix* command, which works for any bonded or non-bonded potential in LAMMPS. The most expensive part of the calculation is the sampling of the stress fluctuations. Several examples of this method are provided in the `examples/ELASTIC_T/BORN_MATRIX` directory described on the *Examples* doc page.

A second way is to measure the change in average stress tensor in an NVT simulations when the cell volume undergoes a finite deformation. In order to balance the systematic and statistical errors in this method, the magnitude of the deformation must be chosen judiciously, and care must be taken to fully equilibrate the deformed cell before sampling the stress tensor. An example of this method is provided in the `examples/ELASTIC_T/DEFORMATION` directory described on the *Examples* doc page.

Another approach is to sample the triclinic cell fluctuations that occur in an NPT simulation. This method can also be slow to converge and requires careful post-processing *(Shinoda)*. We do not provide an example of this method.

A nice review of the advantages and disadvantages of all of these methods is provided in the paper by Clavier et al. *(Clavier)*.

---

**(Ray)** J. R. Ray and A. Rahman, J Chem Phys, 80, 4423 (1984).

**(Shinoda)** Shinoda, Shiga, and Mikami, Phys Rev B, 69, 134103 (2004).

**(Clavier)** G. Clavier, N. Desbiens, E. Bourasseau, V. Lachet, N. Brusselle-Dupend and B. Rousseau, Mol Sim, 43, 1413 (2017).

### 10.3.7 Calculate thermal conductivity

The thermal conductivity $\kappa$ of a material can be measured in at least 4 ways using various options in LAMMPS. See the `examples/KAPPA` directory for scripts that implement the 4 methods discussed here for a simple Lennard-Jones fluid model. Also, see the *Howto viscosity* page for an analogous discussion for viscosity.

The thermal conductivity tensor $\kappa$ is a measure of the propensity of a material to transmit heat energy in a diffusive manner as given by Fourier's law

$$J = -\kappa \cdot \text{grad}(T)$$

where $J$ is the heat flux in units of energy per area per time and grad($T$) is the spatial gradient of temperature. The thermal conductivity thus has units of energy per distance per time per degree K and is often approximated as an isotropic quantity, i.e. as a scalar.

The first method is to setup two thermostatted regions at opposite ends of a simulation box, or one in the middle and one at the end of a periodic box. By holding the two regions at different temperatures with a *thermostatting fix*, the energy added to the hot region should equal the energy subtracted from the cold region and be proportional to the heat flux moving between the regions. See the papers by *Ikeshoji and Hafskjold* and *Wirnsberger et al* for details of this idea. Note that thermostatting fixes such as *fix nvt*, *fix langevin*, and *fix temp/rescale* store the cumulative energy they add/subtract.

Alternatively, as a second method, the *fix heat* or *fix ehex* commands can be used in place of thermostats on each of two regions to add/subtract specified amounts of energy to both regions. In both cases, the resulting temperatures of the two regions can be monitored with the "compute temp/region" command and the temperature profile of the intermediate region can be monitored with the *fix ave/chunk* and *compute ke/atom* commands.

The third method is to perform a reverse non-equilibrium MD simulation using the *fix thermal/conductivity* command which implements the rNEMD algorithm of Muller-Plathe. Kinetic energy is swapped between atoms in two different layers of the simulation box. This induces a temperature gradient between the two layers which can be monitored with

the *fix ave/chunk* and *compute ke/atom* commands. The fix tallies the cumulative energy transfer that it performs. See the *fix thermal/conductivity* command for details.

The fourth method is based on the Green-Kubo (GK) formula which relates the ensemble average of the auto-correlation of the heat flux to $\kappa$. The heat flux can be calculated from the fluctuations of per-atom potential and kinetic energies and per-atom stress tensor in a steady-state equilibrated simulation. This is in contrast to the two preceding non-equilibrium methods, where energy flows continuously between hot and cold regions of the simulation box.

The *compute heat/flux* command can calculate the needed heat flux and describes how to implement the Green_Kubo formalism using additional LAMMPS commands, such as the *fix ave/correlate* command to calculate the needed auto-correlation. See the page for the *compute heat/flux* command for an example input script that calculates the thermal conductivity of solid Ar via the GK formalism.

---

**(Ikeshoji)** Ikeshoji and Hafskjold, Molecular Physics, 81, 251-261 (1994).

**(Wirnsberger)** Wirnsberger, Frenkel, and Dellago, J Chem Phys, 143, 124104 (2015).

### 10.3.8 Calculate viscosity

The shear viscosity $\eta$ of a fluid can be measured in at least 6 ways using various options in LAMMPS. See the `examples/VISCOSITY` directory for scripts that implement the 5 methods discussed here for a simple Lennard-Jones fluid model and 1 method for SPC/E water model. Also, see the *page on calculating thermal conductivity* for an analogous discussion for thermal conductivity.

$\eta$ is a measure of the propensity of a fluid to transmit momentum in a direction perpendicular to the direction of velocity or momentum flow. Alternatively it is the resistance the fluid has to being sheared. It is given by

$$J = -\eta \cdot \text{grad}(V_{\text{stream}})$$

where $J$ is the momentum flux in units of momentum per area per time. and $\text{grad}(V_{\text{stream}})$ is the spatial gradient of the velocity of the fluid moving in another direction, normal to the area through which the momentum flows. Viscosity thus has units of pressure-time.

The first method is to perform a non-equilibrium MD (NEMD) simulation by shearing the simulation box via the *fix deform* command, and using the *fix nvt/sllod* command to thermostat the fluid via the SLLOD equations of motion. Alternatively, as a second method, one or more moving walls can be used to shear the fluid in between them, again with some kind of thermostat that modifies only the thermal (non-shearing) components of velocity to prevent the fluid from heating up.

> **Note**
>
> A recent (2017) book by *(Daivis and Todd)* discusses use of the SLLOD method and non-equilibrium MD (NEMD) thermostatting generally, for both simple and complex fluids, e.g. molecular systems. The latter can be tricky to do correctly.

In both cases, the velocity profile setup in the fluid by this procedure can be monitored by the *fix ave/chunk* command, which determines $\text{grad}(V_{\text{stream}})$ in the equation above. E.g. the derivative in the y-direction of the $V_x$ component of fluid motion or $\text{grad}(V_{\text{stream}}) = \frac{\text{d}V_x}{\text{d}y}$. The $P_{xy}$ off-diagonal component of the pressure or stress tensor, as calculated by the *compute pressure* command, can also be monitored, which is the $J$ term in the equation above. See the *Howto nemd* page for details on NEMD simulations.

The third method is to perform a reverse non-equilibrium MD simulation using the *fix viscosity* command which implements the rNEMD algorithm of Muller-Plathe. Momentum in one dimension is swapped between atoms in two different layers of the simulation box in a different dimension. This induces a velocity gradient which can be monitored with the *fix ave/chunk* command. The fix tallies the cumulative momentum transfer that it performs. See the *fix viscosity* command for details.

The fourth method is based on the Green-Kubo (GK) formula which relates the ensemble average of the auto-correlation of the stress/pressure tensor to $\eta$. This can be done in a fully equilibrated simulation which is in contrast to the two preceding non-equilibrium methods, where momentum flows continuously through the simulation box.

Here is an example input script that calculates the viscosity of liquid Ar via the GK formalism:

```
# Sample LAMMPS input script for viscosity of liquid Ar

units       real
variable    T equal 200.0       # run temperature
variable    Tinit equal 250.0   # equilibration temperature
variable    V equal vol
variable    dt equal 4.0
variable    p equal 400     # correlation length
variable    s equal 5       # sample interval
variable    d equal $p*$s   # dump interval

# convert from LAMMPS real units to SI

variable    kB equal 1.3806504e-23    # [J/K] Boltzmann
variable    atm2Pa equal 101325.0
variable    A2m equal 1.0e-10
variable    fs2s equal 1.0e-15
variable    convert equal ${atm2Pa}*${atm2Pa}*${fs2s}*${A2m}*${A2m}*${A2m}

# setup problem

dimension    3
boundary     p p p
lattice      fcc 5.376 orient x 1 0 0 orient y 0 1 0 orient z 0 0 1
region       box block 0 4 0 4 0 4
create_box   1 box
create_atoms 1 box
mass         1 39.948
pair_style   lj/cut 13.0
pair_coeff   * * 0.2381 3.405
timestep     ${dt}
thermo       $d

# equilibration and thermalization

velocity     all create ${Tinit} 102486 mom yes rot yes dist gaussian
fix          NVT all nvt temp ${Tinit} ${Tinit} 10 drag 0.2
run          8000

# viscosity calculation, switch to NVE if desired

velocity     all create $T 102486 mom yes rot yes dist gaussian
fix          NVT all nvt temp $T $T 10 drag 0.2
#unfix       NVT
#fix         NVE all nve

reset_timestep 0
variable     pxy equal pxy
```

(continues on next page)

(continued from previous page)

```
variable     pxz equal pxz
variable     pyz equal pyz
fix          SS all ave/correlate $s $p $d &
             v_pxy v_pxz v_pyz type auto file S0St.dat ave running
variable     scale equal ${convert}/(${kB}*$T)*$V*$s*${dt}
variable     v11 equal trap(f_SS[3])*${scale}
variable     v22 equal trap(f_SS[4])*${scale}
variable     v33 equal trap(f_SS[5])*${scale}
thermo_style custom step temp press v_pxy v_pxz v_pyz v_v11 v_v22 v_v33
run          100000
variable     v equal (v_v11+v_v22+v_v33)/3.0
variable     ndens equal count(all)/vol
print        "average viscosity: $v [Pa.s] @ $T K, ${ndens} atoms/A^3"
```

The fifth method is related to the above Green-Kubo method, but uses the Einstein formulation, analogous to the Einstein mean-square-displacement formulation for self-diffusivity. The time-integrated momentum fluxes play the role of Cartesian coordinates, whose mean-square displacement increases linearly with time at sufficiently long times.

The sixth is the periodic perturbation method, which is also a non-equilibrium MD method. However, instead of measuring the momentum flux in response to an applied velocity gradient, it measures the velocity profile in response to applied stress. A cosine-shaped periodic acceleration is added to the system via the *fix accelerate/cos* command, and the *compute viscosity/cos* command is used to monitor the generated velocity profile and remove the velocity bias before thermostatting.

**Note**

An article by *(Hess)* discussed the accuracy and efficiency of these methods.

---

**(Daivis and Todd)** Daivis and Todd, Nonequilibrium Molecular Dynamics (book), Cambridge University Press, https://doi.org/10.1017/9781139017848, (2017).

**(Hess)** Hess, B. The Journal of Chemical Physics 2002, 116 (1), 209-217.

### 10.3.9 Calculate diffusion coefficients

The diffusion coefficient $D$ of a material can be measured in at least 2 ways using various options in LAMMPS. See the `examples/DIFFUSE` directory for scripts that implement the 2 methods discussed here for a simple Lennard-Jones fluid model.

The first method is to measure the mean-squared displacement (MSD) of the system, via the *compute msd* command. The slope of the MSD versus time is proportional to the diffusion coefficient. The instantaneous MSD values can be accumulated in a vector via the *fix vector* command, and a line fit to the vector to compute its slope via the *variable slope* function, and thus extract $D$.

The second method is to measure the velocity auto-correlation function (VACF) of the system, via the *compute vacf* command. The time-integral of the VACF is proportional to the diffusion coefficient. The instantaneous VACF values can be accumulated in a vector via the *fix vector* command, and time integrated via the *variable trap* function, and thus extract $D$.

## 10.4 Force fields howto

### 10.4.1 Some general force field considerations

A compact summary of the concepts, definitions, and properties of force fields with explicit bonded interactions (like the ones discussed in this HowTo) is given in *(Gissinger)*.

A force field has 2 parts: the formulas that define its potential functions and the coefficients used for a particular system. To assign parameters it is first required to assign atom types. Those are not only based on the elements, but also on the chemical environment due to the atoms bound to them. This often follows the chemical concept of *functional groups*. Example: a carbon atom bound with a single bond to a single OH-group (alcohol) would be a different atom type than a carbon atom bound to a methyl CH3 group (aliphatic carbon). The atom types usually then determine the non-bonded Lennard-Jones parameters and the parameters for bonds, angles, dihedrals, and impropers. On top of that, partial charges have to be applied. Those are usually independent of the atom types and are determined either for groups of atoms called residues with some fitting procedure based on quantum mechanical calculations, or based on some increment system that add or subtract increments from the partial charge of an atom based on the types of the neighboring atoms.

Force fields differ in the strategies they employ to determine the parameters and charge distribution in how generic or specific they are which in turn has an impact on the accuracy (compare for example CGenFF to CHARMM and GAFF to AMBER). Because of the different strategies, it is not a good idea to use a mix of parameters from different force field *families* (like CHARMM, AMBER, or GROMOS) and that extends to the parameters for the solvent, especially water. The publication describing the parameterization of a force field will describe which water model to use. Changing the water model usually leads to overall worse results (even if it may improve on the water itself).

In addition, one has to consider that *families* of force fields like CHARMM, AMBER, OPLS, or GROMOS have evolved over time and thus provide different *revisions* of the force field parameters. These often corresponds to changes in the functional form or the parameterization strategies. This may also result in changes required for simulation settings like the preferred cutoff or how Coulomb interactions are computed (cutoff, smoothed/shifted cutoff, or long-range with Ewald summation or equivalent). Unless explicitly stated in the publication describing the force field, the Coulomb interaction cannot be chosen at will but must match the revision of the force field. That said, liberties may be taken during the initial equilibration of a system to speed up the process, but not for production simulations.

---

**(Gissinger)** J. R. Gissinger, I. Nikiforov, Y. Afshar, B. Waters, M. Choi, D. S. Karls, A. Stukowski, W. Im, H. Heinz, A. Kohlmeyer, and E. B. Tadmor, J Phys Chem B, 128, 3282-3297 (2024).

### 10.4.2 CHARMM, AMBER, COMPASS, ClassII-xe, DREIDING, and OPLS force fields

Here we only discuss formulas implemented in LAMMPS that correspond to formulas commonly used in the CHARMM, AMBER, COMPASS, and DREIDING force fields. Setting coefficients is done either from special sections in an input data file via the *read_data* command or in the input script with commands like *pair_coeff* or *bond_coeff* and so on. See the *Tools* doc page for additional tools that can use CHARMM, AMBER, or Materials Studio generated files to assign force field coefficients and convert their output into LAMMPS input. LAMMPS input scripts can also be generated by charmm-gui.org.

### CHARMM and AMBER

The CHARMM force field *(MacKerell)* and AMBER force field *(Cornell)* have potential energy function of the form

$$V = \sum_{bonds} E_b + \sum_{angles} E_a + \overbrace{\sum_{dihedral} E_d}^{\substack{\text{charmm}\\ \text{charmmfsw}}} + \sum_{impropers} E_i$$

$$+ \underbrace{\sum_{pairs} (E_{LJ} + E_{coul})}_{\substack{\text{lj/charmm/coul/charmm}\\ \text{lj/charmm/coul/charmm/implicit}\\ \text{lj/charmm/coul/long}\\ \text{lj/charmm/coul/msm}\\ \text{lj/charmmfsw/coul/charmmfsh}\\ \text{lj/charmmfsw/coul/long}}} + \sum_{special} E_s + \sum_{residues} \text{CMAP}(\phi,\psi)$$

The terms are computed by bond styles (relationship between two atoms), angle styles (between 3 atoms) , dihedral/improper styles (between 4 atoms), pair styles (non-covalently bonded pair interactions) and special bonds. The CMAP term (see *fix cmap* command for details) corrects for pairs of dihedral angles ("Correction MAP") to significantly improve the structural and dynamic properties of proteins in crystalline and solution environments *(Brooks)*. The AMBER force field does not include the CMAP term.

The interaction styles listed below compute force field formulas that are consistent with common options in CHARMM or AMBER. See each command's documentation for the formula it computes.

- *bond_style harmonic*
- *angle_style charmm*
- *dihedral_style charmmfsh*
- *dihedral_style charmm*
- *pair_style lj/charmmfsw/coul/charmmfsh*
- *pair_style lj/charmmfsw/coul/long*
- *pair_style lj/charmm/coul/charmm*
- *pair_style lj/charmm/coul/charmm/implicit*
- *pair_style lj/charmm/coul/long*
- *special_bonds charmm*
- *special_bonds amber*

The pair styles compute Lennard Jones (LJ) and Coulombic interactions with additional switching or shifting functions that ramp the energy and/or force smoothly to zero between an inner ($a$) and outer ($b$) cutoff. The older styles with *charmm* (not *charmmfsw* or *charmmfsh*) in their name compute the LJ and Coulombic interactions with an energy switching function (esw) $S(r)$ which ramps the energy smoothly to zero between the inner and outer cutoff. This can cause irregularities in pairwise forces (due to the discontinuous second derivative of energy at the boundaries of the switching region), which in some cases can result in complications in energy minimization and detectable artifacts in MD simulations.

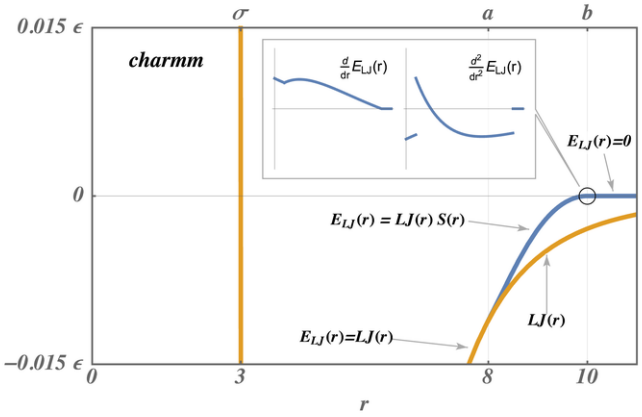
$$LJ(r) = 4\varepsilon \left[ \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]$$

$$C(r) = \frac{Cq_iq_j}{\varepsilon r}$$

$$S(r) = \frac{\left(b^2 - r^2\right)^2 \left(b^2 + 2r^2 - 3a^2\right)}{\left(b^2 - a^2\right)^3}$$

$$E_{LJ}(r) = \begin{cases} LJ(r), & r \le a \\ LJ(r)S(r), & a < r \le b \\ 0, & r > b \end{cases}$$

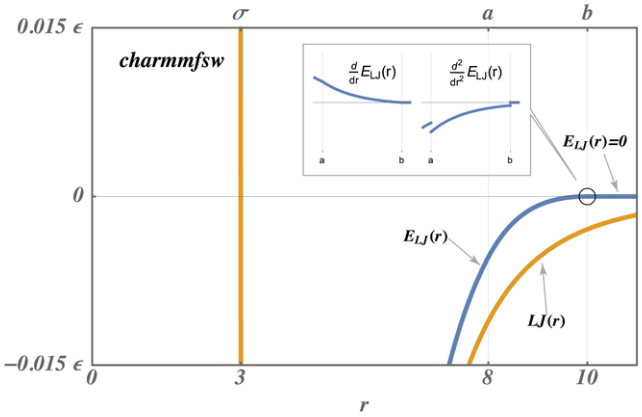
$$E_{coul}(r) = \begin{cases} C(r), & r \le a \\ C(r)S(r), & a < r \le b \\ 0, & r > b \end{cases}$$




The newer styles with *charmmfsw* or *charmmfsh* in their name replace energy switching with force switching (fsw) for LJ interactions and force shifting (fsh) functions for Coulombic interactions *(Steinbach)*

$$E_{LJ}(r) = \begin{cases} 4\varepsilon\sigma^6 \left( \dfrac{\sigma^6 - r^6}{r^{12}} - \dfrac{\sigma^6}{a^6 b^6} + \dfrac{1}{a^3 b^3} \right) & r \le a \\ \dfrac{4\varepsilon\sigma^6 \left( \sigma^6 \left(b^6 - r^6\right)^2 - b^3 r^6 \left(a^3 + b^3\right) \left(b^3 - r^3\right)^2 \right)}{b^6 r^{12} \left(b^6 - a^6\right)} & a < r \le b \\ 0, & r > b \end{cases}$$

$$E_{coul}(r) = \begin{cases} C(r)\dfrac{(b-r)^2}{rb^2}, & r \le b \\ 0, & r > b \end{cases}$$




These styles are used by LAMMPS input scripts generated by https://charmm-gui.org/ *(Brooks)*.

**Note**

For CHARMM, newer *charmmfsw* or *charmmfsh* styles were released in March 2017. We recommend they be used instead of the older *charmm* styles. See discussion of the differences on the *pair charmm* and *dihedral charmm* doc pages.

**Note**

The TIP3P water model is strongly recommended for use with the CHARMM force field. In fact, "using the SPC model with CHARMM parameters is a bad idea" and "to enable TIP4P style water in CHARMM, you would have to write a new pair style" . LAMMPS input scripts generated by Solution Builder on https://charmm-gui.org use TIP3P molecules for solvation. Any other water model can and probably will lead to false conclusions.

### COMPASS

COMPASS is a general force field for atomistic simulation of common organic molecules, inorganic small molecules, and polymers which was developed using ab initio and empirical parameterization techniques *(Sun)*. See the *Tools* page for the msi2lmp tool for creating LAMMPS template input and data files from BIOVIA's Materials Studio files. Please note that the msi2lmp tool is very old and largely unmaintained, so it does not support all features of Materials Studio provided force field files, especially additions during the last decade. You should watch the output carefully and compare results, where possible. See *(Sun)* for a description of the COMPASS force field.

These interaction styles listed below compute force field formulas that are consistent with the COMPASS force field. See each command's documentation for the formula it computes.

- *bond_style class2*
- *angle_style class2*
- *dihedral_style class2*
- *improper_style class2*
- *pair_style lj/class2*
- *pair_style lj/class2/coul/cut*
- *pair_style lj/class2/coul/long*

- *special_bonds lj/coul 0 0 1*

### ClassII-xe

Added in version 30Mar2026.

The computationally efficient simulation of condensed-phase materials such as metals, polymers, and composites can be achieved with fixed-bond force fields. Two popular fixed-bond force fields for such materials are COMPASS *(Sun)* and PCFF *(Maple)*. Both use a Class II-based functional form that incorporates anharmonic vibrational modes via the quartic Taylor-series expansion of the Hamiltonian. In addition, Class II-based force fields add intermolecular coupling via cross-term interactions that also account for more complex vibrational modes. The underlying harmonic bonds and harmonic cross-terms can limit the reliability of the physics, e.g., when calculating mechanical properties. To overcome the harmonic bonding limitations, a Morse bond *(Morse)* can be used in place of the quartic bond, but this improvement when modeling bond dissociation is limited when the cross-terms remain harmonic. To improve the physicality of the response of the cross-terms that couple bond stretch to other higher-order interactions (angles, dihedrals, and impropers), those respective cross-terms also need to be modified.

The cross-term potentials that model bond stretch coupling to higher-order interactions are similar in functional form to the harmonic bonding potential. Thus, the exponential function that *(Morse)* derived for converting a harmonic bonding potential to a bonding potential that accommodates larger bond stretches and can model bond dissociation, can also be implemented into the cross-terms. This defines the naming convention for the ClassII-xe functional form, where x refers to cross-term and e refers to an exponential function, and defines the purpose of the ClassII-xe functional form. The purpose is to allow bond dissociation in a ClassII-based force field via a consistent definition of bond dissociation via the Morse bonding potential and the higher-order cross-term coupling potentials. See *(Kemppainen)* for a description of the ClassII-xe functional form. The interaction styles listed below compute the force field formulas for the ClassII-xe functional form:

- *bond_style morse*
- *angle_style class2xe*
- *dihedral_style class2xe*
- *improper_style class2*
- *pair_style lj/class2*
- *pair_style lj/class2/coul/cut*
- *pair_style lj/class2/coul/long*
- *special_bonds lj/coul 0 0 1*

Since both COMPASS and PCFF are Class II-based, their parameters can be converted into the ClassII-xe functional form. The conversion from ClassII to ClassII-xe functional form requires reparameterizing the cross-terms, which can be accomplished via the LUNAR tool, for which a link is provided on the Pre/Post processing page. LUNAR can be used to build a model from scratch in either COMPASS or PCFF (using its 'atom_typing' and 'all2lmp' modules) and then convert that model to COMPASS-xe or PCFF-xe (using its 'auto_morse_bond_update' module). To establish a consistent naming convention for the new ClassII-xe functional form when converting from a parent force field, the -xe suffix should be added to the parent force field name (e.g. PCFF vs. PCFF-xe).

The usage of the ClassII-xe functional form can only let a bond dissociate; however, to disconnect the dissociated bond and remove the higher-order interactions, other LAMMPS commands are required, such as *fix bond/react* or *fix bond/break*. It was demonstrated that PCFF-xe (without *fix bond/react* or *fix bond/break*) can model a material up to fracture *(Kemppainen)*, allowing for calculation of mechanical properties such such as tensile and shear modulus, tensile and shear yield strength, Poisson's ratio, etc. However, the simulation may crash after fracture if bonds beyond than the processor communication cutoff distance. Thus, post-fracture phenomena cannot be simulated without the usage of *fix bond/react* or *fix bond/break*. Note that using *fix bond/break* can cause instabilities and crashes because a discontinuity is introduced to the energy landscape when a bond is removed. On the other hand, *fix bond/react* can relax high-energy configurations via the `stabilization` keyword, whereby a small local group of atoms involved in

the discontinuity are integrated with *fix nve/limit*. LUNAR can also be used to assist with setting up simulations that include *fix bond/react*.

### DREIDING

DREIDING is a generic force field developed by the Goddard group at Caltech and is useful for predicting structures and dynamics of organic, biological and main-group inorganic molecules. The philosophy in DREIDING is to use general force constants and geometry parameters based on simple hybridization considerations, rather than individual force constants and geometric parameters that depend on the particular combinations of atoms involved in the bond, angle, or torsion terms. DREIDING has an *explicit hydrogen bond term* to describe interactions involving a hydrogen atom on very electronegative atoms (N, O, F). Unlike CHARMM or AMBER, the DREIDING force field has not been parameterized for considering solvents (like water) and has no rules for assigning (partial) charges. That will seriously limit its accuracy when used for simulating systems where those matter.

See *(Mayo)* for a description of the DREIDING force field

The interaction styles listed below compute force field formulas that are consistent with the DREIDING force field. See each command's documentation for the formula it computes.

- *bond_style harmonic*
- *bond_style morse*
- *angle_style cosine/squared*
- *angle_style harmonic*
- *angle_style cosine*
- *angle_style cosine/periodic*
- *dihedral_style charmm*
- *improper_style umbrella*
- *pair_style buck*
- *pair_style buck/coul/cut*
- *pair_style buck/coul/long*
- *pair_style lj/cut*
- *pair_style lj/cut/coul/cut*
- *pair_style lj/cut/coul/long*
- *pair_style hbond/dreiding/lj*
- *pair_style hbond/dreiding/morse*
- *special_bonds dreiding*

### OPLS

OPLS (Optimized Potentials for Liquid Simulations) is a general force field for atomistic simulation of organic molecules in solvent. It was developed by the Jorgensen group at Purdue University and later at Yale University. Multiple versions of the OPLS parameters exist for united atom representations (OPLS-UA) and for all-atom representations (OPLS-AA).

This force field is based on atom types mapped to specific functional groups in organic and biological molecules. Each atom includes a static, partial atomic charge reflecting the oxidation state of the element derived from its bonded neighbors *(Jorgensen)* and computed based on increments determined by the atom type of the atoms bonded to it.

The interaction styles listed below compute force field formulas that are fully or in part consistent with the OPLS style force fields. See each command's documentation for the formula it computes. Some are only compatible with a subset of OPLS interactions.

- *bond_style harmonic*
- *angle_style harmonic*
- *dihedral_style opls*
- *improper_style cvff*
- *improper_style fourier*
- *improper_style harmonic*
- *pair_style lj/cut/coul/cut*
- *pair_style lj/cut/coul/long*
- *pair_modify geometric*
- *special_bonds lj/coul 0.0 0.0 0.5*

---

**(MacKerell)** MacKerell, Bashford, Bellott, Dunbrack, Evanseck, Field, Fischer, Gao, Guo, Ha, et al (1998). J Phys Chem, 102, 3586 . https://doi.org/10.1021/jp973084f

**(Cornell)** Cornell, Cieplak, Bayly, Gould, Merz, Ferguson, Spellmeyer, Fox, Caldwell, Kollman (1995). JACS 117, 5179-5197. https://doi.org/10.1021/ja00124a002

**(Steinbach)** Steinbach, Brooks (1994). J Comput Chem, 15, 667. https://doi.org/10.1002/jcc.540150702

**(Brooks)** Brooks, et al (2009). J Comput Chem, 30, 1545. https://onlinelibrary.wiley.com/doi/10.1002/jcc.21287

**(Sun)** Sun (1998). J. Phys. Chem. B, 102, 7338-7364. https://doi.org/10.1021/jp980939v

**(Mayo)** Mayo, Olfason, Goddard III (1990). J Phys Chem, 94, 8897-8909. https://doi.org/10.1021/j100389a010

**(Jorgensen)** Jorgensen, Tirado-Rives (1988). J Am Chem Soc, 110, 1657-1666. https://doi.org/10.1021/ja00214a001

**(Maple)** Maple, Journal of Computational Chemistry, 15, 162-182 (1994). https://doi.org/10.1002/jcc.540150207

**(Morse)** Morse, Physical review, 34, 57 (1929). https://doi.org/10.1103/PhysRev.34.57

**(Kemppainen)** Kemppainen, npj Computational Materials 11, 341 (2025). https://doi.org/10.1038/s41524-025-01838-5

### 10.4.3 AMBER to LAMMPS Tutorial

**written by Arun Srikanth Sridhar**

Added in version 11Feb2026.

AMBER2LAMMPS is a Python utility that converts AMBER topology (`.prmtop`) and coordinate/charge (`.crd`) files into LAMMPS data and parameter files. It provides both a command-line interface and a Python API with built-in validation. This tutorial provides a detailed workflow with examples using the AMBER2LAMMPS python utility. The legacy scripts previously distributed in `tools/amber2lmp` have been removed due to their reliance on Python 2 and lack of maintenance.

This tutorial assumes familiarity with molecular dynamics simulations and molecular mechanics force fields, including atom typing, bond topologies, partial charge assignment, and different force field families (AMBER, CHARMM, GROMOS, OPLS/AA). For an introduction to force field concepts, see *Some general force field considerations* before starting this tutorial. AMBER offers various force fields for different biomolecules, including ff19SB/ff14SB (proteins), Bsc1/OL3 (DNA/RNA), GLYCAM06 (carbohydrates), and Lipid17 (lipids), alongside the General AMBER

Force Field (GAFF/GAFF2) for small organic molecules (see https://ambermd.org/AmberModels.php and Wang, J., et al., J. Comput. Chem., 25, 1157-1174 (2004) for GAFF force field development).

**Important Force Field Compatibility Warning**

Different force field families (AMBER, CHARMM, GROMOS, OPLS/AA) use distinct atom typing schemes, charge assignment methods, and parameterization strategies. **Never mix and match parameters from different force field families; this leads to unphysical results and simulation failures.** Always use a consistent force field family throughout your system.


- *What the AMBER2LAMMPS tool does*
- *AMBER2LAMMPS homepage and download*
- *Requirements*
- *Installing dependencies*
  - *AmberTools*
  - *Python packages*
  - *Open Babel (optional, for SMILES to PDB conversion)*
- *Command Reference*
- *Conversion Process*
  - *Charge normalization*
- *Workflow Examples*
- *Prepare input files (ethanol example)*
  - *Convert SMILES to PDB (optional)*
  - *Generate AMBER topology and coordinates*
- *Basic conversion workflow*
  - *LAMMPS input script*
  - *CLI usage with LAMMPS execution*
    - *Additional CLI examples*
  - *Python API usage with LAMMPS execution*
    - *Additional API examples*
- *Validation of AMBER2LAMMPS*
- *Getting Help*
- *Citation*

### What the AMBER2LAMMPS tool does

This tool helps you run molecular dynamics simulations in **LAMMPS** when you have your molecular system set up in **AMBER** format. AMBER2LAMMPS uses AmberTools utilities (antechamber and tleap) to perform the conversion.

**Typical workflow:**

1. Start with a molecular structure (PDB file or SMILES string to be converted to PDB)
2. Use AMBER tools to create AMBER files (`.prmtop`, `.crd`)
3. **Convert AMBER files to LAMMPS format** using this tool
4. Run your simulation in LAMMPS

**What gets converted:**

- ``**.prmtop**`` (topology file): Contains bonds, angles, atom types, and force field parameters -> LAMMPS data and parameter files
- ``**.crd**`` (coordinates file): Contains atomic positions -> LAMMPS coordinates

**What you get as output:**

- **LAMMPS data file** (e.g., `data.lammps`): Contains atomic coordinates, box dimensions, and molecular topology
- **LAMMPS parameter file** (e.g., `parm.lammps`): Contains force field parameters for bonds, angles, and non-bonded interactions

---

### AMBER2LAMMPS homepage and download

AMBER2LAMMPS is developed and maintained outside the LAMMPS repository. Download its source code from its GitHub project page or clone its repository from GitHub:

```
# direct download
curl -L -o AMBER2LAMMPS-main.tar.gz https://github.com/askforarun/AMBER2LAMMPS/archive/
↪refs/heads/main.tar.gz
tar -xzvvf AMBER2LAMMPS-main.tar.gz
cd AMBER2LAMMPS-main

# clone repository
git clone https://github.com/askforarun/AMBER2LAMMPS.git
cd AMBER2LAMMPS
```

---

### Requirements

**Platform Compatibility**

AMBER2LAMMPS works on all major platforms:

- **Linux**: Full support (Ubuntu, CentOS, Fedora, etc.)
- **macOS**: Full support (Intel and Apple Silicon)
- **Windows**: Support via WSL2 or Git Bash

**System Requirements**

- **Structure**: PDB file (or a SMILES string that you can convert to PDB).

- **AmberTools utilities**: `antechamber` and `tleap` to build `.prmtop` and `.crd`.
- **Python packages**: `parmed` and `numpy`.
- **LAMMPS**: On your `PATH` (`which lmp` or `lmp -help`) and built with `MOLECULE`, `KSPACE`, and `EXTRA-MOLECULE` packages.
- **Optional**: Open Babel (`obabel`) if starting from SMILES.

### Installing dependencies

#### AmberTools

Install from https://ambermd.org/GetAmber.php#ambertools and activate the environment:

```
conda activate Ambertools23  # or your AmberTools environment
```

To see the available force fields, run antechamber -h

#### Python packages

Using conda (recommended):

```
conda install -c conda-forge parmed numpy
```

Using pip:

```
pip install --user parmed numpy
```

#### Open Babel (optional, for SMILES to PDB conversion)

Using conda (recommended):

```
conda install -c conda-forge openbabel
```

Using pip:

```
pip install --user openbabel
```

Using system package manager:

```
# Ubuntu/Debian
sudo apt-get install openbabel

# Fedora/Red Hat
sudo dnf install openbabel

# macOS
brew install open-babel
```

### Command Reference

| Argument | Required? | Description |
|---|---|---|
| **Input Files** | | |
| `topology` | yes | AMBER topology file (`.prmtop`) |
| `crd` | yes | AMBER coordinate file (`.crd`) |
| **Output Files** | | |
| `data_file` | yes | Output LAMMPS data filename |
| `param_file` | yes | Output LAMMPS parameter filename |
| **Options** | | |
| `-b, --buffer` | optional | Vacuum padding (Angstrom) for the simulation box. Default: `3.8`. |
| `--charge` | yes | Target net charge (integer). Applies a uniform offset to every atom to reach this charge (1e-6 tolerance). |
| | | Does not add counterions; add those with the AMBER tools before conversion. |
| `--keep-temp` | optional | Keep temporary files (bonds.txt, angles.txt, dihedrals.txt, pairs.txt) after conversion. Default: `False`. |
| `--verbose` | optional | Print step-by-step progress, counts, and box size. Default: `False`. |
| `-h, --help` | optional | Show help message. |

### Conversion Process

The converter implements the following sequence (see `amber_to_lammps.py`):

1. **Input validation**: `validate_files` checks that `topology` and `crd` exist and are readable. The CLI performs this validation automatically, while the Python API requires manual validation if desired.
2. **AMBER topology load**: ParmEd reads atoms, bonds, angles, and dihedrals from `.prmtop`; counts are printed with `--verbose`.
3. **Atom typing and masses**: Atom types and masses are extracted from the topology file.
4. **Coordinates and charges**: Coordinates are read from the CRD file and charges from the topology file.
5. **Box creation (``–buffer``)**: A bounding box around the coordinates is expanded by the buffer on all sides.
6. **Charge normalization**: Charges are shifted uniformly to achieve the target net charge specified by `--charge`.
7. **Non-bonded parameters**: Lennard-Jones parameters are extracted from the topology and become `pair_coeff` entries in the parameter file.
8. **Topology terms**: Bonds, angles, and dihedrals are exported via ParmEd to temporary files and written to the LAMMPS data/parameter files.
9. **Cleanup**: Temporary helper files (`bonds.txt`, `angles.txt`, `dihedrals.txt`, `pairs.txt`) are removed unless `--keep-temp` is specified.
10. **Verbose diagnostics**: With `--verbose`, the script reports counts, box extents after buffering, and the final total charge.

### Charge normalization

**AMBER Charge Schemes:** AMBER uses various charge methods including:

- **RESP** (Restrained Electrostatic Potential): Derived from quantum mechanical electrostatic potential
- **AM1-BCC**: Semi-empirical charges with bond charge corrections

- **CM5** or **CM1A**: Charge models based on atomic charges

In this tutorial, we will use AM1-BCC charges.

**Charge Normalization in AMBER2LAMMPS:**

- Most systems should be overall neutral for PME to converge; small residual charges (+/-0.003) from AM1-BCC (or other methods) calculated from antechamber are common.
- For neutral molecules, run with `--charge 0` so AMBER2LAMMPS applies a uniform offset that removes the residual charge; this prevents the error from scaling up when the system is replicated in LAMMPS.
- For intentionally charged species (e.g., protonated or deprotonated), add counterions in tleap/packmol before conversion. AMBER2LAMMPS never adds ions; it only shifts existing charges to your requested total.
- How the flag works: - `--charge 0`: uniform shift makes the summed charge 0 within 1e-6. - `--charge +1` (or any integer): uniform shift makes the total that integer within 1e-6. - The same constant is added to every atom, so relative charge differences are preserved.

**Example: If your system has net charge +0.003 and you specify**
`--charge 0`, each atom's charge will be reduced by `(0.003 / number_of_atoms)` to reach neutrality.

---

### Workflow Examples

### Prepare input files (ethanol example)

### Convert SMILES to PDB (optional)

If you start from a SMILES string, generate a PDB with `obabel`:

```
obabel -:CCO -h -opdb -O ethanol.pdb --gen3d  # adds hydrogens, recommended
obabel -:c1ccccc1 -h -opdb -O benzene.pdb --gen3d
obabel -:"CC(=O)OC1=CC=CC=C1C(=O)O" -h -opdb -O aspirin.pdb --gen3d
```

### Generate AMBER topology and coordinates

Assuming `ethanol.pdb` is your structure:

1. Generate a MOL2 file with charges:

   ```
   antechamber -j 4 -at gaff2 -dr yes -fi pdb -fo mol2 \
      -i ethanol.pdb -o ethanol.mol2 -c bcc
   ```

   The `-c bcc` assigns AM1-BCC charges; use the `-at` option to choose the force field.

2. Create `tleap.in`:

   ```
   source leaprc.gaff2
   ETH = loadmol2 ethanol.mol2
   check ETH
   saveamberparm SUS ethanol.prmtop ethanol.crd
   quit
   ```

   leaprc.gaff2 loads the gaff2 force field parameters that were used in step 1

3. Run tleap and inspect `leap.log` for errors:

```
tleap -f tleap.in
```

Outputs: `ethanol.prmtop` and `ethanol.crd`.

---

### Basic conversion workflow

#### LAMMPS input script

Save the following LAMMPS commands in a file called `example_lammps_input.lmp`:

```
# LAMMPS Input Script for Converted AMBER System

units real
dimension 3
boundary p p p
atom_style full

read_data data.lammps

pair_style      lj/cut/coul/long 9 9
bond_style      harmonic
angle_style     harmonic
dihedral_style  fourier
special_bonds lj 0.0 0.0 0.5 coul 0.0 0.0 0.83333333

include parm.lammps

thermo_style custom ebond eangle edihed eimp epair evdwl ecoul elong etail pe
run 0
```

The above script includes the parameter file via `include parm.lammps` and loads the coordinate data via `read_data data.lammps`.

#### CLI usage with LAMMPS execution

Ensure amber_to_lammps.py is executable and the input files are accessible.

Run the converter:

```
python3 amber_to_lammps.py data.lammps parm.lammps \
   ethanol.prmtop ethanol.crd --charge 0 --verbose -b 4.5
```

Outputs: `data.lammps` (coordinates/topology) and `parm.lammps`

Run with LAMMPS:

```
lmp -in example_lammps_input.lmp
```

#### Additional CLI examples

```
# Custom buffer and verbose logging (adds 5 Angstroms of padding)
python3 amber_to_lammps.py my_data.lammps my_params.lammps \
```

(continues on next page)

(continued from previous page)

```
   ethanol.prmtop ethanol.crd --charge 0 --verbose -b 5.0

# Custom output names
python3 amber_to_lammps.py system.data system.parm system.prmtop \
   system.crd --charge 0

# Minimal output without verbose logging
python3 amber_to_lammps.py small.data small.parm \
   ethanol.prmtop ethanol.crd --charge 0 -b 3.0

# Using absolute paths with custom buffer
python3 amber_to_lammps.py /home/user/lammps/output/data.lammps /home/user/lammps/output/
↪param.lammps /home/user/amber/topology.prmtop /home/user/amber/coords.crd --charge 0 -
↪b 4.5

# Keep temporary files for debugging
python3 amber_to_lammps.py debug_data.lammps debug_parm.lammps molecule.prmtop molecule.
↪crd --charge 0 --keep-temp --verbose
```

Make sure to rename the files in example_lammps_input.lmp to match the names of the files generated by the conversion when custom file names are used.

### Python API usage with LAMMPS execution

This example script uses the *LAMMPS Python module* to execute LAMMPS directly from Python.

```
from amber_to_lammps import amber2lammps, validate_files
from lammps import lammps

validate_files('ethanol.prmtop', 'ethanol.crd')

amber2lammps(
    data_file='data.lammps',
    param_file='parm.lammps',
    topology='ethanol.prmtop',
    crd='ethanol.crd',
    charge=0,
    buffer=3.8,
    verbose=True,
)

lmp = lammps()
lmp.file('example_lammps_input.lmp')
lmp.close()
```

### Additional API examples

```
# Example 1: Basic conversion without validation
from amber_to_lammps import amber2lammps

amber2lammps(
```

(continues on next page)

(continued from previous page)

```
    data_file='benzene.data',
    param_file='benzene.parm',
    topology='benzene.prmtop',
    crd='benzene.crd',
    charge=0
)

# Example 2: Custom buffer and verbose output
amber2lammps(
    data_file='ethanol.data',
    param_file='ethanol.parm',
    topology='ethanol.prmtop',
    crd='ethanol.crd',
    charge=0,
    buffer=5.0,
    verbose=True,
    keep_temp=True  # Keep temporary files for inspection
)

# Example 3: Batch processing multiple molecules
from amber_to_lammps import amber2lammps, validate_files

molecules = ['ethanol', 'benzene', 'aspirin']

for mol in molecules:
    validate_files(f'{mol}.prmtop', f'{mol}.crd')
    amber2lammps(
        data_file=f'{mol}.data',
        param_file=f'{mol}.parm',
        topology=f'{mol}.prmtop',
        crd=f'{mol}.crd',
        charge=0,
        verbose=True
    )
```

### Validation of AMBER2LAMMPS

AMBER2LAMMPS has been validated against InterMol output. See the project page for details: https://github.com/askforarun/AMBER2LAMMPS

### Getting Help

- **Submit Issues:** https://github.com/askforarun/AMBER2LAMMPS/issues
- **Feature Requests:** Use GitHub Issues or Discussions
- **Questions:** Use GitHub Discussions or Issues

### Citation

If you use the AMBER2LAMMPS tool in your research, please cite it as:

**DOI:** 10.5281/zenodo.18114886

### 10.4.4 AMOEBA and HIPPO force fields

The AMOEBA and HIPPO polarizable force fields were developed by Jay Ponder's group at the U Washington at St Louis. The LAMMPS implementation is based on Fortran 90 code provided by the Ponder group in their Tinker MD software.

The current implementation (July 2022) of AMOEBA in LAMMPS matches the version discussed in *(Ponder)*, *(Ren)*, and *(Shi)*. Likewise the current implementation of HIPPO in LAMMPS matches the version discussed in *(Rackers)*.

These force fields can be used when polarization effects are desired in simulations of water, organic molecules, and biomolecules including proteins, provided that parameterizations (Tinker PRM force field files) are available for the systems you are interested in. Files in the LAMMPS potentials directory with a "amoeba" or "hippo" suffix can be used. The Tinker distribution and website have additional force field files as well: https://github.com/TinkerTools/tinker/tree/release/params.

Note that currently, HIPPO can only be used for water systems, but HIPPO files for a variety of small organic and biomolecules are in preparation by the Ponder group. Those force field files will be included in the LAMMPS distribution when available.

To use the AMOEBA or HIPPO force fields, a simulation must be 3d, and fully periodic or fully non-periodic, and use an orthogonal (not triclinic) simulation box.

---

The AMOEBA and HIPPO force fields contain the following terms in their energy (U) computation. Further details for AMOEBA equations are in *(Ponder)*, further details for the HIPPO equations are in *(Rackers)*.

$$
\begin{aligned}
U &= U_{intermolecular} + U_{intramolecular} \\
U_{intermolecular} &= U_{hal} + U_{repulsion} + U_{dispersion} + U_{multipole} + U_{polar} + U_{qxfer} \\
U_{intramolecular} &= U_{bond} + U_{angle} + U_{torsion} + U_{oop} + U_{b\theta} + U_{UB} + U_{pitorsion} + U_{bitorsion}
\end{aligned}
$$

For intermolecular terms, the AMOEBA force field includes only the $U_{hal}$, $U_{multipole}$, $U_{polar}$ terms. The HIPPO force field includes all but the $U_{hal}$ term. In LAMMPS, these are all computed by the *pair_style amoeba or hippo* command. Note that the $U_{multipole}$ and $U_{polar}$ terms in this formula are not the same for the AMOEBA and HIPPO force fields.

For intramolecular terms, the $U_{bond}$, $U_{angle}$, $U_{torsion}$, $U_{oop}$ terms are computed by the *bond_style class2 angle_style amoeba*, *dihedral_style fourier*, and *improper_style amoeba* commands respectively. The *angle_style amoeba* command includes the $U_{b\theta}$ bond-angle cross term, and the $U_{UB}$ term for a Urey-Bradley bond contribution between the I,K atoms in the IJK angle.

The $U_{pitorsion}$ term is computed by the *fix amoeba/pitorsion* command. It computes 6-body interaction between a pair of bonded atoms which each have 2 additional bond partners.

The $U_{bitorsion}$ term is computed by the *fix amoeba/bitorsion* command. It computes 5-body interaction between two 4-body torsions (dihedrals) which overlap, having 3 atoms in common.

These command doc pages have additional details on the terms they compute:

- *pair_style amoeba or hippo*
- *bond_style class2*
- *angle_style amoeba*
- *dihedral_style fourier*
- *improper_style amoeba*
- *fix amoeba/pitorsion*
- *fix amoeba/bitorsion*

To use the AMOEBA or HIPPO force fields in LAMMPS, use commands like the following appropriately in your input script. The only change needed for AMOEBA vs HIPPO simulation is for the *pair_style* and *pair_coeff* commands, as shown below. See examples/amoeba for example input scripts for both AMOEBA and HIPPO.

```
units             real                              # required
atom_style        amoeba
bond_style        class2                            # CLASS2 package
angle_style       amoeba
dihedral_style    fourier                           # EXTRA-MOLECULE package
improper_style    amoeba
                                                    # required per-atom data
fix               amtype all property/atom i_amtype ghost yes
fix               extra all property/atom &
                  i_amgroup i_ired i_xaxis i_yaxis i_zaxis d_pval ghost yes
fix               polaxe all property/atom i_polaxe

fix               pit all amoeba/pitorsion          # PiTorsion terms in FF
fix_modify        pit energy yes
                                                    # Bitorsion terms in FF
fix               bit all amoeba/bitorsion bitorsion.ubiquitin.data
fix_modify        bit energy yes

read_data         data.ubiquitin fix amtype NULL "Tinker Types" &
                  fix pit "pitorsion types" "PiTorsion Coeffs" &
                  fix pit pitorsions PiTorsions &
                  fix bit bitorsions BiTorsions

pair_style        amoeba                            # AMOEBA FF
pair_coeff        * * amoeba_ubiquitin.prm amoeba_ubiquitin.key

pair_style        hippo                             # HIPPO FF
pair_coeff        * * hippo_water.prm hippo_water.key

special_bonds     lj/coul 0.5 0.5 0.5 one/five yes     # 1-5 neighbors
```

The data file read by the *read_data* command should be created by the tools/tinker/tinker2lmp.py conversion program described below. It will create a section in the data file with the header "Tinker Types". A *fix property/atom* command for the data must be specified before the read_data command. In the example above the fix ID is *amtype*.

Similarly, if the system you are simulating defines AMOEBA/HIPPO pitorsion or bitorsion interactions, there will be entries in the data file for those interactions. They require a *fix amoeba/pitortion* and *fix amoeba/bitorsion* command be defined. In the example above, the IDs for these two fixes are *pit* and *bit*.

Of course, if the system being modeled does not have one or more of the following – bond, angle, dihedral, improper, pitorsion, bitorsion interactions – then the corresponding style and fix commands above do not need to be used. See the example scripts in examples/amoeba for water systems as examples; they are simpler than what is listed above.

The two *fix property/atom* commands with IDs (in the example above) *extra* and *polaxe* are also needed to define internal per-atom quantities used by the AMOEBA and HIPPO force fields.

The *pair_coeff* command used for either the AMOEBA or HIPPO force field takes two arguments for Tinker force field files, namely a PRM and KEY file. The keyfile can be specified as NULL and default values for a various settings will be used. Note that these 2 files are meant to allow use of native Tinker files as-is. However LAMMPS does not support all the options which can be included in a Tinker PRM or KEY file. See specifics below.

A *special_bonds* command with the *one/five* option is required, since the AMOEBA/HIPPO force fields define weighting factors for not only 1-2, 1-3, 1-4 interactions, but also 1-5 interactions. This command will trigger a per-atom list of 1-5 neighbors to be generated. The AMOEBA and HIPPO force fields define their own custom weighting factors for all the 1-2, 1-3, 1-4, 1-5 terms which in the Tinker PRM and KEY files; they can be different for different terms in the force field.

In addition to the list above, these command doc pages have additional details:

- *atom_style amoeba*
- *fix property/atom*
- *special_bonds*

---

Tinker PRM and KEY files

A Tinker PRM file is composed of sections, each of which has multiple lines. This is the list of PRM sections LAMMPS knows how to parse and use. Any other sections are skipped:

- Angle Bending Parameters
- Atom Type Definitions
- Atomic Multipole Parameters
- Bond Stretching Parameters
- Charge Penetration Parameters
- Charge Transfer Parameters
- Dipole Polarizability Parameters
- Dispersion Parameters
- Force Field Definition
- Literature References
- Out-of-Plane Bend Parameters
- Pauli Repulsion Parameters
- Pi-Torsion Parameters
- Stretch-Bend Parameters
- Torsion-Torsion Parameters
- Torsional Parameters
- Urey-Bradley Parameters
- Van der Waals Pair Parameters
- Van der Waals Parameters

A Tinker KEY file is composed of lines, each of which has a keyword followed by zero or more parameters. This is the list of keywords LAMMPS knows how to parse and use in the same manner Tinker does. Any other keywords are skipped. The value in parenthesis is the default value for the keyword if it is not specified, or if the keyfile in the *pair_coeff* command is specified as NULL:

- a-axis (0.0)
- b-axis (0.0)
- c-axis (0.0)

- ctrn-cutoff (6.0)
- ctrn-taper (0.9 * ctrn-cutoff)
- cutoff
- delta-halgren (0.07)
- dewald (no long-range dispersion unless specified)
- dewald-alpha (0.4)
- dewald-cutoff (7.0)
- dispersion-cutoff (9.0)
- dispersion-taper (9.0 * dispersion-cutoff)
- dpme-grid
- dpme-order (4)
- ewald (no long-range electrostatics unless specified)
- ewald-alpha (0.4)
- ewald-cutoff (7.0)
- gamma-halgren (0.12)
- mpole-cutoff (9.0)
- mpole-taper (0.65 * mpole-cutoff)
- pcg-guess (enabled by default)
- pcg-noguess (disable pcg-guess if specified)
- pcg-noprecond (disable pcg-precond if specified)
- pcg-peek (1.0)
- pcg-precond (enabled by default)
- pewald-alpha (0.4)
- pme-grid
- pme-order (5)
- polar-eps (1.0e-6)
- polar-iter (100)
- polar-predict (no prediction operation unless specified)
- ppme-order (5)
- repulsion-cutoff (6.0)
- repulsion-taper (0.9 * repulsion-cutoff)
- taper
- usolve-cutoff (4.5)
- usolve-diag (2.0)
- vdw-cutoff (9.0)
- vdw-taper (0.9 * vdw-cutoff)

Tinker2lmp.py tool

This conversion tool is found in the tools/tinker directory. As shown in examples/amoeba/README, these commands produce the data files found in examples/amoeba, and also illustrate all the options available to use with the tinker2lmp.py script:

```
python tinker2lmp.py -xyz water_dimer.xyz -amoeba amoeba_water.prm -data data.water_
↪dimer.amoeba                # AMOEBA non-periodic system
python tinker2lmp.py -xyz water_dimer.xyz -hippo hippo_water.prm -data data.water_dimer.
↪hippo                   # HIPPO non-periodic system
python tinker2lmp.py -xyz water_box.xyz -amoeba amoeba_water.prm -data data.water_box.
↪amoeba -pbc 18.643 18.643 18.643    # AMOEBA periodic system
python tinker2lmp.py -xyz water_box.xyz -hippo hippo_water.prm -data data.water_box.
↪hippo -pbc 18.643 18.643 18.643       # HIPPO periodic system
python tinker2lmp.py -xyz ubiquitin.xyz -amoeba amoeba_ubiquitin.prm -data data.
↪ubiquitin.new -pbc 54.99 41.91 41.91 -bitorsion bitorsion.ubiquitin.data.new   #␣
↪system with bitorsions
```

Switches and their arguments may be specified in any order.

The -xyz switch is required and specifies an input XYZ file as an argument. The format of this file is an extended XYZ format defined and used by Tinker for its input. Example *.xyz files are in the examples/amoeba directory. The file lists the atoms in the system. Each atom has the following information: Tinker species name (ignored by LAMMPS), xyz coordinates, Tinker numeric type, and a list of atom IDs the atom is bonded to.

Here is more information about the extended XYZ format defined and used by Tinker, and links to programs that convert standard PDB files to the extended XYZ format:

- https://dasher.wustl.edu/tinker/distribution/doc/sphinx/tinker/_build/html/text/file-types.html
- https://openbabel.org/docs/FileFormats/Tinker_XYZ_format.html
- https://github.com/emleddin/pdbxyz-xyzpdb
- https://github.com/TinkerTools/tinker/blob/release/source/pdbxyz.f

The -amoeba or -hippo switch is required. It specifies an input AMOEBA or HIPPO PRM force field file as an argument. This should be the same file used by the *pair_style* command in the input script.

The -data switch is required. It specifies an output file name for the LAMMPS data file that will be produced.

For periodic systems, the -pbc switch is required. It specifies the periodic box size for each dimension (x,y,z). For a Tinker simulation these are specified in the KEY file.

The -bitorsion switch is only needed if the system contains Tinker bitorsion interactions. The data for each type of bitorsion interaction will be written to the specified file, and read by the *fix amoeba/bitorsion* command. The data includes 2d arrays of values to which splines are fit, and thus is not compatible with the LAMMPS data file format.

---

**(Ponder)** Ponder, Wu, Ren, Pande, Chodera, Schnieders, Haque, Mobley, Lambrecht, DiStasio Jr, M. Head-Gordon, Clark, Johnson, T. Head-Gordon, J Phys Chem B, 114, 2549-2564 (2010).

**(Rackers)** Rackers, Silva, Wang, Ponder, J Chem Theory Comput, 17, 7056-7084 (2021).

**(Ren)** Ren and Ponder, J Phys Chem B, 107, 5933 (2003).

**(Shi)** Shi, Xia, Zhang, Best, Wu, Ponder, Ren, J Chem Theory Comp, 9, 4046, 2013.

### 10.4.5 TIP3P water model

The TIP3P water model as implemented in CHARMM *(MacKerell)* specifies a 3-site rigid water molecule with charges and Lennard-Jones parameters assigned to each of the three atoms.

One suitable pair style with cutoff Coulomb would for instance be:

- *pair_style lj/cut/coul/cut*

These commands are examples for a long-range Coulomb model:

- *pair_style lj/cut/coul/long*
- *pair_style lj/cut/coul/long/soft*
- *kspace_style pppm*
- *pair_style lj/long/coul/long*
- *kspace_style pppm/disp*

And these pair styles are compatible with the CHARMM force field:

- *pair_style lj/charmm/coul/charmm*
- *pair_style lj/charmm/coul/long*
- *pair_style lj/charmmfsw/coul/long*

In LAMMPS the *fix shake or fix rattle* command can be used to hold the two O-H bonds and the H-O-H angle rigid. A bond style of *harmonic* and an angle style of *harmonic* or *charmm* should also be used. In case of rigid bonds also bond style *zero* and angle style *zero* can be used.

The table below lists the force field parameters (in real *units*) to for the water molecule atoms to run a rigid or flexible TIP3P-CHARMM model with a cutoff, the original 1983 TIP3P model *(Jorgensen)*, or a TIP3P model with parameters optimized for a long-range Coulomb solver (e.g. Ewald or PPPM in LAMMPS) *(Price)*. The K values can be used if a flexible TIP3P model (without fix shake) is desired, for rigid bonds/angles they are ignored.

| Parameter | TIP3P-CHARMM | TIP3P (original) | TIP3P (Ewald) |
|---|---|---|---|
| O mass (amu) | 15.9994 | 15.9994 | 15.9994 |
| H mass (amu) | 1.008 | 1.008 | 1.008 |
| O charge ($e$) | -0.834 | -0.834 | -0.834 |
| H charge ($e$) | 0.417 | 0.417 | 0.417 |
| LJ $\varepsilon$ of OO (kcal/mole) | 0.1521 | 0.1521 | 0.1020 |
| LJ $\sigma$ of OO (Å) | 3.1507 | 3.1507 | 3.188 |
| LJ $\varepsilon$ of HH (kcal/mole) | 0.0460 | 0.0 | 0.0 |
| LJ $\sigma$ of HH (Å) | 0.4 | 1.0 | 1.0 |
| LJ $\varepsilon$ of OH (kcal/mole) | 0.0836 | 0.0 | 0.0 |
| LJ $\sigma$ of OH (Å) | 1.7753 | 1.0 | 1.0 |
| K of OH bond (kcal/mole/Å$^2$) | 450 | 450 | 450 |
| $r_0$ of OH bond (Å) | 0.9572 | 0.9572 | 0.9572 |
| K of HOH angle (kcal/mole) | 55.0 | 55.0 | 55.0 |
| $\theta_0$ of HOH angle | 104.52° | 104.52° | 104.52° |

Below is the code for a LAMMPS input file and a molecule file (`tip3p.mol`) of TIP3P water for use with the *molecule command* demonstrating how to set up a small bulk water system for TIP3P with rigid bonds. For simplicity and speed the example uses a cutoff Coulomb. Most production simulations require long-range Coulomb instead.

```
units real
atom_style full
region box block -5 5 -5 5 -5 5
create_box 2 box bond/types 1 angle/types 1 &
            extra/bond/per/atom 2 extra/angle/per/atom 1 extra/special/per/atom 2

mass 1 15.9994
mass 2 1.008

pair_style lj/cut/coul/cut 8.0
pair_coeff 1 1 0.1521 3.1507
pair_coeff 2 2 0.0    1.0

bond_style zero
bond_coeff 1 0.9574

angle_style zero
angle_coeff 1 104.52

molecule water tip3p.mol
create_atoms 0 random 33 34564 NULL mol water 25367 overlap 1.33

fix rigid all shake 0.001 10 10000 b 1 a 1
minimize 0.0 0.0 1000 10000

reset_timestep 0
timestep 1.0
velocity all create 300.0 5463576
fix integrate all nvt temp 300 300 100.0

thermo_style custom step temp press etotal pe

thermo 1000
run 20000
write_data tip3p.data nocoeff
```

```
# Water molecule. TIP3P geometry

3 atoms
2 bonds
1 angles

Coords

1    0.00000  -0.06556   0.00000
2    0.75695   0.52032   0.00000
3   -0.75695   0.52032   0.00000

Types

1        1   # O
2        2   # H
```

(continues on next page)

(continued from previous page)

```
3        2   # H

Charges

1       -0.834
2        0.417
3        0.417

Bonds

1   1      1      2
2   1      1      3

Angles

1   1      2      1      3

Shake Flags

1 1
2 1
3 1

Shake Atoms

1 1 2 3
2 1 2 3
3 1 2 3

Shake Bond Types

1 1 1 1
2 1 1 1
3 1 1 1

Special Bond Counts

1 2 0 0
2 1 1 0
3 1 1 0

Special Bonds

1 2 3
2 1 3
3 1 2
```

Wikipedia also has a nice article on water models.

---

**(MacKerell)** MacKerell, Bashford, Bellott, Dunbrack, Evanseck, Field, Fischer, Gao, Guo, Ha, et al, J Phys Chem, 102, 3586 (1998).

**(Jorgensen)** Jorgensen, Chandrasekhar, Madura, Impey, Klein, J Chem Phys, 79, 926 (1983).

**(Price)** Price and Brooks, J Chem Phys, 121, 10096 (2004).

### 10.4.6 TIP4P and OPC water models

The four-point TIP4P rigid water model extends the traditional *three-point TIP3P* model by adding an additional site M, usually massless, where the charge associated with the oxygen atom is placed. This site M is located at a fixed distance away from the oxygen along the bisector of the HOH bond angle. A bond style of *harmonic* and an angle style of *harmonic* or *charmm* should also be used. In case of rigid bonds also bond style *zero* and angle style *zero* can be used. Very similar to the TIP4P model is the OPC water model. It can be realized the same way as TIP4P but has different geometry and force field parameters.

There are two ways to implement TIP4P-like water in LAMMPS:

1. Use a specially written pair style that uses the *TIP3P geometry* without the point M. The point M location is then implicitly derived from the other atoms or each water molecule and used during the force computation. The forces on M are then projected on the oxygen and the two hydrogen atoms. This is computationally very efficient, but the charge distribution in space is only correct within the tip4p labeled styles. So all other computations using charges will "see" the negative charge incorrectly located on the oxygen atom unless they are specially written for using the TIP4P geometry internally as well, e.g. *compute dipole/tip4p*, *fix efield/tip4p*, or *kspace_style pppm/tip4p*.

   This can be done with the following pair styles for Coulomb with a cutoff:

   - *pair_style tip4p/cut*
   - *pair_style lj/cut/tip4p/cut*

   or these commands for a long-range Coulomb treatment:

   - *pair_style tip4p/long*
   - *pair_style lj/cut/tip4p/long*
   - *pair_style lj/long/tip4p/long*
   - *pair_style tip4p/long/soft*
   - *pair_style lj/cut/tip4p/long/soft*
   - *kspace_style pppm/tip4p*
   - *kspace_style pppm/disp/tip4p*

   The bond lengths and bond angles should be held fixed using the *fix shake* or *fix rattle* command, unless a parameterization for a flexible TIP4P model is used. The parameter sets listed below are all for rigid TIP4P model variants and thus the bond and angle force constants are not used and can be set to any legal value; only equilibrium length and angle are used.

2. Use an *explicit 4 point TIP4P geometry* where the oxygen atom carries no charge and the M point no Lennard-Jones interactions. Since *fix shake* or *fix rattle* may not be applied to this kind of geometry, *fix rigid or fix rigid/small* or its thermostatted variants are required to maintain a rigid geometry. This avoids some of the issues with respect to analysis and non-tip4p styles, but it is a more costly force computation (more atoms in the same volume and thus more neighbors in the neighbor lists) and requires a much shorter timestep for stable integration of the rigid body motion. Since no bonds or angles are required, they do not need to be defined and atom style charge would be sufficient for a bulk TIP4P water system. In order to avoid that LAMMPS produces an error due to the massless M site a tiny non-zero mass needs to be assigned.

The table below lists the force field parameters (in real *units*) to for a selection of popular variants of the TIP4P model. There is the rigid TIP4P model with a cutoff *(Jorgensen)*, the TIP4/Ice model *(Abascal1)*, the TIP4P/2005 model *(Abascal2)* and a version of TIP4P parameters adjusted for use with a long-range Coulombic solver (e.g. Ewald or

PPPM in LAMMPS). Note that for implicit TIP4P models the OM distance is specified in the *pair_style* command, not as part of the pair coefficients. Also parameters for the OPC model (*Izadi*) are provided.

| Parameter | TIP4P (original) | TIP4P/Ice | TIP4P/2005 | TIP4P (Ewald) | OPC |
|---|---|---|---|---|---|
| O mass (amu) | 15.9994 | 15.9994 | 15.9994 | 15.9994 | 15.9994 |
| H mass (amu) | 1.008 | 1.008 | 1.008 | 1.008 | 1.008 |
| O or M charge ($e$) | -1.040 | -1.1794 | -1.1128 | -1.04844 | -1.3582 |
| H charge ($e$) | 0.520 | 0.5897 | 0.5564 | 0.52422 | 0.6791 |
| LJ $\varepsilon$ of OO (kcal/mole) | 0.1550 | 0.21084 | 0.1852 | 0.16275 | 0.21280 |
| LJ $\sigma$ of OO (Å) | 3.1536 | 3.1668 | 3.1589 | 3.16435 | 3.1660 |
| LJ $\varepsilon$ of HH, MM, OH, OM, HM (kcal/mole) | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| LJ $\sigma$ of HH, MM, OH, OM, HM (Å) | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| $r_0$ of OH bond (Å) | 0.9572 | 0.9572 | 0.9572 | 0.9572 | 0.8724 |
| $\theta_0$ of HOH angle | 104.52° | 104.52° | 104.52° | 104.52° | 103.60° |
| OM distance (Å) | 0.15 | 0.1577 | 0.1546 | 0.1250 | 0.1594 |

Note that the when using a TIP4P pair style, the neighbor list cutoff for Coulomb interactions is effectively extended by a distance 2 * (OM distance), to account for the offset distance of the fictitious charges on O atoms in water molecules. Thus, it is typically best in an efficiency sense to use a LJ cutoff >= Coulomb cutoff + 2*(OM distance), to shrink the size of the neighbor list. This leads to slightly larger cost for the long-range calculation, so you can test the trade-off for your model. The OM distance and the LJ and Coulombic cutoffs are set in the *pair_style lj/cut/tip4p/long* command.

In addition, each TIP4P water molecule's O atom and its two H atoms must be present within the ghost-atom communication cutoff, so that the position of the massless M-site charge can be reconstructed. This is normally ensured automatically by the bond style (which sets a bond-length-based communication cutoff) together with the *neighbor* skin distance. If you use a no-op bond style (e.g. *bond_style zero*) or unusually short cutoffs, LAMMPS will warn that the communication cutoff is shorter than the bond-length-based estimate and the run may abort with an "Out of range atoms" error; in that case enlarge the ghost cutoff with *comm_modify cutoff*.

Below is the code for a LAMMPS input file using the implicit method and the *TIP3P molecule file*. Because the TIP4P charges are different from TIP3P they need to be reset (or the molecule file changed). For simplicity and speed the example uses a cutoff Coulomb. Most production simulations require long-range Coulomb instead.

```
units real
atom_style full
region box block -5 5 -5 5 -5 5
create_box 2 box bond/types 1 angle/types 1 &
           extra/bond/per/atom 2 extra/angle/per/atom 1 extra/special/per/atom 2

mass 1 15.9994
mass 2 1.008

pair_style lj/cut/tip4p/cut 1 2 1 1 0.15 8.0
pair_coeff 1 1 0.1550 3.1536
pair_coeff 2 2 0.0    1.0

bond_style zero
bond_coeff 1 0.9574

angle_style zero
angle_coeff 1 104.52
```

(continues on next page)

(continued from previous page)

```
molecule water tip3p.mol  # this uses the TIP3P geometry
create_atoms 0 random 33 34564 NULL mol water 25367 overlap 1.33
# must change charges for TIP4P
set type 1 charge -1.040
set type 2 charge  0.520

fix rigid all shake 0.001 10 10000 b 1 a 1
minimize 0.0 0.0 1000 10000

reset_timestep 0
timestep 1.0
velocity all create 300.0 5463576
fix integrate all nvt temp 300 300 100.0

thermo_style custom step temp press etotal pe

thermo 1000
run 20000
write_data tip4p-implicit.data nocoeff
```

When constructing an OPC model, we cannot use the `tip3p.mol` file due to the different geometry. Below is a molecule file providing the 3 sites of an implicit OPC geometry for use with TIP4P styles. Note, that the "Shake" and "Special" sections are missing here. Those will be auto-generated by LAMMPS when the molecule file is loaded *after* the simulation box has been created. These sections are required only when the molecule file is loaded *before*.

```
# Water molecule. 3 point geometry for OPC model

3 atoms
2 bonds
1 angles

Coords

1    0.00000  -0.06037   0.00000
2    0.68558   0.50250   0.00000
3   -0.68558   0.50250   0.00000

Types

1        1   # O
2        2   # H
3        2   # H

Charges

1       -1.3582
2        0.6791
3        0.6791

Bonds

1   1      1      2
```

(continues on next page)

(continued from previous page)

```
2   1       1       3

Angles

1   1       2       1       3
```

Below is a LAMMPS input file using the implicit method to implement the OPC model using the molecule file from above and including the PPPM long-range Coulomb solver.

```
units real
atom_style full
region box block -5 5 -5 5 -5 5
create_box 2 box bond/types 1 angle/types 1 &
           extra/bond/per/atom 2 extra/angle/per/atom 1 extra/special/per/atom 2

mass 1 15.9994
mass 2 1.008

pair_style lj/cut/tip4p/long 1 2 1 1 0.1594 12.0
pair_coeff 1 1 0.2128 3.166
pair_coeff 2 2 0.0    1.0

bond_style zero
bond_coeff 1 0.8724

angle_style zero
angle_coeff 1 103.6

kspace_style pppm/tip4p 1.0e-5

molecule water opc3p.mol  # this file has the OPC geometry but is without M
create_atoms 0 random 33 34564 NULL mol water 25367 overlap 1.33

fix rigid all shake 0.001 10 10000 b 1 a 1
minimize 0.0 0.0 1000 10000

reset_timestep 0
timestep 1.0
velocity all create 300.0 5463576
fix integrate all nvt temp 300 300 100.0

thermo_style custom step temp press etotal pe

thermo 1000
run 20000
write_data opc-implicit.data nocoeff
```

Below is the code for a LAMMPS input file using the explicit method and a TIP4P molecule file. Because of using *fix rigid/small* no bonds need to be defined and thus no extra storage needs to be reserved for them, but we need to either switch to atom style full or use *fix property/atom mol* so that fix rigid/small can identify rigid bodies by their molecule ID. Also a *neigh_modify exclude* command is added to exclude computing intramolecular non-bonded interactions, since those are removed by the rigid fix anyway:

```
units real
atom_style charge
atom_modify map array
region box block -5 5 -5 5 -5 5
create_box 3 box

mass 1 15.9994
mass 2 1.008
mass 3 1.0e-100

pair_style lj/cut/coul/cut 8.0
pair_coeff 1 1 0.1550 3.1536
pair_coeff 2 2 0.0    1.0
pair_coeff 3 3 0.0    1.0

fix mol all property/atom mol ghost yes
molecule water tip4p.mol
create_atoms 0 random 33 34564 NULL mol water 25367 overlap 1.33
neigh_modify exclude molecule/intra all

timestep 0.5
fix integrate all rigid/small molecule langevin 300.0 300.0 100.0 2345634

thermo_style custom step temp press etotal density pe ke
thermo 2000
run 40000
write_data tip4p-explicit.data nocoeff
```

```
# Water molecule. Explicit TIP4P geometry for use with fix rigid

4 atoms

Coords

1    0.00000  -0.06556   0.00000
2    0.75695   0.52032   0.00000
3   -0.75695   0.52032   0.00000
4    0.00000   0.08444   0.00000

Types

1        1   # O
2        2   # H
3        2   # H
4        3   # M

Charges

1        0.000
2        0.520
3        0.520
4       -1.040
```

Wikipedia also has a nice article on water models.

**(Jorgensen)** Jorgensen, Chandrasekhar, Madura, Impey, Klein, J Chem Phys, 79, 926 (1983).

**(Abascal1) Abascal, Sanz, Fernandez, Vega, J Chem Phys, 122, 234511 (2005)**
https://doi.org/10.1063/1.1931662

**(Abascal2) Abascal, J Chem Phys, 123, 234505 (2005)**
https://doi.org/10.1063/1.2121687

**(Izadi) Izadi, Anandakrishnan, Onufriev, J. Phys. Chem. Lett., 5, 21, 3863 (2014)**
https://doi.org/10.1021/jz501780a

### 10.4.7 TIP5P water model

The five-point TIP5P rigid water model extends the *three-point TIP3P model* by adding two additional sites L, usually massless, where the charge associated with the oxygen atom is placed. These sites L are located at a fixed distance away from the oxygen atom, forming a tetrahedral angle that is rotated by 90 degrees from the HOH plane. Those sites thus somewhat approximate lone pairs of the oxygen and consequently improve the water structure to become even more "tetrahedral" in comparison to the *four-point TIP4P model*.

A suitable pair style with cutoff Coulomb would be:

- *pair_style lj/cut/coul/cut*

or these commands for a long-range model:

- *pair_style lj/cut/coul/long*
- *pair_style lj/cut/coul/long/soft*
- *kspace_style pppm*
- *kspace_style pppm/disp*

A TIP5P model *must* be run using a *rigid fix* since there is no other option to keep this kind of structure rigid in LAMMPS. In order to avoid that LAMMPS produces an error due to the massless L sites, those need to be assigned a tiny non-zero mass.

The table below lists the force field parameters (in real *units*) to for a the TIP5P model with a cutoff *(Mahoney)* and the TIP5P-E model *(Rick)* for use with a long-range Coulombic solver (e.g. Ewald or PPPM in LAMMPS).

| Parameter | TIP5P | TIP5P-E |
|---|---|---|
| O mass (amu) | 15.9994 | 15.9994 |
| H mass (amu) | 1.008 | 1.008 |
| O charge ($e$) | 0.0 | 0.0 |
| L charge ($e$) | -0.241 | -0.241 |
| H charge ($e$) | 0.241 | 0.241 |
| LJ $\varepsilon$ of OO (kcal/mole) | 0.1600 | 0.1780 |
| LJ $\sigma$ of OO (Å) | 3.1200 | 3.0970 |
| LJ $\varepsilon$ of HH, LL, OH, OL, HL (kcal/mole) | 0.0 | 0.0 |
| LJ $\sigma$ of HH, LL, OH, OL, HL (Å) | 1.0 | 1.0 |
| $r_0$ of OH bond (Å) | 0.9572 | 0.9572 |
| $\theta_0$ of HOH angle | 104.52° | 104.52° |
| OL distance (Å) | 0.70 | 0.70 |
| $\theta_0$ of LOL angle | 109.47° | 109.47° |

Below is the code for a LAMMPS input file for setting up a simulation of TIP5P water with a molecule file. Because of using *fix rigid/small* no bonds need to be defined and thus no extra storage needs to be reserved for them, but we need to either switch to atom style full or use *fix property/atom mol* so that fix rigid/small can identify rigid bodies by their molecule ID. Also a *neigh_modify exclude* command is added to exclude computing intramolecular non-bonded interactions, since those are removed by the rigid fix anyway. For simplicity and speed the example uses a cutoff Coulomb. Most production simulations require long-range Coulomb instead.

```
units real
atom_style charge
atom_modify map array
region box block -5 5 -5 5 -5 5
create_box 3 box

mass 1 15.9994
mass 2 1.008
mass 3 1.0e-100

pair_style lj/cut/coul/cut 8.0
pair_coeff 1 1 0.160  3.12
pair_coeff 2 2 0.0    1.0
pair_coeff 3 3 0.0    1.0

fix mol all property/atom mol
molecule water tip5p.mol
create_atoms 0 random 33 34564 NULL mol water 25367 overlap 1.33
neigh_modify exclude molecule/intra all

timestep 0.5
fix integrate all rigid/small molecule langevin 300.0 300.0 50.0 235664
reset_timestep 0

thermo_style custom step temp press etotal density pe ke
thermo 1000
run 20000
write_data tip5p.data nocoeff
```

```
# Water molecule. Explicit TIP5P geometry for use with fix rigid

5 atoms

Coords

1    0.00000  -0.06556   0.00000
2    0.75695   0.52032   0.00000
3   -0.75695   0.52032   0.00000
4    0.00000  -0.46971   0.57154
5    0.00000  -0.46971  -0.57154

Types

1        1   # O
2        2   # H
3        2   # H
```

(continues on next page)

(continued from previous page)

```
4        3   # L
5        3   # L

Charges

1        0.000
2        0.241
3        0.241
4       -0.241
5       -0.241
```

Wikipedia also has a nice article on water models.

---

**(Mahoney)** Mahoney, Jorgensen, J Chem Phys 112, 8910 (2000)

**(Rick)** Rick, J Chem Phys 120, 6085 (2004)

### 10.4.8 SPC and SPC/E water model

The SPC water model specifies a 3-site rigid water molecule with charges and Lennard-Jones parameters assigned to each of the three atoms. In LAMMPS the *fix shake* command can be used to hold the two O-H bonds and the H-O-H angle rigid. A bond style of *harmonic* and an angle style of *harmonic* or *charmm* should also be used.

One suitable pair style with cutoff Coulomb would for instance be:

- *pair_style lj/cut/coul/cut*

These commands are examples for a long-range Coulomb model:

- *pair_style lj/cut/coul/long*
- *pair_style lj/cut/coul/long/soft*
- *kspace_style pppm*
- *pair_style lj/long/coul/long*
- *kspace_style pppm/disp*

These are the additional parameters (in real units) to set for O and H atoms and the water molecule to run a rigid SPC model.

O mass = 15.9994
H mass = 1.008
O charge = -0.820
H charge = 0.410
LJ $\varepsilon$ of OO = 0.1553
LJ $\sigma$ of OO = 3.166
LJ $\varepsilon$, $\sigma$ of OH, HH = 0.0
$r_0$ of OH bond = 1.0
$\theta_0$ of HOH angle = 109.47°

Note that as originally proposed, the SPC model was run with a 9 Angstrom cutoff for both LJ and Coulomb terms. It can also be used with long-range electrostatic solvers (e.g. Ewald or PPPM in LAMMPS) without changing any of the parameters above, although it becomes a different model in that mode of usage.

The SPC/E (extended) water model is the same, except the partial charge assignments change:

O charge = -0.8476
H charge = 0.4238

See the *(Berendsen2)* reference for more details on both the SPC and SPC/E models.

Below is the code for a LAMMPS input file and a molecule file (`spce.mol`) of SPC/E water for use with the *molecule command* demonstrating how to set up a small bulk water system for SPC/E with rigid bonds. For simplicity and speed the example uses a cutoff Coulomb. Most production simulations require long-range Coulomb instead.

```
units real
atom_style full
region box block -5 5 -5 5 -5 5
create_box 2 box  bond/types 1 angle/types 1 &
                extra/bond/per/atom 2 extra/angle/per/atom 1 extra/special/per/atom 2

mass 1 15.9994
mass 2 1.008

pair_style lj/cut/coul/cut 10.0
pair_coeff 1 1 0.1553 3.166
pair_coeff 1 2 0.0    1.0
pair_coeff 2 2 0.0    1.0

bond_style zero
bond_coeff 1 1.0

angle_style zero
angle_coeff 1 109.47

molecule water spce.mol
create_atoms 0 random 33 34564 NULL mol water 25367 overlap 1.33

timestep 1.0
fix rigid     all shake 0.0001 10 10000 b 1 a 1
minimize 0.0 0.0 1000 10000
velocity all create 300.0 5463576
fix integrate all nvt temp 300.0 300.0 100.0

thermo_style custom step temp press etotal density pe ke
thermo 1000
run 20000 upto
write_data spce.data nocoeff
```

```
# Water molecule. SPC/E geometry

3 atoms
```

(continues on next page)

(continued from previous page)

```
2 bonds
1 angles

Coords

1    0.00000  -0.06461   0.00000
2    0.81649   0.51275   0.00000
3   -0.81649   0.51275   0.00000

Types

1        1   # O
2        2   # H
3        2   # H

Charges

1       -0.8476
2        0.4238
3        0.4238

Bonds

1   1      1      2
2   1      1      3

Angles

1   1      2      1      3

Shake Flags

1 1
2 1
3 1

Shake Atoms

1 1 2 3
2 1 2 3
3 1 2 3

Shake Bond Types

1 1 1 1
2 1 1 1
3 1 1 1

Special Bond Counts

1 2 0 0
2 1 1 0
```

(continues on next page)

(continued from previous page)

```
3 1 1 0

Special Bonds

1 2 3
2 1 3
3 1 2
```

Wikipedia also has a nice article on water models.

---

**(Berendsen2)** Berendsen, Grigera, Straatsma, J Phys Chem, 91, 6269-6271 (1987).

### 10.4.9 Two-band (and multi-band) EAM potentials with hybrid/overlay

Some embedded-atom-method (EAM) potentials add a *second* embedding term to the standard EAM energy expression in order to capture effects that a single density and embedding function cannot. The best known examples are the *two-band* (or *double-band*, "2BM") models for Fe-Cr, which add an "s-band" embedding term on top of the usual "d-band" EAM so that the sign of the alloy mixing enthalpy changes with chromium concentration *(Olsson)*, *(Bonny)*.

This Howto shows that such a model can be run with the existing *pair_style eam/fs* and *pair_style hybrid/overlay* commands, **without modifying or recompiling LAMMPS**, and how to build the required potential files.

#### The two-band energy expression

In a two-band model the energy of atom $i$ is

$$E_i = \frac{1}{2}\sum_{j\neq i} V(r_{ij}) + F^d\left(\rho_i^d\right) + F^s\left(\rho_i^s\right), \qquad \rho_i^d = \sum_{j\neq i}\phi^d(r_{ij}), \quad \rho_i^s = \sum_{j\neq i}\phi^s(r_{ij}).$$

This is an ordinary EAM (pair term $V$, d-band density $\phi^d$, d-band embedding $F^d$) plus one *extra* embedding term $F^s$ acting on a second, independent density $\rho^s$. Grouping the terms,

$$E_i = \underbrace{\left[\frac{1}{2}\sum_j V + F^d(\rho^d)\right]}_{\text{a standard eam/fs}} + \underbrace{\left[F^s(\rho^s)\right]}_{\text{eam/fs with } V\equiv 0}$$

shows that the total energy is the sum of two EAM potentials. Adding energies, forces, and virials of two potentials acting on the same atoms is exactly what *pair_style hybrid/overlay* does.

#### Why the s-band file must be *eam/fs*

In the two-band models the second (s-band) density is non-zero only between *unlike* atoms; it vanishes between atoms of the same element *(Bonny)*:

$$\phi^s_{AA} = \phi^s_{BB} = 0, \qquad \phi^s_{AB} = \phi^s_{BA} \neq 0.$$

A density that depends on the element of *both* the source and the target atom can only be expressed in the Finnis-Sinclair form, i.e. with *pair_style eam/fs*, whose density functions $\rho_{\alpha\beta}(r)$ are indexed per element pair. It *cannot* be written as an *eam/alloy* potential, which has a single density per element summed over all neighbors. Therefore the s-band file is always an *eam/fs* file in which the like-element density functions are set to zero and only the unlike-element density is non-zero. Its pair-potential ($r\,\phi$) section is set entirely to zero, because the pair term belongs to the d-band file.

The d-band file is an ordinary binary EAM (the pair term plus the d-band density and embedding). Because its density depends only on the neighbor's element it could equally be written as *eam/alloy*; using *eam/fs* for both files keeps the recipe uniform.

### The recipe

```
pair_style hybrid/overlay eam/fs eam/fs
pair_coeff * * eam/fs 1 FeCr.dband.eam.fs Fe Cr   # pair V + d-band F_d(rho_d)
pair_coeff * * eam/fs 2 FeCr.sband.eam.fs Fe Cr   # s-band F_s(rho_s), no pair
```

The numeric 1 and 2 after `eam/fs` are required because the same sub-style is used twice; see *pair_hybrid*. Each *eam/fs* instance keeps its own density and embedding-derivative arrays and performs its own communication, so the two densities $\rho^d$ and $\rho^s$ never mix. `hybrid/overlay` sums the two contributions, and because the s-band file has no pair term there is no double counting: the combined energy is exactly $\frac{1}{2}\sum V + F^d(\rho^d) + F^s(\rho^s)$.

The two files are independent *eam/fs* files and may use different radial grids and cutoffs; the neighbor list is built with the larger of the two cutoffs. Because both files store the same element masses, you may also set masses explicitly with the *mass* command to avoid any ambiguity.

This generalizes to more than two bands: overlay $N$ *eam/fs* files to add $N$ embedding terms. The same construction works for any model of the form "standard EAM plus one or more additive embedding terms."

### Building the s-band file

The helper script `tools/eam_generate/two_band_fecr.py` writes the s-band *eam/fs* file (cross-only density, $F^s$ embedding, zero pair term) and can also emit an illustrative demo d-band file plus a ready-to-run input deck:

```
cd tools/eam_generate
# self-contained runnable demo (writes both files + a sample input):
./two_band_fecr.py --demo --prefix FeCr_demo

# real use: add an s-band to your own trusted Fe-Cr d-band eam/fs file:
./two_band_fecr.py --dband MyFeCr.eam.fs --preset olsson-vasp --prefix FeCr_2bm
```

The script implements the s-band density as the square of a 4s Slater function with a smooth cutoff and the s-band embedding as $F^s(\rho) = c_1\sqrt{\rho} + c_2\rho^2 + c_3\rho^4$, matching the forms used in the published models. It ships parameter presets taken from the literature, but note that the absolute normalization of $\rho^s$ and the matching $F^s$ coefficients are convention-sensitive: validate a preset against a known result (for instance the concentration dependence of the mixing enthalpy) before using it for production work. The default `--demo` parameters are self-consistent and illustrative only; they exercise the overlay mechanics but are not a published fit.

### A worked check

Running the demo input produced by `--demo` and decomposing the energy with *compute pair* confirms the construction:

```
compute ed all pair eam/fs 1 epair   # d-band sub-style energy
compute es all pair eam/fs 2 epair   # s-band sub-style energy
thermo_style custom step pe c_ed c_es
```

The total potential energy is exactly `c_ed + c_es`. For a *pure* element the s-band energy `c_es` is identically zero, because the like-element s-density is zero, so $\rho^s = 0$ and $F^s(0) = 0$. Introducing unlike neighbors (an alloy) makes `c_es` non-zero: the s-band term samples the local concentration, which is the entire purpose of the model.

### Restrictions and notes

- The s-band embedding of these models typically has a $\sqrt{\rho}$ term, whose slope diverges as $\rho \to 0$. This is harmless here: LAMMPS tabulates $F^s$ as a spline (finite at every grid point) and the force contribution is multiplied by the s-density derivative, which is identically zero for like pairs. A pure-element atom therefore contributes no spurious s-band force.

- This Howto reproduces the two-band *functional form* exactly. Reproducing a specific published *parameterization* additionally requires the correct d-band/pair files (for Fe-Cr, the Mendelev iron potential plus the refit chromium and cross interactions) and validated s-band coefficients.

---

**(Olsson)** Olsson, Wallenius, Domain, Nordlund, Malerba, Physical Review B, 72, 214119 (2005).

**(Bonny)** Bonny, Pasianot, Terentyev, Malerba, Philosophical Magazine, 91, 1724 (2011).

## 10.5 Packages howto

### 10.5.1 Finite-size spherical and aspherical particles

Typical MD models treat atoms or particles as point masses. Sometimes it is desirable to have a model with finite-size particles such as spheroids, ellipsoids, superellipsoids, or generalized aspherical bodies. The difference is that such particles have a moment of inertia, rotational energy, and angular momentum. Rotation is induced by torque coming from interactions with other particles.

LAMMPS has several options for running simulations with these kinds of particles. The following aspects are discussed in turn:

- atom styles
- pair potentials
- time integration
- computes, thermodynamics, and dump output
- rigid bodies composed of finite-size particles

Example input scripts for these kinds of models are in the body, colloid, dipole, ellipse, line, peri, pour, and tri directories of the *examples directory* in the LAMMPS distribution.

#### Atom styles

There are several *atom styles* that allow for definition of finite-size particles: sphere, dipole, ellipsoid, line, tri, peri, and body.

The *sphere* style defines particles that are spheroids and each particle can have a unique diameter and mass (or density). These particles store an angular velocity (omega) and can be acted upon by torque. The "set" command can be used to modify the diameter and mass of individual particles, after then are created.

The *dipole* style does not actually define finite-size particles, but is often used in conjunction with spherical particles, via a command like

```
atom_style hybrid sphere dipole
```

This is because when dipoles interact with each other, they induce torques, and a particle must be finite-size (i.e. have a moment of inertia) in order to respond and rotate. See the *atom_style dipole* command for details. The "set" command can be used to modify the orientation and length of the dipole moment of individual particles, after then are created.

The *ellipsoid* style defines particles that are ellipsoids or superellipsoids and thus can be aspherical. Each particle has a shape, specified by 3 diameters, and mass (or density). Superellipsoid particles can be defined by additionally specifying 2 blockiness exponents (block) and adding the *superellipsoid* keyword to the *atom_style ellipsoid* command.

These particles store an angular momentum and their orientation (quaternion), and can be acted upon by torque. They do not store an angular velocity (omega), which can be in a different direction than angular momentum, rather they compute it as needed. The *set command* can be used to modify the diameter, orientation, and mass of individual

particles, after they are created. The *set command* can also be used to modify the blockiness of superellipsoid particles. It also has a brief explanation of what quaternions are.

The *line* style defines line segment particles with two end points and a mass (or density). They can be used in 2d simulations, and they can be joined together to form rigid bodies which represent arbitrary polygons.

The *tri* style defines triangular particles with three corner points and a mass (or density). They can be used in 3d simulations, and they can be joined together to form rigid bodies which represent arbitrary particles with a triangulated surface.

The *peri* style is used with *Peridynamic models* and defines particles as having a volume, that is used internally in the *pair_style peri* potentials.

The *body* style allows for definition of particles which can represent complex entities, such as surface meshes of discrete points, collections of sub-particles, deformable objects, etc. The body style is discussed in more detail on the *Howto body* doc page.

Note that if one of these atom styles is used (or multiple styles via the *atom_style hybrid* command), not all particles in the system are required to be finite-size or aspherical.

For example, in the ellipsoid style, if the 3 shape parameters are set to the same value, the particle will be a sphere rather than an ellipsoid. If the 3 shape parameters are all set to 0.0 or if the diameter is set to 0.0, it will be a point particle. In the line or tri style, if the lineflag or triflag is specified as 0, then it will be a point particle. Similarly, if a superellipsoid has both blockiness parameters set to 1.0, the superellipsoid will be a regular ellipsoid.

Some of the pair styles used to compute pairwise interactions between finite-size particles also compute the correct interaction with point particles as well, e.g. the interaction between a point particle and a finite-size particle or between two point particles. If necessary, *pair_style hybrid* can be used to ensure the correct interactions are computed for the appropriate style of interactions. Likewise, using groups to partition particles (ellipsoids versus spheres versus point particles) will allow you to use the appropriate time integrators and temperature computations for each class of particles. See the doc pages for various commands for details.

Also note that for *2d simulations*, atom styles sphere and ellipsoid still use 3d particles, rather than as circular disks or ellipses. This means they have the same moment of inertia as the 3d object. When temperature is computed, the correct degrees of freedom are used for rotation in a 2d versus 3d system.

### Pair potentials

When a system with finite-size particles is defined, the particles will only rotate and experience torque if the force field computes such interactions. These are the various *pair styles* that generate torque:

- *pair_style granular*
- *pair_style gran/hooke*
- *pair_style gran/hooke/history*
- *pair_style gran/hertz/history*
- *pair_style granular/superellipsoid*
- *pair_style dipole/cut*
- *pair_style gayberne*
- *pair_style resquared*
- *pair_style brownian*
- *pair_style lubricate*
- *pair_style line/lj*
- *pair_style tri/lj*

- *pair_style body/nparticle*

Most of the granular pair styles are used with spherical particles with the exception of the *granular/superellipsoid* pair style which is used with superellipsoid particles. The dipole pair style is used with the dipole atom style, which could be applied to spherical or ellipsoidal particles. The GayBerne and REsquared potentials require ellipsoidal particles, though they will also work if the 3 shape parameters are the same (a sphere). The Brownian and lubrication potentials are used with spherical particles. The line, tri, and body potentials are used with line segment, triangular, and body particles respectively.

### Time integration

There are several fixes that perform time integration on finite-size spherical particles, meaning the integrators update the rotational orientation and angular velocity or angular momentum of the particles:

- *fix nve/sphere*
- *fix nvt/sphere*
- *fix npt/sphere*

Likewise, there are 3 fixes that perform time integration on ellipsoidal particles:

- *fix nve/asphere*
- *fix nvt/asphere*
- *fix npt/asphere*

The advantage of these fixes is that those which thermostat the particles include the rotational degrees of freedom in the temperature calculation and thermostatting. The *fix langevin* command can also be used with its *omgea* or *angmom* options to thermostat the rotational degrees of freedom for spherical or ellipsoidal particles. Other thermostatting fixes only operate on the translational kinetic energy of finite-size particles.

These fixes perform constant NVE time integration on line segment, triangular, and body particles:

- *fix nve/line*
- *fix nve/tri*
- *fix nve/body*

Note that for mixtures of point and finite-size particles, these integration fixes can only be used with *groups* which contain finite-size particles.

### Computes, thermodynamics, and dump output

There are several computes that calculate the temperature or rotational energy of spherical or ellipsoidal particles:

- *compute temp/sphere*
- *compute temp/asphere*
- *compute erotate/sphere*
- *compute erotate/asphere*

These include rotational degrees of freedom in their computation. If you wish the thermodynamic output of temperature or pressure to use one of these computes (e.g. for a system entirely composed of finite-size particles), then the compute can be defined and the *thermo_modify* command used. Note that by default thermodynamic quantities will be calculated with a temperature that only includes translational degrees of freedom. See the *thermo_style* command for details.

These commands can be used to output various attributes of finite-size particles:

- *dump custom*
- *compute property/atom*

- *dump local*
- *compute body/local*

Attributes include the dipole moment, the angular velocity, the angular momentum, the quaternion, the torque, the end-point and corner-point coordinates (for line and tri particles), and sub-particle attributes of body particles.

#### Rigid bodies composed of finite-size particles

The *fix rigid* command treats a collection of particles as a rigid body, computes its inertia tensor, sums the total force and torque on the rigid body each timestep due to forces on its constituent particles, and integrates the motion of the rigid body.

If any of the constituent particles of a rigid body are finite-size particles (spheres or ellipsoids or line segments or triangles), then their contribution to the inertia tensor of the body is different than if they were point particles. This means the rotational dynamics of the rigid body will be different. Thus a model of a dimer is different if the dimer consists of two point masses versus two spheroids, even if the two particles have the same mass. Finite-size particles that experience torque due to their interaction with other particles will also impart that torque to a rigid body they are part of.

See the "fix rigid" command for example of complex rigid-body models it is possible to define in LAMMPS.

Note that the *fix shake* command can also be used to treat 2, 3, or 4 particles as a rigid body, but it always assumes the particles are point masses.

Also note that body particles cannot be modeled with the *fix rigid* command. Body particles are treated by LAMMPS as single particles, though they can store an internal state, such as a list of sub-particles. Individual body particles are typically treated as rigid bodies, and their motion integrated with a command like *fix nve/body*. Interactions between pairs of body particles are computed via a command like *pair_style body/nparticle*.

### 10.5.2 Granular models

Granular systems are composed of finite-size spherical or aspherical particles with explicit diameters, as opposed to point particles. This means they have an angular velocity and torque can be imparted to them to cause them to rotate.

The various atom, pair, fix, and compute styles listed below on this page are useful for creating granular models with such particles.

You can also define granular "surfaces" which are a collection of line segments (2d systems) or triangles (3d systems), which act as boundaries interacting with the particles. This is particularly useful for defining a complex wall or boundary geometry. Particle/surface interactions can be specified with similar options as the pair styles listed below for particle/particle interactions.

These doc pages describe how to define and use granular surfaces:

- *Howto granular surfaces*
- *fix surface/global*
- *fix surface/local*
- *pair_style surf/granular*

---

To run a simulation of particles with a granular model, you will typically want to use the following commands:

- *atom_style sphere*
- *fix nve/sphere*
- *fix gravity*

Aspherical granular particles can be simulated by creating clusters of spherical particles using either the *rigid* or *BPM* package or by using *superellipsoids*.

This compute

- *compute erotate/sphere*

calculates rotational kinetic energy which can be *output with thermodynamic info*.

The compute

- *compute fabric*

calculates various versions of the fabric tensor for granular and non-granular pair styles.

Use one of these 4 pair potentials, which compute forces and torques between interacting pairs of particles:

- *pair_style gran/history*
- *pair_style gran/no_history*
- *pair_style gran/hertzian*
- *pair_style granular*

The first 3 are the older, original granular pair styles implemented in LAMMPS. The 4th *pair_style gran/no_history* is the newest and most flexible. It's pairwise interaction models include options similar to the first 3 pair styles, plus many additional models.

To add your own custom granular contact model to the *pair_style granular* command, see the *Modifying granular sub-models* doc page.

---

These commands implement fix options specific to granular systems:

- *fix freeze*
- *fix pour*
- *fix viscous*
- *fix wall/gran*
- *fix wall/gran/region*

The fix style *freeze* zeroes both the force and torque of frozen atoms, and should be used for granular systems instead of the fix style *setforce*.

To model heat conduction, one must add the temperature and heatflow atom variables with:

- *fix property/atom*

plus a temperature integration fix

- *fix heat/flow*

and a heat conduction option defined in both these commands:

- *pair_style granular*
- *fix wall/gran*

For computational efficiency, you can eliminate needless pairwise computations between frozen atoms by using this command:

- *neigh_modify* exclude

**Note**

By default, for 2d systems, granular particles are still modeled as 3d spheres, not 2d discs (circles), meaning their moment of inertia will be the same as in 3d. If you wish to model granular particles in 2d as 2d discs, see the note on this topic on the *Howto 2d* doc page, where 2d simulations are discussed.

### 10.5.3 Granular surfaces

Added in version 4Jul2026.

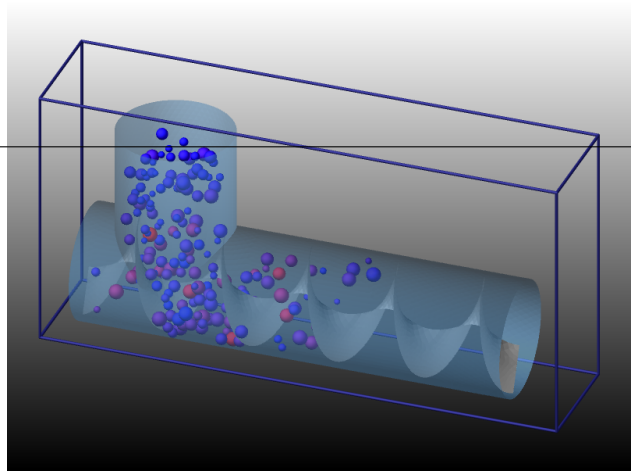
As explained on the *Howto granular* doc page, granular systems are composed of finite-size spherical or aspherical particles with explicit diameters, as opposed to point particles. This means they have an angular velocity and torque can be imparted to them to cause them to rotate.

The *Howto granular* doc page lists various atom, pair, fix, and compute styles useful for creating granular models with such particles.

This page explains how you can also define granular surfaces which are a collection of triangles (3d systems) or line segments (2d systems), which act as boundaries interacting with the particles. This is particularly useful for defining a complex wall or boundary geometry. As described below, particle/surface interactions can be specified with similar options as those for particle/particle interactions.

In the examples directory, several examples of these boundaries are found in the gransurf folder. In particular, this includes a screw feeder geometry where a cylindrical container is being fed a stream of granular particles from above which are conveyed forward using a rotating screw. An illustration is rendered in the below figure using the *dump image* command.

Furthermore, as another illustration of possible applications, an image is included of a complex geometry based on the actual shape of the Itokawa asteroid. Here, a surface is used to create a container which is filled by a polydisperse granular packing.

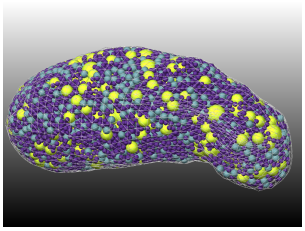


#### Global versus local surfaces

A key point to understand is that LAMMPS supports two forms of granular surfaces. You cannot use both in the same simulation.

The first is *global* which means that each processor stores a copy of all the triangles/lines. This is suitable when only a modest number of triangles/lines is needed. They can be large triangles/lines, any of which span a significant fraction of the simulation box size in one or more dimensions.

The second is *local* which means that the collection of triangles/lines is distributed across processors in the same manner that particles are distributed. Each processor is assigned to a sub-domain of the simulation box and owns whichever triangles/lines have their center point in the processor's sub-domain. Similar to particles, each processor may also own ghost copies of triangles/lines whose finite size overlaps with the processor's sub-domain. The total number of triangles/lines in the system can now be very large. For effective distribution and minimal communication, all the triangles/lines should be small, no more than a few particle diameters in size. If even one larger triangle or line is



defined then the neighbor list cutoff and communication cutoff will be set correspondingly larger, which can slow down the simulation. Note that a large triangle or line can instead be defined as multiple connected smaller triangles or lines without changing the topology of the collective surface. The *multi* *neighbor* option may be useful, however, in such scenarios.

One of these two commands must be specified to use *global* or *local* surfaces in your granular simulation:

- *fix surface/global*
- *fix surface/local*

The *fix surface/global* command reads in the global surfaces in one of two ways. The first option is from a molecule file(s) previously defined by the *molecule* command. The file should define triangles or lines with header keywords and a Triangles or Lines section. The second option is from a text or binary STL (stereolithography) file which defines a set of triangles. It can only be used with 3d simulations.

The *fix surface/local* command defines local surface in one of three ways. The first two options are the same molecule and STL files explained in the previous paragraph. In this case, the list of triangles/lines is distributed across processors based on the center point of each triangle/line. The third option is to include them in a LAMMPS data file which has been previously read in via the *read_data* command. If the file has a Triangles or Lines section, then triangles/lines will be read in and distributed along with any particles the data file includes, assuming an appropriate *atom_style* has been specified, as explained below.

---

### Surface attributes

For both global and local surfaces, each triangle/line is assigned a *type* and a *molecule ID*. This is done when surfaces are read-in from a molecule, STL, or data file. Since STL files do not define types or molecule IDs, the *fix surface/global* and *fix surface/local* commands specify the type and molecule ID that will be assigned to the read-in triangles. The *fix surface/global* command also allows use of the *fix_modify type/region* command to assign types based on a geometric region. Since local surfaces are effectively particles, the *set* command can be used to alter the *type* or *molecule ID* of any triangle or line.

For both global and local surfaces, types are used to define the style of granular interactions for individual triangles/lines. Different types, and thus styles, can be used within a single object consisting of connected triangles/lines.

Molecule IDs can be used to limit which triangles/lines are considered to be "connected". They are therefore intended to be assigned uniquely to each inter-connected set of triangles/lines, as if each object were a "molecule". See the Surface Connectivity section below.

For local surfaces, the molecule ID can be used to define groups which enables assignment of different motions to different surface objects. See the Surface Motion section below. Various other LAMMPS commands operate on groups or molecules and can thus be used to gather statistics about or output information about individual surface objects.

---

### Atom styles for granular surfaces

For all three ways of defining *local* surfaces, the triangles/lines are stored internally in LAMMPS as triangle-style or line-style particles. This means you must use a hybrid atom style which includes one of these two sub-styles (for 3d or 2d):

- *atom_style tri* for 3d simulations
- *atom_style line* for 2d simulations

The atom_style hybrid command must also define a *atom_style sphere* sub-style for the granular particles which interact with the surfaces.

Note that for molecule or STL file input, the *fix surface/local* command reads the file(s) and uses the values for each surface to create a single new triangle or line particle. For data file input, the triangle/line particles are created when the data file is read.

For granular simulations with *global* surfaces, a hybrid atom style which defines triangle-style or line-style particles should NOT be used. Typically only an *atom_style sphere* command is needed to define the properties of particles in the simulation. The triangles/lines are stored by the *fix surface/global* command and not as triangle-style or line-style particles.

---

### Rules for surface topology

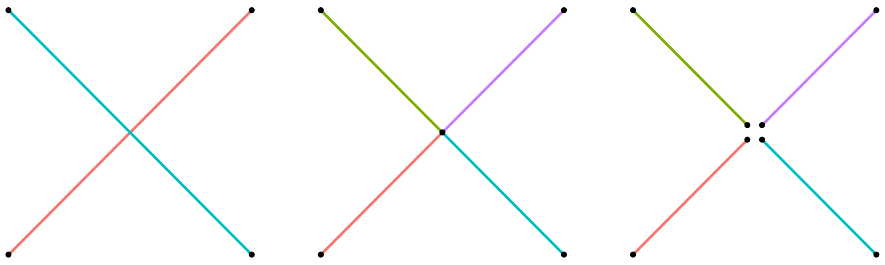
For both *global* and *local* surfaces, granular particles interact with both sides of each triangle or line segment. This means a surface such as a mixer blade can be infinitely thin.

Triangles and line segments are considered to be "connected" and form a contiguous surface, if they (1) have the same molecule ID and (2) share common edges or corner point (triangles) or end points (line segments). Each triangle edge or corner point can be shared by multiple adjacent triangles. A triangle edge is shared by two triangles if both the end points of the edge are corner points of both triangles. A triangle corner point is shared by two triangles if it is a corner point of both. Likewise a line segment end point is shared by two line segments if it is an end point of both segments.

There is no requirement that a triangle edge or triangle corner point or line segment end point be connected to another triangle or line segment. If an edge or point has no connection, it is a free (unconnected) edge or point. Particles interact with the free edge or corner point in a manner consistent with forces generated by particles overlapping with the interior of a triangle or line segment.

No check is made to see if two triangles or line segments intersect each other; this is allowed if it makes sense for the geometry of the collection of surfaces. However, intersections can cause issues if the missing connectivity leads to inaccurate forces.

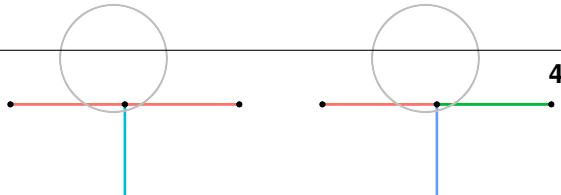
As an example of a valid intersection, consider a 2d simulation which mixes a container of granular particles. *Global* line segments are used to define both the box-shaped container and the mixer in the center. The 4 mixer blades are in the shape of a large X and are made to rotate using the *fix_modify* command (see below).



The 2 blades could be defined by 2 line segments which cross each other at their centers (left). Or the 2 blades could be defined by 4 line segments, all of which have a common endpoint at the center of the mixer (middle). Or the 2 blades could be defined by 4 non-touching line segments, all of which have a distinct endpoint near the center of the mixer, but displaced from it by a distance less than the radius of a granular particle (right). In any of these cases, when a particle gets very close to the center of the mixer it will interact with both nearby line segments as expected.

As an example of an invalid intersection, consider a 2d simulation of a T shaped object (next figure, left) defined by only 2 line segments (red, blue). The vertical line segment (blue) ends at the midpoint of the horizontal line segment (red). Without proper connectivity, there is no way to censor a force from the geometrically hidden vertical segment as a particle (gray) moves horizontally across the top of the T. In contrast, the same T shape can be defined in a valid manner (right) by using 3 line segments (red, green, blue) which all share a common endpoint at the center of the top of the T. The connectivity will then correctly censor the force from the vertical segment (blue).

See the next section on connectivity for how two triangles or line segments are treated if they share a common edge (triangle) or point (triangle or line).



### Surface connectivity

If multiple triangles/lines are used to define a contiguous surface which is flat or gently curved or has sharp edges or corners, LAMMPS will detect when two or more line segments (2d) in the same molecule share the same end-point. Or when two or more triangles (3d) in the same molecule share the same edge or same corner point.

This connectivity is stored internally and is used when appropriate to calculate accurate forces on particles which simultaneously overlap with 2 or more connected triangles or line segments.

Consider the simulation model of the previous section for a 2d mixer now defined by *local* line segments. The flat surface of each mixer blade (and container box faces) is defined by multiple small line segments. It is important that these line segments be "connected" so that when a particle contacts two adjacent line segments at the same time, the resulting force on the particle is the same as it would be if it were contacting the middle of a single long line segment.

Here is how to ensure that LAMMPS detects the appropriate connections.

For either *global* or *local* surfaces, if the triangles/lines are defined in a molecule or STL file, then 3 corner points (triangle) or 2 end points (line) will be listed for each triangle/line in the file. LAMMPS will only make a connection between 2 triangles or lines if a shared point is EXACTLY the same in both. This is a single point in both for a corner point or end point connection. It is two points in both triangles for an edge connection.

For *local* surfaces, if the triangles/lines are defined in a data file, then 3 corner points (triangle) or 2 end points (line) will be listed for each triangle/line in the file. However in this case, LAMMPS will allow for an INEXACT match of a shared point to make a connection between 2 triangles or lines. Again, this is a single point in both for a corner point or end point connection. It is two points in both triangles for an edge connection.

An INEXACT match means that the two points can be EPSILON apart. EPSILON is defined as a tiny fraction (1.0e-4) of the size of the smallest triangle or line in the system. The reason INEXACT matches are allowed is that data files can be created in a variety of manners, including by LAMMPS itself as a simulation runs via the *write_data* command. Internally, triangle-style and line-style particles do not store their corner points directly. Instead, the center point of the triangle/line is stored, along with an orientation of the triangle/line and a displacement vector from the center point for each corner point. This means that when new corner points values are written to a data file for two different triangles/line, they may differ slightly due to round-offs in finite-precision arithmetic.

Note that due to how connectivity is defined, two triangles/lines will not be connected if their corner points are separated by small distances (greater than EPSILON, but still small). Likewise they will not be connected if the corner point of one triangle/line is very close to (or even on) the surface of another triangle or middle of another line segment. In general these kinds of granular surfaces could be problematic and should be avoided, but LAMMPS does not check for these conditions.

In addition, note that connectivity is only defined between two triangles/lines with the same molecule ID. This way surfaces of two molecules can move independently, as described in the following section.

Note that if a triangle or line segment has a free edge or free corner/end point (not connected to any other triangle/line), granular particles will still interact with the triangle/line if the nearest contact point to the spherical particle center is on the free edge or is the free corner/end point.

### Surface motion

By default, surface triangles/lines are motionless during a simulation, whether they are *global* or *local*. Triangles/lines impart forces and torques to granular particles, but the inverse forces/torques on the triangles/lines do not cause them to move.

However, triangles/lines can be made to move in a prescribed manner. E.g. the rotation of 2d mixer blades in the example described above. These two commands can be used for that purpose:

- *fix_modify move* for *global* surfaces
- *fix move* for *local* surfaces

For *global* surfaces, the *fix_modify move* command can move a specified subset of the triangles/lines in various ways (translation, rotation, etc). Which triangles move is specified based on the *molecule ID* of each triangle. Molecule IDs are specified when surfaces are defined by the *fix surface/global* command. They can also be defined by the *fix_modify mol/region* command.

For *local* surfaces, the *fix move* command can move a specified subset of the triangles/lines in various ways (translation, rotation, etc). Which triangles move is specified based on the group-ID argument to the *fix move* command. Groups of *local* surfaces can be defined by the *group* command.

**Note**

For an object defined by two or more connected triangles/lines, it is an error to assign a motion and not include all the connected triangles/lines, since this would break the connections. LAMMPS checks this for *global* surfaces but only checks that the fix group of any instances of *fix move* include all or none of a set of connected *local* triangles/lines.

---

### Calculation of forces

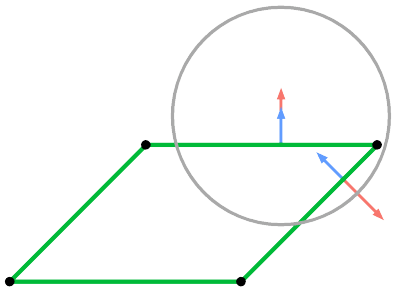
After generating the surface connectivity, LAMMPS classifies each connection as being flat or non-flat based on the angular difference between normal vectors. The two sides are then classified as being concave or convex based on their normal vectors. In scenarios where flat surfaces are perfectly flat (parallel normal vectors) the concave vs convex designation is arbitrary.

Each point or edge of a line or triangle is then classified as being internal, external, or unconnected based on the connectivity. For lines, an end point is internal if it only has flat connections, external if it has at least one non-flat (concave or convex) connection, and unconnected if it has no connections. The same is true for edges on a triangle. Corners on triangles inherit their classification from the two edges that meet at it. If either edge is unconnected, the corner is unconnected. Otherwise, the corner is external if either edge is external, and internal if both edges are internal.

To calculate force on a particle, LAMMPS finds a set of all lines or triangles that are geometrically in contact with the particle, $S_{\text{all}}$. For each line or triangle $i$, LAMMPS calculates the geometric overlap $\delta_i$ and the normal vector between the contact point on that line/triangle and the particle, $\hat{n}_{r,i}$.

Depending on the contact model, the force between a particle and a surface can depend on many variables including relative velocities, angular velocities, sliding history, etc. Most importantly, the force depends on an effective overlap distance $\delta_f$ and an effective point of contact $\vec{r}_c$ or (equivalently) the direction of the normal force $\hat{n}_f$. If a particle is in contact with a single line or triangle $i$, there is an unambiguous geometrically-determined point of contact and overlap such that $\delta_f = \delta_i$ and $\hat{n}_f = \hat{n}_{r,i}$. If a particle is in contact with two (or more) lines/triangles that are not connected, then two forces are applied with overlaps and directions determined from the geometric values for each line/triangle. However, if the particle is in contact with two (or more) connected lines/triangles, the calculation of $\delta_f$ and $\hat{n}_f$ is more complicated and is described in the remainder of this section.

First, LAMMPS needs to identify a set of consistent sides of contact for each line/triangle in $S_{\text{all}}$. For a single line or triangle, which side or face of the surface is in contact is unambiguous. However, if a particle is in contact with two or more connected lines/triangles this depends on the network of connectivity. For instance, the below figure highlights a 2D system with a particle (gray circle) in contact with two (green) lines that are part of a rhombus shaped object. From a naive local determination, one would determine the particle is in contact with the sides/faces of the two lines with each surface normal oriented in the blue direction, one of which points inwards, into the object. However, through the context of the convex connection, one can identify the physical (correct) surface normals (red). LAMMPS evaluates a consistent set of sides/faces (setting the sign of $\hat{n}_s$) by walking through all connections of contacted surfaces starting from the primary contact. If there are still unchecked surfaces, LAMMPS finds the unchecked surface with the largest overlap and repeats the process.



Next, LAMMPS clusters all contacted and connected lines/triangles into distinct composite sets each consisting of mutually flat line/triangle surfaces that act as one physical object, denoted $S_n$ where $n$ labels a particular composite. For instance, if a particle is touching two flat-connected surfaces and a third concave-connected surface, it will group them into two composite surfaces of size two and one. Each composite set of surfaces calculates an effective contact point and applies a single force on the particle. This clustering is performed by starting with the primary contact and checking all of its connections. Flat connections are added to the current set of surfaces $S_n$ and to a list of surfaces which need to be walked. Non-flat connections are skipped, however, convex connections with smaller overlaps are first flagged to be hidden (described below). LAMMPS then iterates one-by-one through the list of surfaces to be walked until it is empty. This effectively searches all 1st flat neighbors of the primary contact, then flat 2nd neighbors, etc. LAMMPS then identifies the next unprocessed surface with the largest overlap (the new primary contact) and repeats the process to create a new set of composite surfaces $S_m$ and continues until all contacting surfaces in $S_{\text{all}}$ have been added to a composite set (which can consist of single surface). While performing this walk, any hidden flags are passed along to subsequently walked flat connecting surfaces such that the hidden status cascades around a convex turn.

For each composite set of flat surfaces $S_n$, LAMMPS calculates a single force from an effective net overlap $\delta_f$ and direction $\hat{n}_f$. Before calculating $\delta_f$, the individual overlap of any hidden surface $i$ is zeroed, $\delta_i = 0$. Then $\delta_f = \max(\delta_i | i \in S_n)$. The direction of the force is a weighted average across all surfaces in the set $S_n$ or $\hat{n}_f = A \sum W_i \delta_i \hat{n}_{f,i}$ where $\hat{n}_{f,i}$ is a calculated direction of force for that surface, $W_i$ is a per-surface weight, and $A$ is a normalization factor.

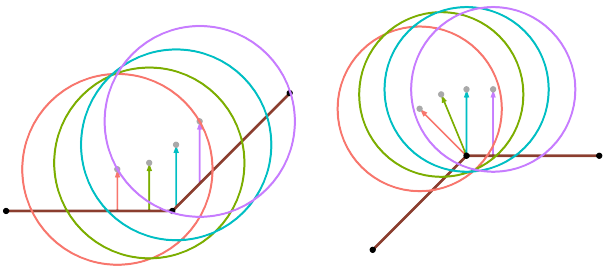
Before describing the calculation of individual directions, $\hat{n}_{f,i}$, and weights, $W_i$, for surfaces in a composite, LAMMPS calculates a general weight for externally vs. internally contacted surfaces defined as $W_{\text{ext}} \equiv (\delta_{\text{max,ext}}/\delta_{\text{max}})^2$ and $W_{\text{int}} \equiv 1 - W_{\text{ext}}$, respectively, where $\delta_{\text{max,ext}}$ is the maximum overlap with an externally contacted surface in that composite set $S_n$. This weighting is used to emphasize contributions from surfaces on a convex boundary as a particle moves along the convex turn.

In 2D, a surface $i$'s weight $W_i$ is either $W_{\text{ext}}$ or $W_{\text{int}}$ based on its status. By default, $\hat{n}_{f,i}$ is simply the direction from the local contact point on that surface to the particle, $\hat{n}_{r,i}$. For example, if the contact point is inside of the line, then $\hat{n}_{r,i}$ is equivalent to the surface normal $\hat{n}_{s,i}$. However, there are two exceptions: (a) if the contact is at a concave-connected point then $\hat{n}_{f,i} = \hat{n}_{s,i}$, and (b) if the contact is at a convex-connected point and $\hat{n}_{r,i}$ has a component pointing into the neighboring line vector indexed $j$ then $\hat{n}_{f_i} = \hat{n}_{s,j}$. These rules place limits on how how much a resulting force can point into the connected line $j$ to ensure $\hat{n}_{f_i}$ smoothly varies as the particle turns the bend. A few details are worth noting. First, as non-flat connecting surfaces are hidden behind convex turns ($\delta_i = 0$), the limit in the convex scenario is only relevant for connections that are both flat and convex. Secondly, if two flat lines are perfectly parallel, then $\hat{n}_{f,i} = \hat{n}_{s,i} = \hat{n}_{s,j}$ implying the concave/convex designation has no effect. Lastly, if a point is shared by more than two lines, then LAMMPS finds which connecting line has a normal vector closest to $\hat{n}_{s,i}$ to determining whether it's a concave or convex connection.

To illustrate, two scenarios are visualized in the figure below. In the left panel, a particle at various positions (red, green, blue, and purple) contacts a concave bend made up of two lines (coral brown). Here the leftmost line is labeled $i$ and the rightmost line is labeled $j$. The direction of the force $\hat{n}_{f,i}$ from line $i$ is indicated by arrows. Along the entire

contact, $\hat{n}_{f,i} = \hat{n}_{s,i}$. In the right panel, the contact point is at a convex corner such that $\hat{n}_{f,i} = \hat{n}_{r,i}$ (red, green) unless $\hat{n}_{r,i}$ has a component pointing into the adjacent line $j$, in which case $\hat{n}_{f,i} = \hat{n}_{s,j}$ (blue, purple). These rules therefore simply enforce sensible continuity of forces as atoms move across line segments.

In 3D, first consider a scenario where there are no contacts with free or unconnected edges (all edges have at least one connection). Again, each composite set of surfaces $S_n$ calculates a single force. For each surface, if its contact point is inside of the triangle, then $\hat{n}_{f,i} = \hat{n}_{s,i}$. If the contact is on an edge, it acts much like a point in 2D: a concave connection implies $\hat{n}_{f,i} = \hat{n}_{s,i}$ and a convex connection implies $\hat{n}_{f,i} = \hat{n}_{r,i}$ unless $\hat{n}_{r,i}$ points into the adjacent triangle $j$ in which case $\hat{n}_{f,i} = \hat{n}_{s,j}$. This is determined by comparing the three dot products between $\hat{n}_{s,i}$, $\hat{n}_{s,j}$, and $\hat{n}_{r,i}$.



If a particle contacts a corner, then the corner first calculates what the direction of $\hat{n}_{f,i}$ would be had the particle contacted either of the two edges, labeled $a$ and $b$, $\hat{n}_{f,a}$ and $\hat{n}_{f,b}$ (where the $i$ is implied from context). Let us dnote the normalized line vectors of these edges as $\hat{l}_a$ and $\hat{l}_b$, where the orientation is chosen to point towards the corner in consideration.

If $\hat{n}_{f,a} \cdot \hat{l}_b < 0$ (or if the force from that edge has a component pointing into the other edge), then $\hat{n}_{f,a}$ is replaced with $\hat{n}_{s,i}$, and vice versa for edge $b$, to avoid forces pointing into the triangle. A resulting $\hat{n}_{f,i}$ is then calculated by performing a weighted average of these two edge contributions such that the result interpolates between the two limits as a particle moves from contacting one edge to the corner to the other edge.

The calculation of these two edge weights is complicated, but a brief description is provided below to contextualize the code. Each weight is primarily a function of several dot products including $C_a = \hat{n}_{r,i} \cdot \hat{l}_a$ and $D_a = \hat{n}^p_{r,i} \cdot \hat{l}_a$ as well as equivalents for edge $b$ where $\hat{n}^p_{r,i}$ is $\hat{n}_{r,i}$ projected into the plane of the triangle. If the contact is directly above the corner such that $\hat{n}_{r,i}$ is parallel to $\hat{n}_{s,i}$, then $\hat{n}_{f,i}$ is simply set to $\hat{n}_{s,i}$ and this procedure is skipped. The base weights are then $W_a = (1 - C_a)D_b$ and $W_b = (1 - C_b)D_a$. This construction ensures the weight for, say, edge $a$ approaches zero as $\hat{n}_{r,i}$ either aligns with $\hat{l}_a$ (at which point the edge cannot easily calculate a realistic force) or becomes perpendicular to $\hat{l}_b$ such that its contribution dominates.

When evaluating corner contacts, if either of the two edges, say $a$, has a convex connection and is hiding another triangle, an additional weight is required for that edge to ensure $\hat{n}_{f,i}$ continuously transitions as the particle moves across that convex turn to surface $j$. This is calculated in terms of the dot product $E_a = \hat{n}^a_{r,i} \cdot (\delta_{s,i} + \delta_{s,j})/2$ where $\hat{n}^a_{r,i}$ is the vector $\hat{n}_{r,i}$ minus any component along $\hat{l}_a$. $W_b$ is then multiplied by $(1 - E_a)$, which goes to zero as the particle approaches the threshold before switching to contacting surface $j$ which shares edge $a$.

Lastly, there are forces on particles from unconnected (free) edges or corners of triangles. For example, if one defines a circular plate made up of triangles, such unconnected edges make up the perimeter of the plate. Currently, LAMMPS does not calculate or store connectivity information between unconnected edges along such a border and therefore forces are not guaranteed to vary in a physically realistic manner as a particle moves along it (e.g. if there is a convex bend in the border, forces are not censored from the far edge as particles mov around it). With this constraint, the below calculation is designed to prioritize continuity in the direction of force and ensure forces always point away from the surface's border as expected. However, tt is generally recommended that if a surface has an exposed border, it should be designed to be relatively simple and smooth.

First assume there is only one contacted triangle in a composite set. Forces from edges are calculated identically to a convex-connected edge. Unconnected corners (those that have at least one unconnected edge) also average contributions from their two edges. If both edges are connected (or unconnected), this average is the same. However, if one is connected and the other is unconnected then the contribution from the connected edge, say $a$, is modified.

For interpolating between unconnected and connected limits, a general weighting $W_c = \max(0.0, 1.0 - dr_{uc}/\delta_{\max})$ is used where $dr_{uc}$ is the maximum in-plane distance away from a triangle's unconnected corner or edge in the composite set. If the particle sits on top of the effective surface from the set, this weight is zero. As the particle moves off of and away from this effective surface, the weight approaches one. $W_i$ for all surfaces that are not contacted on an unconnected

feature is then multiplied by $W_c$. Additionally, these surfaces multiply $W_i$ by a factor of $W_{ip} = \min(1.0, \hat{n}_{s,i} \cdot \hat{n}_{r,i}/(1-W_c))$. This term goes to zero as the particle moves within the plane of the surface, which practically should only happen when a particle is outside and away from the surface. This term avoids complications when internal connections cannot determine which side of the surface is being contacted.

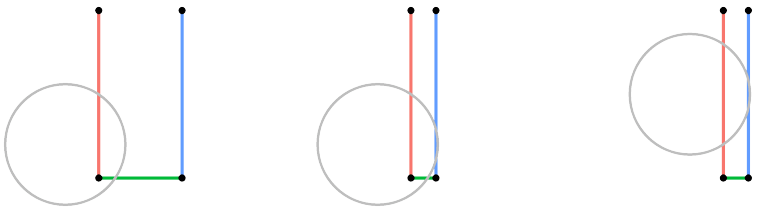
Lastly, when contacting an unconnected corner which has one unconnected edge and one connected edge, the contribution from the connected edge is multiplied by these two factors, $W_c$ and $W_{ip}$.

---

### Recommendations for geometries

When designing geometries for granular surfaces, there are several things to keep in mind to avoid unintended force contributions and to improve efficiency.

While convex corners are identified and used to censor forces from physically hidden sections of a wall, if a particle is not in contact with the entirety of a convex turn, then forces cannot be properly censored. For example, consider a 2d system with a U shaped wall defined by 3 line segments (see figure). If the width of the U is wider than the typical particle-wall overlap (left), no issues are anticipated. However, if the width of the U is very thin relative to the typical particle-wall overlap (middle, right), then a particle could potentially be in contact with both vertical legs of the U. If the particle is also in contact with the base of the U (middle, green), then it can identify the convex turn and censor forces from the rightmost vertical leg (blue). However, if the particle is towards the top of the U (right) and not in contact with the base (green), then there is no way for the particle to identify the convex turn and censor forces from the far vertical wall (blue). Therefore, walls with very thin features separated by a gap less than the expected overlap distance between a particle and a surface can lead to unintended additional forces.



As mentioned in the above section, forces resulting from contact with unconnected endpoints of lines do point in the expected direction and experience no discontinuities as the particle moves around the endpoint. However, in 3D, contacts with unconnected edges only produce reasonably directed forces oriented away from the edge. However, the exact direction of a force can wobble as the contact moves across a series of disconnected edges and convex turns may not be appropriately censored. Therefore, it is recommended to avoid complex geometries along unconnected boundaries such as rapid oscillations in- or out-of-plane such as pleats or sawteeth, relative to the length of an edge.

To build a neighbor list between particles and lines/triangles, LAMMPS assigns a radius to each line/triangle that corresponds to the radius of the circle/sphere that encloses the object. Therefore, one must be aware that systems with large disparities between triangle/line and particle radii may slow down the neighbor list build, and could benefit from using the *the multi neighbor style* command for *local* surfaces. Furthermore, triangles with very high aspect ratios should generally be avoided as they can lead to large neighbor lists containing many particles which are not actually close to being contact with the triangle (false positives).

---

### Example scripts

The `examples/gransurf` directory has example input scripts which use both *global* and *local* surfaces. Both 2d and 3d models are included.

Each script produces a series of snapshot images using the *dump image* command. The snapshots visualize both the particles and granular surfaces. The snapshots can be animated to view a movie of the simulation.

### 10.5.4 Body particles

**Overview:**

In LAMMPS, body particles are generalized finite-size particles. Individual body particles can represent complex entities, such as surface meshes of discrete points, collections of sub-particles, deformable objects, etc. Note that other kinds of finite-size spherical and aspherical particles are also supported by LAMMPS, such as spheres, ellipsoids, line segments, and triangles, but they are simpler entities than body particles. See the *Howto spherical* page for a general overview of all these particle types.

Body particles are used via the *atom_style body* command. It takes a body style as an argument. The current body styles supported by LAMMPS are as follows. The name in the first column is used as the *bstyle* argument for the *atom_style body* command.

| | |
|---|---|
| *nparticle* | rigid body with N sub-particles |
| *rounded/polygon* | 2d polygons with N vertices |
| *rounded/polyhedron* | 3d polyhedra with N vertices, E edges and F faces |

The body style determines what attributes are stored for each body and thus how they can be used to compute pairwise body/body or bond/non-body (point particle) interactions. More details of each style are described below.

More styles may be added in the future. See the *page on creating new body styles* for details on how to add a new body style to the code.

---

**When to use body particles:**

You should not use body particles to model a rigid body made of simpler particles (e.g. point, sphere, ellipsoid, line segment, triangular particles), if the interaction between pairs of rigid bodies is just the summation of pairwise interactions between the simpler particles. LAMMPS already supports this kind of model via the *fix rigid* command. Any of the numerous pair styles that compute interactions between simpler particles can be used. The *fix rigid* command time integrates the motion of the rigid bodies. All of the standard LAMMPS commands for thermostatting, adding constraints, performing output, etc will operate as expected on the simple particles.

By contrast, when body particles are used, LAMMPS treats an entire body as a single particle for purposes of computing pairwise interactions, building neighbor lists, migrating particles between processors, output of particles to a dump file, etc. This means that interactions between pairs of bodies or between a body and non-body (point) particle need to be encoded in an appropriate pair style. If such a pair style were to mimic the *fix rigid* model, it would need to loop over the entire collection of interactions between pairs of simple particles within the two bodies, each time a single body/body interaction was computed.

Thus it only makes sense to use body particles and develop such a pair style, when particle/particle interactions are more complex than what the *fix rigid* command can already calculate. For example, consider particles with one or more of the following attributes:

- represented by a surface mesh
- represented by a collection of geometric entities (e.g. planes + spheres)
- deformable
- internal stress that induces fragmentation

For these models, the interaction between pairs of particles is likely to be more complex than the summation of simple pairwise interactions. An example is contact or frictional forces between particles with planar surfaces that interpenetrate. Likewise, the body particle may store internal state, such as a stress tensor used to compute a fracture criterion.

These are additional LAMMPS commands that can be used with body particles of different styles

| | |
|---|---|
| *fix nve/body* | integrate motion of a body particle in NVE ensemble |
| *fix nvt/body* | ditto for NVT ensemble |
| *fix npt/body* | ditto for NPT ensemble |
| *fix nph/body* | ditto for NPH ensemble |
| *compute body/local* | store sub-particle attributes of a body particle |
| *compute temp/body* | compute temperature of body particles |
| *dump local* | output sub-particle attributes of a body particle |
| *dump image* | output body particle attributes as an image |

The pair styles currently defined for use with specific body styles are listed in the sections below.

Note that for all the body styles, if the data file defines a general triclinic box, then the orientation of the body particle and its corresponding 6 moments of inertia and other orientation-dependent values should reflect the fact the body is defined within a general triclinic box with edge vectors **A**,**B**,**C**. LAMMPS will rotate the box to convert it to a restricted triclinic box. This operation will also rotate the orientation of the body particles. See the *Howto triclinic* doc page for more details. The sections below highlight the orientation-dependent values specific to each body style.

---

**Specifics of body style nparticle:**

The *nparticle* body style represents body particles as a rigid body with a variable number N of sub-particles. It is provided as a vanilla, prototypical example of a body particle, although as mentioned above, the *fix rigid* command already duplicates its functionality.

The atom_style body command for this body style takes two additional arguments:

```
atom_style body nparticle Nmin Nmax
Nmin = minimum # of sub-particles in any body in the system
Nmax = maximum # of sub-particles in any body in the system
```

The Nmin and Nmax arguments are used to bound the size of data structures used internally by each particle.

When the *read_data* command reads a data file for this body style, the following information must be provided for each entry in the *Bodies* section of the data file:

```
atom-ID 1 M
N
ixx iyy izz ixy ixz iyz
x1 y1 z1
...
xN yN zN
```

where M = 6 + 3*N, and N is the number of sub-particles in the body particle.

The integer line has a single value N. The floating point line(s) list 6 moments of inertia followed by the coordinates of the N sub-particles (x1 to zN) as 3N values. These values can be listed on as many lines as you wish; see the *read_data* command for more details.

The 6 moments of inertia (ixx,iyy,izz,ixy,ixz,iyz) should be the values consistent with the current orientation of the rigid body around its center of mass. The values are with respect to the simulation box XYZ axes, not with respect to the principal axes of the rigid body itself. LAMMPS performs the latter calculation internally.

The coordinates of each sub-particle are specified as its x,y,z displacement from the center-of-mass of the body particle. The center-of-mass position of the particle is specified by the x,y,z values in the *Atoms* section of the data file, as is the total mass of the body particle.

Note that if the data file defines a general triclinic simulation box, these sub-particle displacements are orientation-dependent and, as mentioned above, should reflect the body particle's orientation within the general triclinic box.

The *pair_style body/nparticle* command can be used with this body style to compute body/body and body/non-body interactions.

---

**Specifics of body style rounded/polygon:**

The *rounded/polygon* body style represents body particles as a 2d polygon with a variable number of N vertices. This style can only be used for 2d models; see the *dimension* command. See the *pair_style body/rounded/polygon* page for a diagram of two squares with rounded circles at the vertices. Special cases for N = 1 (circle) and N = 2 (rod with rounded ends) can also be specified.

One use of this body style is for 2d discrete element models, as described in *Fraige*.

Similar to body style *nparticle*, the atom_style body command for this body style takes two additional arguments:

```
atom_style body rounded/polygon Nmin Nmax
Nmin = minimum # of vertices in any body in the system
Nmax = maximum # of vertices in any body in the system
```

The Nmin and Nmax arguments are used to bound the size of data structures used internally by each particle.

When the *read_data* command reads a data file for this body style, the following information must be provided for each body in the *Bodies* section of the data file:

```
atom-ID 1 M
N
ixx iyy izz ixy ixz iyz
x1 y1 z1
...
xN yN zN
diameter
```

where M = 6 + 3*N + 1, and N is the number of vertices in the body particle.

The integer line has a single value N. The floating point line(s) list 6 moments of inertia, followed by the coordinates of the N vertices (x1 to zN) as 3N values (with z = 0.0 for each), followed by a diameter value = the rounded diameter of the circle that surrounds each vertex. The diameter value can be different for each body particle. These floating-point values can be listed on as many lines as you wish; see the *read_data* command for more details.

**Note**

It is important that the vertices for each polygonal body particle be listed in order around its perimeter, so that edges can be inferred. LAMMPS does not check that this is the case.

The 6 moments of inertia (ixx,iyy,izz,ixy,ixz,iyz) should be the values consistent with the current orientation of the rigid body around its center of mass. The values are with respect to the simulation box XYZ axes, not with respect to the principal axes of the rigid body itself. LAMMPS performs the latter calculation internally.

The coordinates of each vertex are specified as its x,y,z displacement from the center-of-mass of the body particle. The center-of-mass position of the particle is specified by the x,y,z values in the *Atoms* section of the data file.

For example, the following information would specify a square particle whose edge length is sqrt(2) and rounded diameter is 1.0. The orientation of the square is aligned with the xy coordinate axes which is consistent with the 6 moments of inertia: ixx iyy izz ixy ixz iyz = 1 1 4 0 0 0. Note that only Izz matters in 2D simulations.

```
3 1 19
4
1 1 4 0 0 0
-0.7071 -0.7071 0
-0.7071 0.7071 0
0.7071 0.7071 0
0.7071 -0.7071 0
1.0
```

A rod in 2D, whose length is 4.0, mass 1.0, rounded at two ends by circles of diameter 0.5, is specified as follows:

```
1 1 13
2
1 1 1.33333 0 0 0
-2 0 0
2 0 0
0.5
```

A disk, whose diameter is 3.0, mass 1.0, is specified as follows:

```
1 1 10
1
1 1 4.5 0 0 0
0 0 0
3.0
```

Note that if the data file defines a general triclinic simulation box, these polygon vertex displacements are orientation-dependent and, as mentioned above, should reflect the body particle's orientation within the general triclinic box.

The *pair_style body/rounded/polygon* command can be used with this body style to compute body/body interactions. The *fix wall/body/polygon* command can be used with this body style to compute the interaction of body particles with a wall.

---

**Specifics of body style rounded/polyhedron:**

The *rounded/polyhedron* body style represents body particles as a 3d polyhedron with a variable number of N vertices, E edges and F faces. This style can only be used for 3d models; see the *boundary* command. See the "pair_style body/rounded/polygon" page for a diagram of a two 2d squares with rounded circles at the vertices. A 3d cube with rounded spheres at the 8 vertices and 12 rounded edges would be similar. Special cases for N = 1 (sphere) and N = 2 (rod with rounded ends) can also be specified.

This body style is for 3d discrete element models, as described in *Wang*.

Similar to body style *rounded/polygon*, the atom_style body command for this body style takes two additional arguments:

```
atom_style body rounded/polyhedron Nmin Nmax
Nmin = minimum # of vertices in any body in the system
Nmax = maximum # of vertices in any body in the system
```

The Nmin and Nmax arguments are used to bound the size of data structures used internally by each particle.

When the *read_data* command reads a data file for this body style, the following information must be provided for each entry in the *Bodies* section of the data file:

```
atom-ID 3 M
N E F
ixx iyy izz ixy ixz iyz
x1 y1 z1
...
xN yN zN
0 1
1 2
2 3
...
0 1 2 -1
0 2 3 -1
...
1 2 3 4
diameter
```

where M = 6 + 3*N + 2*E + 4*F + 1, and N is the number of vertices in the body particle, E = number of edges, F = number of faces. For N = 1 or 2, the format is simpler. E and F are ignored and no edges or faces are listed, so that M = 6 + 3*N + 1.

The integer line has three values: number of vertices (N), number of edges (E) and number of faces (F). The floating point line(s) list 6 moments of inertia followed by the coordinates of the N vertices (x1 to zN) as 3N values, followed by 2E vertex indices corresponding to the end points of the E edges, then 4*F vertex indices defining F faces. The last value is the diameter value = the rounded diameter of the sphere that surrounds each vertex. The diameter value can be different for each body particle. These floating-point values can be listed on as many lines as you wish; see the *read_data* command for more details.

Note that vertices are numbered from 0 to N-1 inclusive. The order of the 2 vertices in each edge does not matter. Faces can be triangles or quadrilaterals. In both cases 4 vertices must be specified. For a triangle the 4th vertex is -1. The 4 vertices within each triangle or quadrilateral face should be ordered by the right-hand rule so that the normal vector of the face points outwards from the center of mass. For polyhedron with faces with more than 4 vertices, you should split the complex face into multiple simple faces, each of which is a triangle or quadrilateral.

**Note**

If a face is a quadrilateral then its 4 vertices must be co-planar. LAMMPS does not check that this is the case. If you have a quad-face of a polyhedron that is not planar (e.g. a cube whose vertices have been randomly displaced), then you should represent the single quad face as two triangle faces instead.

The 6 moments of inertia (ixx,iyy,izz,ixy,ixz,iyz) should be the values consistent with the current orientation of the rigid body around its center of mass. The values are with respect to the simulation box XYZ axes, not with respect to the principal axes of the rigid body itself. LAMMPS performs the latter calculation internally.

The coordinates of each vertex are specified as its x,y,z displacement from the center-of-mass of the body particle. The center-of-mass position of the particle is specified by the x,y,z values in the *Atoms* section of the data file.

For example, the following information would specify a cubic particle whose edge length is 2.0 and rounded diameter is 0.5. The orientation of the cube is aligned with the xyz coordinate axes which is consistent with the 6 moments of inertia: ixx iyy izz ixy ixz iyz = 0.667 0.667 0.667 0 0 0.

```
1 3 79
8 12 6
0.667 0.667 0.667 0 0 0
1 1 1
```

(continues on next page)

(continued from previous page)

```
1 -1 1
-1 -1 1
-1 1 1
1 1 -1
1 -1 -1
-1 -1 -1
-1 1 -1
0 1
1 2
2 3
3 0
4 5
5 6
6 7
7 4
0 4
1 5
2 6
3 7
0 1 2 3
4 5 6 7
0 1 5 4
1 2 6 5
2 3 7 6
3 0 4 7
0.5
```

A rod in 3D, whose length is 4.0, mass 1.0 and rounded at two ends by circles of diameter 0.5, is specified as follows:

```
1 3 13
2 1 1
0 1.33333 1.33333 0 0 0
-2 0 0
2 0 0
0.5
```

A sphere whose diameter is 3.0 and mass 1.0, is specified as follows:

```
1 3 10
1 1 1
0.9 0.9 0.9 0 0 0
0 0 0
3.0
```

The number of edges and faces for a rod or sphere must be listed, but is ignored.

Note that if the data file defines a general triclinic simulation box, these polyhedron vertex displacements are orientation-dependent and, as mentioned above, should reflect the body particle's orientation within the general triclinic box.

The *pair_style body/rounded/polyhedron* command can be used with this body style to compute body/body interactions. The *fix wall/body/polyhedron* command can be used with this body style to compute the interaction of body particles with a wall.

---

**Output specifics for all body styles:**

For the *compute body/local* and *dump local* commands, all 3 of the body styles described on this page produces one datum for each of the N vertices (of sub-particles) in a body particle. The datum has 3 values:

```
1 = x position of vertex (or sub-particle)
2 = y position of vertex
3 = z position of vertex
```

These values are the current position of the vertex within the simulation domain, not a displacement from the center-of-mass (COM) of the body particle itself. These values are calculated using the current COM and orientation of the body particle.

The *dump image* command and its *body* keyword can be used to render body particles.

For the *nparticle* body style, each body is drawn as a collection of spheres, one for each sub-particle. The size of each sphere is determined by the *bflag1* parameter for the *body* keyword. The *bflag2* argument is ignored.

For the *rounded/polygon* body style, each body is drawn as a polygon. The *bflag2* argument selects how the bodies are represented: for a value of 1, only the faces are drawn; for a value of 2 instead only a wireframe outline is drawn; and for a value of 3 both the faces and the wireframe outline are drawn. The diameter of the wireframe cylinders is determined by the *bflag1* parameter for the *body* keyword.

For the *rounded/polyhedron* body style, each body is drawn as a polyhedron consisting of triangles and squares. The *bflag2* argument selects how the bodies are represented: for a value of 1, only the faces are drawn with some shading added; for a value of 2 instead only a wireframe mesh is drawn; for a value of 3 both the faces and the wireframe mesh are drawn; and for a value of 4 only the faces are drawn, but without shading. The diameter of the wireframe cylinders is determined by the *bflag1* parameter for the *body* keyword.

Note that for both the *rounded/polygon* and *rounded/polyhedron* styles, line segments are drawn between the pairs of vertices. Depending on the diameters of the line segments this may be slightly different than the physical extent of the body as calculated by the *pair_style rounded/polygon* or *pair_style rounded/polyhedron* commands. Conceptually, the pair styles define the surface of a 2d or 3d body by lines or planes that are tangent to the finite-size spheres of specified diameter which are placed on each vertex position.

---

**(Fraige)** F. Y. Fraige, P. A. Langston, A. J. Matchett, J. Dodds, Particuology, 6, 455 (2008).

**(Wang)** J. Wang, H. S. Yu, P. A. Langston, F. Y. Fraige, Granular Matter, 13, 1 (2011).

### 10.5.5 Bonded particle models

The BPM package implements bonded particle models which can be used to simulate mesoscale solids. Solids are constructed as a collection of particles, which each represent a coarse-grained region of space much larger than the atomistic scale. Particles within a solid region are then connected by a network of bonds to model solid elasticity. There are many names for methods that are based on similar (or equivalent) capabilities to those in this package, including, but not limited to, cohesive beam models, bonded DEMs, lattice spring models, mass spring models, and lattice particle methods.

Unlike traditional bonds in molecular dynamics, the equilibrium bond length can vary between bonds. Bonds store the reference state. This includes setting the equilibrium length equal to the initial distance between the two particles, but can also include data on the bond orientation for rotational models. This produces a stress-free initial state. Furthermore, bonds are allowed to break under large strains, producing fracture. The examples/bpm directory has sample input scripts for simulations of the fragmentation of an impacted plate and the pouring of extended, elastic bodies. See *(Clemmer)* for more general information on the approach and the LAMMPS implementation. Example movies illustrating some of these capabilities are found at https://www.lammps.org/movies.html#bpmpackage.

---

Bonds can be created using a *read data* or *create bonds* command. Alternatively, a *molecule* template with bonds can be used with *fix deposit* or *fix pour* to create solid grains.

In this implementation, bonds store their reference state when they are first computed in the setup of the first simulation run. Data is then preserved across run commands and is written to *binary restart files* such that restarting the system will not reset the reference state of a bond. Bonds that are created midway into a run, such as those created by pouring grains using *fix pour*, are initialized on that timestep.

---

Currently, there are three types of bonds included in the BPM package. The first bond style, *bond bpm/spring*, only applies pairwise, central body forces. Point particles must have *bond atom style* and may be thought of as nodes in a spring network. An optional multibody term can be used to adjust the network's Poisson's ratio. The *bpm/spring/plastic* bond style is similar except it adds a plastic yield strain. Alternatively, the third bond style, *bond bpm/rotational*, resolves tangential forces and torques arising with the shearing, bending, and twisting of the bond due to rotation or displacement of particles. Particles are similar to those used in the *granular package*, *atom style sphere*. However, they must also track the current orientation of particles and store bonds, and therefore use a *bpm/sphere atom style*. This also requires a unique integrator *fix nve/bpm/sphere* which numerically integrates orientation similar to *fix nve/asphere*.

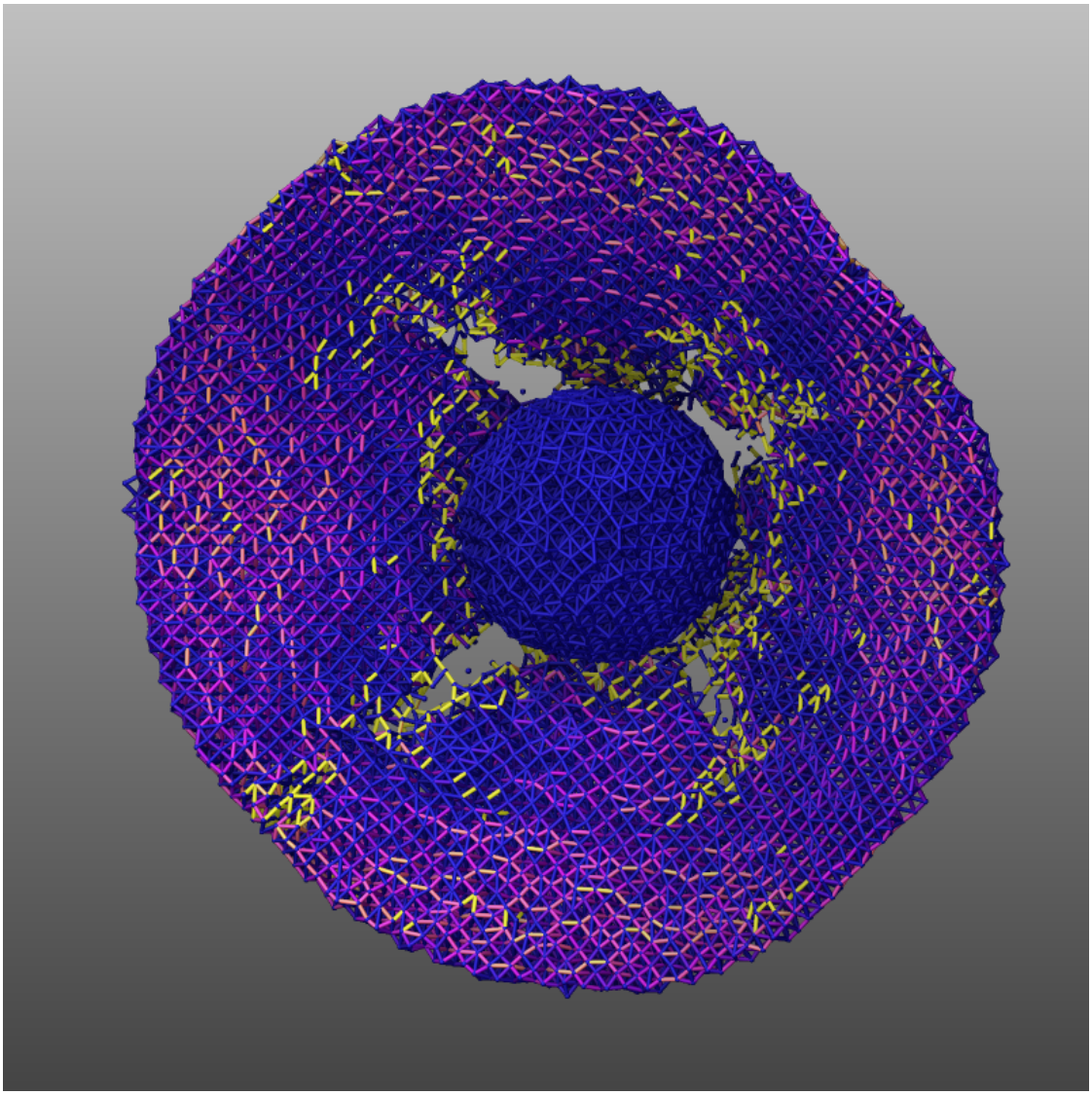
In addition to bond styles, a new pair style *pair bpm/spring* was added to accompany the bpm/spring bond style. By default, this pair style is simply a hookean repulsion with similar velocity damping as its sister bond style, but optional arguments can be used to modify the force.

---

Bond data can be output using a combination of standard LAMMPS commands. A list of IDs for bonded atoms can be generated using the *compute property/local* command. Various properties of bonds can be computed using the *compute bond/local* command. This command allows one to access data saved to the bond's history, such as the reference length of the bond. More information on bond history data can be found on the documentation pages for the specific BPM bond styles. Finally, this data can be output using a *dump local* command. As one may output many columns from the same compute, the *dump modify* ***colname*** option may be used to provide more helpful column names. An example of this procedure is found in `/examples/bpm/pour/`. External software, such as OVITO, can read these dump files to render bond data.

Added in version 4Jul2026.

The *compute bond/local* property can also be visualized using *dump image* to color bonds by property using a color map. The `/examples/GRAPHICS/in.bpm.fracture` example file shows how such a visualization can be constructed.



As bonds can potentially be broken between neighbor list builds, BPM bond styles may place restrictions on the *special_bonds* command. There are three possible scenarios which determine how pair interactions between bonded particles and special bond weights work.

The first option is the simplest. If bonds cannot break, then one can use any special bond settings to control pair forces. Namely, this is accomplished by setting the *break* keyword to *no*. Note that a zero coul weight for 1-2 bonds can be used to exclude bonded atoms from the neighbor list builds

```
special_bonds lj 0 1 1 coul 0 1 1
```

This can be useful for post-processing, or to determine pair interaction properties between distinct bonded particles.

If bonds can break, the second scenario is if pair forces are overlaid on top of bond forces such that atoms can simultaneously exchange both types of forces. This is accomplished by setting the *overlay/pair* keyword present in all bpm bond styles to *yes*. This case requires the following special bond settings

```
special_bonds lj/coul 1 1 1
```

Note that this scenario does not update special bond lists when bonds break, hence why fractional weights are not allowed. Whether or not two particles are bonded has no bearing on pair forces.

In the third scenario, bonds can break but pair forces are disabled between bonded particles. This is the default behavior of BPM bond styles. Unlike *bond quartic*, pair forces are not removed by subtracting pair forces during the bond computation, but rather by dynamically updating the 1-2 special bond list. To do this, LAMMPS requires *newton* bond off such that all processors containing an atom know when a bond breaks. Additionally, one must use the following special bond settings

```
special_bonds lj 0 1 1 coul 1 1 1
```

These settings accomplish two goals. First, they turn off 1-3 and 1-4 special bond lists, which are not currently supported for breakable BPMs. As BPMs often have dense bond networks, generating/updating 1-3 and 1-4 special bond lists can be expensive. By setting the lj weight for 1-2 bonds to zero, this turns off pairwise interactions. Even though there are no charges in BPM models, setting a nonzero coul weight for 1-2 bonds ensures all bonded neighbors are still included in the neighbor list in case bonds break between neighbor list builds.

To monitor the fracture of bonds in the system, all BPM bond styles have the ability to record instances of bond breakage to output using the *dump local* command. Since one may frequently output a list of broken bonds and the time they broke, the *dump modify* option *header no* may be useful to avoid repeatedly printing the header of the dump file. An example of this procedure is found in /examples/bpm/impact/. Additionally, one can use *compute nbond/atom* to tally the current number of bonds per atom.

See the *Howto* page on broken bonds for more information.

---

While LAMMPS has many utilities to create and delete bonds, *only* the following are currently compatible with BPM bond styles:

- *create_bonds*
- *delete_bonds*
- *fix bond/create*
- *fix bond/break*
- *fix bond/swap*

**Note**

The *create_bonds* command requires certain *special_bonds* settings. To subtract pair interactions, one will need to switch between different *special_bonds* settings in the input script. An example is found in `examples/bpm/impact`.

---

**(Clemmer)** Clemmer, Monti, Lechman, Soft Matter, 20, 1702 (2024).

### 10.5.6 Polarizable models

In polarizable force fields the charge distributions in molecules and materials respond to their electrostatic environments. Polarizable systems can be simulated in LAMMPS using three methods:

- the fluctuating charge method, implemented in the *QEQ* package,
- the adiabatic core-shell method, implemented in the *CORESHELL* package,
- the thermalized Drude dipole method, implemented in the *DRUDE* package.

The fluctuating charge method calculates instantaneous charges on interacting atoms based on the electronegativity equalization principle. It is implemented in the *fix qeq* which is available in several variants. It is a relatively efficient technique since no additional particles are introduced. This method allows for charge transfer between molecules or atom groups. However, because the charges are located at the interaction sites, off-plane components of polarization cannot be represented in planar molecules or atom groups.

The two other methods share the same basic idea: polarizable atoms are split into one core atom and one satellite particle (called shell or Drude particle) attached to it by a harmonic spring. Both atoms bear a charge and they represent collectively an induced electric dipole. These techniques are computationally more expensive than the QEq method because of additional particles and bonds. These two charge-on-spring methods differ in certain features, with the core-shell model being normally used for ionic/crystalline materials, whereas the so-called Drude model is normally used for molecular systems and fluid states.

The core-shell model is applicable to crystalline materials where the high symmetry around each site leads to stable trajectories of the core-shell pairs. However, bonded atoms in molecules can be so close that a core would interact too strongly or even capture the Drude particle of a neighbor. The Drude dipole model is relatively more complex in order to remedy this and other issues. Specifically, the Drude model includes specific thermostatting of the core-Drude pairs and short-range damping of the induced dipoles.

The three polarization methods can be implemented through a self-consistent calculation of charges or induced dipoles at each timestep. In the fluctuating charge scheme this is done by the matrix inversion method in *fix qeq/point*, but for core-shell or Drude-dipoles the relaxed-dipoles technique would require an slow iterative procedure. These self-consistent solutions yield accurate trajectories since the additional degrees of freedom representing polarization are massless. An alternative is to attribute a mass to the additional degrees of freedom and perform time integration using an extended Lagrangian technique. For the fluctuating charge scheme this is done by *fix qeq/dynamic*, and for the charge-on-spring models by the methods outlined in the next two sections. The assignment of masses to the additional degrees of freedom can lead to unphysical trajectories if care is not exerted in choosing the parameters of the polarizable models and the simulation conditions.

In the core-shell model the vibration of the shells is kept faster than the ionic vibrations to mimic the fast response of the polarizable electrons. But in molecular systems thermalizing the core-Drude pairs at temperatures comparable to the rest of the simulation leads to several problems (kinetic energy transfer, too short a timestep, etc.) In order to avoid these problems the relative motion of the Drude particles with respect to their cores is kept "cold" so the vibration of the core-Drude pairs is very slow, approaching the self-consistent regime. In both models the temperature is regulated using the velocities of the center of mass of core+shell (or Drude) pairs, but in the Drude model the actual relative core-Drude particle motion is thermostatted separately as well.

### 10.5.7 Adiabatic core/shell model

The adiabatic core-shell model by *Mitchell and Fincham* is a simple method for adding polarizability to a system. In order to mimic the electron shell of an ion, a satellite particle is attached to it. This way the ions are split into a core and a shell where the latter is meant to react to the electrostatic environment inducing polarizability. See the *Howto polarizable* page for a discussion of all the polarizable models available in LAMMPS.

Technically, shells are attached to the cores by a spring force f = k*r where k is a parameterized spring constant and r is the distance between the core and the shell. The charges of the core and the shell add up to the ion charge, thus q(ion) = q(core) + q(shell). This setup introduces the ion polarizability (alpha) given by alpha = q(shell)^2 / k. In a similar fashion the mass of the ion is distributed on the core and the shell with the core having the larger mass.

To run this model in LAMMPS, *atom_style full* can be used since atom charge and bonds are needed. Each kind of core/shell pair requires two atom types and a bond type. The core and shell of a core/shell pair should be bonded to each other with a harmonic bond that provides the spring force. For example, a data file for NaCl, as found in examples/coreshell, has this format:

```
432   atoms  # core and shell atoms
216   bonds  # number of core/shell springs
```

(continues on next page)

(continued from previous page)

```
4     atom types  # 2 cores and 2 shells for Na and Cl
2     bond types

0.0 24.09597 xlo xhi
0.0 24.09597 ylo yhi
0.0 24.09597 zlo zhi

Masses       # core/shell mass ratio = 0.1

1 20.690784  # Na core
2 31.90500   # Cl core
3 2.298976   # Na shell
4 3.54500    # Cl shell

Atoms

1    1    2   1.5005    0.00000000   0.00000000   0.00000000 # core of core/shell pair 1
2    1    4  -2.5005    0.00000000   0.00000000   0.00000000 # shell of core/shell pair 1
3    2    1   1.5056    4.01599500   4.01599500   4.01599500 # core of core/shell pair 2
4    2    3  -0.5056    4.01599500   4.01599500   4.01599500 # shell of core/shell pair 2
(...)

Bonds   # Bond topology for spring forces

1     2     1     2   # spring for core/shell pair 1
2     2     3     4   # spring for core/shell pair 2
(...)
```

Non-Coulombic (e.g. Lennard-Jones) pairwise interactions are only defined between the shells. Coulombic interactions are defined between all cores and shells. If desired, additional bonds can be specified between cores.

The *special_bonds* command should be used to turn-off the Coulombic interaction within core/shell pairs, since that interaction is set by the bond spring. This is done using the *special_bonds* command with a 1-2 weight = 0.0, which is the default value. It needs to be considered whether one has to adjust the *special_bonds* weighting according to the molecular topology since the interactions of the shells are bypassed over an extra bond.

Note that this core/shell implementation does not require all ions to be polarized. One can mix core/shell pairs and ions without a satellite particle if desired.

Since the core/shell model permits distances of r = 0.0 between the core and shell, a pair style with a "cs" suffix needs to be used to implement a valid long-range Coulombic correction. Several such pair styles are provided in the CORESHELL package. See *this page* for details. All of the core/shell enabled pair styles require the use of a long-range Coulombic solver, as specified by the *kspace_style* command. Either the PPPM or Ewald solvers can be used.

For the NaCL example problem, these pair style and bond style settings are used:

```
pair_style      born/coul/long/cs 20.0 20.0
pair_coeff      * *      0.0 1.000   0.00  0.00   0.00
pair_coeff      3 3    487.0 0.23768 0.00  1.05   0.50 #Na-Na
pair_coeff      3 4 145134.0 0.23768 0.00  6.99   8.70 #Na-Cl
pair_coeff      4 4 405774.0 0.23768 0.00 72.40 145.40 #Cl-Cl

bond_style      harmonic
```

(continues on next page)

(continued from previous page)

```
bond_coeff      1 63.014 0.0
bond_coeff      2 25.724 0.0
```

When running dynamics with the adiabatic core/shell model, the following issues should be considered. The relative motion of the core and shell particles corresponds to the polarization, hereby an instantaneous relaxation of the shells is approximated and a fast core/shell spring frequency ensures a nearly constant internal kinetic energy during the simulation. Thermostats can alter this polarization behavior, by scaling the internal kinetic energy, meaning the shell will not react freely to its electrostatic environment. Therefore it is typically desirable to decouple the relative motion of the core/shell pair, which is an imaginary degree of freedom, from the real physical system. To do that, the *compute temp/cs* command can be used, in conjunction with any of the thermostat fixes, such as *fix nvt* or *fix langevin*. This compute uses the center-of-mass velocity of the core/shell pairs to calculate a temperature, and ensures that velocity is what is rescaled for thermostatting purposes. This compute also works for a system with both core/shell pairs and non-polarized ions (ions without an attached satellite particle). The *compute temp/cs* command requires input of two groups, one for the core atoms, another for the shell atoms. Non-polarized ions which might also be included in the treated system should not be included into either of these groups, they are taken into account by the *group-ID* (second argument) of the compute. The groups can be defined using the *group *type** command. Note that to perform thermostatting using this definition of temperature, the *fix modify temp* command should be used to assign the compute to the thermostat fix. Likewise the *thermo_modify temp* command can be used to make this temperature be output for the overall system.

For the NaCl example, this can be done as follows:

```
group cores type 1 2
group shells type 3 4
compute CSequ all temp/cs cores shells
fix thermoberendsen all temp/berendsen 1427 1427 0.4    # thermostat for the true␣
↪physical system
fix thermostatequ all nve                               # integrator as needed for the␣
↪berendsen thermostat
fix_modify thermoberendsen temp CSequ
thermo_modify temp CSequ                                # output of center-of-mass␣
↪derived temperature
```

The pressure for the core/shell system is computed via the regular LAMMPS convention by *treating the cores and shells as individual particles*. For the thermo output of the pressure as well as for the application of a barostat, it is necessary to use an additional *pressure* compute based on the default *temperature* and specifying it as a second argument in *fix modify* and *thermo_modify* resulting in:

```
(...)
compute CSequ all temp/cs cores shells
compute thermo_press_lmp all pressure thermo_temp       # pressure for individual␣
↪particles
thermo_modify temp CSequ press thermo_press_lmp         # modify thermo to regular␣
↪pressure
fix press_bar all npt temp 300 300 0.04 iso 0 0 0.4
fix_modify press_bar temp CSequ press thermo_press_lmp  # pressure modification for␣
↪correct kinetic scalar
```

If *compute temp/cs* is used, the decoupled relative motion of the core and the shell should in theory be stable. However numerical fluctuation can introduce a small momentum to the system, which is noticeable over long trajectories. Therefore it is recommendable to use the *fix momentum* command in combination with *compute temp/cs* when equilibrating the system to prevent any drift.

When initializing the velocities of a system with core/shell pairs, it is also desirable to not introduce energy into the

relative motion of the core/shell particles, but only assign a center-of-mass velocity to the pairs. This can be done by using the *bias* keyword of the *velocity create* command and assigning the *compute temp/cs* command to the *temp* keyword of the *velocity* command, e.g.

```
velocity all create 1427 134 bias yes temp CSequ
velocity all scale 1427 temp CSequ
```

To maintain the correct polarizability of the core/shell pairs, the kinetic energy of the internal motion shall remain nearly constant. Therefore the choice of spring force and mass ratio need to ensure much faster relative motion of the two atoms within the core/shell pair than their center-of-mass velocity. This allows the shells to effectively react instantaneously to the electrostatic environment and limits energy transfer to or from the core/shell oscillators. This fast movement also dictates the timestep that can be used.

The primary literature of the adiabatic core/shell model suggests that the fast relative motion of the core/shell pairs only allows negligible energy transfer to the environment. The mentioned energy transfer will typically lead to a small drift in total energy over time. This internal energy can be monitored using the *compute chunk/atom* and *compute temp/chunk* commands. The internal kinetic energies of each core/shell pair can then be summed using the sum() special function of the *variable* command. Or they can be time/averaged and output using the *fix ave/time* command. To use these commands, each core/shell pair must be defined as a "chunk". If each core/shell pair is defined as its own molecule, the molecule ID can be used to define the chunks. If cores are bonded to each other to form larger molecules, the chunks can be identified by the *fix property/atom* via assigning a core/shell ID to each atom using a special field in the data file read by the *read_data* command. This field can then be accessed by the *compute property/atom* command, to use as input to the *compute chunk/atom* command to define the core/shell pairs as chunks.

For example if core/shell pairs are the only molecules:

```
read_data NaCl_CS_x0.1_prop.data
compute prop all property/atom molecule
compute cs_chunk all chunk/atom c_prop
compute cstherm all temp/chunk cs_chunk temp internal com yes cdof 3.0     # note the␣
↪chosen degrees of freedom for the core/shell pairs
fix ave_chunk all ave/time 10 1 10 c_cstherm file chunk.dump mode vector
```

For example if core/shell pairs and other molecules are present:

```
fix csinfo all property/atom i_CSID                          # property/atom command
read_data NaCl_CS_x0.1_prop.data fix csinfo NULL CS-Info  # atom property added in the␣
↪data-file
compute prop all property/atom i_CSID
(...)
```

The additional section in the data file would be formatted like this:

```
CS-Info         # header of additional section

1   1           # column 1 = atom ID, column 2 = core/shell ID
2   1
3   2
4   2
5   3
6   3
7   4
8   4
(...)
```

**(Mitchell and Fincham)** Mitchell, Fincham, J Phys Condensed Matter, 5, 1031-1038 (1993).

**(Fincham)** Fincham, Mackrodt and Mitchell, J Phys Condensed Matter, 6, 393-404 (1994).

### 10.5.8 Drude induced dipoles

The thermalized Drude model represents induced dipoles by a pair of charges (the core atom and the Drude particle) connected by a harmonic spring. See the *Howto polarizable* doc page for a discussion of all the polarizable models available in LAMMPS.

The Drude model has a number of features aimed at its use in molecular systems (*Lamoureux and Roux*):

- Thermostatting of the additional degrees of freedom associated with the induced dipoles at very low temperature, in terms of the reduced coordinates of the Drude particles with respect to their cores. This makes the trajectory close to that of relaxed induced dipoles.
- Consistent definition of 1-2 to 1-4 neighbors. A core-Drude particle pair represents a single (polarizable) atom, so the special screening factors in a covalent structure should be the same for the core and the Drude particle. Drude particles have to inherit the 1-2, 1-3, 1-4 special neighbor relations from their respective cores.
- Stabilization of the interactions between induced dipoles. Drude dipoles on covalently bonded atoms interact too strongly due to the short distances, so an atom may capture the Drude particle of a neighbor, or the induced dipoles within the same molecule may align too much. To avoid this, damping at short range can be done by Thole functions (for which there are physical grounds). This Thole damping is applied to the point charges composing the induced dipole (the charge of the Drude particle and the opposite charge on the core, not to the total charge of the core atom).

A detailed tutorial covering the usage of Drude induced dipoles in LAMMPS is on the *here*.

As with the core-shell model, the cores and Drude particles should appear in the data file as standard atoms. The same holds for the springs between them, which are described by standard harmonic bonds. The nature of the atoms (core, Drude particle or non-polarizable) is specified via the *fix drude* command. The special list of neighbors is automatically refactored to account for the equivalence of core and Drude particles as regards special 1-2 to 1-4 screening. It may be necessary to use the *extra/special/per/atom* keyword of the *read_data* command. If using *fix shake*, make sure no Drude particle is in this fix group.

There are three ways to thermostat the Drude particles at a low temperature: use either *fix langevin/drude* for a Langevin thermostat, or *fix drude/transform/** for a Nose-Hoover thermostat, or *fix tgnvt/drude* for a temperature-grouped Nose-Hoover thermostat. The first and third require use of the command *comm_modify vel yes*. The second requires two separate integration fixes like *nvt* or *npt*. The correct temperatures of the reduced degrees of freedom can be calculated using the *compute temp/drude*. This requires also to use the command *comm_modify vel yes*.

Short-range damping of the induced dipole interactions can be achieved using Thole functions through the *pair style thole* in *pair_style hybrid/overlay* with a Coulomb pair style. It may be useful to use *coul/long/cs* or similar from the CORESHELL package if the core and Drude particle come too close, which can cause numerical issues.

---

**(Lamoureux and Roux)** G. Lamoureux, B. Roux, J. Chem. Phys 119, 3025 (2003)

### 10.5.9 Tutorial for Thermalized Drude oscillators in LAMMPS

This Howto document explains how to use Drude oscillators in LAMMPS to simulate polarizable systems using the DRUDE package. As an illustration, the input files for a simulation of 250 phenol molecules are documented. First of all, LAMMPS has to be compiled with the DRUDE package included. Then, the data file and input scripts have to be modified to include the Drude dipoles and how to handle them.

Example input scripts are available: `examples/PACKAGES/drude`

---

### Overview of Drude induced dipoles

Polarizable atoms acquire an induced electric dipole moment under the action of an external electric field, for example the electric field created by the surrounding particles. Drude oscillators represent these dipoles by two fixed charges: the core (DC) and the Drude particle (DP) bound by a harmonic potential. The Drude particle can be thought of as the electron cloud whose center can be displaced from the position of the corresponding nucleus.

The sum of the masses of a core-Drude pair should be the mass of the initial (unsplit) atom, $m_C + m_D = m$. The sum of their charges should be the charge of the initial (unsplit) atom, $q_C + q_D = q$. A harmonic potential between the core and Drude partners should be present, with force constant $k_D$ and an equilibrium distance of zero. The (half-)stiffness of the *harmonic bond* $K_D = k_D/2$ and the Drude charge $q_D$ are related to the atom polarizability $\alpha$ by

$$K_D = \frac{1}{2}\frac{q_D^2}{\alpha}$$

Ideally, the mass of the Drude particle should be small, and the stiffness of the harmonic bond should be large, so that the Drude particle remains close to the core. The values of Drude mass, Drude charge, and force constant can be chosen following different strategies, as in the following examples of polarizable force fields:

- *Lamoureux and Roux* suggest adopting a global half-stiffness, $K_D$ = 500 kcal/(mol Ang $^2$) - which corresponds to a force constant $k_D$ = 4184 kJ/(mol Ang $^2$) - for all types of core-Drude bond, a global mass $m_D$ = 0.4 g/mol (or u) for all types of Drude particles, and to calculate the Drude charges for individual atom types from the atom polarizabilities using equation (1). This choice is followed in the polarizable CHARMM force field.
- Alternately, *Schroeder and Steinhauser* suggest adopting a global charge $q_D$ = -1.0e and a global mass $m_D$ = 0.1 g/mol (or u) for all Drude particles, and to calculate the force constant for each type of core-Drude bond from equation (1). The timesteps used by these authors are between 0.5 and 2 fs, with the degrees of freedom of the Drude oscillators kept cold at 1 K.
- In both these force fields hydrogen atoms are treated as non-polarizable.

The motion of the Drude particles can be calculated by minimizing the energy of the induced dipoles at each timestep, by an iterative, self-consistent procedure. The Drude particles can be massless and therefore do not contribute to the kinetic energy. However, the relaxed method is computationally slow. An extended-lagrangian method can be used to calculate the positions of the Drude particles, but this requires them to have mass. It is important in this case to decouple the degrees of freedom associated with the Drude oscillators from those of the normal atoms. Thermalizing the Drude dipoles at temperatures comparable to the rest of the simulation leads to several problems (kinetic energy transfer, very short timestep, etc.), which can be remedied by the "cold Drude" technique (*Lamoureux and Roux*).

Two closely related models are used to represent polarization through "charges on a spring": the core-shell model and the Drude model. Although the basic idea is the same, the core-shell model is normally used for ionic/crystalline materials, whereas the Drude model is normally used for molecular systems and fluid states. In ionic crystals the symmetry around each ion and the distance between them are such that the core-shell model is sufficiently stable. But to be applicable to molecular/covalent systems the Drude model includes two important features:

1. The possibility to thermostat the additional degrees of freedom associated with the induced dipoles at very low temperature, in terms of the reduced coordinates of the Drude particles with respect to their cores. This makes the trajectory close to that of relaxed induced dipoles.
2. The Drude dipoles on covalently bonded atoms interact too strongly due to the short distances, so an atom may capture the Drude particle (shell) of a neighbor, or the induced dipoles within the same molecule may align too much. To avoid this, damping at short range of the interactions between the point charges composing the induced dipole can be done by *Thole* functions.

---

### Preparation of the data file

The data file is similar to a standard LAMMPS data file for *atom_style full*. The DPs and the *harmonic bonds* connecting them to their DC should appear in the data file as normal atoms and bonds.

You can use the *polarizer* tool (Python script distributed with the DRUDE package) to convert a non-polarizable data file (here *data.102494.lmp*) to a polarizable data file (*data-p.lmp*)

```
polarizer -q -f phenol.dff data.102494.lmp data-p.lmp
```

This will automatically insert the new atoms and bonds. The masses and charges of DCs and DPs are computed from *phenol.dff*, as well as the DC-DP bond constants. The file *phenol.dff* contains the polarizabilities of the atom types and the mass of the Drude particles, for instance:

```
# units: kJ/mol, A, deg
# kforce is in the form k/2 r_D^2
# type  m_D/u   q_D/e    k_D   alpha/A3  thole
OH      0.4    -1.0    4184.0   0.63     0.67
CA      0.4    -1.0    4184.0   1.36     2.51
CAI     0.4    -1.0    4184.0   1.09     2.51
```

The hydrogen atoms are absent from this file, so they will be treated as non-polarizable atoms. In the non-polarizable data file *data.102494.lmp*, atom names corresponding to the atom type numbers have to be specified as comments at the end of lines of the *Masses* section. You probably need to edit it to add these names. It should look like

```
Masses

1 12.011 # CAI
2 12.011 # CA
3 15.999 # OH
4 1.008  # HA
5 1.008  # HO
```

---

**Basic input file**

The atom style should be set to (or derive from) *full*, so that you can define atomic charges and molecular bonds, angles, dihedrals…

The *polarizer* tool also outputs certain lines related to the input script (the use of these lines will be explained below). In order for LAMMPS to recognize that you are using Drude oscillators, you should use the fix *drude*. The command is

```
fix DRUDE all drude C C C N N D D D
```

The N, C, D following the *drude* keyword have the following meaning: There is one tag for each atom type. This tag is C for DCs, D for DPs and N for non-polarizable atoms. Here the atom types 1 to 3 (C and O atoms) are DC, atom types 4 and 5 (H atoms) are non-polarizable and the atom types 6 to 8 are the newly created DPs.

By recognizing the fix *drude*, LAMMPS will find and store matching DC-DP pairs and will treat DP as equivalent to their DC in the *special bonds* relations. It may be necessary to extend the space for storing such special relations. In this case extra space should be reserved by using the *extra/special/per/atom* keyword of either the *read_data* or *create_box* command. With our phenol, there is 1 more special neighbor for which space is required. Otherwise LAMMPS crashes and gives the required value.

```
read_data data-p.lmp extra/special/per/atom 1
```

Let us assume we want to run a simple NVT simulation at 300 K. Note that Drude oscillators need to be thermalized at a low temperature in order to approximate a self-consistent field (SCF), therefore it is not possible to simulate an NVE ensemble with this package. Since dipoles are approximated by a charged DC-DP pair, the *pair_style* must include Coulomb interactions, for instance *lj/cut/coul/long* with *kspace_style pppm*. For example, with a cutoff of 10. and a precision 1.e-4:

```
pair_style lj/cut/coul/long 10.0
kspace_style pppm 1.0e-4
```

As compared to the non-polarizable input file, *pair_coeff* lines need to be added for the DPs. Since the DPs have no Lennard-Jones interactions, their $\varepsilon$ is 0. so the only *pair_coeff* line that needs to be added is

```
pair_coeff * 6* 0.0 0.0 # All-DPs
```

Now for the thermalization, the simplest choice is to use the *fix langevin/drude*.

```
fix LANG all langevin/drude 300. 100 12435 1. 20 13977
```

This applies a Langevin thermostat at temperature 300. to the centers of mass of the DC-DP pairs, with relaxation time 100 and with random seed 12345. This fix applies also a Langevin thermostat at temperature 1. to the relative motion of the DPs around their DCs, with relaxation time 20 and random seed 13977. Only the DCs and non-polarizable atoms need to be in this fix's group. LAMMPS will thermostat the DPs together with their DC. For this, ghost atoms need to know their velocities. Thus you need to add the following command:

```
comm_modify vel yes
```

In order to avoid that the center of mass of the whole system drifts due to the random forces of the Langevin thermostat on DCs, you can add the *zero yes* option at the end of the fix line.

If the fix *shake* is used to constrain the C-H bonds, it should be invoked after the fix *langevin/drude* for more accuracy.

```
fix SHAKE ATOMS shake 0.0001 20 0 t 4 5
```

**Note**

The group of the fix *shake* must not include the DPs. If the group *ATOMS* is defined by non-DPs atom types, you could use

Since the fix *langevin/drude* does not perform time integration (just modification of forces but no position/velocity updates), the fix *nve* should be used in conjunction.

```
fix NVE all nve
```

To avoid the flying ice cube artifact, where the atoms progressively freeze and the center of mass of the whole system drifts faster and faster, the *fix momentum* can be used. For instance:

```
fix MOMENTUM all momentum 100 linear 1 1 1
```

Finally, do not forget to update the atom type elements if you use them in a *dump_modify* ... *element* ... command, by adding the element type of the DPs. Here for instance

```
dump DUMP all custom 10 dump.lammpstrj id mol type element x y z ix iy iz
dump_modify DUMP element C C O H H D D D
```

The input file should now be ready for use!

You will notice that the global temperature *thermo_temp* computed by LAMMPS is not 300. K as wanted. This is because LAMMPS treats DPs as standard atoms in its default compute. If you want to output the temperatures of the DC-DP pair centers of mass and of the DPs relative to their DCs, you should use the *compute temp_drude*

```
compute TDRUDE all temp/drude
```

And then output the correct temperatures of the Drude oscillators using *thermo_style custom* with respectively *c_TDRUDE[1]* and *c_TDRUDE[2]*. These should be close to 300.0 and 1.0 on average.

```
thermo_style custom step temp c_TDRUDE[1] c_TDRUDE[2]
```

---

**Thole screening**

Dipolar interactions represented by point charges on springs may not be stable, for example if the atomic polarizability is too high for instance, a DP can escape from its DC and be captured by another DC, which makes the force and energy diverge and the simulation crash. Even without reaching this extreme case, the correlation between nearby dipoles on the same molecule may be exaggerated. Often, special bond relations prevent bonded neighboring atoms to see the charge of each other's DP, so that the problem does not always appear. It is possible to use screened dipole-dipole interactions by using the **pair_style thole**. This is implemented as a correction to the Coulomb pair_styles, which dampens at short distance the interactions between the charges representing the induced dipoles. It is to be used as *hybrid/overlay* with any standard *coul* pair style. In our example, we would use

```
pair_style hybrid/overlay lj/cut/coul/long 10.0 thole 2.6 10.0
```

This tells LAMMPS that we are using two pair_styles. The first one is as above (*lj/cut/coul/long 10.0*). The second one is a *thole* pair_style with default screening factor 2.6 (*Noskov*) and cutoff 10.0.

Since *hybrid/overlay* does not support mixing rules, the interaction coefficients of all the pairs of atom types with i < j should be explicitly defined. The output of the *polarizer* script can be used to complete the *pair_coeff* section of the input file. In our example, this will look like:

```
pair_coeff    1    1 lj/cut/coul/long    0.0700   3.550
pair_coeff    1    2 lj/cut/coul/long    0.0700   3.550
pair_coeff    1    3 lj/cut/coul/long    0.1091   3.310
pair_coeff    1    4 lj/cut/coul/long    0.0458   2.985
pair_coeff    2    2 lj/cut/coul/long    0.0700   3.550
pair_coeff    2    3 lj/cut/coul/long    0.1091   3.310
pair_coeff    2    4 lj/cut/coul/long    0.0458   2.985
pair_coeff    3    3 lj/cut/coul/long    0.1700   3.070
pair_coeff    3    4 lj/cut/coul/long    0.0714   2.745
pair_coeff    4    4 lj/cut/coul/long    0.0300   2.420
pair_coeff    *    5 lj/cut/coul/long    0.0000   0.000
pair_coeff    *   6* lj/cut/coul/long    0.0000   0.000
pair_coeff    1    1 thole   1.090   2.510
pair_coeff    1    2 thole   1.218   2.510
pair_coeff    1    3 thole   0.829   1.590
pair_coeff    1    6 thole   1.090   2.510
pair_coeff    1    7 thole   1.218   2.510
pair_coeff    1    8 thole   0.829   1.590
pair_coeff    2    2 thole   1.360   2.510
pair_coeff    2    3 thole   0.926   1.590
pair_coeff    2    6 thole   1.218   2.510
pair_coeff    2    7 thole   1.360   2.510
pair_coeff    2    8 thole   0.926   1.590
pair_coeff    3    3 thole   0.630   0.670
pair_coeff    3    6 thole   0.829   1.590
pair_coeff    3    7 thole   0.926   1.590
```

(continues on next page)

(continued from previous page)

```
pair_coeff    3    8 thole   0.630   0.670
pair_coeff    6    6 thole   1.090   2.510
pair_coeff    6    7 thole   1.218   2.510
pair_coeff    6    8 thole   0.829   1.590
pair_coeff    7    7 thole   1.360   2.510
pair_coeff    7    8 thole   0.926   1.590
pair_coeff    8    8 thole   0.630   0.670
```

For the *thole* pair style the coefficients are

1. the atom polarizability in units of cubic length
2. the screening factor of the Thole function (optional, default value specified by the pair_style command)
3. the cutoff (optional, default value defined by the pair_style command)

The special neighbors have charge-charge and charge-dipole interactions screened by the *coul* factors of the *special_bonds* command (0.0, 0.0, and 0.5 in the example above). Without using the pair_style *thole*, dipole-dipole interactions are screened by the same factor. By using the pair_style *thole*, dipole-dipole interactions are screened by Thole's function, whatever their special relationship (except within each DC-DP pair of course). Consider for example 1-2 neighbors: using the pair_style *thole*, their dipoles will see each other (despite the *coul* factor being 0.) and the interactions between these dipoles will be damped by Thole's function.

---

**Thermostats and barostats**

Using a Nose-Hoover barostat with the *langevin/drude* thermostat is straightforward using fix *nph* instead of *nve*. For example:

```
fix NPH all nph iso 1. 1. 500
```

It is also possible to use a Nose-Hoover instead of a Langevin thermostat. This requires to use **fix drude/transform** just before and after the time integration fixes. The *fix drude/transform/direct* converts the atomic masses, positions, velocities and forces into a reduced representation, where the DCs transform into the centers of mass of the DC-DP pairs and the DPs transform into their relative position with respect to their DC. The *fix drude/transform/inverse* performs the reverse transformation. For a NVT simulation, with the DCs and atoms at 300 K and the DPs at 1 K relative to their DC one would use

```
fix DIRECT all drude/transform/direct
fix NVT1 ATOMS nvt temp 300. 300. 100
fix NVT2 DRUDES nvt temp 1. 1. 20
fix INVERSE all drude/transform/inverse
```

For our phenol example, the groups would be defined as

```
group ATOMS  type 1 2 3 4 5 # DCs and non-polarizable atoms
group CORES  type 1 2 3     # DCs
group DRUDES type 6 7 8     # DPs
```

Note that with the fixes *drude/transform*, it is not required to specify *comm_modify vel yes* because the fixes do it anyway (several times and for the forces also).

It is a bit more tricky to run a NPT simulation with Nose-Hoover barostat and thermostat. First, the volume should be integrated only once. So the fix for DCs and atoms should be *npt* while the fix for DPs should be *nvt* (or vice versa). Second, the *fix npt* computes a global pressure and thus a global temperature whatever the fix group. We do want the pressure to correspond to the whole system, but we want the temperature to correspond to the fix group only. We must

then use the *fix_modify* command for this. In the end, the block of instructions for thermostatting and barostatting will look like

```
compute TATOMS ATOMS temp
fix DIRECT all drude/transform/direct
fix NPT ATOMS npt temp 300. 300. 100 iso 1. 1. 500
fix_modify NPT temp TATOMS press thermo_press
fix NVT DRUDES nvt temp 1. 1. 20
fix INVERSE all drude/transform/inverse
```

Another option for thermalizing the Drude model is to use the temperature-grouped Nose-Hoover (TGNH) thermostat proposed by *(Son)*. This is implemented as *fix tgnvt/drude* and *fix tgnpt/drude*. It separates the kinetic energy into three contributions: the molecular center of mass (COM) motion, the motion of atoms or atom-Drude pairs relative to molecular COMs, and the relative motion of atom-Drude pairs. An independent Nose-Hoover chain is applied to each type of motion. When TGNH is used, the temperatures of molecular, atomic and Drude motion can be printed out with *thermo_style command* command.

NVT simulation with TGNH thermostat

```
comm_modify vel yes
fix TGNVT all tgnvt/drude temp 300. 300. 100 1. 20
thermo_style custom f_TGNVT[1] f_TGNVT[2] f_TGNVT[3]
```

NPT simulation with TGNH thermostat

```
comm_modify vel yes
fix TGNPT all tgnpt/drude temp 300. 300. 100 1. 20 iso 1. 1. 500
thermo_style custom f_TGNPT[1] f_TGNPT[2] f_TGNPT[3]
```

---

**Rigid bodies**

You may want to simulate molecules as rigid bodies (but polarizable). Common cases are water models such as *SWM4-NDP*, which is a kind of polarizable TIP4P water. The rigid bodies and the DPs should be integrated separately, even with the Langevin thermostat. Let us review the different thermostats and ensemble combinations.

NVT ensemble using Langevin thermostat:

```
comm_modify vel yes
fix LANG all langevin/drude 300. 100 12435 1. 20 13977
fix RIGID ATOMS rigid/nve/small molecule
fix NVE DRUDES nve
```

NVT ensemble using Nose-Hoover thermostat:

```
fix DIRECT all drude/transform/direct
fix RIGID ATOMS rigid/nvt/small molecule temp 300. 300. 100
fix NVT DRUDES nvt temp 1. 1. 20
fix INVERSE all drude/transform/inverse
```

NPT ensemble with Langevin thermostat:

```
comm_modify vel yes
fix LANG all langevin/drude 300. 100 12435 1. 20 13977
fix RIGID ATOMS rigid/nph/small molecule iso 1. 1. 500
fix NVE DRUDES nve
```

NPT ensemble using Nose-Hoover thermostat:

```
compute TATOM ATOMS temp
fix DIRECT all drude/transform/direct
fix RIGID ATOMS rigid/npt/small molecule temp 300. 300. 100 iso 1. 1. 500
fix_modify RIGID temp TATOM press thermo_press
fix NVT DRUDES nvt temp 1. 1. 20
fix INVERSE all drude/transform/inverse
```

---

**(Lamoureux and Roux)** Lamoureux and Roux, J Chem Phys, 119, 3025-3039 (2003)

**(Schroeder)** Schroeder and Steinhauser, J Chem Phys, 133, 154511 (2010).

**(Thole)** Chem Phys, 59, 341 (1981).

**(Noskov)** Noskov, Lamoureux and Roux, J Phys Chem B, 109, 6705 (2005).

**(SWM4-NDP)** Lamoureux, Harder, Vorobyov, Roux, MacKerell, Chem Phys Let, 418, 245-249 (2006)

**(Son)** Son, McDaniel, Cui and Yethiraj, J Phys Chem Lett, 10, 7523 (2019).

## 10.5.10 Peridynamics with LAMMPS

This Howto is based on the Sandia report 2010-5549 by Michael L. Parks, Pablo Seleson, Steven J. Plimpton, Richard B. Lehoucq, and Stewart A. Silling.

### Overview

Peridynamics is a nonlocal extension of classical continuum mechanics. The discrete peridynamic model has the same computational structure as a molecular dynamics model. This Howto provides a brief overview of the peridynamic model of a continuum, then discusses how the peridynamic model is discretized within LAMMPS as described in the original article *(Parks)*. An example problem with comments is also included.

### Quick Start

The peridynamics styles are included in the optional *PERI package*. If your LAMMPS executable does not already include the PERI package, you can see the *build instructions for packages* for how to enable the package when compiling a custom version of LAMMPS from source.

Here is a minimal example for setting up a peridynamics simulation.

```
units         si
boundary      s s s
lattice       sc 0.0005
atom_style    peri
atom_modify   map array
neighbor      0.0010 bin
region        target cylinder y 0.0 0.0 0.0050 -0.0050 0.0 units box
create_box    1 target
create_atoms  1 region target

pair_style    peri/pmb
pair_coeff    * * 1.6863e22 0.0015001 0.0005 0.25
set           group all density 2200
set           group all volume 1.25e-10
velocity      all set 0.0 0.0 0.0 sum no units box
```

(continues on next page)

(continued from previous page)

```
fix           1 all nve
compute       1 all damage/atom
timestep      1.0e-7
```

Some notes on this input example:

- peridynamics simulations typically use SI *units*
- particles must be created on a *simple cubic lattice*
- using the *atom style peri* is required
- an *atom map* is required for indexing particles
- The *skin distance* used when computing neighbor lists should be defined appropriately for your choice of simulation parameters. The *skin* should be set to a value such that the peridynamic horizon plus the skin distance is larger than the maximum possible distance between two bonded particles (before their bond breaks). Here it is set to 0.001 meters.
- a *peridynamics pair style* is required. Available choices are currently: *peri/eps*, *peri/lps*, *peri/pmb*, and *peri/ves*. The model parameters are set with a *pair_coeff* command.
- the mass density and volume fraction for each particle must be defined. This is done with the two *set* commands for *density* and *volume*. For a simple cubic lattice, the volume of a particle should be equal to the cube of the lattice constant, here $V_i = \Delta x^3$.
- with the *velocity* command all particles are initially at rest
- a plain *velocity-Verlet time integrator* is used, which is algebraically equivalent to a centered difference in time, but numerically more stable
- you can compute the damage at the location of each particle with *compute damage/atom*
- finally, the timestep is set to 0.1 microseconds with the *timestep* command.

### Peridynamic Model of a Continuum

The following is not a complete overview of peridynamics, but a discussion of only those details specific to the model we have implemented within LAMMPS. For more on the peridynamic theory, the reader is referred to *(Silling 2007)*. To begin, we define the notation we will use.

### Basic Notation

Within the peridynamic literature, the following notational conventions are generally used. The position of a given point in the reference configuration is $\mathbf{x}$. Let $\mathbf{u}(\mathbf{x},t)$ and $\mathbf{y}(\mathbf{x},t)$ denote the displacement and position, respectively, of the point $\mathbf{x}$ at time $t$. Define the relative position and displacement vectors of two bonded points $\mathbf{x}$ and $\mathbf{x}'$ as $\xi = \mathbf{x}' - \mathbf{x}$ and $\eta = \mathbf{u}(\mathbf{x}',t) - \mathbf{u}(\mathbf{x},t)$, respectively. We note here that $\eta$ is time-dependent, and that $\xi$ is not. It follows that the relative position of the two bonded points in the current configuration can be written as $\xi + \eta = \mathbf{y}(\mathbf{x}',t) - \mathbf{y}(\mathbf{x},t)$.

Peridynamic models are frequently written using *states*, which we briefly describe here. For the purposes of our discussion, all states are operators that act on vectors in $\mathbb{R}^3$. For a more complete discussion of states, see *(Silling 2007)*. A *vector state* is an operator whose image is a vector, and may be viewed as a generalization of a second-rank tensor. Similarly, a *scalar state* is an operator whose image is a scalar. Of particular interest is the vector force state $\underline{\mathbf{T}}[\mathbf{x},t]\langle\mathbf{x}'-\mathbf{x}\rangle$, which is a mapping, having units of force per volume squared, of the vector $\mathbf{x}'-\mathbf{x}$ to the force vector state field. The vector state operator $\underline{\mathbf{T}}$ may itself be a function of $\mathbf{x}$ and $t$. The constitutive model is completely contained within $\underline{\mathbf{T}}$.

In the peridynamic theory, the deformation at a point depends collectively on all points interacting with that point. Using the notation of *(Silling 2007)*, we write the peridynamic equation of motion as

$$\rho(\mathbf{x})\ddot{\mathbf{u}}(\mathbf{x},t) = \int_{\mathcal{H}_\mathbf{x}} \left\{ \underline{\mathbf{T}}\left[\mathbf{x},t\right]\left\langle \mathbf{x}'-\mathbf{x}\right\rangle - \underline{\mathbf{T}}\left[\mathbf{x}',t\right]\left\langle \mathbf{x}-\mathbf{x}'\right\rangle \right\} dV_{\mathbf{x}'} + \mathbf{b}(\mathbf{x},t), \tag{1}$$

where $\rho$ represents the mass density, $\underline{\mathbf{T}}$ the force vector state, and $\mathbf{b}$ an external body force density. A point $\mathbf{x}$ interacts with all the points $\mathbf{x}'$ within the neighborhood $\mathcal{H}_\mathbf{x}$, assumed to be a spherical region of radius $\delta > 0$ centered at $\mathbf{x}$. $\delta$ is called the *horizon*, and is analogous to the cutoff radius used in molecular dynamics. Conditions on $\underline{\mathbf{T}}$ for which *(1)* satisfies the balance of linear and angular momentum are given in *(Silling 2007)*.

We consider only force vector states that can be written as

$$\underline{\mathbf{T}} = \underline{t}\,\underline{\mathbf{M}},$$

with $\underline{t}$ a *scalar force state* and $\underline{\mathbf{M}}$ the *deformed direction vector state*, defined by

$$\underline{\mathbf{M}}\left\langle \xi\right\rangle = \begin{cases} \frac{\xi+\eta}{\|\xi+\eta\|} & \|\xi+\eta\| \neq 0 \\ 0 & \text{otherwise} \end{cases}. \tag{2}$$

Such force states correspond to so-called *ordinary* materials *(Silling 2007)*. These are the materials for which the force between any two interacting points $\mathbf{x}$ and $\mathbf{x}'$ acts along the line between the points.

### Linear Peridynamic Solid (LPS) Model

We summarize the linear peridynamic solid (LPS) material model. For more on this model, the reader is referred to *(Silling 2007)*. This model is a nonlocal analogue to a classical linear elastic isotropic material. The elastic properties of a classical linear elastic isotropic material are determined by (for example) the bulk and shear moduli. For the LPS model, the elastic properties are analogously determined by the bulk and shear moduli, along with the horizon $\delta$.

The LPS model has a force scalar state

$$\underline{t} = \frac{3K\theta}{m}\underline{\omega}\,\underline{x} + \alpha\underline{\omega}\,\underline{e}^{\mathrm{d}}, \tag{3}$$

with $K$ the bulk modulus and $\alpha$ related to the shear modulus $G$ as

$$\alpha = \frac{15G}{m}.$$

The remaining components of the model are described as follows. Define the reference position scalar state $\underline{x}$ so that $\underline{x}\left\langle \xi\right\rangle = \|\xi\|$. Then, the weighted volume $m$ is defined as

$$m\left[\mathbf{x}\right] = \int_{\mathcal{H}_\mathbf{x}} \underline{\omega}\left\langle \xi\right\rangle \underline{x}\left\langle \xi\right\rangle \underline{x}\left\langle \xi\right\rangle dV_\xi. \tag{4}$$

Let

$$\underline{e}\left[\mathbf{x},t\right]\left\langle \xi\right\rangle = \|\xi+\eta\| - \|\xi\|$$

be the extension scalar state, and

$$\theta\left[\mathbf{x},t\right] = \frac{3}{m\left[\mathbf{x}\right]}\int_{\mathcal{H}_\mathbf{x}} \underline{\omega}\left\langle \xi\right\rangle \underline{x}\left\langle \xi\right\rangle \underline{e}\left[\mathbf{x},t\right]\left\langle \xi\right\rangle dV_\xi$$

be the dilatation. The isotropic and deviatoric parts of the extension scalar state are defined, respectively, as

$$\underline{e}^{\mathrm{i}} = \frac{\theta\underline{x}}{3}, \qquad \underline{e}^{\mathrm{d}} = \underline{e} - \underline{e}^{\mathrm{i}},$$

where the arguments of the state functions and the vectors on which they operate are omitted for simplicity. We note that the LPS model is linear in the dilatation $\theta$, and in the deviatoric part of the extension $\underline{e}^{\mathrm{d}}$.

**Note**

The weighted volume $m$ is time-independent, and does not change as bonds break. It is computed with respect to the bond family defined at the reference (initial) configuration.

The non-negative scalar state $\underline{\omega}$ is an *influence function (Silling 2007)*. For more on influence functions, see *(Seleson 2010)*. If an influence function $\underline{\omega}$ depends only upon the scalar $\|\xi\|$, (i.e., $\underline{\omega}\langle\xi\rangle = \underline{\omega}\langle\|\xi\|\rangle$), then $\underline{\omega}$ is a spherical influence function. For a spherical influence function, the LPS model is isotropic *(Silling 2007)*.

**Note**

In the LAMMPS implementation of the LPS model, the influence function $\underline{\omega}\langle\|\xi\|\rangle = 1/\|\xi\|$ is used. However, the user can define their own influence function by altering the method "influence_function" in the file `pair_peri_lps.cpp`. The LAMMPS peridynamics code permits both spherical and non-spherical influence functions (e.g., isotropic and non-isotropic materials).

### Prototype Microelastic Brittle (PMB) Model

We summarize the prototype microelastic brittle (PMB) material model. For more on this model, the reader is referred to *(Silling 2000)* and *(Silling 2005)*. This model is a special case of the LPS model; see *(Seleson 2010)* for the derivation. The elastic properties of the PMB model are determined by the bulk modulus $K$ and the horizon $\delta$.

The PMB model is expressed using the scalar force state field

$$\underline{t}[\mathbf{x},t]\langle\xi\rangle = \frac{1}{2} f\left(\eta,\xi\right), \qquad (5)$$

with $f$ a scalar-valued function. We assume that $f$ takes the form

$$f = cs,$$

where

$$c = \frac{18K}{\pi\delta^4}, \qquad (6)$$

with $K$ the bulk modulus and $\delta$ the horizon, and $s$ the bond stretch, defined as

$$s(t,\eta,\xi) = \frac{\|\eta+\xi\| - \|\xi\|}{\|\xi\|}.$$

Bond stretch is a unitless quantity, and identical to a one-dimensional definition of strain. As such, we see that a bond at its equilibrium length has stretch $s = 0$, and a bond at twice its equilibrium length has stretch $s = 1$. The constant $c$ given above is appropriate for 3D models only. For more on the origins of the constant $c$, see *(Silling 2005)*. For the derivation of $c$ for 1D and 2D models, see *(Emmrich)*.

Given *(5)*, *(1)* reduces to

$$\rho(\mathbf{x})\ddot{\mathbf{u}}(\mathbf{x},t) = \int_{\mathcal{H}_\mathbf{x}} \mathbf{f}\left(\eta,\xi\right) dV_\xi + \mathbf{b}(\mathbf{x},t), \qquad (7)$$

with

$$\mathbf{f}\left(\eta,\xi\right) = f\left(\eta,\xi\right)\frac{\xi+\eta}{\|\xi+\eta\|}.$$

Unlike the LPS model, the PMB model has a Poisson ratio of $\nu = 1/4$ in 3D, and $\nu = 1/3$ in 2D. This is reflected in the input for the PMB model, which requires only the bulk modulus of the material, whereas the LPS model requires both the bulk and shear moduli.

### Damage

Bonds are made to break when they are stretched beyond a given limit. Once a bond fails, it is failed forever *(Silling)*. Further, new bonds are never created during the course of a simulation. We discuss only one criterion for bond breaking, called the *critical stretch* criterion.

Define $\mu$ to be the history-dependent scalar boolean function

$$\mu(t,\eta,\xi) = \left\{ \begin{array}{ll} 1 & \text{if } s(t',\eta,\xi) < \min\left(s_0(t',\eta,\xi), s_0(t',\eta',\xi')\right) \text{ for all } 0 \le t' \le t \\ 0 & \text{otherwise} \end{array} \right\}. \tag{8}$$

where $\eta' = \mathbf{u}(\mathbf{x}'',t) - \mathbf{u}(\mathbf{x}',t)$ and $\xi' = \mathbf{x}'' - \mathbf{x}'$. Here, $s_0(t,\eta,\xi)$ is a critical stretch defined as

$$s_0(t,\eta,\xi) = s_{00} - \alpha s_{\min}(t,\eta,\xi), \qquad s_{\min}(t) = \min_{\xi} s(t,\eta,\xi), \tag{9}$$

where $s_{00}$ and $\alpha$ are material-dependent constants. The history function $\mu$ breaks bonds when the stretch $s$ exceeds the critical stretch $s_0$.

Although $s_0(t,\eta,\xi)$ is expressed as a property of a particle, bond breaking must be a symmetric operation for all particle pairs sharing a bond. That is, particles $\mathbf{x}$ and $\mathbf{x}'$ must utilize the same test when deciding to break their common bond. This can be done by any method that treats the particles symmetrically. In the definition of $\mu$ above, we have chosen to take the minimum of the two $s_0$ values for particles $\mathbf{x}$ and $\mathbf{x}'$ when determining if the $\mathbf{x}$–$\mathbf{x}'$ bond should be broken.

Following *(Silling)*, we can define the damage at a point $\mathbf{x}$ as

$$\varphi(\mathbf{x},t) = 1 - \frac{\int_{\mathcal{H}_\mathbf{x}} \mu(t,\eta,\xi) dV_{\mathbf{x}'}}{\int_{\mathcal{H}_\mathbf{x}} dV_{\mathbf{x}'}}. \tag{10}$$

### Discrete Peridynamic Model and LAMMPS Implementation

In LAMMPS, instead of *(1)*, we model this equation of motion:

$$\rho(\mathbf{x})\ddot{\mathbf{y}}(\mathbf{x},t) = \int_{\mathcal{H}_\mathbf{x}} \left\{ \underline{\mathbf{T}}\left[\mathbf{x},t\right] \langle \mathbf{x}' - \mathbf{x} \rangle - \underline{\mathbf{T}}\left[\mathbf{x}',t\right] \langle \mathbf{x} - \mathbf{x}' \rangle \right\} dV_{\mathbf{x}'} + \mathbf{b}(\mathbf{x},t),$$

where we explicitly track and store at each timestep the positions and not the displacements of the particles. We observe that $\ddot{\mathbf{y}}(\mathbf{x},t) = \ddot{\mathbf{x}} + \ddot{\mathbf{u}}(\mathbf{x},t) = \ddot{\mathbf{u}}(\mathbf{x},t)$, so that this is equivalent to *(1)*.

### Spatial Discretization

The region defining a peridynamic material is discretized into particles forming a simple cubic lattice with lattice constant $\Delta x$, where each particle $i$ is associated with some volume fraction $V_i$. For any particle $i$, let $\mathcal{F}_i$ denote the family of particles for which particle $i$ shares a bond in the reference configuration. That is,

$$\mathcal{F}_i = \{p \mid \|\mathbf{x}_p - \mathbf{x}_i\| \le \delta\}. \tag{11}$$

The discretized equation of motion replaces *(1)* with

$$\rho \ddot{\mathbf{y}}_i^n = \sum_{p \in \mathcal{F}_i} \left\{ \underline{\mathbf{T}}\left[\mathbf{x}_i, t\right] \left\langle \mathbf{x}_p' - \mathbf{x}_i \right\rangle - \underline{\mathbf{T}}\left[\mathbf{x}_p, t\right] \left\langle \mathbf{x}_i - \mathbf{x}_p \right\rangle \right\} V_p + \mathbf{b}_i^n, \tag{12}$$

where $n$ is the timestep number and subscripts denote the particle number.

### Short-Range Forces

In the model discussed so far, particles interact only through their bond forces. A particle with no bonds becomes a free non-interacting particle. To account for contact forces, short-range forces are introduced *(Silling 2007)*. We add to the force in *(12)* the following force

$$\mathbf{f}_S(\mathbf{y}_p, \mathbf{y}_i) = \min\left\{0, \frac{c_S}{\delta}(\left\|\mathbf{y}_p - \mathbf{y}_i\right\| - d_{pi})\right\} \frac{\mathbf{y}_p - \mathbf{y}_i}{\left\|\mathbf{y}_p - \mathbf{y}_i\right\|}, \tag{13}$$

where $d_{pi}$ is the short-range interaction distance between particles $p$ and $i$, and $c_S$ is a multiple of the constant $c$ from *(6)*. Note that the short-range force is always repulsive, never attractive. In practice, we choose

$$c_S = 15\frac{18K}{\pi\delta^4}. \tag{14}$$

For the short-range interaction distance, we choose *(Silling 2007)*

$$d_{pi} = \min\left\{0.9\left\|\mathbf{x}_p - \mathbf{x}_i\right\|, 1.35(r_p + r_i)\right\}, \tag{15}$$

where $r_i$ is called the *node radius* of particle $i$. Given a discrete lattice, we choose $r_i$ to be half the lattice constant.

**Note**

For a simple cubic lattice, $\Delta x = \Delta y = \Delta z$.

Given this definition of $d_{pi}$, contact forces appear only when particles are under compression.

When accounting for short-range forces, it is convenient to define the short-range family of particles

$$\mathcal{F}_i^S = \{p \mid \left\|\mathbf{y}_p - \mathbf{y}_i\right\| \leq d_{pi}\}.$$

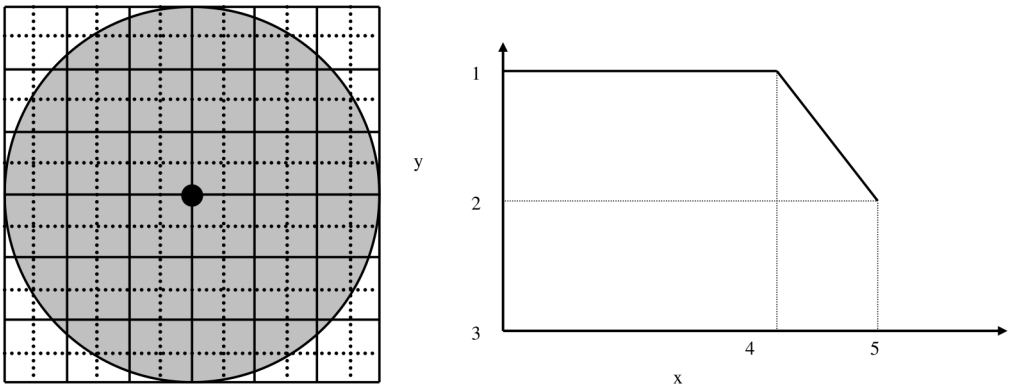
### Modification to the Particle Volume

The right-hand side of *(12)* may be thought of as a midpoint quadrature of *(1)*. To slightly improve the accuracy of this quadrature, we discuss a modification to the particle volume used in *(12)*. In a situation where two particles share a bond with $\left\|\mathbf{x}_p - \mathbf{x}_i\right\| = \delta$, for example, we suppose that only approximately half the volume of each particle is "seen" by the other *(Silling 2007)*. When computing the force of each particle on the other we use $V_p/2$ rather than $V_p$ in *(12)*. As such, we introduce a nodal volume scaling function for all bonded particles where $\delta - r_i \leq \left\|\mathbf{x}_p - \mathbf{x}_i\right\| \leq \delta$ (see the Figure below).

We choose to use a linear unitless nodal volume scaling function

$$\nu(\mathbf{x}_p - \mathbf{x}_i) = \left\{ \begin{array}{cc} -\frac{1}{2r_i}\left\|\mathbf{x}_p - \mathbf{x}_i\right\| + \left(\frac{\delta}{2r_i} + \frac{1}{2}\right) & \text{if } \delta - r_i \leq \left\|\mathbf{x}_p - \mathbf{x}_i\right\| \leq \delta \\ 1 & \text{if } \left\|\mathbf{x}_p - \mathbf{x}_i\right\| \leq \delta - r_i \\ 0 & \text{otherwise} \end{array} \right\}$$

If $\left\|\mathbf{x}_p - \mathbf{x}_i\right\| = \delta$, $\nu = 0.5$, and if $\left\|\mathbf{x}_p - \mathbf{x}_i\right\| = \delta - r_i$, $\nu = 1.0$, for example.




(a) Two-dimensional diagram showing particle on mesh (solid lines) with neighborhood $\mathcal{H}_{\mathbf{x}}$ as grey circular region. Dual mesh (dotted lines) shows boundaries of each particle.

(b) Plot of $\nu(\mathbf{x}_p - \mathbf{x}_i)$ vs. $\|\mathbf{x}_p - \mathbf{x}_i\|$.

Fig. 3: Diagram showing horizon of a particular particle, demonstrating that the volume associated with particles near the boundary of the horizon is not completely contained within the horizon.

### Temporal Discretization

When discretizing time in LAMMPS, we use a velocity-Verlet scheme, where both the position and velocity of the particle are stored explicitly. The velocity-Verlet scheme is generally expressed in three steps. In *Algorithm 1*, $\rho_i$ denotes the mass density of a particle and $\widetilde{\mathbf{f}}_i^n$ denotes the net force density on particle $i$ at timestep $n$. The LAMMPS command *fix nve* performs a velocity-Verlet integration.

**Algorithm 1: Velocity Verlet**

1: $\mathbf{v}_i^{n+1/2} = \mathbf{v}_i^n + \frac{\Delta t}{2\rho_i}\widetilde{\mathbf{f}}_i^n$

2: $\mathbf{y}_i^{n+1} = \mathbf{y}_i^n + \Delta t \mathbf{v}_i^{n+1/2}$

3: $\mathbf{v}_i^{n+1} = \mathbf{v}_i^{n+1/2} + \frac{\Delta t}{2\rho_i}\widetilde{\mathbf{f}}_i^{n+1}$

### Breaking Bonds

During the course of simulation, it may be necessary to break bonds, as described in the *Damage section*. Bonds are recorded as broken in a simulation by removing them from the bond family $\mathcal{F}_i$ (see *(11)*).

A naive implementation would have us first loop over all bonds and compute $s_{min}$ in *(9)*, then loop over all bonds again and break bonds with a stretch $s > s_0$ as in *(8)*, and finally loop over all particles and compute forces for the next step of *Algorithm 1*. For reasons of computational efficiency, we instead store the per-particle minimum stretch $s_{min}$ from the *previous* timestep and, when deciding whether to break a bond, form its critical stretch $s_0 = s_{00} - \alpha\, s_{min}$ from that bond's own $s_{00}$ and $\alpha$. This keeps the criterion correct when $s_{00}$ and $\alpha$ differ between atom-type pairs (for example, a deliberately weakened interface).

Changed in version 4Jul2026.

Earlier versions stored a single per-particle critical stretch $s_0$ (the maximum of $s_{00} - \alpha s$ over a particle's bonds), which reproduces *(9)* only when $s_{00}$ and $\alpha$ are identical for all atom-type pairs.

**Note**

For the first timestep, $s_0$ is initialized to $\infty$ for all nodes. This means that no bonds may be broken until the second timestep. As such, it is recommended that the first few timesteps of the peridynamic simulation not involve any actions that might result in the breaking of bonds. As a practical example, the projectile in the *commented example below* is placed such that it does not impact the target brittle plate until several timesteps into the simulation.

### LPS Pseudocode

A sketch of the LPS model implementation in the PERI package appears in *Algorithm 2*. This algorithm makes use of the routines in *Algorithm 3* and *Algorithm 4*.

**Algorithm 2: LPS Peridynamic Model Pseudocode**

Fix $s_{00}$, $\alpha$, horizon $\delta$, bulk modulus $K$, shear modulus $G$, timestep $\Delta t$, and generate initial lattice of particles with lattice constant $\Delta x$. Let there be $N$ particles. Define constant $c_S$ for repulsive short-range forces.
Initialize bonds between all particles $i \neq j$ where $\|\mathbf{x}_j - \mathbf{x}_i\| \leq \delta$
Initialize weighted volume $m$ for all particles using *Algorithm 3*
Initialize $s_0 = \infty$ {*Initialize each entry to MAX_DOUBLE*}
**while** not done **do**
  Perform step 1 of *Algorithm 1*, updating velocities of all particles
  Perform step 2 of *Algorithm 1*, updating positions of all particles
  $\tilde{s}_0 = \infty$ {*Initialize each entry to MAX_DOUBLE*}
  **for** $i$ = 1 to $N$ **do**
    {Compute short-range forces}
    **for all** particles $j \in \mathcal{F}_i^S$ (the short-range family of nodes for particle $i$) **do**
      $r = \left\|\mathbf{y}_j - \mathbf{y}_i\right\|$
      $dr = \min\{0, r - d\}$ {*Short-range forces are only repulsive, never attractive*}
      $k = \frac{c_S}{\delta} V_k dr$ {$c_S$ *defined in :ref: `(14) <pericS> `*}
      $\mathbf{f} = \mathbf{f} + k \frac{\mathbf{y}_j - \mathbf{y}_i}{\left\|\mathbf{y}_j - \mathbf{y}_i\right\|}$
    **end for**
  **end for**
  Compute the dilatation for each particle using *Algorithm 4*
  **for** $i$ = 1 to $N$ **do**
    {Compute bond forces}
    **for all** particles $j$ sharing an unbroken bond with particle $i$ **do**
      $e = \left\|\mathbf{y}_j - \mathbf{y}_i\right\| - \|\mathbf{x}_j - \mathbf{x}_i\|$
      $\omega_+ = \underline{\omega}\left\langle \mathbf{x}_j - \mathbf{x}_i \right\rangle$ {*Influence function evaluation*}
      $\omega_- = \underline{\omega}\left\langle \mathbf{x}_i - \mathbf{x}_j \right\rangle$ {*Influence function evaluation*}
      $\hat{f} = \left[(3K - 5G)\left(\frac{\theta(i)}{m(i)}\omega_+ + \frac{\theta(j)}{m(j)}\omega_-\right)\|\mathbf{x}_j - \mathbf{x}_i\| + 15G\left(\frac{\omega_+}{m(i)} + \frac{\omega_-}{m(j)}\right)e\right] v(\mathbf{x}_j - \mathbf{x}_i)V_j$
      $\mathbf{f} = \mathbf{f} + \hat{f}\frac{\mathbf{y}_j - \mathbf{y}_i}{\left\|\mathbf{y}_j - \mathbf{y}_i\right\|}$
      **if** $(dr / \|\mathbf{x}_j - \mathbf{x}_i\|) > \min(s_0(i), s_0(j))$ **then**
        Break $i$'s bond with $j$ {$j$ *'s bond with i will be broken when this loop iterates on* $j$}

**end if**
$\tilde{s}_0(i) = \min(\tilde{s}_0(i), s_{00} - \alpha(dr/\|\mathbf{x}_j - \mathbf{x}_i\|))$
**end for**
**end for**
$s_0 = \tilde{s}_0$ {*Store for use in next timestep*}
Perform step 3 of *Algorithm 1*, updating velocities of all particles
**end while**

**Algorithm 3: Computation of Weighted Volume *m***

**for** $i = 1$ to $N$ **do**
$m(i) = 0.0$
**for all** particles $j$ sharing a bond with particle $i$ **do**
$m(i) = m(i) + \underline{\omega}\langle \mathbf{x}_j - \mathbf{x}_i \rangle \|\mathbf{x}_j - \mathbf{x}_i\|^2 \nu(\mathbf{x}_j - \mathbf{x}_i)V_j$
**end for**
**end for**

**Algorithm 4: Computation of Dilatation $\theta$**

**for** $i = 1$ to $N$ **do**
$\theta(i) = 0.0$
**for all** particles $j$ sharing an unbroken bond with particle $i$ **do**
$e = \|\mathbf{y}_i - \mathbf{y}_j\| - \|\mathbf{x}_i - \mathbf{x}_j\|$
$\theta(i) = \theta(i) + \underline{\omega}\langle \mathbf{x}_j - \mathbf{x}_i \rangle \|\mathbf{x}_j - \mathbf{x}_i\| e\nu(\mathbf{x}_j - \mathbf{x}_i)V_j$
**end for**
$\theta(i) = \frac{3}{m(i)}\theta(i)$
**end for**

### PMB Pseudocode

A sketch of the PMB model implementation in the PERI package appears in *Algorithm 5*.

**Algorithm 5: PMB Peridynamic Model Pseudocode**

Fix $s_{00}$, $\alpha$, horizon $\delta$, spring constant $c$, timestep $\Delta t$, and generate initial lattice of particles with lattice constant $\Delta x$. Let there be $N$ particles.
Initialize bonds between all particles $i \neq j$ where $\|\mathbf{x}_j - \mathbf{x}_i\| \leq \delta$
Initialize $s_0 = \infty$ {*Initialize each entry to MAX_DOUBLE*}
**while** not done **do**
Perform step 1 of *Algorithm 1*, updating velocities of all particles
Perform step 2 of *Algorithm 1*, updating positions of all particles

$\tilde{s}_0 = \infty$ {*Initialize each entry to MAX_DOUBLE*}
**for** $i$ = 1 to $N$ **do**
{Compute short-range forces}
**for all** particles $j \in \mathcal{F}_i^S$ (the short-range family of nodes for particle $i$) **do**
$r = \left\|\mathbf{y}_j - \mathbf{y}_i\right\|$
$dr = \min\{0, r - d\}$ {*Short-range forces are only repulsive, never attractive*}
$k = \frac{c_S}{\delta} V_k dr$ {*$c_S$ defined in :ref:`(14) <pericS>`*}
$\mathbf{f} = \mathbf{f} + k\frac{\mathbf{y}_j - \mathbf{y}_i}{\left\|\mathbf{y}_j - \mathbf{y}_i\right\|}$
**end for**
**end for**
**for** $i$ = 1 to $N$ **do**
{Compute bond forces}
**for all** particles $j$ sharing an unbroken bond with particle $i$ **do**
$r = \left\|\mathbf{y}_j - \mathbf{y}_i\right\|$
$dr = r - \left\|\mathbf{x}_j - \mathbf{x}_i\right\|$
$k = \frac{c}{\left\|\mathbf{x}_i - \mathbf{x}_j\right\|} \nu(\mathbf{x}_i - \mathbf{x}_j) V_j dr$ {*$c$ defined in :ref:`(6) <peric>`*}
$\mathbf{f} = \mathbf{f} + k\frac{\mathbf{y}_j - \mathbf{y}_i}{\left\|\mathbf{y}_j - \mathbf{y}_i\right\|}$
**if** $(dr / \left\|\mathbf{x}_j - \mathbf{x}_i\right\|) > \min(s_0(i), s_0(j))$ **then**
Break $i$'s bond with $j$ {*$j$'s bond with $i$ will be broken when this loop iterates on $j$*}
**end if**
$\tilde{s}_0(i) = \min(\tilde{s}_0(i), s_{00} - \alpha(dr / \left\|\mathbf{x}_j - \mathbf{x}_i\right\|))$
**end for**
**end for**
$s_0 = \tilde{s}_0$ {*Store for use in next timestep*}
Perform step 3 of *Algorithm 1*, updating velocities of all particles
**end while**

### Damage

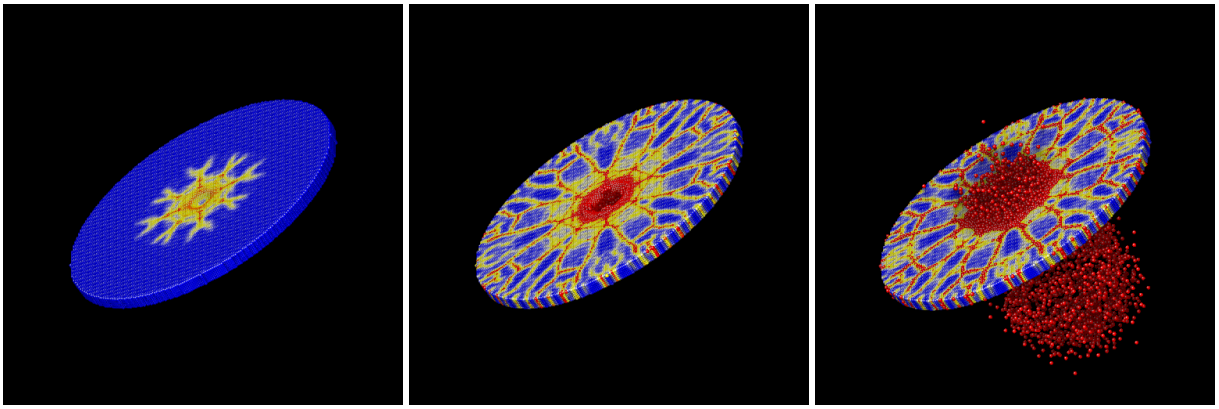
The damage associated with every particle (see *(10)*) can optionally be computed and output with a LAMMPS data dump. To do this, your input script must contain the command *compute damage/atom* This enables a LAMMPS per-atom compute to calculate the damage associated with each particle every time a LAMMPS *data dump* frame is written.

### Visualizing Simulation Results

There are multiple ways to visualize the simulation results. Typically, you want to display the particles and color code them by the value computed with the *compute damage/atom* command.

This can be done, for example, by using the built-in visualizer of the *dump image or dump movie* command to create snapshot images or a movie. Below are example command for using dump image with the *example listed below* and a set of images created for steps 300, 600, and 2000 this way.

```
dump            D2 all image 100 dump.peri.*.png c_C1 type box no 0.0 view 30 60 zoom 1.
↪5 up 0 0 -1  ssao yes 4539 0.6
dump_modify     D2 pad 5 adiam * 0.001 amap 0.0 1.0 ca 0.1 3 min blue 0.5 yellow max red
```



For interactive visualization, the Ovito is very convenient to use. Below are steps to create a visualization of the *same example from below* now using the generated trajectory in the `dump.peri` file.

- Launch Ovito
- File -> Load File -> `dump.peri`
- Select "-> Particle types" and under "Appearance" set "Display radius:" to 0.0005
- From the "Add modification:" drop down list select "Color coding"
- Under "Color coding" select from the "Color gradient" drop down list "Jet"
- Also under "Color coding" set "Start value:" to 0 and "End value:" to 1
- You can improve the image quality by adding the "Ambient occlusion" modification

## Pitfalls

### Parallel Scalability

LAMMPS operates in parallel in a *spatial-decomposition mode*, where each processor owns a spatial subdomain of the overall simulation domain and communicates with its neighboring processors via distributed-memory message passing (MPI) to acquire ghost atom information to allow forces on the atoms it owns to be computed. LAMMPS also uses Verlet neighbor lists which are recomputed every few timesteps as particles move. On these timesteps, particles also migrate to new processors as needed. LAMMPS decomposes the overall simulation domain so that spatial subdomains of nearly equal volume are assigned to each processor. When each subdomain contains nearly the same number of particles, this results in a reasonable load balance among all processors. As is more typical with some peridynamic simulations, some subdomains may contain many particles while other subdomains contain few particles, resulting in a load imbalance that impacts parallel scalability.

### Setting the "skin" distance

The *neighbor* command with LAMMPS is used to set the so-called "skin" distance used when building neighbor lists. All atom pairs within a cutoff distance equal to the horizon $\delta$ plus the skin distance are stored in the list. Unexpected crashes in LAMMPS may be due to too small a skin distance. The skin should be set to a value such that $\delta$ plus the skin distance is larger than the maximum possible distance between two bonded particles. For example, if $s_{00}$ is increased, the skin distance may also need to be increased.

### "Lost" particles

All particles are contained within the "simulation box" of LAMMPS. The boundaries of this box may change with time, or not, depending on how the LAMMPS *boundary* command has been set. If a particle drifts outside the simulation box during the course of a simulation, it is called *lost*.

As an option of the *thermo_modify* command of LAMMPS, the lost keyword determines whether LAMMPS checks for lost atoms each time it computes thermodynamics and what it does if atoms are lost. If the value is *ignore*, LAMMPS

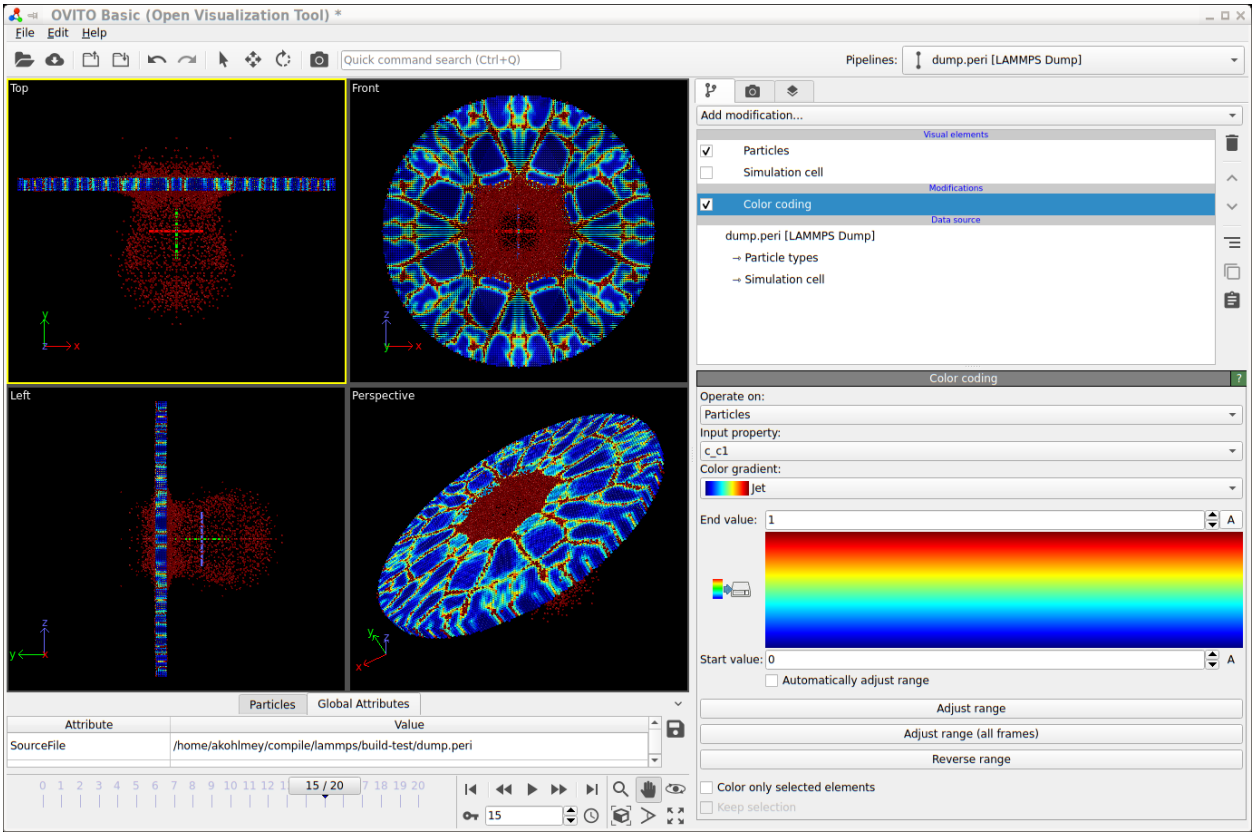



Fig. 4: Screenshot of visualizing a trajectory with Ovito

does not check for lost atoms. If the value is *error* or *warn*, LAMMPS checks and either issues an error or warning. The code will exit with an error and continue with a warning. This can be a useful debugging option. The default behavior of LAMMPS is to exit with an error if a particle is lost.

The peridynamic module within LAMMPS does not check for lost atoms. If a particle with unbroken bonds is lost, those bonds are marked as broken by the remaining particles.

**Defining the peridynamic horizon $\delta$**

In the *pair_coeff* command, the user must specify the horizon $\delta$. This argument determines which particles are bonded when the simulation is initialized. It is recommended that $\delta$ be set to a small fraction of a lattice constant larger than desired.

For example, if the lattice constant is 0.0005 and you wish to set the horizon to three times the lattice constant, then set $\delta$ to be 0.0015001, a value slightly larger than three times the lattice constant. This guarantees that particles three lattice constants away from each other are still bonded. If $\delta$ is set to 0.0015, for example, floating point error may result in some pairs of particles three lattice constants apart not being bonded.

**Breaking bonds too early**

For technical reasons, the bonds in the simulation are not created until the end of the first timestep of the simulation. Therefore, one should not attempt to break bonds until at least the second step of the simulation.

## Bugs

The user is cautioned that this code is a beta release. If you are confident that you have found a bug in the peridynamic module, please report it in a *GitHub Issue <https://github.com/lammps/lammps/issues>* or send an email to the LAMMPS developers. First, check the New features and bug fixes section of the LAMMPS website site to see if the bug has already been reported or fixed. If not, the most useful thing you can do for us is to isolate the problem. Run it on the smallest number of atoms and fewest number of processors and with the simplest input script that reproduces the bug. In your message, describe the problem and any ideas you have as to what is causing it or where in the code the problem might be. We'll request your input script and data files if necessary.

## Modifying and Extending the Peridynamic Module

To add new features or peridynamic potentials to the peridynamic module, the user is referred to the *Modifying & extending LAMMPS* section. To develop a new bond-based material, start with the *peri/pmb* pair style as a template. To develop a new state-based material, start with the *peri/lps* pair style as a template.

## A Numerical Example

To introduce the peridynamic implementation within LAMMPS, we replicate a numerical experiment taken from section 6 of *(Silling 2005)*.

## Problem Description and Setup

We consider the impact of a rigid sphere on a homogeneous disk of brittle material. The sphere has diameter 0.01 m and velocity 100 m/s directed normal to the surface of the target. The target material has density $\rho$ = 2200 kg/m:math:^3. A PMB material model is used with $K$ = 14.9 GPa and critical bond stretch parameters given by $s_{00}$ = 0.0005 and $\alpha$ = 0.25. A three-dimensional simple cubic lattice is constructed with lattice constant 0.0005 m and horizon 0.0015 m. (The horizon is three times the lattice constant.) The target is a cylinder of diameter 0.074 m and thickness 0.0025 m, and the associated lattice contains 103,110 particles. Each particle $i$ has volume fraction $V_i = 1.25 \times 10^{-10}\text{m}^3$.

The spring constant in the PMB material model is (see *(6)*)

$$c = \frac{18k}{\pi\delta^4} = \frac{18(14.9 \times 10^9)}{\pi(1.5 \times 10^{-3})^4} \approx 1.6863 \times 10^{22}.$$

The CFL analysis from *(Silling2005)* shows that a timestep of $1.0 \times 10^{-7}$ is safe.

We observe here that in IEEE double-precision floating point arithmetic when computing the bond stretch $s(t, \eta, \xi)$ at each iteration where $\|\eta + \xi\|$ is computed during the iteration and $\|\xi\|$ was computed and stored for the initial lattice, it may be that $fl(s) = \varepsilon$ with $|\varepsilon| \leq \varepsilon_{machine}$ for an unstretched bond. Taking $\varepsilon = 2.220446049250313 \times 10^{-16}$, we see that the value $csV_i \approx 4.68 \times 10^{-4}$, computed when determining $f$, is perhaps larger than we would like, especially when the true force should be zero. One simple way to avoid this issue is to insert the following instructions in Algorithm *Algorithm 5* after instruction 21 (and similarly for Algorithm *Algorithm 2*):

**if** $|dr| < \varepsilon_{machine}$ **then**
    $dr = 0$
**end if**

Qualitatively, this says that displacements from equilibrium on the order of $10^{-16}$m are taken to be exactly zero, a seemingly reasonable assumption.

### The Projectile

The projectile used in the following experiments is not the one used in *(Silling 2005)*. The projectile used here exerts a force

$$F(r) = -k_s(r - R)^2$$

on each atom where $k_s$ is a specified force constant, $r$ is the distance from the atom to the center of the indenter, and $R$ is the radius of the projectile. The force is repulsive and $F(r) = 0$ for $r > R$. For our problem, the projectile radius $R = 0.05$ m, and we have chosen $k_s = 1.0 \times 10^{17}$ (compare with *(6)* above).

### Writing the LAMMPS Input File

We discuss the example input script *listed below*. In line 2 we specify that SI units are to be used. We specify the dimension (3) and boundary conditions ("shrink-wrapped") for the computational domain in lines 3 and 4. In line 5 we specify that peridynamic particles are to be used for this simulation. In line 7, we set the "skin" distance used in building the LAMMPS neighbor list. In line 8 we set the lattice constant (in meters) and in line 10 we define the spatial region where the target will be placed. In line 12 we specify a rectangular box enclosing the target region that defines the simulation domain. Line 14 fills the target region with atoms. Lines 15 and 17 define the peridynamic material model, and lines 19 and 21 set the particle density and particle volume, respectively. The particle volume should be set to the cube of the lattice constant for a simple cubic lattice. Line 23 sets the initial velocity of all particles to zero. Line 25 instructs LAMMPS to integrate time with velocity-Verlet, and lines 27-30 create the spherical projectile, sending it with a velocity of 100 m/s towards the target. Line 32 declares a compute style for the damage (percentage of broken bonds) associated with each particle. Line 33 sets the timestep, line 34 instructs LAMMPS to provide a screen dump of thermodynamic quantities every 200 timesteps, and line 35 instructs LAMMPS to create a data file (`dump.peri`) with a complete snapshot of the system every 100 timesteps. This file can be used to create still images or movies. Finally, line 36 instructs LAMMPS to run for 2000 timesteps.

Listing 5: Peridynamics Example LAMMPS Input Script

```
1  # 3D Peridynamic simulation with projectile"
2  units           si
3  dimension       3
4  boundary        s s s
5  atom_style      peri
6  atom_modify     map array
7  neighbor        0.0010 bin
8  lattice         sc 0.0005
9  # Create desired target
10 region          target cylinder y 0.0 0.0 0.037 -0.0025 0.0 units box
11 # Make 1 atom type
```

(continues on next page)

(continued from previous page)

```
create_box      1 target
# Create the atoms in the simulation region
create_atoms    1 region target
pair_style      peri/pmb
#               <type1> <type2>    <c>    <horizon>  <s00> <alpha>
pair_coeff         *       *    1.6863e22 0.0015001 0.0005  0.25
# Set mass density
set             group all density 2200
# volume = lattice constant^3
set             group all volume 1.25e-10
# Zero out velocities of particles
velocity        all set 0.0 0.0 0.0 sum no units box
# Use velocity-Verlet time integrator
fix             F1 all nve
# Construct spherical indenter to shatter target
variable        y0 equal 0.00510
variable        vy equal -100
variable        y equal "v_y0 + step*dt*v_vy"
fix             F2 all indent 1e17 sphere 0.0000 v_y 0.0000 0.0050 units box
# Compute damage for each particle
compute         C1 all damage/atom
timestep        1.0e-7
thermo          200
dump            D1 all custom 100 dump.peri id type x y z c_C1
run             2000
```

**Note**

To use the LPS model, replace line 15 with *pair_style peri/lps* and modify line 16 accordingly.

### Numerical Results and Discussion

We ran the *input script from above*. Images of the disk (projectile not shown) appear in Figure below. The plot of damage on the top monolayer was created by coloring each particle according to its damage.

The symmetry in the computed solution arises because a "perfect" lattice was used, and a because a perfectly spherical projectile impacted the lattice at its geometric center. To break the symmetry in the solution, the nodes in the peridynamic body may be perturbed slightly from the lattice sites. To do this, the lattice of points can be slightly perturbed using the *displace_atoms* command.

---

**(Emmrich)** Emmrich, Weckner, Commun. Math. Sci., 5, 851-864 (2007),

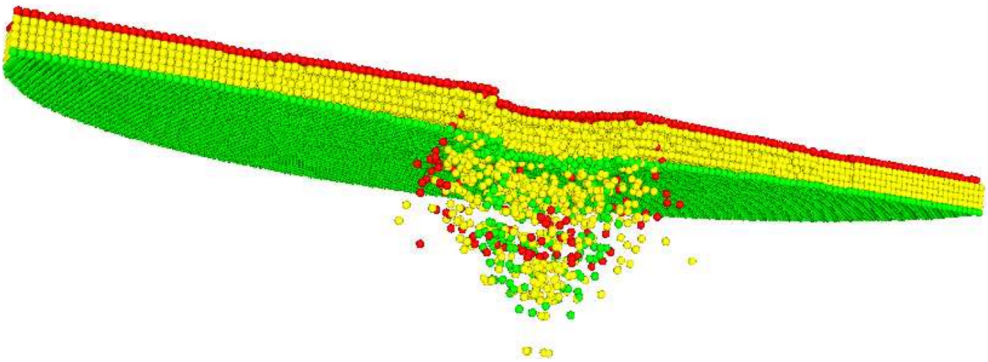
**(Parks)** Parks, Lehoucq, Plimpton, Silling, Comp Phys Comm, 179(11), 777-783 (2008).

**(Silling 2000)** Silling, J Mech Phys Solids, 48, 175-209 (2000).

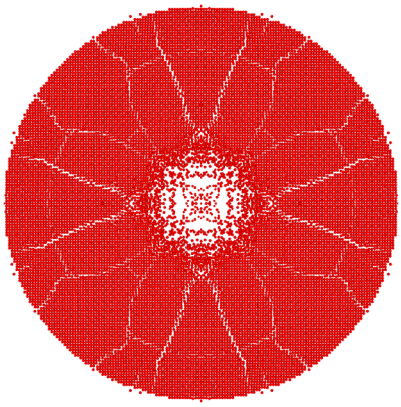
**(Silling 2005)** Silling Askari, Computer and Structures, 83, 1526-1535 (2005).

**(Silling 2007)** Silling, Epton, Weckner, Xu, Askari, J Elasticity, 88, 151-184 (2007).

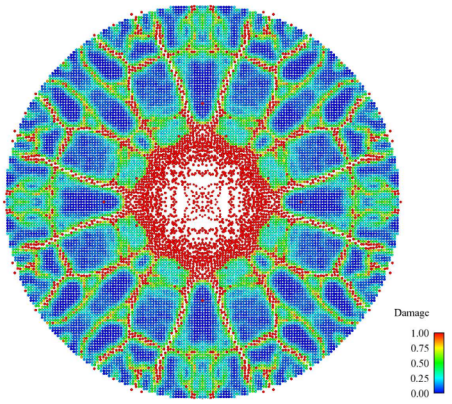
**(Seleson 2010)** Seleson, Parks, Int J Mult Comp Eng 9(6), pp. 689-706, 2011.



(a) Cut view of target during impact.



(b) Top monolayer showing fragmentation.




(c) Top monolayer showing damage. (blue = 0% broken bonds; red = 100% broken bonds)

Fig. 5: Target during (a) and after (b,c) impact

### 10.5.11 Manifolds (surfaces)

**Overview:**

This page is not about a LAMMPS input script command, but about manifolds, which are generalized surfaces, as defined and used by the MANIFOLD package, to track particle motion on the manifolds. See the src/MANIFOLD/README file for more details about the package and its commands.

Below is a list of currently supported manifolds by the MANIFOLD package, their parameters and a short description of them. The parameters listed here are in the same order as they should be passed to the relevant fixes.

| *manifold* | *parameters* | *equation* | *description* |
|---|---|---|---|
| cylinder | R | x^2 + y^2 - R^2 = 0 | Cylinder along z-axis, axis going through (0,0,0) |
| cylinder_de | R l a | x^2 + y^2 - r(z)^2 = 0, r(x) = R if \| z \| > l, r(z) = R - a*(1 + cos(z/l))/2 otherwise | A cylinder with a dent around z = 0 |
| dumbbell | a A B c | -( x^2 + y^2 ) + (a^2 - z^2/c^2) * ( 1 + (A*sin(B*z^2))^4) = 0 | A dumbbell |
| ellipsoid | a b c | (x/a)^2 + (y/b)^2 + (z/c)^2 - 1 = 0 | An ellipsoid |
| gaussian_bu | A l rc1 rc2 | if( x < rc1) -z + A * exp( -x^2 / (2 l^2) ); else if( x < rc2 ) -z + a + b*x + c*x^2 + d*x^3; else z | A Gaussian bump at x = y = 0, smoothly tapered to a flat plane z = 0. |
| plane | a b c x0 y0 z0 | a*(x-x0) + b*(y-y0) + c*(z-z0) = 0 | A plane with normal (a,b,c) going through point (x0,y0,z0) |
| plane_v | a w | z - a*sin(w*x) = 0 | A plane with a sinusoidal modulation on z along x. |
| sphere | R | x^2 + y^2 + z^2 - R^2 = 0 | A sphere of radius R |
| supersphere | R q | \| x \|^q + \| y \|^q + \| z \|^q - R^q = 0 | A supersphere of hyperradius R |
| spine | a, A, B, B2, c | -(x^2 + y^2) + (a^2 - z^2/f(z)^2)*(1 + (A*sin(g(z)*z^2))^4), f(z) = c if z > 0, 1 otherwise; g(z) = B if z > 0, B2 otherwise | An approximation to a dendritic spine |
| spine_t | a, A, B, B2, c | -(x^2 + y^2) + (a^2 - z^2/f(z)^2)*(1 + (A*sin(g(z)*z^2))^2), f(z) = c if z > 0, 1 otherwise; g(z) = B if z > 0, B2 otherwise | Another approximation to a dendritic spine |
| thylakoid | wB LB lB | Various, see *(Paquay)* | A model grana thylakoid consisting of two block-like compartments connected by a bridge of width wB, length LB and taper length lB |
| torus | R r | (R - sqrt( x^2 + y^2 ) )^2 + z^2 - r^2 | A torus with large radius R and small radius r, centered on (0,0,0) |

**(Paquay)** Paquay and Kusters, Biophys. J., 110, 6, (2016). preprint available at arXiv:1411.3019.

### 10.5.12 Reproducing hydrodynamics and elastic objects (RHEO)

The RHEO package is a hybrid implementation of smoothed particle hydrodynamics (SPH) for fluid flow, which can couple to the *BPM package* to model solid elements. RHEO combines these methods to enable mesh-free modeling of multi-phase material systems. Its SPH solver supports many advanced options including reproducing kernels, particle shifting, free surface identification, and solid surface reconstruction. To model fluid-solid systems, the status of particles can dynamically change between a fluid and solid state, e.g. during melting/solidification, which determines how they interact and their physical behavior. The package is designed with modularity in mind, so one can easily turn various features on/off, adjust physical details of the system, or develop new capabilities. For instance, the numerics associated with calculating gradients, reproducing kernels, etc. are separated into distinct classes to simplify the development of new integration schemes which can call these calculations. Additional numerical details can be found in *(Palermo)* and *(Clemmer)*. Example movies illustrating some of these capabilities are found at https://www.lammps.org/movies.html#rheopackage.

Note, if you simply want to run a traditional SPH simulation, the *SPH package* package is likely better suited for your application. It has fewer advanced features and therefore benefits from improved performance. The *MACHDYN* package for solids may also be relevant for fluid-solid problems.

---

At the core of the package is *fix rheo* which integrates particle trajectories and controls many optional features (e.g. the use of reproducing kernels). In conjunction to fix rheo, one must specify an instance of *fix rheo/pressure* and *fix rheo/viscosity* to define a pressure equation of state and viscosity model, respectively. Optionally, one can model a heat equation with *fix rheo/thermal*, which also allows the user to specify equations for a particle's thermal conductivity, specific heat, latent heat, and melting temperature. The ordering of these fixes in an an input script matters. Fix rheo must be defined prior to all other RHEO fixes.

Typically, RHEO requires atom style rheo. In addition to typical atom properties like positions and forces, particles store a local density, viscosity, pressure, and status. If thermal evolution is modeled, one must use atom style rheo/thermal which also includes a local energy, temperature, and conductivity. Note that the temperature is always derived from the energy. This implies the *temperature* attribute of *the set command* does not affect particles. Instead, one should use the *sph/e* attribute.

The status variable uses bit-masking to track various properties of a particle such as its current state of matter (fluid or solid) and its location relative to a surface. Some of these properties (and others) can be accessed using *compute rheo/property/atom*. The *status* attribute in *the set command* only allows control over the first bit which sets the state of matter, 0 is fluid and 1 is solid.

Fluid interactions, including pressure forces, viscous forces, and heat exchange, are calculated using *pair rheo*. Unlike typical pair styles, pair rheo ignores the *special bond* settings. Instead, it determines whether to calculate forces based on the status of particles: e.g., hydrodynamic forces are only calculated if a fluid particle is involved.

---

To model elastic objects, there are currently two mechanisms in RHEO, one designed for bulk solid bodies and the other for thin shells. Both mechanisms rely on introducing bonded forces between particles and therefore require a hybrid of atom style bond and rheo (or rheo/thermal).

To create an elastic solid body, one has to (a) change the status of constituent particles to solid (e.g. with the *set* command), (b) create bpm bonds between the particles (see the *bpm howto* page for more details), and (c) use *pair rheo/solid* to apply repulsive contact forces between distinct solid bodies. Akin to pair rheo, pair rheo/solid considers a particle's fluid/solid phase to determine whether to apply forces. However, unlike pair rheo, pair rheo/solid does obey special bond settings such that contact forces do not have to be calculated between two bonded solid particles in the same elastic body.

In systems with thermal evolution, fix rheo/thermal can optionally set a melting/solidification temperature allowing particles to dynamically swap their state between fluid and solid when the temperature exceeds or drops below the critical temperature, respectively. Using the *react* option, one can specify a maximum bond length and a bond type. Then, when solidifying, particles search their local neighbors and automatically create bonds with any neighboring

solid particles in range. For BPM bond styles, bonds then use the immediate position of the two particles to calculate a reference state. When melting, particles delete any bonds of the specified type when reverting to a fluid state. Special bonds are updated as bonds are created/broken.

The other option for elastic objects is an elastic shell that is nominally much thinner than a particle diameter, e.g. a oxide skin which gradually forms over time on the surface of a fluid. Currently, this is implemented using *fix rheo/oxidation* and bond style *rheo/shell*. Essentially, fix rheo/oxidation creates candidate bonds of a specified type between surface fluid particles within a specified distance. a newly created rheo/shell bond will then start a timer. While the timer is counting down, the bond will delete itself if particles move too far apart or move away from the surface. However, if the timer reaches a user-defined threshold, then the bond will activate and apply additional forces to the fluid particles. Bond style rheo/shell then operates very similarly to a BPM bond style, storing a reference length and breaking if stretched too far. Unlike the above method, this option does not remove the underlying fluid interactions (although particle shifting is turned off) and does not modify special bond settings of particles.

While these two options are not expected to be appropriate for every system, either framework can be modified to create more suitable models (e.g. by changing the criteria for creating/deleting a bond or altering force calculations).

---

**(Palermo)** Palermo, Wolf, Clemmer, O'Connor, Phys. Fluids, 36, 113337 (2024).

**(Clemmer)** Clemmer, Pierce, O'Connor, Nevins, Jones, Lechman, Tencer, Appl. Math. Model., 130, 310-326 (2024).

### 10.5.13 Magnetic spins

The magnetic spin simulations are enabled by the SPIN package, whose implementation is detailed in *Tranchida*.

The model represents the simulation of atomic magnetic spins coupled to lattice vibrations. The dynamics of those magnetic spins can be used to simulate a broad range a phenomena related to magneto-elasticity, or or to study the influence of defects on the magnetic properties of materials.

The magnetic spins are interacting with each others and with the lattice via pair interactions. Typically, the magnetic exchange interaction can be defined using the *pair/spin/exchange* command. This exchange applies a magnetic torque to a given spin, considering the orientation of its neighboring spins and their relative distances. It also applies a force on the atoms as a function of the spin orientations and their associated inter-atomic distances.

The command *fix precession/spin* allows to apply a constant magnetic torque on all the spins in the system. This torque can be an external magnetic field (Zeeman interaction), and an uniaxial or cubic magnetic anisotropy.

A Langevin thermostat can be applied to those magnetic spins using *fix langevin/spin*. Typically, this thermostat can be coupled to another Langevin thermostat applied to the atoms using *fix langevin* in order to simulate thermostatted spin-lattice systems.

The magnetic damping can also be applied using *fix langevin/spin*. It allows to either dissipate the thermal energy of the Langevin thermostat, or to perform a relaxation of the magnetic configuration toward an equilibrium state.

The command *fix setforce/spin* allows to set the components of the magnetic precession vectors (while erasing and replacing the previously computed magnetic precession vectors on the atom). This command can be used to freeze the magnetic moment of certain atoms in the simulation by zeroing their precession vector.

The command *fix nve/spin* can be used to perform a symplectic integration of the combined dynamics of spins and atomic motions.

The minimization style *min/spin* can be applied to the spins to perform a minimization of the spin configuration.

All the computed magnetic properties can be output by two main commands. The first one is *compute spin*, that enables to evaluate magnetic averaged quantities, such as the total magnetization of the system along x, y, or z, the spin temperature, or the magnetic energy. The second command is *compute property/atom*. It enables to output all the per atom magnetic quantities. Typically, the orientation of a given magnetic spin, or the magnetic force acting on this spin.

---

**(Tranchida)** Tranchida, Plimpton, Thibaudeau and Thompson, Journal of Computational Physics, 372, 406-425, (2018).

### 10.5.14 Adaptive-precision interatomic potentials (APIP)

The *PKG-APIP* enables use of adaptive-precision potentials as described in *(Immel2025)* and *(Immel2026)*. In the context of this package, precision refers to the accuracy of an interatomic potential.

Modern machine-learning (ML) potentials translate the accuracy of DFT simulations into MD simulations, i.e., ML potentials are more accurate compared to traditional empirical potentials. However, this accuracy comes at a cost: there is a considerable performance gap between the evaluation of classical and ML potentials, e.g., the force calculation of a classical EAM potential is 100-1000 times faster compared to the ML-based ACE method. The evaluation time difference results in a conflict between large time and length scales on the one hand and accuracy on the other. This conflict is resolved by an APIP model for simulations, in which the highest precision is required only locally but not globally.

An APIP model uses a precise but expensive ML potential only for a subset of atoms, while a fast potential is used for the remaining atoms. Whether the precise or the fast potential is used is determined by a continuous switching parameter $\lambda_i$ that can be defined for each atom $i$. The switching parameter can be adjusted dynamically during a simulation or kept constant as explained below.

The potential energy $E_i$ of an atom $i$ described by an adaptive-precision interatomic potential is given by *(Immel2025)*

$$E_i = \lambda_i E_i^{(\mathrm{fast})} + (1 - \lambda_i) E_i^{(\mathrm{precise})},$$

where $E_i^{(\mathrm{fast})}$ is the potential energy of atom $i$ according to a fast interatomic potential, $E_i^{(\mathrm{precise})}$ is the potential energy according to a precise interatomic potential and $\lambda_i \in [0, 1]$ is the switching parameter that decides how the potential energies are weighted.

Adaptive-precision saves computation time when the computation of the precise potential is not required for many atoms, i.e., when $\lambda_i = 1$ applies for many atoms.

The currently implemented potentials are:

| Fast potential | Precise potential |
|---|---|
| *ACE* | *ACE* |
| *EAM* | |

In theory, any short-range potential can be used for an adaptive-precision interatomic potential. How to implement a new (fast or precise) adaptive-precision potential is explained in *here*.

The switching parameter $\lambda_i$ that combines the two potentials can be dynamically calculated during a simulation. There are two ways to calculate dynamic switching parameters.

1. according to *(Immel2026)* with a differentiable switching parameter that results in a conservative potential. Energy and momentum are (in the absence of external forces) conserved by design.

2. according to *(Immel2025)* with a non-differentiable switching parameter. In this case, the implementation can be optimized for performance by using the switching parameters of the previous timestep. Thereby, one can perform most of the switching-parameter calculation within the force-calculation routine and include this calculations in the load-balancing. The potential is not conservative and energy- and momentum-conservation are achieved through a local correction.

Alternatively, one can set a constant switching parameter before the start of a simulation. Using constant switching parameters results in a conservative potential.

To run a simulation with an adaptive-precision potential, one needs the following components:

**conservative dynamic switching parameter**

1. *atom_style apip conservative* so that the switching parameter $\lambda_i$ can be stored.
2. A fast potential: *eam/apip* or *pace/fast/apip*.
3. A precise potential: *pace/precise/apip*.
4. *fix lambda/la/csp/apip* to calculate a differentiable switching parameter $\lambda_i$.
5. *pair_style hybrid/overlay* to combine the previously mentioned pair_styles.
6. *fix atom_weight/apip* to approximate the load caused by every atom, as the computations of the pair_styles are only required for a subset of atoms.
7. *fix balance* to perform dynamic load balancing with the calculated load.

**dynamic switching parameter with correction**

1. *atom_style apip* so that the switching parameter $\lambda_i$ can be stored.
2. A fast potential: *eam/apip* or *pace/fast/apip*.
3. A precise potential: *pace/precise/apip*.
4. *pair_style lambda/input/apip* to calculate $\lambda_i^{\text{input}}$, from which $\lambda_i$ is calculated.
5. *fix lambda/apip* to calculate the switching parameter $\lambda_i$.
6. *pair_style lambda/zone/apip* to calculate the spatial transition zone of the switching parameter.
7. *pair_style hybrid/overlay* to combine the previously mentioned pair_styles.
8. *fix lambda_thermostat/apip* to conserve the energy when switching parameters change.
9. *fix atom_weight/apip* to approximate the load caused by every atom, as the computations of the pair_styles are only required for a subset of atoms.
10. *fix balance* to perform dynamic load balancing with the calculated load.

**constant switching parameter**

1. *atom_style apip* so that the switching parameter $\lambda_i$ can be stored.
2. A fast potential: *eam/apip* or *pace/fast/apip*.
3. A precise potential: *pace/precise/apip*.
4. *set* command to set the switching parameter $\lambda_i$.
5. *pair_style hybrid/overlay* to combine the previously mentioned pair_styles.
6. *fix atom_weight/apip* to approximate the load caused by every atom, as the computations of the pair_styles are only required for a subset of atoms.
7. *fix balance* to perform dynamic load balancing with the calculated load.

### Example

**Note**

How to select the values of the parameters of an adaptive-precision interatomic potential is discussed in detail in *(Immel2025)* and *(Immel2026)*.

**conservative dynamic switching parameter**

Lines like these would appear in the input script:

```
atom_style apip conservative
comm_style tiled

pair_style hybrid/overlay eam/fs/apip pace/precise/apip
pair_coeff * * eam/fs/apip Cu.eam.fs Cu
pair_coeff * * pace/precise/apip Cu.yace Cu

# calculate a differentiable switching parameter to obtain
# a conservative potential
fix 2 all lambda/la/csp/apip 0.24 1.5 11.0 12.0 bcc

fix 4 all atom_weight/apip 100 eam ace 0 0 all

variable myweight atom f_4

fix 5 all balance 100 1.1 rcb weight var myweight
```

First, the *atom_style apip conservative* and the communication style are set.

**Note**

Note, that *comm_style tiled* is required for the style *rcb* of *fix balance*, but not for APIP. However, the flexibility offered by the balancing style *rcb*, compared to the balancing style *shift*, is advantageous for APIP.

A conservative adaptive-precision EAM-ACE potential is defined based on the differentiable switching parameter calculated by *fix lambda/la/csp/apip*, i.e., $\lambda$ is calculated from the locally averaged CSP. The *pair_style hybrid/overlay* combines the fast EAM and the precise ACE potential, interpolated by the corresponding switching parameter. Furthermore, the *fix lambda/la/csp/apip* calculates the forces caused by the differentiation of the switching parameter. Thereby, one obtains a conservative potential ($\boldsymbol{F}_i = -\nabla_i \sum_k E_k$) that by design conserves energy and momentum in the absence of external forces.
The fix 4 calculates the computational weight that is used by fix 5 to balance the load between different processors.

**dynamic switching parameter with correction**

Lines like these would appear in the input script:

```
atom_style apip
comm_style tiled

pair_style hybrid/overlay eam/fs/apip pace/precise/apip lambda/input/csp/apip fcc␣
↪cutoff 5.0 lambda/zone/apip 12.0
pair_coeff * * eam/fs/apip Cu.eam.fs Cu
pair_coeff * * pace/precise/apip Cu.yace Cu
pair_coeff * * lambda/input/csp/apip
pair_coeff * * lambda/zone/apip

fix 2 all lambda/apip 2.5 3.0 time_averaged_zone 4.0 12.0 110 110 min_delta_lambda␣
↪0.01
fix 3 all lambda_thermostat/apip N_rescaling 200
fix 4 all atom_weight/apip 100 eam ace lambda/input lambda/zone all

variable myweight atom f_4

fix 5 all balance 100 1.1 rcb weight var myweight
```

First, the *atom_style apip* and the communication style are set.

**Note**

Note, that *comm_style tiled* is required for the style *rcb* of *fix balance*, but not for APIP. However, the flexibility offered by the balancing style *rcb*, compared to the balancing style *shift*, is advantageous for APIP.

An adaptive-precision EAM-ACE potential, for which the switching parameter $\lambda$ is calculated from the CSP, is defined via *pair_style hybrid/overlay*. The fixes ensure that the switching parameter is calculated, the energy conserved, the weight for the load balancing calculated and the load-balancing itself is done.

**constant switching parameter**

Lines like these would appear in the input script:

```
atom_style apip
comm_style tiled

pair_style hybrid/overlay eam/fs/apip pace/precise/apip
pair_coeff * * eam/fs/apip Cu.eam.fs Cu
pair_coeff * * pace/precise/apip Cu.yace Cu

# calculate lambda somehow
variable lambda atom ...
set group all apip/lambda v_lambda

fix 4 all atom_weight/apip 100 eam ace lambda/input lambda/zone all

variable myweight atom f_4

fix 5 all balance 100 1.1 rcb weight var myweight
```

First, the *atom_style apip* and the communication style are set.

**Note**

Note, that *comm_style tiled* is required for the style *rcb* of *fix balance*, but not for APIP. However, the flexibility offered by the balancing style *rcb*, compared to the balancing style *shift*, is advantageous for APIP.

An adaptive-precision EAM-ACE potential is defined via *pair_style hybrid/overlay*. The switching parameter $\lambda_i$ of the adaptive-precision EAM-ACE potential is set via the *set command*. The parameter is not updated during the simulation. Therefore, the potential is conservative. The fixes ensure that the weight for the load balancing is calculated and the load-balancing itself is done.

### Implementing new APIP pair styles

One can introduce adaptive-precision to an existing pair style by modifying the original pair style. One should calculate the force $F_i = -\nabla_i \sum_j E_j^{\text{original}}$ for a fast potential or $F_i = -(1-\nabla_i) \sum_j E_j^{\text{original}}$ for a precise potential from the original potential energy $E_j^{\text{original}}$ to see where the switching parameter $\lambda_i$ needs to be introduced in the force calculation. The switching parameter $\lambda_i$ is known for all atoms $i$ in force calculation routine. One needs to introduce an abortion criterion based on $\lambda_i$ to ensure that all not required calculations are skipped and compute time can be saved. Furthermore, one needs to provide the number of calculations and measure the computation time. Communication within the force calculation needs to be prevented to allow effective load-balancing. With communication, the load balancer cannot balance few calculations of the precise potential on one processor with many computations of the fast potential on another processor.

All changes in the pair_style pace/apip compared to the pair_style pace are annotated and commented. Thus, the pair_style pace/apip can serve as an example for the implementation of new adaptive-precision potentials.

**(Immel2025)** Immel, Drautz and Sutmann, J Chem Phys, 162, 114119 (2025)

**(Immel2026)** Immel, Drautz and Sutmann, arXiv:2512.07693

## 10.5.15 Local Density Dependent Potentials via LDD


- *Overview*
- *Local Density Definitions*
- *Local Density/ Square Gradient Potentials*
- *Input File Examples*
  - *Example 1) A simple atomic input example using only tabulated LD potentials*
  - *Example 2) Molecular input example that layers tabulated pair, LD and SG interactions via* pair_style hybrid
- *read_data file format*
- *Outputting the local-density data*
- *When to use LD/SG potentials:*


### Overview

The *LDD pair style* in the BOCS package implements potentials that are a function of the local density, $\rho_{\beta|I}$, of type $\beta$ particles around a central particle, $I$. There are two types of local density dependent potentials implemented. There are Local Density (LD) potentials $U_{LD}$ , which are direct functions of $\rho_{\beta|I}$. There are also Square Gradient (SG) potentials $U_{SG}$. SG potentials use a direct function of the local density, $U_{\nabla}(\rho_{\beta|I})$, as a coefficient of the square gradient, $|\frac{\partial \rho_{\beta|I}}{\partial R_I}|^2$ of a particular local density. Such potentials are useful for coarse grained models and can be parameterized from trajectory data using BOCS version 5 and higher.

### Local Density Definitions

Each site (atom) in a molecular dynamics simulation $I$ is assigned a type $t_I$. These site types may be labeled e.g. $\alpha$, $\beta$, ..., etc. The local density, $\rho_{\beta|I}$, of $\beta$ particles around a central particle, $I$, of type $t_I = \alpha$ is defined:

$$\rho_{\beta|I} = \sum_{J \in S_{\beta|I}} \bar{w}_{\beta|\alpha}(r_{IJ})$$

where $S_{\beta|I}$ is the set of particles of type $\beta$ that are not excluded from pairwise interactions involving site $I$. $\bar{w}_{\beta|\alpha}(r_{IJ})$ is a normalized indicator function, it determines the pair weighted contribution from each particle $J$, of type $t_J = \beta$ to the $\beta$ local density around a central particle $I$ of type $t_I = \alpha$.

The normalized indicator function is defined by dividing a continuous/differentiable non-negative non-increasing indicator function $w(r)$ by its spatial integral $[w]$. Therefore, $\bar{w}(r)$ is defined for each $\beta|\alpha$ pair as:

$$\bar{w}(r) = \frac{w(r)}{[w]}\theta(r - r_c)$$
$$[w] = \int_0^{r_c} 4\pi r^2 w(r) dr$$

where $\theta$ is the Heaviside function, $r_c$ is the local density length-scale, and $\bar{w}(r)$ is defined such that $\bar{w}(r) = 0$, if $r > r_c$. The local density of $\beta$ particles that surround a given particle $I$ is defined by the sum of contributions from $\beta$ particles that are within (<=) the cutoff $r_c$. The choice of indicator function is specified by the user in the *indicator* argument of the *ldd pair_coeff* command. Different options are listed under different pages indexed on the *pair ldd* page. Note that the weighting function and its length-scale can be defined independently for each pair of site types.

If particle $I$ is of type $t_I = \beta$, its contribution to the local density $\rho_{t_I|I}$ can be included or excluded from the sum without changing the net force on a pair of particles. Practically, the only difference is a horizontal shift in the argument of the LD potential function $U_{\beta|\beta}(\rho)$. Whether or not to include this contribution is set by the user in the *ldd pair_coeff* command via the *self yes* or *self no* arguments.

For molecular systems, particles included in the local density around a given particle $I$ are identified by the neighbor list for the non-bonded interactions. That is, a particle excluded by the neighbor list for a central particle will also be excluded from the sum defining its local density.

For example, consider a 5 bead chain that is simulated with default 1-2, 1-3, and 1-4 non-bonded exclusions:

$A_1 - B_2 - C_3 - D_4 - E_5$

The local density of particles of type E surrounding site 1, $\rho_{E|1}$ will include the contribution from the end chain particle $\bar{w}_{E|A}(r_{15})$. Conversely, the local density of particles of type B surrounding site 1, $\rho_{B|1}$ will only include intermolecular contributions to the local density. Whether or not the local density of A particles surrounding site 1, $\rho_{A|1}$, will include the *self* intramolecular contribution $\bar{w}_{A|A}(0)$ is determined by whether *self yes* or *self no* is listed as an argument by the user in the *ldd pair_coeff* command.

The choice of local density definition used for a particular interaction between types, $\beta|\alpha$ is set by the user via the *ldd pair_coeff* command with the different available options for indicator functions. Note, that the weighting function, $\bar{w}_{\beta|\alpha}$, and local density potential, $U_{\beta|\alpha}$ are not symmetric with respect to $\alpha$ and $\beta$. For instance, if $\alpha = v$ corresponds to solvents and $\beta = u$ corresponds to solutes, then the package allows the user to specify distinct weighting functions, $\bar{w}_{v|u} \neq \bar{w}_{u|v}$ and distinct LD potentials $U_{v|u} \neq U_{u|v}$, for treating solvent around solute and solute around solvent respectively.

---

### Local Density/ Square Gradient Potentials

The LDD pair style implements forces for two kinds of local density dependent potentials.

LD (Local Density) potentials are defined:

$$U_{LD}(\mathbf{R}) = \sum_I \sum_\beta U_{\beta|t_I}(\rho_{\beta|I}),$$

where the first sum is over all sites, $I$ , and the second sum is over all site types, $\beta$. The form of $U_{\beta|\alpha}$ is specified by the user in the *ldd pair_coeff* command using the *potential* keyword. For instance, $U_{\beta|\alpha}(\rho_{\beta|I})$ can be defined as zero (*noforce*) or as a *constant*, *linear*, or *quadratic* function of the local density. Alternatively, $U_{\beta|\alpha}(\rho_{\beta|I})$ can be specified by a table of values using the *table/lin* or *table/spline* keywords. Additionally the *potential* arg *mdpd* is a quadratic function in local density that can be combined with the dpd pair_style to simulate pairwise mdpd interactions in the LD framework. All of these forms are documented on the *pair_style ldd* page.

SG (Square Gradient) potentials are defined:

$$U_{SG}(\mathbf{R}) = \sum_I \sum_\beta U_{\nabla;\beta|t_I}(\rho_{\beta|I}) \left|\nabla_I \rho_{\beta|I}\right|^2$$

where the first sum is over all sites, $I$, the second sum is over all site types, $\beta$ and the gradient $\nabla_I = \frac{\partial}{\partial \mathbf{R_I}}$ is evaluated with respect to the position of particle $I$. The form of the coefficient, $U_{\nabla;\beta|t_I}$, is specified by the user in the *ldd pair_coeff* command using the (optional) *gradient* keyword. Many of the available functional forms for the LD potential are also available for the SG coefficients, see *pair_ldd* for details.

---

For any given site $I$, once the local densities $\rho_{\beta|I}$ and the gradients of the local density $\nabla_I \rho_{\beta|I}$ have been determined, the corresponding forces are pairwise decomposable with pair forces analogous to those stated in Delyser 2021 and Delyser 2019.

The per-atom local densities and gradients are owned by the *ldd* pair style itself (recomputed every step, not stored in data or restart files); no special atom style is required, and the data is exposed to the rest of LAMMPS through *fix pair* (see the *pair ldd* page).

Interactions are specified per *species* pair. The potential file defines a set of species and the `pair_coeff * * file. ldd S1 S2 ...` command maps each atom type to a species (several types may map to the same species). Each of the $N^2_{\text{species}}$ *ordered* species pairs is given on its own line. In particular, the entry `A B` specifies the interaction for atoms of species A surrounded by atoms of species B, and is distinct from `B A`, the interaction for species B surrounded by species A. This "surrounded by" convention (`A B`) corresponds to the local density often written as `B|A` (species B given a central particle of species A) in published works.

---

### Input File Examples

#### Example 1) A simple atomic input example using only tabulated LD potentials

The input script uses a plain atomic atom style and maps the two atom types to species A and B from the potential file:

```
## Init system/units
units real
dimension 3
atom_style atomic

newton on
timestep 1

read_data my_mix.data
velocity all create 300.0 22345 dist gaussian

## pair_style ldd takes no cutoff (it comes from each indicator rc)
pair_style ldd
pair_coeff * * my_mix.ldd A B   # atom type 1 -> species A, type 2 -> species B

## Run / Output
run_style verlet
neigh_modify    delay 5 every 1 check yes
neighbor        2.0 bin
thermo 500

fix 1 all nvt temp 300.0 300.0 100.0
# expose the per-atom local densities (one column per species) + total energy
fix lddout all pair 1 ldd local_density 0 total_energy 0
dump 1 all custom 500 dump.txt id x y z f_lddout[*]
dump_modify 1 sort id

run 10000
```

with the potential file `my_mix.ldd`:

```
# all four ordered species pairs must be listed (use "ignore" for none)
A A indicator dpd 0.0 6.5 self yes potential table/lin LD_table.1.1.dat
A B ignore
B A indicator dpd 0.0 5.5 self no  potential table/lin LD_table.2.1.dat
B B indicator dpd 0.0 5.5 self yes potential table/lin LD_table.2.2.dat
```

Note in the above example, only species pairs A|A, A|B, and B|B interact according to LD potentials (the A B entry, corresponding to B|A, is ignored).

### Example 2) Molecular input example that layers tabulated pair, LD and SG interactions via *pair_style hybrid*

The molecular topology uses the ordinary *full* atom style; the *ldd* interactions are layered in with *pair_style hybrid/overlay*:

```
## Initialize system /units

units real
dimension 3
atom_style full

newton on
timestep 1

bond_style harmonic

region my_box block 0 40 0 40 0 40
create_box 2 my_box bond/types 1 extra/bond/per/atom 1

mass 1 59.044
mass 2 59.044

pair_style hybrid/overlay ldd table spline 3000

molecule DI DIMER.mol
create_atoms 0 random 250 885401 my_box mol DI 454353

include ff_bonded_params.lammps
velocity all create 300.0 22345 dist gaussian

## Pair styles
pair_coeff * * table lammps_nb_ALL.table nb_All 15.0
pair_coeff * * ldd dimer.ldd A B   # type 1 -> species A, type 2 -> species B

## Run / Output
run_style verlet
neigh_modify    delay 5 every 1 check yes
neighbor        2.0 bin
thermo 500

fix 1 all nvt temp 300.0 300.0 100.0
run 10000
```

with the potential file `dimer.ldd`:

```
A A indicator lucy 0.0 6.5 self no potential table/spline LD.1.1.dat gradient table/
↪gradspline SG.1.1.dat
A B indicator lucy 0.0 6.5 self no potential table/spline LD.1.2.dat
B A indicator lucy 0.0 6.5 self no potential table/spline LD.2.1.dat
B B indicator lucy 0.0 6.5 self no potential table/spline LD.2.2.dat gradient table/
```

(continues on next page)

(continued from previous page)

```
↪gradspline SG.2.2.dat
```

Note in the above example LD cross interactions are both specified, but not necessarily symmetric. Also in this example, the cross interactions opt not to use the optional *gradient* keyword. Conversely the like-species interactions (A|A, B|B) layer SG potentials on top of the already defined LD and pair interactions.

See `examples/PACKAGES/bocs` for full input files for a 4-site chain example (topology like A-B-C-D).

See *pair_style ldd* for all pair_coeff *args* options that exist, all restrictions, and more examples.

---

### read_data file format

The *ldd* pair style does not require a special atom style: the local densities, their gradients, and the associated energies are owned by the pair style and recomputed every step, so they are never read from or written to a data file. The `read_data` input therefore follows the usual format for whatever atom style you choose, e.g. *atomic* for a simple system or *full* for a molecular one.

---

### Outputting the local-density data

For each central particle $I$ and each species $\beta$, there is a local density of $\beta$ particles that surround $I$, $\rho_{\beta|I}$, and a corresponding gradient $\frac{\partial \rho_{\beta|I}}{\partial R_I}$. These quantities (along with the per-species LD/SG energies and the per-atom total energy) are made available to the rest of LAMMPS through the *fix pair* command, which copies the chosen pair-style fields into a per-atom array. *pair_style ldd* describes the available options and quantities output.

For example:

```
fix lddout all pair 1 ldd local_density 0 grad_density 0 total_energy 0
dump 1 all custom 500 dump.txt id x y z vx vy vz fx fy fz f_lddout[*]
dump_modify 1 sort id
```

writes a custom trajectory that includes the local densities and their gradients alongside the usual positions, velocities, and forces (one column per species, three per species for the gradient). Such a trajectory can be used with the Bottom-up Open-source Coarse-graining Software, which can parameterize LD/SG potentials from atomistic data and convert LAMMPS trajectories to .trr files for analysis with gromacs tools.

---

### When to use LD/SG potentials:

This package has been primarily developed in the context of parameterizing/simulating bottom up coarse grained models. We have found that LD/SG potentials are particularly useful for supplementing pair potentials when simulating CG models in the NPT ensemble and in different interfacial environments. In particular, such potentials dramatically improve CG descriptions of the pressure-density equation of state, interfacial profiles, liquid-vapor coexistence, and transferability between bulk and interfacial systems. SG potentials are slower to simulate than LD potentials alone, so where applicable for coarse graining we recommend trying to construct/simulate LD potentials first and then [if necessary] adding SG potentials to refine further. CG force fields with LD and SG potentials can be parameterized via the multi-scale coarse graining method implemented in BOCSv5 and higher.

---

A number of works have been published using this package (with BOCS) in its development stage for different CG studies and are listed below:

**(DeLyser)** DeLyser, Noid. "Extending Pressure-Matching to Inhomogeneous Systems via Local-Density Potentials." The Journal of Chemical Physics, 147, no. 13: 134111 (2017)

**(DeLyser)** DeLyser, Noid. "Analysis of local density potentials." The Journal of Chemical Physics, 151, no. 22:224106 (2019)

**(DeLyser)** DeLyser, Noid "Bottom-up coarse-grained models for external fields and interfaces" The Journal of Chemical Physics 153, 224103 (2020)

**(DeLyser)** DeLyser, Noid "Coarse-grained models for local density gradients" The Journal of Chemical Physics 156, 034106 (2021)

**(Szukalo)** Szukalo, Noid "A temperature-dependent length-scale for transferable local density potentials". The Journal of Chemical Physics, 159, 074104 (2023)

**(Lesniewski)** Lesniewski, Remsing, Noid "Revisiting what we lose by coarse-graining: Modeling cooperative hydrophobic phenomena with short-ranged, pair-additive forces." The Journal of Chemical Physics, 164, 204111 (2026)

**(Dutta)** Dutta, Lesniewski, Qaisrani, Noid, Andrienko, Nikoubashman. "Accurate coarse-graining of conjugated organic molecules in melts and thin films using density-dependent potentials". The Journal of Chemical Theory and Computation, 22, 7, 3697-3708 (2026)

**(Lesniewski)** Lesniewski, DeLyser, W. G. Noid "Progress toward a better BOCS: systematic coarse-graining with local density potentials" In Prep 2026

## 10.6 Tutorials howto

### 10.6.1 Using CMake with LAMMPS

The support for building LAMMPS with CMake is a recent addition to LAMMPS thanks to the efforts of Christoph Junghans (LANL) and Richard Berger (LANL). One of the key strengths of CMake is that it is not tied to a specific platform or build system. Instead it generates the files necessary to build and develop for different build systems and on different platforms. Note, that this applies to the build system itself not the LAMMPS code. In other words, without additional porting effort, it is not possible - for example - to compile LAMMPS with Visual C++ on Windows. The build system output can also include support files necessary to program LAMMPS as a project in integrated development environments (IDE) like Eclipse, Visual Studio, QtCreator, Xcode, CodeBlocks, Kate and others.

A second important feature of CMake is that it can detect and validate available libraries, optimal settings, available support tools and so on, so that by default LAMMPS will take advantage of available tools without requiring to provide the details about how to enable/integrate them.

The downside of this approach is, that there is some complexity associated with running CMake itself and how to customize the building of LAMMPS. This tutorial will show how to manage this through some selected examples. Please see the chapter about *building LAMMPS* for descriptions of specific flags and options for LAMMPS in general and for specific packages.

Changed in version 10Sep2025.

CMake can be used through either the command-line interface (CLI) program `cmake` (or `cmake3`), a text mode interactive user interface (TUI) program `ccmake` (or `ccmake3`), or a graphical user interface (GUI) program `cmake-gui`. All of them are portable software available on all supported platforms and can be used interchangeably. Since LAMMPS version 10Sep2025, the minimum required CMake version is 3.20.

All details about features and settings for CMake are in the CMake online documentation. We focus below on the most important aspects with respect to compiling LAMMPS.

### Prerequisites

This tutorial assumes that you are operating in a command-line environment using a shell like Bash or Zsh.

- Linux: any Terminal window will work or text console
- macOS: launch the Terminal application
- Windows 10 or 11: install and run the *Windows Subsystem for Linux*
- other Unix-like operating systems like FreeBSD

> **Note**
>
> It is also possible to use CMake on Windows 10 or 11 through either the Microsoft Visual Studio IDE with the bundled CMake or from the Windows command prompt using a separately installed CMake package, both using the native Microsoft Visual C++ compilers and (optionally) the Microsoft MPI SDK. This tutorial, however, only covers unix-like command-line interfaces.

We also assume that you have downloaded and unpacked a recent LAMMPS source code package or used Git to create a clone of the LAMMPS sources on your compilation machine.

You should change into the top level directory of the LAMMPS source tree all paths mentioned in the tutorial are relative to that. Immediately after downloading it should look like this:

```
$ ls
bench  doc       lib      potentials  README  tools
cmake  examples  LICENSE  python      src
```

### Build versus source directory

When using CMake the build procedure is separated into multiple distinct phases:

1. **Configuration:** detect or define which features and settings should be enable and used and how LAMMPS should be compiled
2. **Compilation:** generate and compile all necessary source files and build libraries and executables.
3. **Installation:** copy selected files from the compilation into your file system, so they can be used without having to keep the source and build tree around.

The configuration and compilation of LAMMPS has to happen in a dedicated *build directory* which must be different from the source directory. Also the source directory (`src`) must remain pristine, so it is not allowed to "install" packages using the traditional make process and after an compilation attempt all created source files must be removed. This can be achieved with `make no-all purge`.

You can pick **any** folder outside the source tree. We recommend to create a folder `build` in the top-level directory, or multiple folders in case you want to have separate builds of LAMMPS with different options (`build-parallel`, `build-serial`) or with different compilers (`build-gnu`, `build-clang`, `build-intel`) and so on. All the auxiliary files created by one build process (executable, object files, log files, etc) are stored in this directory or subdirectories within it that CMake creates.

### Running CMake

#### CLI version

In the (empty) `build` directory, we now run the command `cmake ../cmake`, which will start the configuration phase and you will see the progress of the configuration printed to the screen followed by a summary of the enabled features, options and compiler settings. A typical summary screen will look like this:

```
$ cmake ../cmake/
-- The CXX compiler identification is GNU 8.2.0
-- Check for working CXX compiler: /opt/tools/gcc-8.2.0/bin/c++
-- Check for working CXX compiler: /opt/tools/gcc-8.2.0/bin/c++ - works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found Git: /usr/bin/git (found version "2.25.2")
-- Running check for auto-generated files from make-based build system
-- Found MPI_CXX: /usr/lib64/mpich/lib/libmpicxx.so (found version "3.1")
-- Found MPI: TRUE (found version "3.1")
-- Looking for C++ include omp.h
-- Looking for C++ include omp.h - found
-- Found OpenMP_CXX: -fopenmp (found version "4.5")
-- Found OpenMP: TRUE (found version "4.5")
-- Found JPEG: /usr/lib64/libjpeg.so (found version "62")
-- Found PNG: /usr/lib64/libpng.so (found version "1.6.37")
-- Found ZLIB: /usr/lib64/libz.so (found version "1.2.11")
-- Found GZIP: /usr/bin/gzip
-- Found FFMPEG: /usr/bin/ffmpeg
-- Performing Test COMPILER_SUPPORTS-ffast-math
-- Performing Test COMPILER_SUPPORTS-ffast-math - Success
-- Performing Test COMPILER_SUPPORTS-march=native
-- Performing Test COMPILER_SUPPORTS-march=native - Success
-- Looking for C++ include cmath
-- Looking for C++ include cmath - found
-- Generating style_angle.h...
[...]
-- Generating lmpinstalledpkgs.h...
-- The following tools and libraries have been found and configured:
 * Git
 * MPI
 * OpenMP
 * JPEG
 * PNG
 * ZLIB

-- <<< Build configuration >>>
   Build type:       RelWithDebInfo
   Install path:     /home/akohlmey/.local
   Generator:        Unix Makefiles using /usr/bin/gmake
-- <<< Compilers and Flags: >>>
-- C++ Compiler:     /opt/tools/gcc-8.2.0/bin/c++
      Type:          GNU
      Version:       8.2.0
      C++ Flags:     -O2 -g -DNDEBUG
      Defines:       LAMMPS_SMALLBIG;LAMMPS_MEMALIGN=64;LAMMPS_JPEG;LAMMPS_PNG;LAMMPS_
↪GZIP;LAMMPS_FFMPEG
      Options:       -ffast-math;-march=native
-- <<< Linker flags: >>>
-- Executable name:  lmp
-- Static library flags:
```

(continues on next page)

(continued from previous page)

```
-- <<< MPI flags >>>
-- MPI includes:     /usr/include/mpich-x86_64
-- MPI libraries:    /usr/lib64/mpich/lib/libmpicxx.so;/usr/lib64/mpich/lib/libmpi.so;
-- Configuring done
-- Generating done
-- Build files have been written to: /home/akohlmey/compile/lammps/build
```

The `cmake` command has one mandatory argument, and that is a folder with either the file `CMakeLists.txt` or `CMakeCache.txt`. The `CMakeCache.txt` file is created during the CMake configuration run and contains all active settings, thus after a first run of CMake all future runs in the build folder can use the folder `.` and CMake will know where to find the CMake scripts and reload the settings from the previous step. This means, that one can modify an existing configuration by re-running CMake, but only needs to provide flags indicating the desired change, everything else will be retained. One can also mix compilation and configuration, i.e. start with a minimal configuration and then, if needed, enable additional features and recompile.

The steps above **will NOT compile the code**. The compilation can be started in a portable fashion with `cmake --build .`, or you use the selected built tool, e.g. `make`.

### TUI version

For the text mode UI CMake program the basic principle is the same. You start the command `ccmake ../cmake` in the `build` folder.

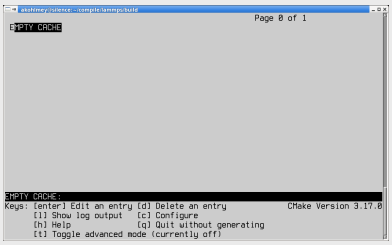



Fig. 6: Initial `ccmake` screen

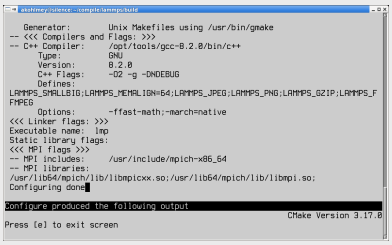



Fig. 7: Configure output of `ccmake`

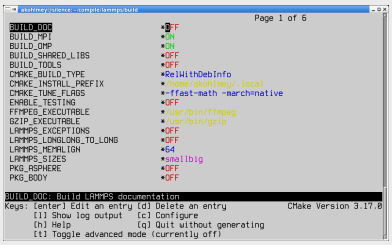



Fig. 8: Options screen of `ccmake`

This will show you the initial screen (left image) with the empty configuration cache. Now you type the 'c' key to run the configuration step. That will do a first configuration run and show the summary (center image). You exit the summary screen with 'e' and see now the main screen with detected options and settings. You can now make changes by moving and down with the arrow keys of the keyboard and modify entries. For on/off settings, the enter key will toggle the state. For others, hitting enter will allow you to modify the value and you commit the change by hitting the enter key again or cancel using the escape key. All "new" settings will be marked with a star '*' and for as long as one setting is marked like this, you have to re-run the configuration by hitting the 'c' key again, sometimes multiple times unless the TUI shows the word "generate" next to the letter 'g' and by hitting the 'g' key the build files will be written to the folder and the TUI exits. You can quit without generating build files by hitting 'q'.

### GUI version

For the graphical CMake program the steps are similar to the TUI version. You can type the command `cmake-gui ../cmake` in the `build` folder. In this case the path to the CMake script folder is not required, it can also be entered from the GUI.

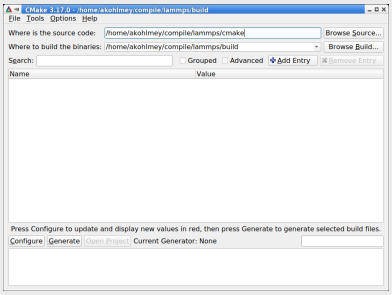


Fig. 9: Initial `cmake-gui` screen

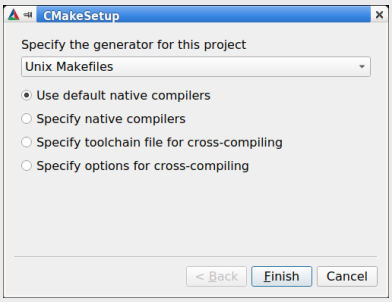



Fig. 10: Generator selection in `cmake-gui`

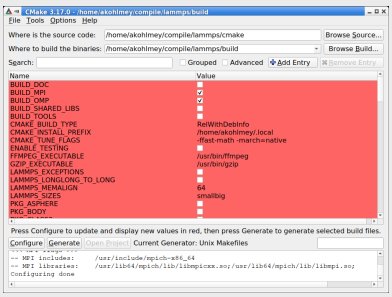


Fig. 11: Options screen of `cmake-gui`

Again, you start with an empty configuration cache (left image) and need to start the configuration step. For the very first configuration in a folder, you will have a pop-up dialog (center image) asking to select the desired build tool and some configuration settings (stick with the default) and then you get the option screen with all new settings highlighted in red. You can modify them (or not) and click on the "configure" button again until satisfied and click on the "generate" button to write out the build files. You can exit the GUI from the "File" menu or hit "ctrl-q".

### Setting options

Options that enable, disable or modify settings are modified by setting the value of CMake variables. This is done on the command-line with the *-D* flag in the format `-D VARIABLE=value`, e.g. `-D CMAKE_BUILD_TYPE=Release` or `-D BUILD_MPI=on`. There is one quirk: when used before the CMake directory, there may be a space between the *-D* flag and the variable, after it must not be. Such CMake variables can have boolean values (on/off, yes/no, or 1/0 are all valid) or are strings representing a choice, or a path, or are free format. If the string would contain whitespace, it must be put in quotes, for example `-D CMAKE_CXX_FLAGS="-O3 -Wall -ftree-vectorize -ffast-math"`.

CMake variables fall into two categories: 1) common CMake variables that are used by default for any CMake configuration setup and 2) project specific variables, i.e. settings that are specific for LAMMPS. Also CMake variables can be flagged as *advanced*, which means they are not shown in the text mode or graphical CMake program in the overview of all settings by default, but only when explicitly requested (by hitting the 't' key or clicking on the 'Advanced' check-box).

### Some common CMake variables

| Variable | Description |
|---|---|
| `CMAKE_INSTALL_PREFIX` | root directory of install location for `make install` (default: `$HOME/.local`) |
| `LAMMPS_INSTALL_RPATH` | set or remove runtime path setting from binaries for `make install` (default: `off`) |
| `CMAKE_BUILD_TYPE` | controls compilation options: one of `RelWithDebInfo` (default), `Release`, `Debug`, `MinSizeRel` |
| `BUILD_SHARED_LIBS` | if set to `on` build the LAMMPS library as shared library (default: `off`) |
| `CMAKE_MAKE_PROGRAM` | name/path of the compilation command (default depends on *-G* option, usually `make`) |
| `CMAKE_VERBOSE_MAKEFI` | if set to `on` echo commands while executing during build (default: `off`) |
| `CMAKE_C_COMPILER` | C compiler to be used for compilation (default: system specific, `gcc` on Linux) |
| `CMAKE_CXX_COMPILER` | C++ compiler to be used for compilation (default: system specific, `g++` on Linux) |
| `CMAKE_Fortran_COMPILI` | Fortran compiler to be used for compilation (default: system specific, `gfortran` on Linux) |
| `CXX_COMPILER_LAUNCHEI` | tool to launch the C++ compiler, e.g. `ccache` or `distcc` for faster compilation (default: empty) |

### Some common LAMMPS specific variables

| Variable | Description |
|---|---|
| BUILD_MPI | build LAMMPS with MPI support (default: on if a working MPI available, else off) |
| BUILD_OMP | build LAMMPS with OpenMP support (default: on if compiler supports OpenMP fully, else off) |
| BUILD_TOOLS | compile some additional executables from the tools folder (default: off) |
| BUILD_DOC | include building the HTML format documentation for packaging/installing (default: off) |
| LAMMPS_MACHIN | when set to name the LAMMPS executable and library will be called lmp_name and liblammps_name.a |
| FFT | select which FFT library to use: FFTW3, MKL, KISS (default, unless FFTW3 is found) |
| FFT_KOKKOS | select which FFT library to use in Kokkos-enabled styles: FFTW3, MKL, HIPFFT, CUFFT, MKL_GPU, KISS (default) |
| FFT_SINGLE | select whether to use single precision FFTs (default: off) |
| WITH_JPEG | whether to support JPEG format in *dump image* (default: on if found) |
| WITH_PNG | whether to support PNG format in *dump image* (default: on if found) |
| WITH_GZIP | whether to support reading and writing compressed files (default: on if found) |
| WITH_FFMPEG | whether to support generating movies with *dump movie* (default: on if found) |

### Enabling or disabling LAMMPS packages

The LAMMPS software is organized into a common core that is always included and a large number of *add-on packages* that have to be enabled to be included into a LAMMPS executable. Packages are enabled through setting variables of the kind PKG_<NAME> to on and disabled by setting them to off (or using yes, no, 1, 0 correspondingly). <NAME> has to be replaced by the name of the package, e.g. MOLECULE or EXTRA-PAIR.

### Using presets

Since LAMMPS has a lot of optional features and packages, specifying them all on the command-line can be tedious. Or when selecting a different compiler tool chain, multiple options have to be changed consistently and that is rather error prone. Or when enabling certain packages, they require consistent settings to be operated in a particular mode. For this purpose, we are providing a selection of "preset files" for CMake in the folder cmake/presets. They represent a way to pre-load or override the CMake configuration cache by setting or changing CMake variables. Preset files are loaded using the *-C* command-line flag. You can combine loading multiple preset files or change some variables later with additional *-D* flags. A few examples:

```
cmake -C ../cmake/presets/basic.cmake -D PKG_MISC=on ../cmake
cmake -C ../cmake/presets/clang.cmake -C ../cmake/presets/most.cmake ../cmake
cmake -C ../cmake/presets/basic.cmake -D BUILD_MPI=off ../cmake
```

The first command will install the packages KSPACE, MANYBODY, MOLECULE, RIGID, GRAPHICS, and MISC; the first five from the preset file and the sixth from the explicit variable definition. The second command will first switch the compiler tool chain to use the Clang compilers and install a large number of packages that are not depending on any special external libraries or tools and are not very unusual. The third command will enable the first four packages like above and then enforce compiling LAMMPS as a serial program (using the MPI STUBS library).

It is also possible to do this incrementally.

```
cmake -C ../cmake/presets/basic.cmake ../cmake
cmake -D PKG_MISC=on .
```

will achieve the same final configuration as in the first example above. In this scenario it is particularly convenient to do the second configuration step using either the text mode or graphical user interface (ccmake or cmake-gui).

> **Note**
>
> Using a preset to select a compiler package (`clang.cmake`, `gcc.cmake`, `intel.cmake`, `oneapi.cmake`, or `pgi.cmake`) are an exception to the mechanism of updating the configuration incrementally, as they will trigger a reset of cached internal CMake settings and thus reset settings to their default values.

### Compilation and build targets

The actual compilation will be started by running the selected build command (on Linux this is by default `make`, see below how to select alternatives). You can also use the portable command `cmake --build .` which will adapt to whatever the selected build command is. This is particularly convenient, if you have set a custom build command via the `CMAKE_MAKE_PROGRAM` variable.

When calling the build program, you can also select which "target" is to be build through appending the `--target` flag and the name of the target to the build command. When using `make` as build tool, you can just append the target name to the command. Example: `cmake --build . --target all` or `make all`. The following abstract targets are available:

| Target | Description |
|---|---|
| `all` | build "everything" (default) |
| `lammps` | build the LAMMPS library and executable |
| `doc` | build the HTML documentation (if configured) |
| `install` | install all target files into folders in `CMAKE_INSTALL_PREFIX` |
| `test` | run some tests (if configured with `-D ENABLE_TESTING=on`) |
| `clean` | remove all generated files |

### Choosing generators

While CMake usually defaults to creating makefiles to compile software with the `make` program, it supports multiple alternate build tools (e.g. `ninja-build` which tends to be faster and more efficient in parallelizing builds than `make`) and can generate project files for integrated development environments (IDEs) like VisualStudio, Eclipse or CodeBlocks. This is specific to how the local CMake version was configured and compiled. The list of available options can be seen at the end of the output of `cmake --help`. Example on Fedora 31 this is:

```
Generators

The following generators are available on this platform (* marks default):
* Unix Makefiles               = Generates standard UNIX makefiles.
  Green Hills MULTI            = Generates Green Hills MULTI files
                                 (experimental, work-in-progress).
  Ninja                        = Generates build.ninja files.
  Ninja Multi-Config           = Generates build-<Config>.ninja files.
  Watcom WMake                 = Generates Watcom WMake makefiles.
  CodeBlocks - Ninja           = Generates CodeBlocks project files.
  CodeBlocks - Unix Makefiles  = Generates CodeBlocks project files.
  CodeLite - Ninja             = Generates CodeLite project files.
  CodeLite - Unix Makefiles    = Generates CodeLite project files.
  Sublime Text 2 - Ninja       = Generates Sublime Text 2 project files.
  Sublime Text 2 - Unix Makefiles
                               = Generates Sublime Text 2 project files.
  Kate - Ninja                 = Generates Kate project files.
  Kate - Unix Makefiles        = Generates Kate project files.
```

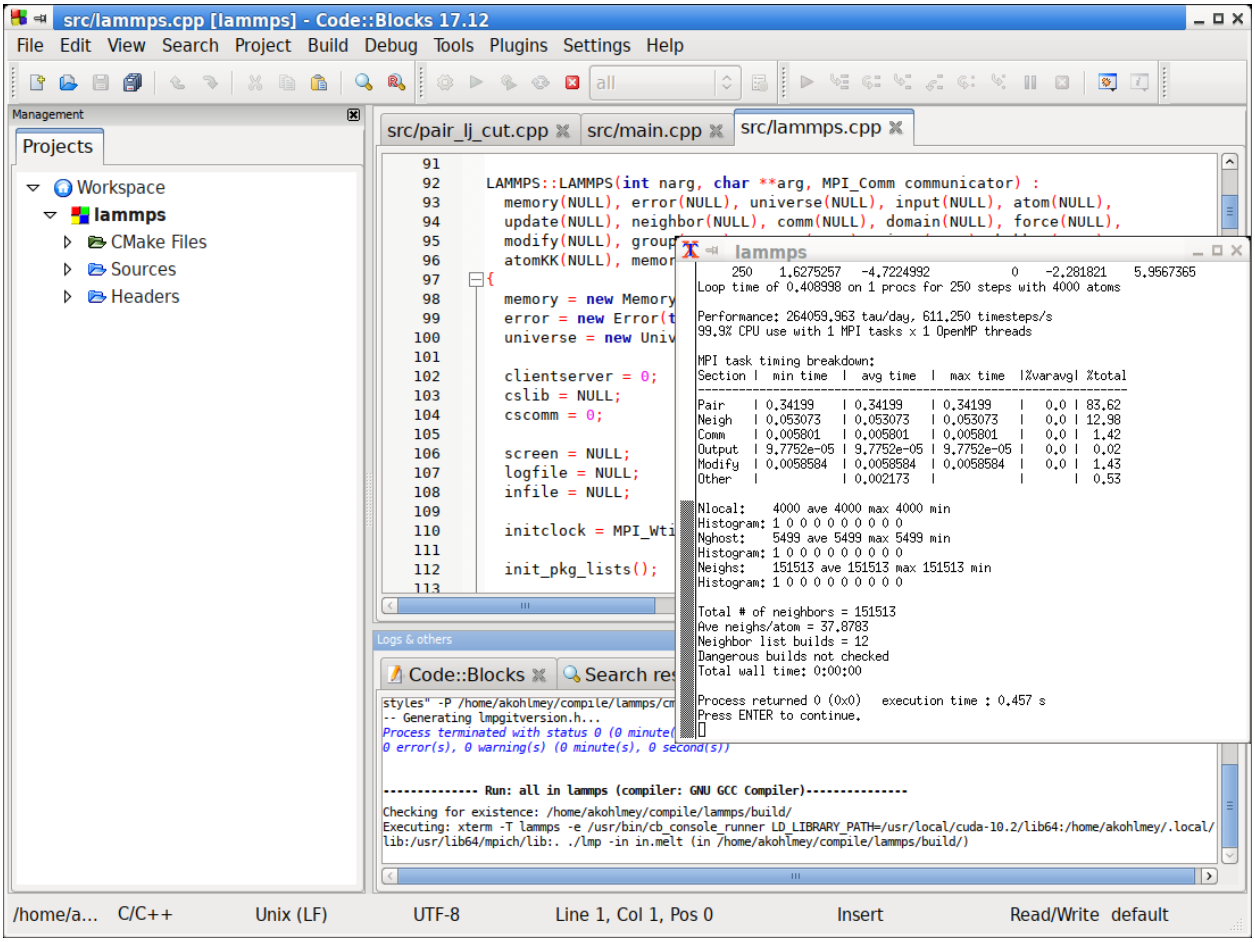
(continues on next page)

(continued from previous page)

```
  Eclipse CDT4 - Ninja         = Generates Eclipse CDT 4.0 project files.
  Eclipse CDT4 - Unix Makefiles= Generates Eclipse CDT 4.0 project files.
```

Below is a screenshot of using the CodeBlocks IDE with the ninja build tool after running CMake as follows:

```
cmake -G 'CodeBlocks - Ninja' ../cmake/presets/most.cmake ../cmake/
```




## 10.6.2 LAMMPS GitHub tutorial

**written by Stefan Paquay**

---

This document describes the process of how to use GitHub to integrate changes or additions you have made to LAMMPS into the official LAMMPS distribution. It uses the process of updating this very tutorial as an example to describe the individual steps and options. You need to be familiar with git and you may want to have a look at the git book to familiarize yourself with some of the more advanced git features used below.

As of fall 2016, submitting contributions to LAMMPS via pull requests on GitHub is the preferred option for integrating contributed features or improvements to LAMMPS, as it significantly reduces the amount of work required by the LAMMPS developers. Consequently, creating a pull request will increase your chances to have your contribution included and will reduce the time until the integration is complete. For more information on the requirements to have your code included into LAMMPS please see *this page*.

---

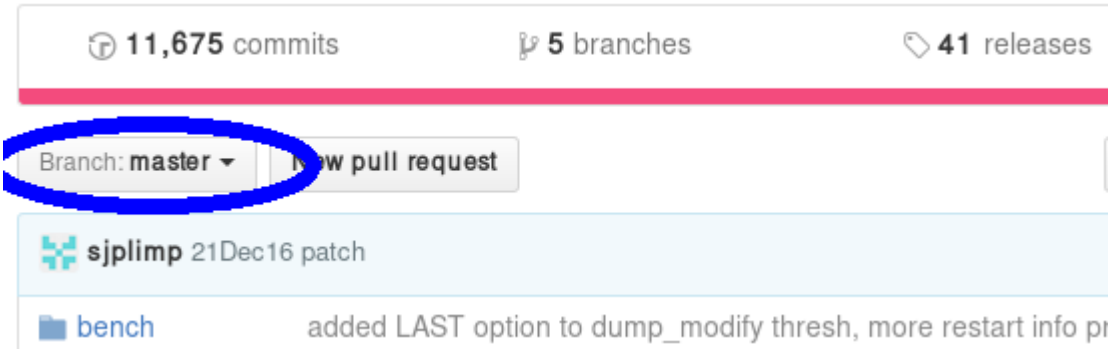
#### Making an account

First of all, you need a GitHub account. This is fairly simple, just go to GitHub and create an account by clicking the "Sign up for GitHub" button. Once your account is created, you can sign in by clicking the button in the top left and filling in your username or e-mail address and password.

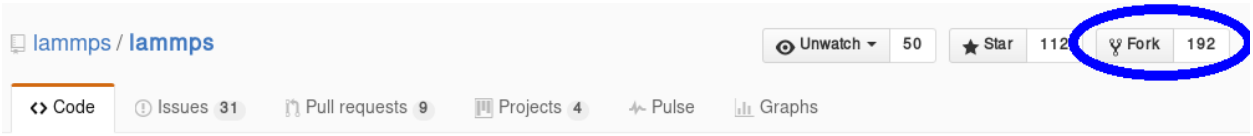
---

#### Forking the repository

To get changes into LAMMPS, you need to first fork the *lammps/lammps* repository on GitHub. At the time of writing, *develop* is the preferred target branch. Thus go to LAMMPS on GitHub and make sure branch is set to "develop", as shown in the figure below.




If it is not, use the button to change it to *develop*. Once it is, use the fork button to create a fork.




This will create a fork (which is essentially a copy, but uses less resources) of the LAMMPS repository under your own GitHub account. You can make changes in this fork and later file *pull requests* to allow the upstream repository to merge changes from your own fork into the one we just forked from (or others that were forked from the same repository). At the same time, you can set things up, so you can include changes from upstream into your repository and thus keep it in sync with the ongoing LAMMPS development.

---

#### Adding changes to your own fork

Additions to the upstream version of LAMMPS are handled using *feature branches*. For every new feature, a so-called feature branch is created, which contains only those modification relevant to one specific feature. For example, adding a single fix would consist of creating a branch with only the fix header and source file and nothing else. It is explained in more detail here: feature branch workflow.

#### Feature branches

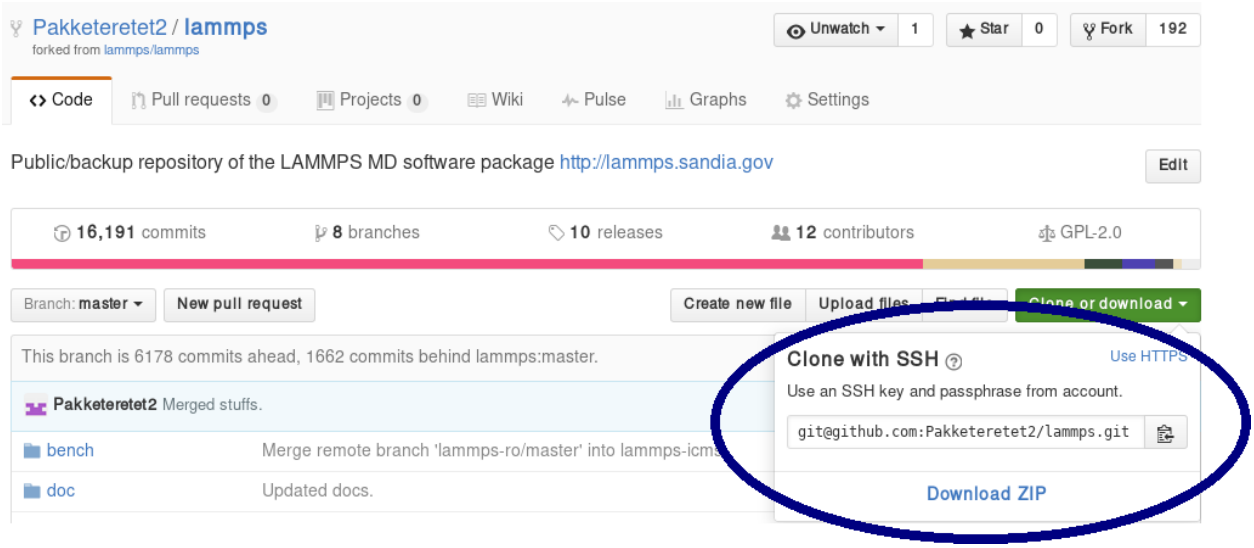
First of all, create a clone of your version on GitHub on your local machine via HTTPS:

```
git clone https://github.com/<your user name>/lammps.git <some name>
```

or, if you have set up your GitHub account for using SSH keys, via SSH:

```
git clone git@github.com:<your user name>/lammps.git
```

You can find the proper URL by clicking the "Clone or download"-button:




The above command copies ("clones") the git repository to your local machine to a directory with the name you chose. If none is given, it will default to "lammps". Typical names are "mylammps" or something similar.

You can use this local clone to make changes and test them without interfering with the repository on GitHub.

To pull changes from upstream into this copy, you can go to the directory and use git pull:

```
cd mylammps
git checkout develop
git pull https://github.com/lammps/lammps develop
```

You can also add this URL as a remote:

```
git remote add upstream https://www.github.com/lammps/lammps
```

From then on you can update your upstream branches with:

```
git fetch upstream
```

and then refer to the upstream repository branches with *upstream/develop* or *upstream/release* and so on.

At this point, you typically make a feature branch from the updated branch for the feature you want to work on. This tutorial contains the workflow that updated this tutorial, and hence we will call the branch "github-tutorial-update":

```
git fetch upstream
git checkout -b github-tutorial-update upstream/develop
```

Now that we have changed branches, we can make our changes to our local repository. Just remember that if you want to start working on another, unrelated feature, you should switch branches!

**Note**

Committing changes to the *develop*, *release*, or *stable* branches is strongly discouraged. While it may be convenient initially, it will create more work in the long run. Various texts and tutorials on using git effectively discuss the motivation for using feature branches instead.

#### After changes are made

After everything is done, add the files to the branch and commit them:

```
git add doc/src/Howto_github.txt
git add doc/src/JPG/tutorial*.png
```

> **Warning**
>
> Do not use *git commit -a* (or *git add -A*). The -a flag (or -A flag) will automatically include **all** modified **and** new files and that is rarely the behavior you want. It can easily lead to accidentally adding unrelated and unwanted changes into the repository. Instead it is preferable to explicitly use *git add*, *git rm*, *git mv* for adding, removing, renaming individual files, respectively, and then *git commit* to finalize the commit. Carefully check all pending changes with *git status* before committing them. If you find doing this on the command-line too tedious, consider using a GUI, for example the one included in git distributions written in Tk, i.e. use *git gui* (on some Linux distributions it may be required to install an additional package to use it).

After adding all files, the change set can be committed with some useful message that explains the change.

```
git commit -m 'Finally updated the GitHub tutorial'
```

After the commit, the changes can be pushed to the same branch on GitHub:

```
git push
```

Git will ask you for your user name and password on GitHub if you have not configured anything. If your local branch is not present on GitHub yet, it will ask you to add it by running

```
git push --set-upstream origin github-tutorial-update
```

If you correctly type your user name and password, the feature branch should be added to your fork on GitHub.

If you want to make really sure you push to the right repository (which is good practice), you can provide it explicitly:

```
git push origin
```

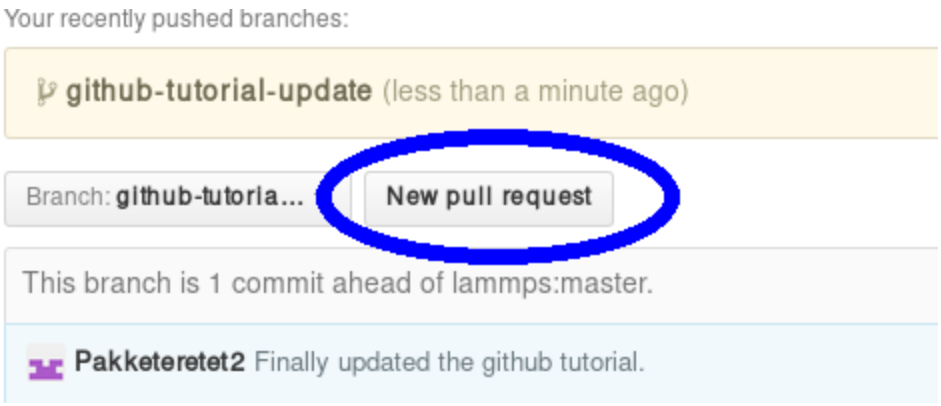
or using an explicit URL:

```
git push git@github.com:Pakketeretet2/lammps.git
```

---

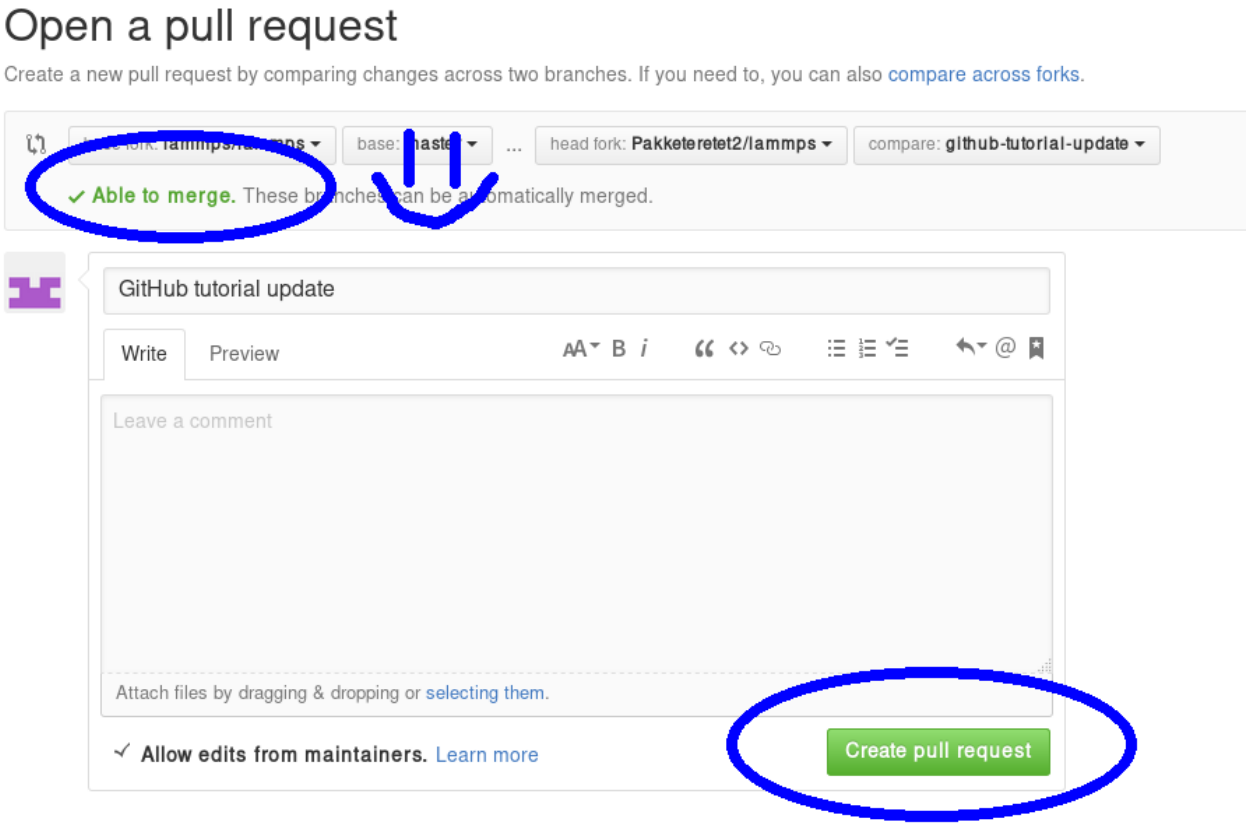
#### Filing a pull request

Up to this point in the tutorial, all changes were to *your* clones of LAMMPS. Eventually, however, you want this feature to be included into the official LAMMPS version. To do this, you will want to file a pull request by clicking on the "New pull request" button:



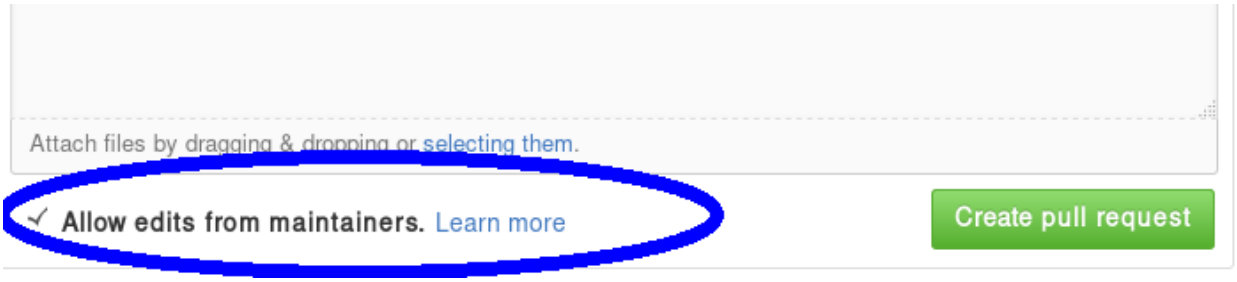

Make sure that the current branch is set to the correct one, which, in this case, is "github-tutorial-update". If done correctly, the only changes you will see are those that were made on this branch.

This will open up a new window that lists changes made to the repository. If you are just adding new files, there is not much to do, but I suppose merge conflicts are to be resolved here if there are changes in existing files. If all changes can automatically be merged, green text at the top will say so and you can click the "Create pull request" button, see image.



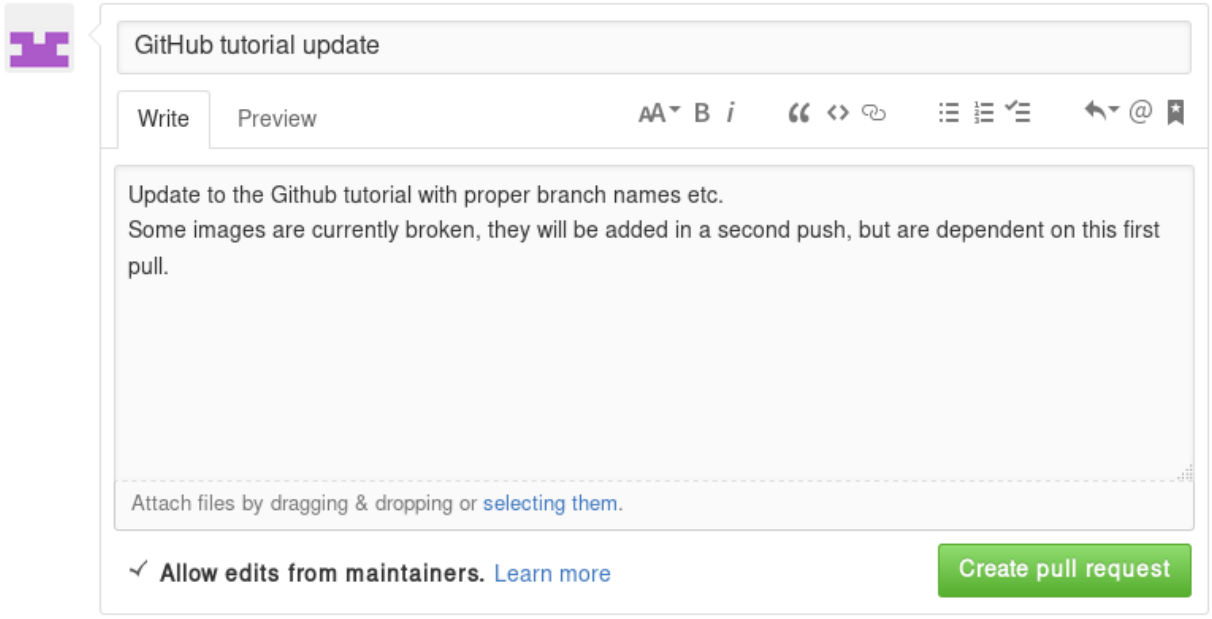

Before creating the pull request, make sure the short title is accurate and add a comment with details about your pull request. Here you write what your modifications do and why they should be incorporated upstream.

Note the checkbox that says "Allow edits from maintainers". This is checked by default checkbox (although in my version of Firefox, only the checkmark is visible):




If it is checked, maintainers can immediately add their own edits to the pull request. This helps the inclusion of your branch significantly, as simple/trivial changes can be added directly to your pull request branch by the LAMMPS maintainers. The alternative would be that they make changes on their own version of the branch and file a reverse pull request to you. Just leave this box checked unless you have a very good reason not to.

Now just write some nice comments and click on "Create pull request".



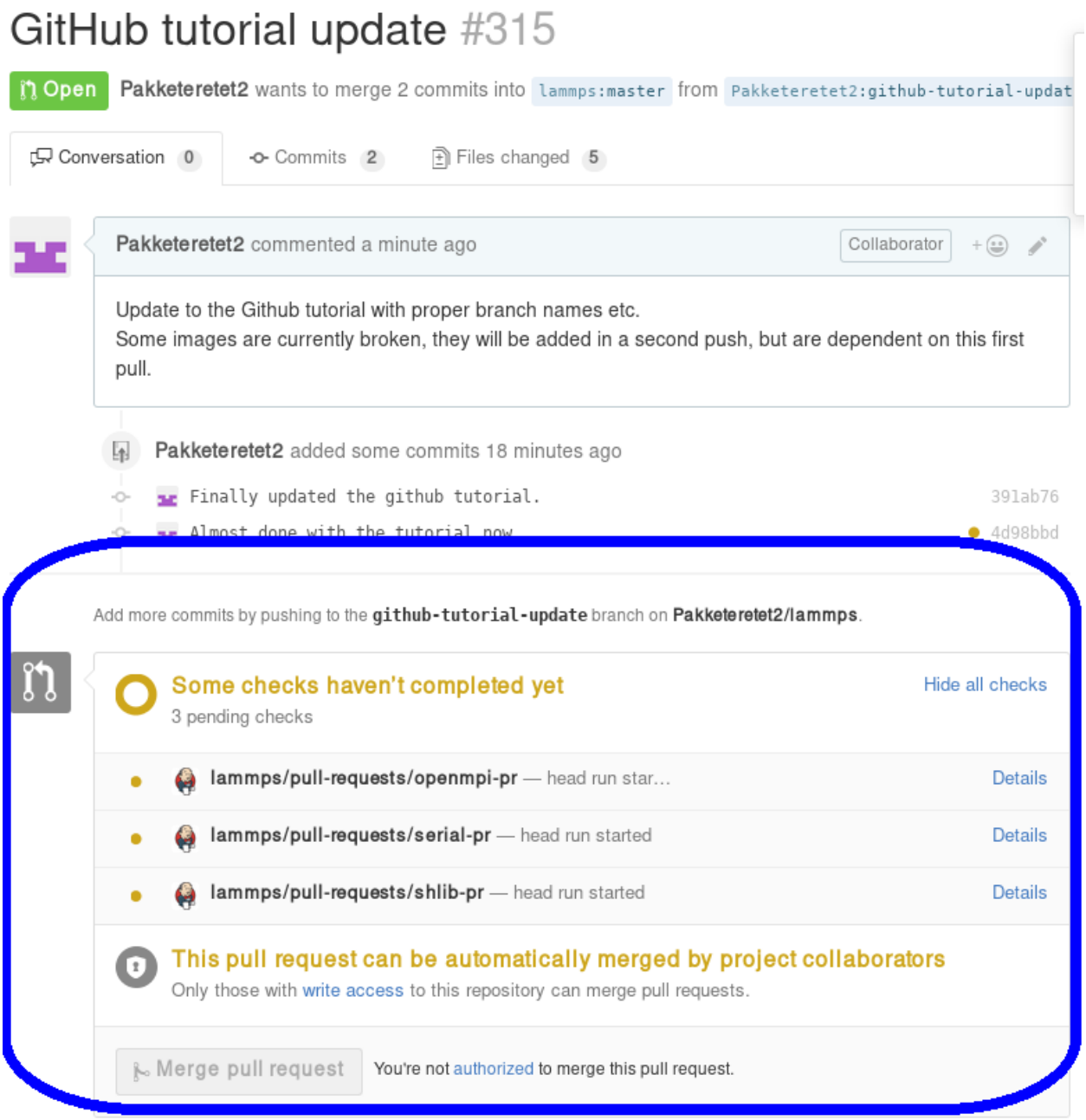

---

**After filing a pull request**

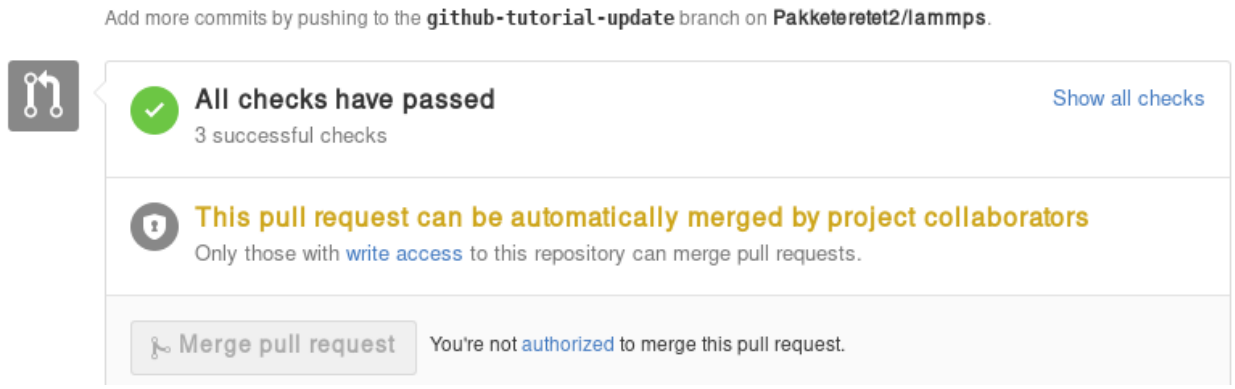
**Note**

When you submit a pull request (or ask for a pull request) for the first time, you will receive an invitation to become a LAMMPS project collaborator. Please accept this invite as being a collaborator will simplify certain administrative tasks and will probably speed up the merging of your feature, too.

You will notice that after filing the pull request, some checks are performed automatically:




If all is fine, you will see this:

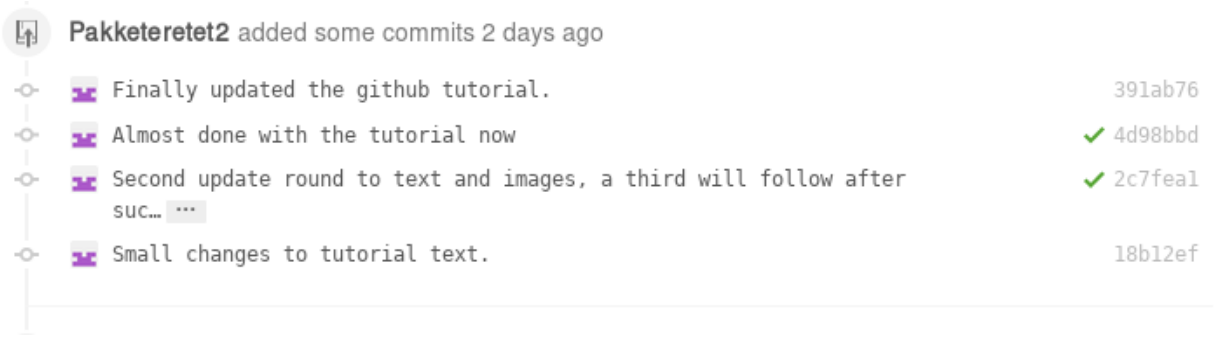



If any of the checks are failing, your pull request will not be processed, as your changes may break compilation for certain configurations or may not merge cleanly. It is your responsibility to remove the reason(s) for the failed test(s). If you need help with this, please contact the LAMMPS developers by adding a comment explaining your problems with resolving the failed tests.

A few further interesting things (can) happen to pull requests before they are included.

**Additional changes**

First of all, any additional changes you push into your branch in your repository will automatically become part of the pull request:




This means you can add changes that should be part of the feature after filing the pull request, which is useful in case you have forgotten them, or if a developer has requested that something needs to be changed before the feature can be accepted into the official LAMMPS version. After each push, the automated checks are run again.

**Labels**

LAMMPS developers may add labels to your pull request to assign it to categories (mostly for bookkeeping purposes), but a few of them are important: *needs_work*, *work_in_progress*, *run_tests*, *test_for_regression*, and *ready_for_merge*. The first two indicate, that your pull request is not considered to be complete. With "needs_work" the burden is on exclusively on you; while "work_in_progress" can also mean, that a LAMMPS developer may want to add changes. Please watch the comments to the pull requests. The two "test" labels are used to trigger extended tests before the code is merged. This is sometimes done by LAMMPS developers, if they suspect that there may be some subtle side effects from your changes. It is not done by default, because those tests are very time-consuming. The *ready_for_merge* label is usually attached when the LAMMPS developer assigned to the pull request considers this request complete and to trigger a final full test evaluation.

**Reviews**

As of Fall 2021, a pull request needs to pass all automatic tests and at least 1 approving review from a LAMMPS developer with write access to the repository before it is eligible for merging. In case your changes touch code that certain developers are associated with, they are auto-requested by the GitHub software. Those associations are set in the file .github/CODEOWNERS Thus if you want to be automatically notified to review when anybody changes files or

packages, that **you** have contributed to LAMMPS, you can add suitable patterns to that file, or a LAMMPS developer may add you.

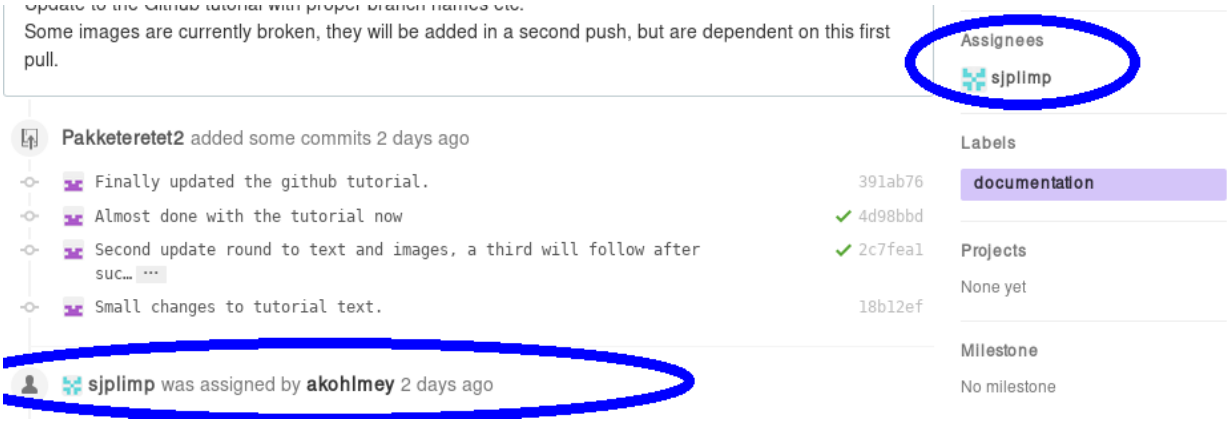
Otherwise, you can also manually request reviews from specific developers, or LAMMPS developers - in their assessment of your pull request - may determine who else should be reviewing your contribution and add that person. Through reviews, LAMMPS developers also may request specific changes from you. If those are not addressed, your pull requests cannot be merged.

**Assignees**

There is an assignee property for pull requests. If the request has not been reviewed by any developer yet, it is not assigned to anyone. After revision, a developer can choose to assign it to either a) you, b) a LAMMPS developer (including him/herself) or c) Axel Kohlmeyer (akohlmey).

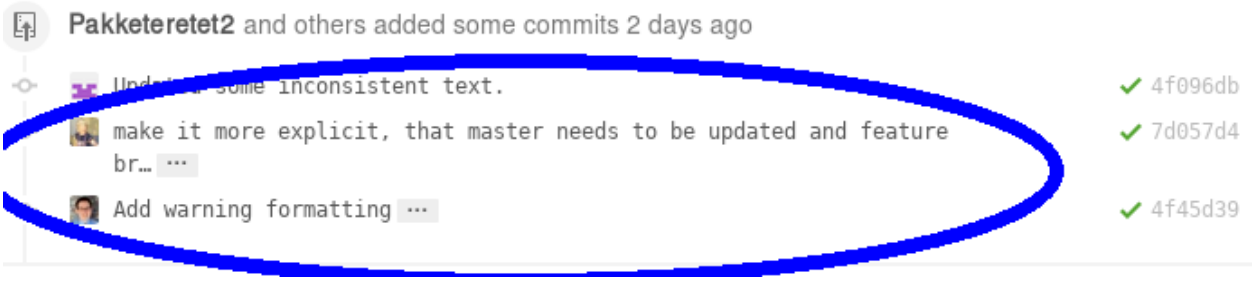
- Case a) happens if changes are required on your part
- Case b) means that at the moment, it is being tested and reviewed by a LAMMPS developer with the expectation that some changes would be required. After the review, the developer can choose to implement changes directly or suggest them to you.
- Case c) means that the pull request has been assigned to the developer overseeing the merging of pull requests into the *develop* branch.

In this case, Axel assigned the tutorial to Steve:




**Edits from LAMMPS maintainers**

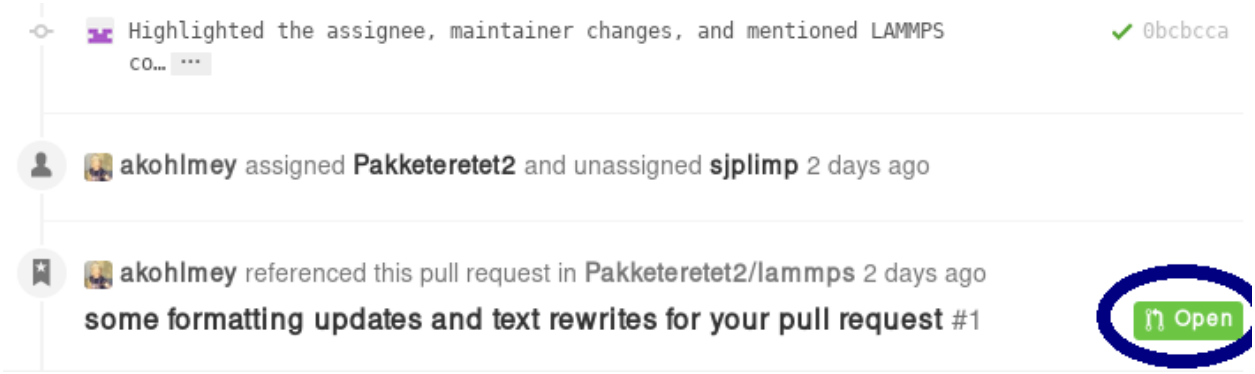
If you allowed edits from maintainers (the default), any LAMMPS maintainer can add changes to your pull request. In this case, both Axel and Richard made changes to the tutorial:



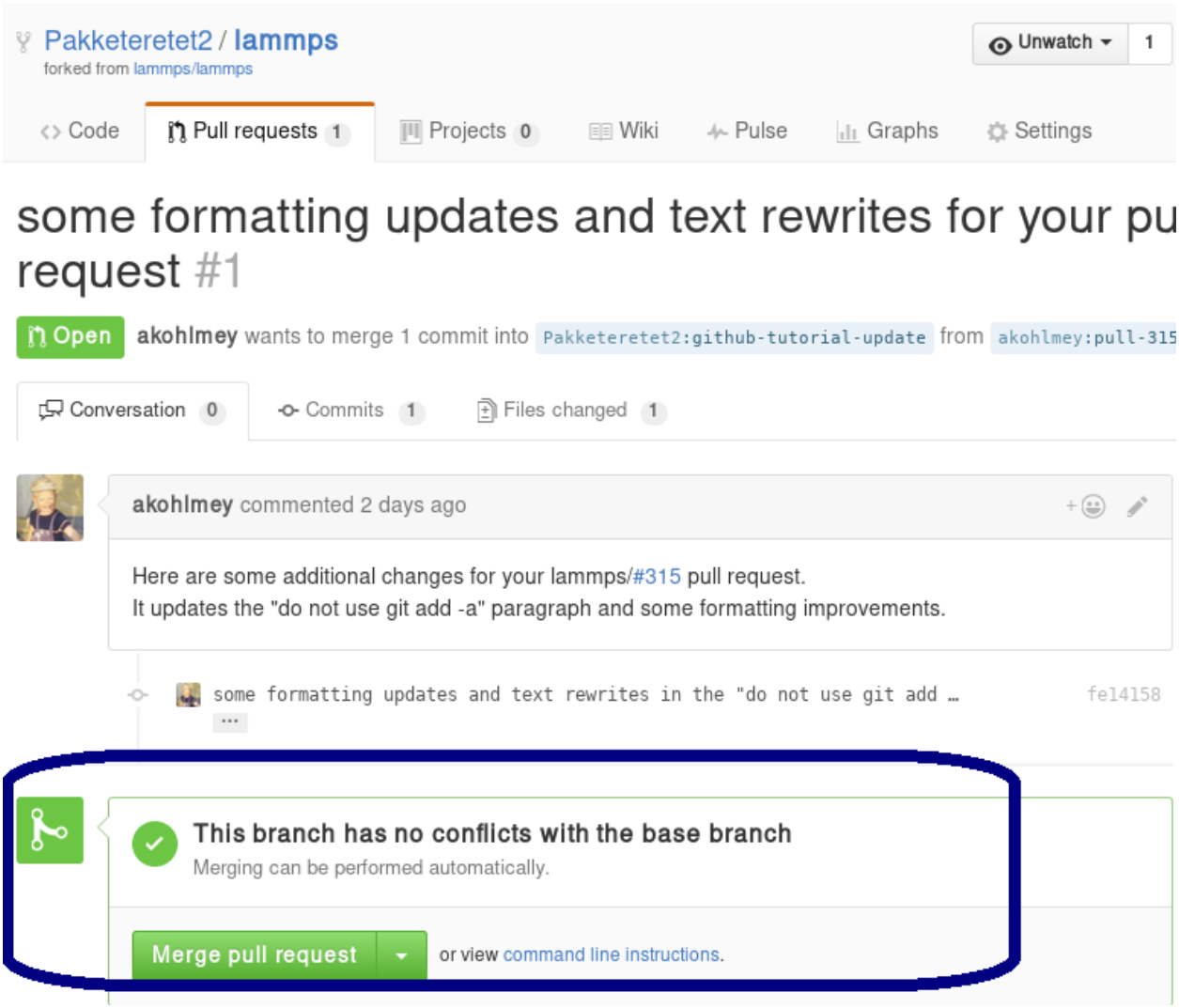

**Reverse pull requests**

Sometimes, however, you might not feel comfortable having other people push changes into your own branch, or maybe the maintainers are not sure their idea was the right one. In such a case, they can make changes, reassign you as the assignee, and file a "reverse pull request", i.e. file a pull request in **your** forked GitHub repository to include changes in the branch, that you have submitted as a pull request yourself. In that case, you can choose to merge their changes

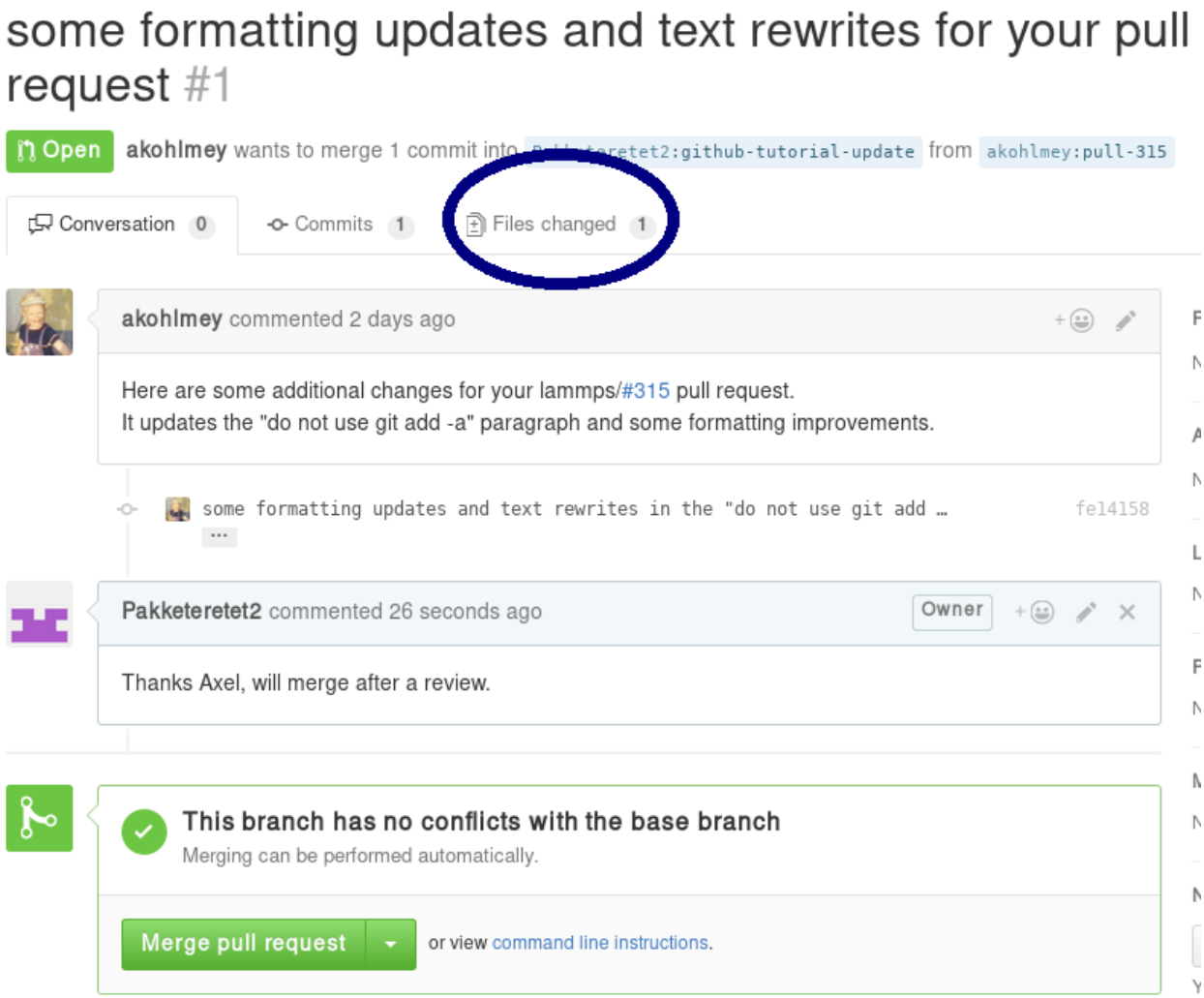
back into your branch, possibly make additional changes or corrections and proceed from there. It looks something like this:



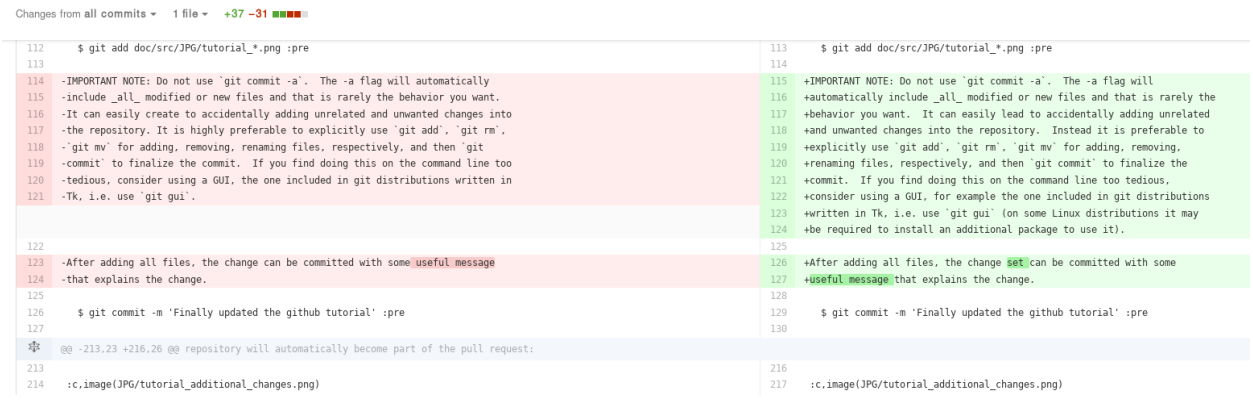

For some reason, the highlighted button did not work in my case, but I can go to my own repository and merge the pull request from there:



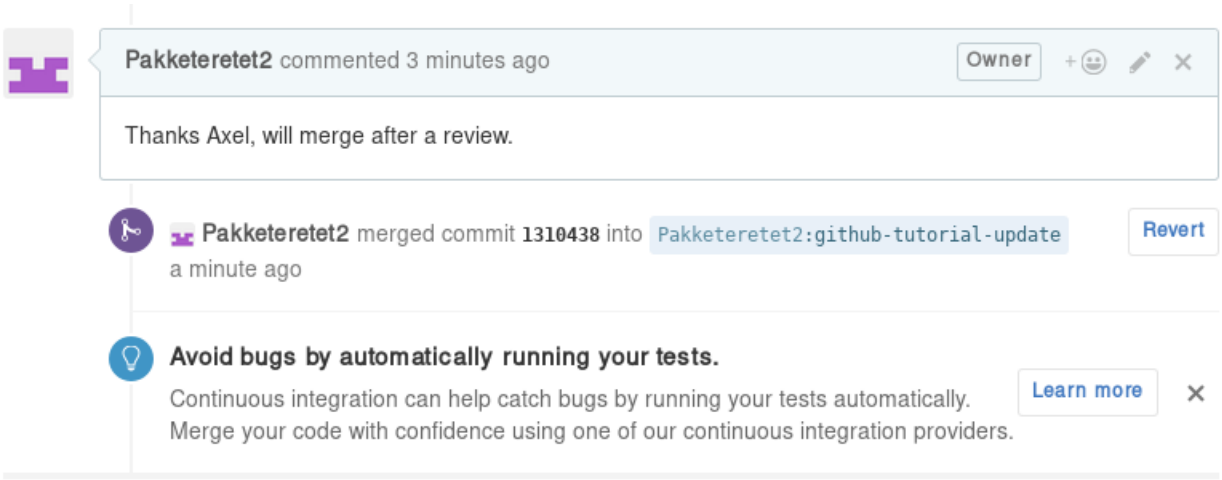

Be sure to check the changes to see if you agree with them by clicking on the tab button:




In this case, most of it is changes in the markup and a short rewrite of Axel's explanation of the "git gui" and "git add" commands.




Because the changes are OK with us, we are going to merge by clicking on "Merge pull request". After a merge it looks like this:



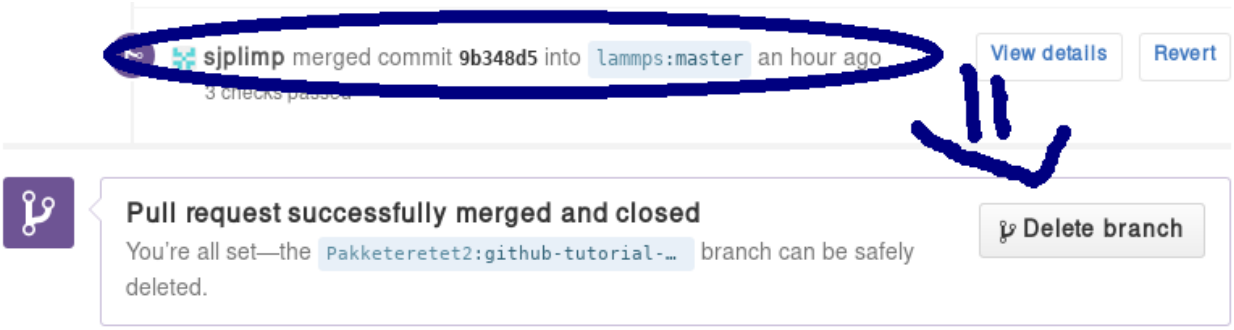

Now, since in the meantime our local text for the tutorial also changed, we need to pull Axel's change back into our branch, and merge them:

```
git add Howto_github.txt
git add JPG/tutorial_reverse_pull_request*.png
git commit -m "Updated text and images on reverse pull requests"
git pull
```

In this case, the merge was painless because git could auto-merge:

```
[ stefan@phys8250 src ] % git pull
Auto-merging doc/src/tutorial_github.txt
Waiting for Emacs...
Merge made by the 'recursive' strategy.
 doc/src/tutorial_github.txt | 68 ++++++
```

When everything is fine, the feature branch is merged into the *develop* branch:




Now one question remains: What to do with the feature branch that got merged into upstream?

It is in principle safe to delete them from your own fork. This helps keep it a bit more tidy. Note that you first have to switch to another branch!

```
git checkout develop
git pull https://github.com/lammps/lammps develop
git branch -d github-tutorial-update
```

If you do not pull first, it is not really a problem but git will warn you at the next statement that you are deleting a local branch that was not yet fully merged into HEAD. This is because git does not yet know your branch just got merged into LAMMPS upstream. If you first delete and then pull, everything should still be fine. You can display all branches that are fully merged by:

Finally, if you delete the branch locally, you might want to push this to your remote(s) as well:

```
git push origin :github-tutorial-update
```

**Recent changes in the workflow**

Some recent changes to the workflow are not captured in this tutorial. For example, in addition to the *develop* branch, to which all new features should be submitted, there is also a *release*, a *stable*, and a *maintenance* branch; the *release* branch is updated from the *develop* branch as part of a "feature release", and *stable* (together with *release*) are updated from *develop* when a "stable release" is made. In between stable releases, selected bug fixes and infrastructure updates are back-ported from the *develop* branch to the *maintenance* branch and occasionally merged to *stable* as an update release.

Furthermore, the naming of the release tags now follow the pattern "patch_<Day><Month><Year>" to simplify comparisons between releases. For stable releases additional "stable_<Day><Month><Year>" tags are applied and update releases are tagged with "stable_<Day><Month><Year>_update<Number>", Finally, all releases and submissions are subject to automatic testing and code checks to make sure they compile with a variety of compilers and popular operating systems. Some unit and regression testing is applied as well.

A detailed discussion of the LAMMPS developer GitHub workflow can be found in the file doc/github-development-workflow.md

### 10.6.3 Using LAMMPS-GUI

Changed in version 10Sep2025.

This page used to contain the documentation for LAMMPS-GUI. LAMMPS-GUI, however, is no longer bundled with LAMMPS and its documentation has moved to https://lammps-gui.lammps.org/

### 10.6.4 Moltemplate Tutorial

In this tutorial, we are going to use the tool *Moltemplate* from https://moltemplate.org/ to set up a classical molecular dynamic simulation using the *OPLS-AA force field*. The first task is to describe an organic compound and create a complete input deck for LAMMPS. The second task is to use moltemplate to build a polymer. The third task is to map the OPLS-AA force field to a molecular sample created with an external tool, e.g. PACKMOL, and exported as a PDB file. The files used in this tutorial can be found in the `tools/moltemplate/tutorial-files` folder of the LAMMPS source code distribution.

Many more examples can be found here: https://moltemplate.org/examples.html

#### Simulating an organic solvent

This example aims to create a cubic box of the organic solvent formamide.

The first step is to create a molecular topology in the LAMMPS-template (LT) file format representing a single molecule, which will be stored in a Moltemplate object called `_FAM inherits OPLSAA {}`. This command states that the object `_FAM` is based on an existing object called `OPLSAA`, which contains OPLS-AA parameters, atom type definitions, partial charges, masses and bond-angle rules for many organic and biological compounds. The atomic structure is the starting point to populate the command `write('Data Atoms') {}`, which will write the `Atoms` section in the LAMMPS data file. The OPLS-AA force field uses the `atom_style full`, therefore, this column format is used: `# atomID molID atomType charge coordX coordY coordZ`. The `atomID`s are replaced with Moltemplate `$`-type variables, which are then substituted with unique numerical IDs. The same logic is applied to the `molID`, except that the same variable is used for the whole molecule. The atom types are assigned using `@`-type variables. The assignment of atom types (e.g. `@atom:177`, `@atom:178`) is done using the OPLS-AA atom types defined in the "In Charges" section of the file `oplsaa2024.lt`, looking for a reasonable match with the description of the atom. The resulting file (`formamide.lt`) follows:

```
import /usr/local/moltemplate/moltemplate/force_fields/oplsaa2024.lt  # defines OPLSAA

_FAM inherits OPLSAA {
```

(continues on next page)

(continued from previous page)

```
  # atomID    molID  atomType  charge  coordX  coordY  coordZ
  write('Data Atoms') {
    $atom:C00  $mol  @atom:235  0.00    0.100   0.490   0.0
    $atom:O01  $mol  @atom:236  0.00    1.091  -0.250   0.0
    $atom:N02  $mol  @atom:237  0.00   -1.121  -0.181   0.0
    $atom:H03  $mol  @atom:240  0.00   -2.013   0.272   0.0
    $atom:H04  $mol  @atom:240  0.00   -1.056  -1.190   0.0
    $atom:H05  $mol  @atom:279  0.00    0.144   1.570   0.0
  }

  # A list of the bonds in the molecule:
  # BondID   AtomID1   AtomID2
  write('Data Bond List') {
    $bond:C1 $atom:C00 $atom:O01
    $bond:C2 $atom:C00 $atom:H05
    $bond:C3 $atom:C00 $atom:N02
    $bond:C4 $atom:N02 $atom:H03
    $bond:C5 $atom:N02 $atom:H04
  }
}
```

You don't have to specify the charge in this example because the OPLSAA force-field assigns charge according to the atom type. (This is not true when using other force fields.) A "Data Bond List" section is needed as the atom type will determine the bond type. The other bonded interactions (e.g. angles, dihedrals, and impropers) will be automatically generated by Moltemplate.

If the simulation is not charge-neutral, or Moltemplate complains that you have missing bond, angle, or dihedral types, this probably means that at least one of your atom types is incorrect (or that perhaps there is no suitable atom type currently defined in the `oplsaa2024.lt` file).

The second step is to create a master file with instructions to build a starting structure and the LAMMPS commands to run an NPT simulation. The master file (`solv_01.lt`) follows:

```
import formamide.lt  # Defines "_FAM" and OPLSAA

# Distribute the molecules on a 5x5x5 cubic grid with spacing 4.6
solv = new _FAM [5].move( 4.6, 0, 0)
                [5].move( 0, 4.6, 0)
                [5].move( 0, 0, 4.6)
solv[*][*][*].move(-11.5, -11.5, -11.5)

# Set the simulation box.
write_once("Data Boundary") {
 -11.5 11.5 xlo xhi
 -11.5 11.5 ylo yhi
 -11.5 11.5 zlo zhi
}

# Note: The lines below in the "In Run" section are often omitted.

write_once("In Run"){
 # Create an input deck for LAMMPS.
```

(continues on next page)

(continued from previous page)

```
 # Run an NPT simulation.
 # Input variables.
 variable run    string solv_01   # output name
 variable ts     equal  1         # timestep
 variable temp   equal  300       # equilibrium temperature
 variable p      equal  1.        # equilibrium pressure
 variable d      equal  1000      # output frequency
 variable equi   equal  5000      # Equilibration steps
 variable prod   equal  30000     # Production steps

 # Derived variables.
 variable tcouple equal \$\{ts\}*100
 variable pcouple equal \$\{ts\}*1000

 # Output.
 thermo          \$d
 thermo_style custom step etotal evdwl ecoul elong ebond eangle &
 edihed eimp ke pe temp press vol density cpu
 thermo_modify flush yes

 # Trajectory.
 dump TRJ all dcd \$d \$\{run\}.dcd
 dump_modify TRJ unwrap yes

 # Thermalisation and relaxation, NPT ensemble.
 timestep       \$\{ts\}
 fix             NPT all npt temp \$\{temp\} \$\{temp\} \$\{tcouple\} iso \$p \$p \$\
↪{pcouple\}
 velocity all create \$\{temp\} 858096 dist gaussian
 # Short runs to update the PPPM settings as the box shrinks.
 run    \$\{equi\} post no
 run    \$\{equi\} post no
 run    \$\{equi\} post no
 run    \$\{equi\}
 # From now on, the density shouldn't change too much.
 run    \$\{prod\}
 unfix NPT
}
```

The first two commands insert the content of files `oplsaa2024.lt` and `formamide.lt` into the master file. At this point, we can use the command `solv = new _FAM [N]` to create N copies of a molecule of type `_FAM`. In this case, we create an array of 5*5*5 molecules on a cubic grid using the coordinate transformation command `.move( 4.6, 0, 0)`. See the Moltemplate documentation to learn more about the syntax. As the sample was created from scratch, we also specify the simulation box size in the "Data Boundary" section.

The LAMMPS setting for the force field are specified in the file `oplsaa2024.lt` and are written automatically in the input deck. We also specify the boundary conditions and a set of variables in the "In Init" section.

The remaining commands to run an NPT simulation are written in the "In Run" section. Note that in this script, LAMMPS variables are protected with the escape character \ to distinguish them from Moltemplate variables, e.g. `\$\{run\}` is a LAMMPS variable that is written in the input deck as `${run}`.

(Note: Moltemplate can be slow to run, so you need to change you run settings frequently, I recommended moving those commands (from "In Run") out of your .lt files and into a separate file. Moltemplate creates a file named `run.`

`in.EXAMPLE` for this purpose. You can put your run settings and fixes that file and then invoke LAMMPS using `mpirun -np 4 lmp -in run.in.EXAMPLE` instead.)

Compile the master file with:

```
moltemplate.sh solv_01.lt
cleanup_moltemplate.sh   # <-- optional: see below
```

(Note: The optional "cleanup_moltemplate.sh" command deletes unused atom types, which sometimes makes LAMMPS run faster. But it does not work with many-body pair styles or dreiding-style h-bonds. Fortunately most force fields, including OPLSAA, don't use those features.)

Then execute the simulation with the following:

```
mpirun -np 4 lmp -in solv_01.in -l solv_01.log
```

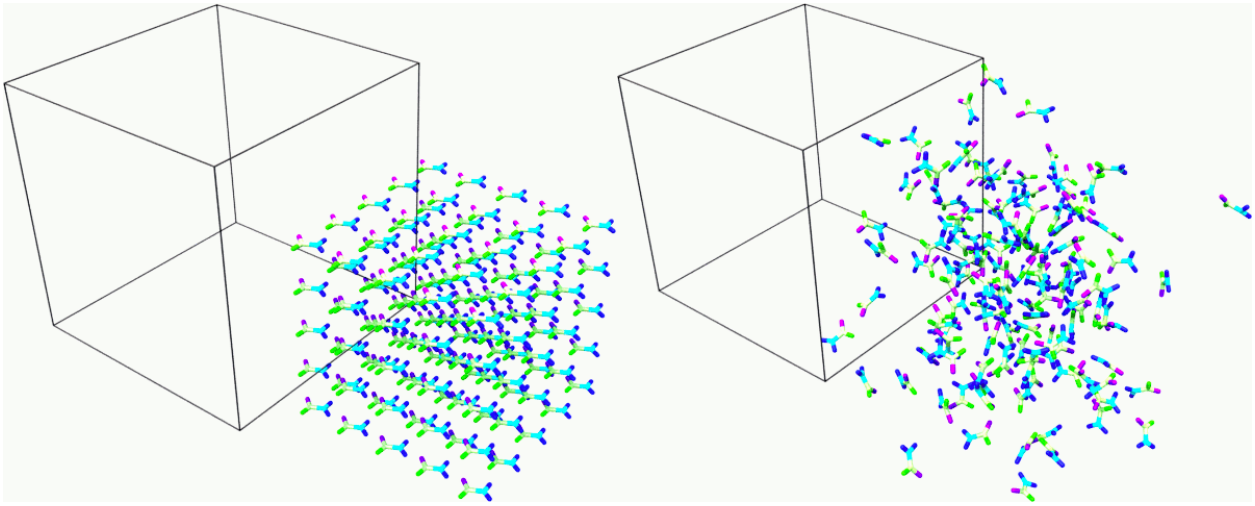


Fig. 12: Snapshot of the sample at the beginning and end of the simulation. Rendered with Ovito.

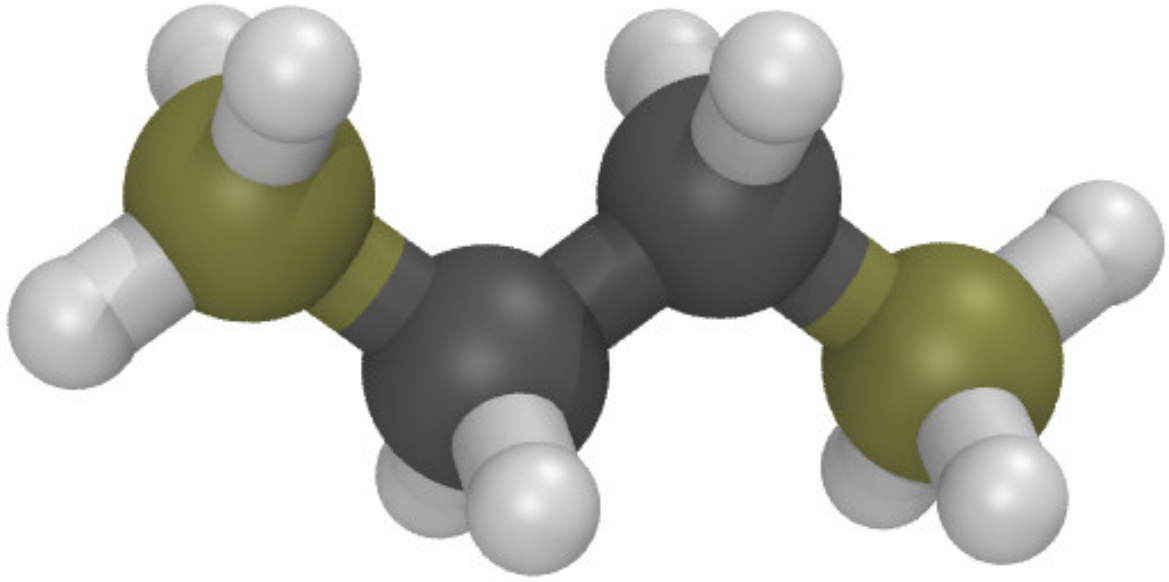
### Building a simple polymer

Moltemplate is particularly useful for building polymers (and other molecules with sub-units). As an simple example, consider butane:

The `butane.lt` file below defines Butane as a polymer containing 4 monomers (of type `CH3`, `CH2`, `CH2`, `CH3`).

```
import /usr/local/moltemplate/moltemplate/force_fields/oplsaa2024.lt  # defines OPLSAA

CH3 inherits OPLSAA {

  # atomID    molID   atomType  charge   coordX   coordY    coordZ
  write("Data Atoms") {
    $atom:c  $mol:... @atom:54   0.0   0.000000  0.4431163  0.000000
    $atom:h1 $mol:... @atom:60   0.0   0.000000  1.0741603  0.892431
    $atom:h2 $mol:... @atom:60   0.0   0.000000  1.0741603 -0.892431
    $atom:h3 $mol:... @atom:60   0.0  -0.892431 -0.1879277  0.000000
  }
  # (Using "$mol:..." indicates this object ("CH3") is part of a larger
  #  molecule. Moltemplate will share the molecule-ID with that molecule.)
```

(continues on next page)



(continued from previous page)

```
  # A list of the bonds within the "CH3" molecular sub-unit:
  # BondID   AtomID1   AtomID2
  write('Data Bond List') {
    $bond:ch1 $atom:c $atom:h1
    $bond:ch2 $atom:c $atom:h2
    $bond:ch3 $atom:c $atom:h3
  }
}

CH2 inherits OPLSAA {

  # atomID    molID   atomType  charge   coordX   coordY    coordZ
  write("Data Atoms") {
    $atom:c  $mol:... @atom:57   0.0   0.000000  0.4431163  0.000000
    $atom:h1 $mol:... @atom:60   0.0   0.000000  1.0741603  0.892431
    $atom:h2 $mol:... @atom:60   0.0   0.000000  1.0741603 -0.892431
  }

  # A list of the bonds within the "CH2" molecular sub-unit:
  # BondID   AtomID1   AtomID2
  write('Data Bond List') {
    $bond:ch1 $atom:c $atom:h1
    $bond:ch2 $atom:c $atom:h2
  }
}

Butane inherits OPLSAA {
```

(continues on next page)

(continued from previous page)

```
  create_var {$mol}  # optional:force all monomers to share the same molecule-ID

  # - Create 4 monomers
  # - Move them along the X axis using ".move()",
  # - Rotate them 180 degrees with respect to the previous monomer
  monomer1 = new CH3
  monomer2 = new CH2.rot(180,1,0,0).move(1.2533223,0,0)
  monomer3 = new CH2.move(2.5066446,0,0)
  monomer4 = new CH3.rot(180,0,0,1).move(3.7599669,0,0)

  # A list of the bonds connecting different monomers together:
  write('Data Bond List') {
    $bond:b1  $atom:monomer1/c $atom:monomer2/c
    $bond:b2  $atom:monomer2/c $atom:monomer3/c
    $bond:b3  $atom:monomer3/c $atom:monomer4/c
  }
}
```

Again, you don't have to specify the charge in this example because OPLSAA assigns charges according to the atom type.

This `Butane` object is a molecule which can be used anywhere other molecules can be used. (You can arrange `Butane` molecules on a lattice, as we did previously. You can also modify individual butane molecules by adding or deleting atoms or bonds. You can add bonds between specific butane molecules or use `Butane` as a sub-unit to define even larger molecules. See the moltemplate manual for details.)

### How to build a complex polymer

A similar procedure can be used to create more complicated polymers, such as the NIPAM polymer example shown below. For details, see:

https://github.com/jewettaij/moltemplate/tree/master/examples/all_atom/force_field_OPLSAA/NIPAM_polymer+water+ions

### Mapping an existing structure

Another helpful way to use Moltemplate is mapping an existing molecular sample to a force field. This is useful when a complex sample is assembled from different simulations or created with specialized software (e.g. PACKMOL). (Note: The previous link shows how to build this entire system from scratch using only moltemplate. However here we will assume instead that we obtained a PDB file for this system using PACKMOL.)

As in the previous examples, all molecular species in the sample are defined using single-molecule Moltemplate objects. For this example, we use a short polymer in a box containing water molecules and ions in the PDB file `model.pdb`.

It is essential to understand that the order of atoms in the PDB file and in the Moltemplate master script must match, as we are using the coordinates from the PDB file in the order they appear. The order of atoms and molecules in the PDB file provided is as follows:

- 500 water molecules, with atoms ordered in this sequence:

```
ATOM      1  O   MOL D   1       5.901   7.384   1.103  0.00  0.00      DUM
ATOM      2  H   MOL D   1       6.047   8.238   0.581  0.00  0.00      DUM
ATOM      3  H   MOL D   1       6.188   7.533   2.057  0.00  0.00      DUM
```

- 1 polymer molecule.
- 1 $Ca^{2+}$ ion.

- 2 $Cl^-$ ions.

In the master LT file, this sequence of molecules is matched with the following commands:

```
# Create the sample.
wat=new SPC[500]
pol=new PolyNIPAM[1]
cat=new Ca[1]
ani=new Cl[2]
```

Note that the first command would create 500 water molecules in the same position in space, and the other commands will use the coordinates specified in the corresponding molecular topology block. However, the coordinates will be overwritten by rendering an external atomic structure file. Note that if the same molecule species are scattered in the input structure, it is recommended to reorder and group together for molecule types to facilitate the creation of the input sample.

The molecular topology for the polymer is created as in the previous example, with the atom types assigned as in the following schema:

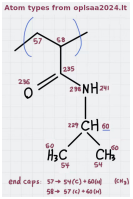



Fig. 13: Atom types assigned to the polymer's repeating unit.

The molecular topology of the water and ions is stated directly into the master file for the sake of space, but they could also be written in a separate file(s) and imported before the sample is created.

The resulting master LT file defining short annealing at a fixed volume (NVT) follows:

```
# Use the OPLS-AA force field for all species.
import /usr/local/moltemplate/moltemplate/force_fields/oplsaa2024.lt
import PolyNIPAM.lt

# Define the SPC water and ions as in the OPLS-AA
Ca inherits OPLSAA {
  write("Data Atoms"){
    $atom:a1  $mol:. @atom:412 0.0  0.00000 0.00000 0.000000
  }
}
Cl inherits OPLSAA {
  write("Data Atoms"){
    $atom:a1  $mol:. @atom:401 0.0  0.00000 0.00000 0.000000
  }
}
SPC inherits OPLSAA {
  write("Data Atoms"){
    $atom:O  $mol:. @atom:9991 0.  0.0000000 0.00000 0.0000000
    $atom:H1 $mol:. @atom:9990 0.  0.8164904 0.00000 0.5773590
    $atom:H2 $mol:. @atom:9990 0. -0.8164904 0.00000 0.5773590
  }
  write("Data Bond List") {
```

(continues on next page)

(continued from previous page)

```
    $bond:OH1 $atom:O $atom:H1
    $bond:OH2 $atom:O $atom:H2
  }
}

# Create the sample.
wat=new SPC[500]
pol=new PolyNIPAM[1]
cat=new Ca[1]
ani=new Cl[2]

# Periodic boundary conditions:
write_once("Data Boundary"){
  0 26 xlo xhi
  0 26 ylo yhi
  0 26 zlo zhi
}

write_once("In Init"){
  boundary p p p  # "p p p" is the default. This line is optional.
  neighbor 3 bin  # (This line is also optional in this example.)
}

# Note: The lines below in the "In Run" section are often omitted.

# Run an NVT simulation.
write_once("In Run"){
  # Input variables.
  variable run    string sample01  # output name
  variable ts     equal  2         # timestep
  variable temp   equal  298.15    # equilibrium temperature
  variable p      equal  1.        # equilibrium pressure
  variable equi   equal  30000     # equilibration steps

  # Set the output.
  thermo          1000
  thermo_style    custom step etotal evdwl ecoul elong ebond eangle &
  edihed eimp pe ke temp press atoms vol density cpu
  thermo_modify flush yes
  compute pe1 all pe/atom pair
  dump TRJ all custom 100 \$\{run\}.dump id xu yu zu c_pe1

  # Minimise the input structure, just in case.
  minimize        .01 .001 1000 100000
  write_data \$\{run\}.min

  # Set the constrains.
  group watergroup type @atom:9991 @atom:9990
  fix 0 watergroup shake 0.0001 10 0 b @bond:spcO_spcH a @angle:spcH_spcO_spcH

  # Short annealing.
  timestep        \$\{ts\}
```

(continues on next page)

(continued from previous page)

```
  fix       1 all nvt temp \$\{temp\} \$\{temp\} \$(100*dt)
  velocity    all create \$\{temp\} 315443
  run             \$\{equi\}
  unfix 1
}
```

In this example, the water model is SPC and it is defined in the `oplsaa2024.lt` file with atom types `@atom:9991` and `@atom:9990`. For water we also use the `group` and `fix shake` commands with Moltemplate `@`-type variables, to ensure consistency with the numerical values assigned during compilation. To identify the bond and angle types, look for the extended `@atom` IDs, which in this case are:

```
replace{ @atom:9991 @atom:9991_bspcO_aspcO_dspcO_ispcO }
replace{ @atom:9990 @atom:9990_bspcH_aspcH_dspcH_ispcH }
```

From which we can identify the following "Data Bonds By Type": `@bond:spcO_spcH @atom:*_bspcO*_a*_d*_i* @atom:*_bspcH*_a*_d*_i*` and "Data Angles By Type": `@angle:spcH_spcO_spcH @atom:*_b*_aspcH*_d*_i* @atom:*_b*_aspcO*_d*_i* @atom:*_b*_aspcH*_d*_i*`

Compile the master file with:

```
moltemplate.sh -pdb model.pdb sample01.lt
cleanup_moltemplate.sh
```

And execute the simulation with the following:

```
mpirun -np 4 lmp -in sample01.in -l sample01.log
```

---

**(OPLS-AA)** Jorgensen, W.L., Ghahremanpour, M.M., Saar, A., Tirado-Rives, J., J. Phys. Chem. B, 128(1), 250-262 (2024).

**(Moltemplate)** Jewett et al., J. Mol. Biol., 433(11), 166841 (2021)

### 10.6.5 LAMMPS Python Tutorial


- *Overview*
- *Quick Start*
  - *System-wide or User Installation*
    - *Step 1: Building LAMMPS as a shared library*
    - *Step 2: Installing the LAMMPS Python module*
  - *Installation inside of a virtual environment*
    - *Benefits of using a virtualenv*
    - *Creating a virtualenv with lammps installed*
- *Creating a new lammps instance*
- *Commands*
- *Accessing atom data*

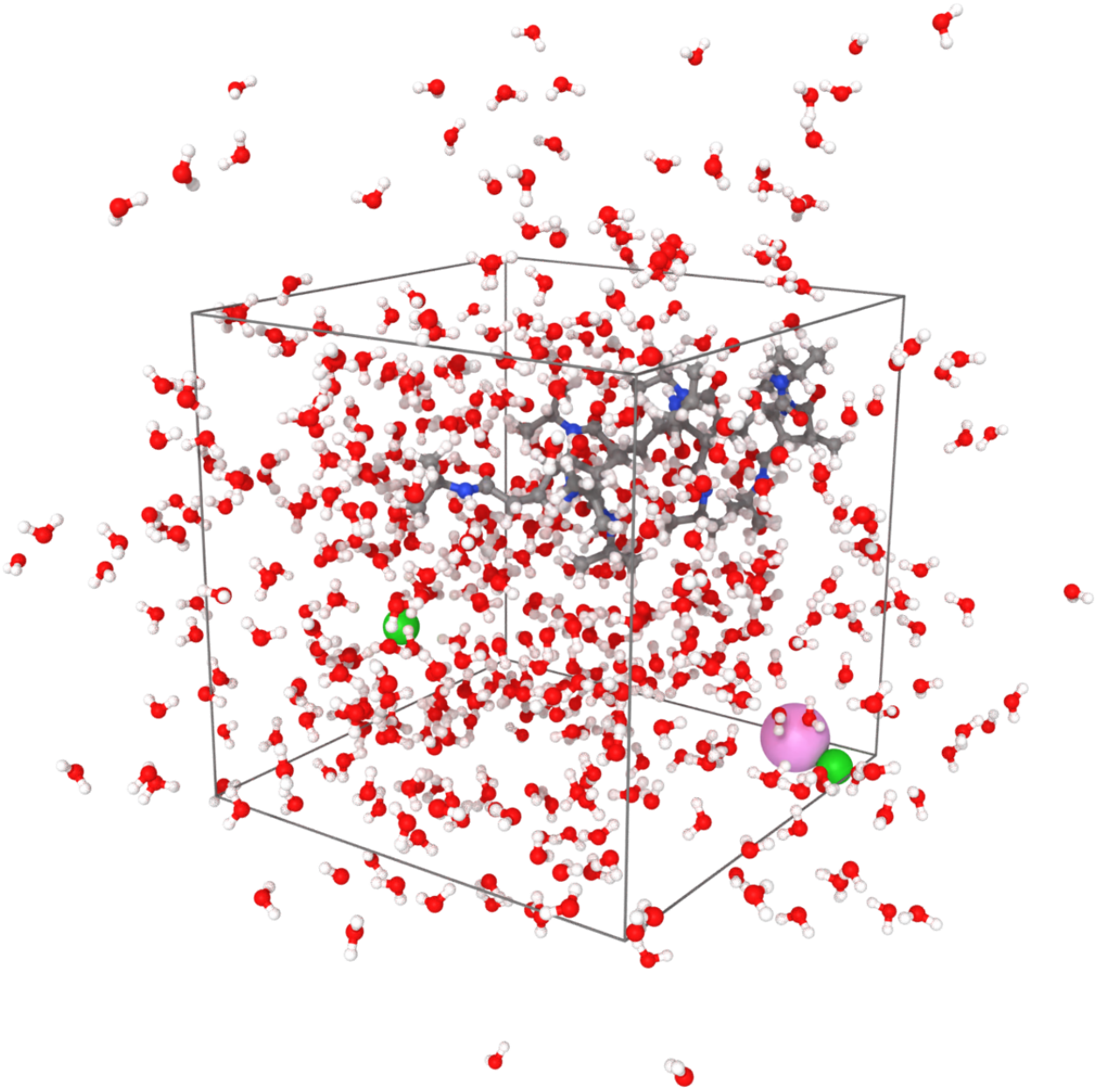


Fig. 14: Sample visualized with Ovito loading the trajectory into the DATA file written after minimization.

- *Retrieving the values of thermodynamic data and variables*
- *Error handling*
- *Using LAMMPS in IPython notebooks and Jupyter*
- *Interactive Python Examples*
  - *Validating a dihedral potential*
  - *Running a Monte Carlo relaxation*


---

### Overview

The `lammps` Python module is a wrapper class for the LAMMPS *C language library interface API* which is written using Python ctypes. The design choice of this wrapper class is to follow the C language API closely with only small changes related to Python specific requirements and to better accommodate object oriented programming.

In addition to this flat ctypes interface, the `lammps` wrapper class exposes a discoverable API that doesn't require as much knowledge of the underlying C language library interface or LAMMPS C++ code implementation.

Finally, the API exposes some additional features for IPython integration into Jupyter notebooks, e.g. for embedded visualization output from *dump style image*.

---

### Quick Start

### System-wide or User Installation

### Step 1: Building LAMMPS as a shared library

To use LAMMPS inside of Python it has to be compiled as shared library. This library is then loaded by the Python interface. In this example we enable the *MOLECULE package* and compile LAMMPS with *PNG, JPEG and FFMPEG output support* enabled.

**CMake build**

```
mkdir $LAMMPS_DIR/build-shared
cd  $LAMMPS_DIR/build-shared

# MPI, PNG, Jpeg, FFMPEG are auto-detected
cmake ../cmake -DPKG_MOLECULE=yes -DPKG_PYTHON=on -DBUILD_SHARED_LIBS=yes
make
```

**Traditional make**

```
cd $LAMMPS_DIR/src

# add LAMMPS packages if necessary
make yes-MOLECULE
make yes-PYTHON
```

```
# compile shared library using Makefile
make mpi mode=shlib LMP_INC="-DLAMMPS_PNG -DLAMMPS_JPEG -DLAMMPS_FFMPEG" JPG_
↪LIB="-lpng -ljpeg"
```

### Step 2: Installing the LAMMPS Python module

Next install the LAMMPS Python module into your current Python installation with:

```
make install-python
```

This will create a so-called "wheel" and then install the LAMMPS Python module from that "wheel" into either into a system folder (provided the command is executed with root privileges) or into your personal Python module folder.

**Note**

Recompiling the shared library requires re-installing the Python package.

**Handling an "externally-managed-environment" Error**

Some Python installations made through Linux distributions (e.g. Ubuntu 24.04LTS or later) will prevent installing the LAMMPS Python module into a system folder or a corresponding folder of the individual user as attempted by `make install-python` with an error stating that an *externally managed* python installation must be only managed by the same package package management tool. This is an optional setting, so not all Linux distributions follow it currently (Spring 2025). The reasoning and explanations for this error can be found in the Python Packaging User Guide

These guidelines suggest to create a virtual environment and install the LAMMPS Python module there (see below). This is generally a good idea and the LAMMPS developers recommend this, too. If, however, you want to proceed and install the LAMMPS Python module regardless, you can install the "wheel" file (see above) manually with the `pip` command by adding the `--break-system-packages` flag.

### Installation inside of a virtual environment

You can use virtual environments to create a custom Python environment specifically tuned for your workflow.

### Benefits of using a virtualenv

- isolation of your system Python installation from your development installation
- installation can happen in your user directory without root access (useful for HPC clusters)
- installing packages through pip allows you to get newer versions of packages than e.g., through apt-get or yum package managers (and without root access)
- you can even install specific old versions of a package if necessary

**Prerequisite (e.g. on Ubuntu)**

```
apt-get install python-venv
```

### Creating a virtualenv with lammps installed

```
# create virtual envrionment named 'testing'
python3 -m venv $HOME/python/testing

# activate 'testing' environment
source $HOME/python/testing/bin/activate
```

Now configure and compile the LAMMPS shared library as outlined above. When using CMake and the shared library has already been build, you need to re-run CMake to update the location of the python executable to the location in the virtual environment with:

```
cmake . -DPython_EXECUTABLE=$(which python)

# install LAMMPS package in virtualenv
(testing) make install-python

# install other useful packages
(testing) pip install matplotlib jupyter mpi4py pandas

...

# return to original shell
(testing) deactivate
```

---

### Creating a new lammps instance

To create a lammps object you need to first import the class from the lammps module. By using the default constructor, a new *lammps* instance is created.

```
from lammps import lammps
L = lammps()
```

See the *LAMMPS Python documentation* for how to customize the instance creation with optional arguments.

---

### Commands

Sending a LAMMPS command with the library interface is done using the `command` method of the lammps object.

For instance, let's take the following LAMMPS command:

```
region box block 0 10 0 5 -0.5 0.5
```

This command can be executed with the following Python code if `L` is a `lammps` instance:

```
L.command("region box block 0 10 0 5 -0.5 0.5")
```

For convenience, the `lammps` class also provides a command wrapper `cmd` that turns any LAMMPS command into a regular function call:

```
L.cmd.region("box block", 0, 10, 0, 5, -0.5, 0.5)
```

Note that each parameter is set as Python number literal. With the wrapper each command takes an arbitrary parameter list and transparently merges it to a single command string, separating individual parameters by white-space.

The benefit of this approach is avoiding redundant command calls and easier parameterization. With the `command` function each call needs to be assembled manually using formatted strings.

```
L.command(f"region box block {xlo} {xhi} {ylo} {yhi} {zlo} {zhi}")
```

The wrapper accepts parameters directly and will convert them automatically to a final command string.

```
L.cmd.region("box block", xlo, xhi, ylo, yhi, zlo, zhi)
```

**Note**

When running in IPython you can use Tab-completion after `L.cmd.` to see all available LAMMPS commands.

### Accessing atom data

All per-atom properties that are part of the *atom style* in the current simulation can be accessed using the `extract_atoms()` method. This can be retrieved as ctypes objects or as NumPy arrays through the lammps.numpy module. Those represent the *local* atoms of the individual sub-domain for the current MPI process and may contain information for the local ghost atoms or not depending on the property. Both can be accessed as lists, but for the ctypes list object the size is not known and hast to be retrieved first to avoid out-of-bounds accesses.

```
nlocal = L.extract_setting("nlocal")
nall = L.extract_setting("nall")
print("Number of local atoms ", nlocal, "  Number of local and ghost atoms ", nall);

# access via ctypes directly
atom_id = L.extract_atom("id")
print("Atom IDs", atom_id[0:nlocal])

# access through numpy wrapper
atom_type = L.numpy.extract_atom("type")
print("Atom types", atom_type)

x = L.numpy.extract_atom("x")
v = L.numpy.extract_atom("v")
print("positions array shape", x.shape)
print("velocity array shape", v.shape)
# turn on communicating velocities to ghost atoms
L.cmd.comm_modify("vel", "yes")
v = L.numpy.extract_atom('v')
print("velocity array shape", v.shape)
```

Some properties can also be set from Python since internally the data of the C++ code is accessed directly:

```
# set position in 2D simulation
x[0] = (1.0, 0.0)

# set position in 3D simulation
x[0] = (1.0, 0.0, 1.)
```

### Retrieving the values of thermodynamic data and variables

To access thermodynamic data from the last completed timestep, you can use the *get_thermo()* method, and to extract the value of (compatible) variables, you can use the *extract_variable()* method.

```
result = L.get_thermo("ke") # kinetic energy
result = L.get_thermo("pe") # potential energy

result = L.extract_variable("t") / 2.0
```

### Error handling

We are using C++ exceptions in LAMMPS for errors and the C language library interface captures and records them. This allows checking whether errors have happened in Python during a call into LAMMPS and then re-throw the error as a Python exception. This way you can handle LAMMPS errors in the conventional way through the Python exception handling mechanism.

**Warning**

Capturing a LAMMPS exception in Python can still mean that the current LAMMPS process is in an illegal state and must be terminated. It is advised to save your data and terminate the Python instance as quickly as possible.

### Using LAMMPS in IPython notebooks and Jupyter

If the LAMMPS Python package is installed for the same Python interpreter as IPython, you can use LAMMPS directly inside of an IPython notebook inside of Jupyter. Jupyter is a powerful integrated development environment (IDE) for many dynamic languages like Python, Julia and others, which operates inside of any web browser. Besides auto-completion and syntax highlighting it allows you to create formatted documents using Markup, mathematical formulas, graphics and animations intermixed with executable Python code. It is a great format for tutorials and showcasing your latest research.

To launch an instance of Jupyter simply run the following command inside your Python environment (this assumes you followed the Quick Start instructions):

```
jupyter notebook
```

### Interactive Python Examples

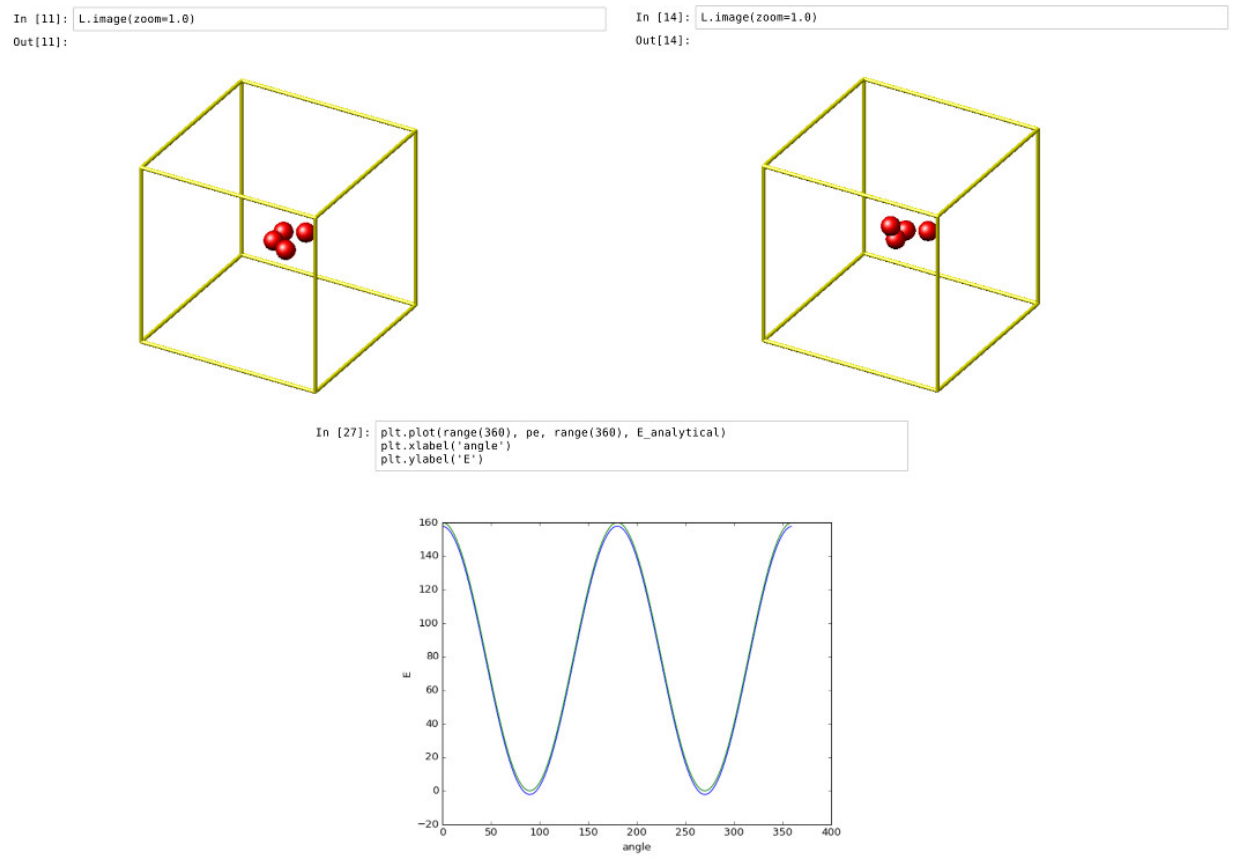
Examples of IPython notebooks can be found in the `python/examples/ipython` subdirectory. To open these notebooks launch `jupyter notebook` inside this directory and navigate to one of them. If you compiled and installed a LAMMPS shared library with PNG, JPEG and FFMPEG support you should be able to rerun all of these notebooks.

### Validating a dihedral potential

This example showcases how an IPython Notebook can be used to compare a simple LAMMPS simulation of a harmonic dihedral potential to its analytical solution. Four atoms are placed in the simulation and the dihedral potential is applied on them using a datafile. Then one of the atoms is rotated along the central axis by setting its position from Python, which changes the dihedral angle.

```
phi = [d \* math.pi / 180 for d in range(360)]

pos = [(1.0, math.cos(p), math.sin(p)) for p in phi]

x = L.numpy.extract_atom("x")

pe = []
for p in pos:
    x[3] = p
    L.cmd.run(0, "post", "no")
    pe.append(L.get_thermo("pe"))
```

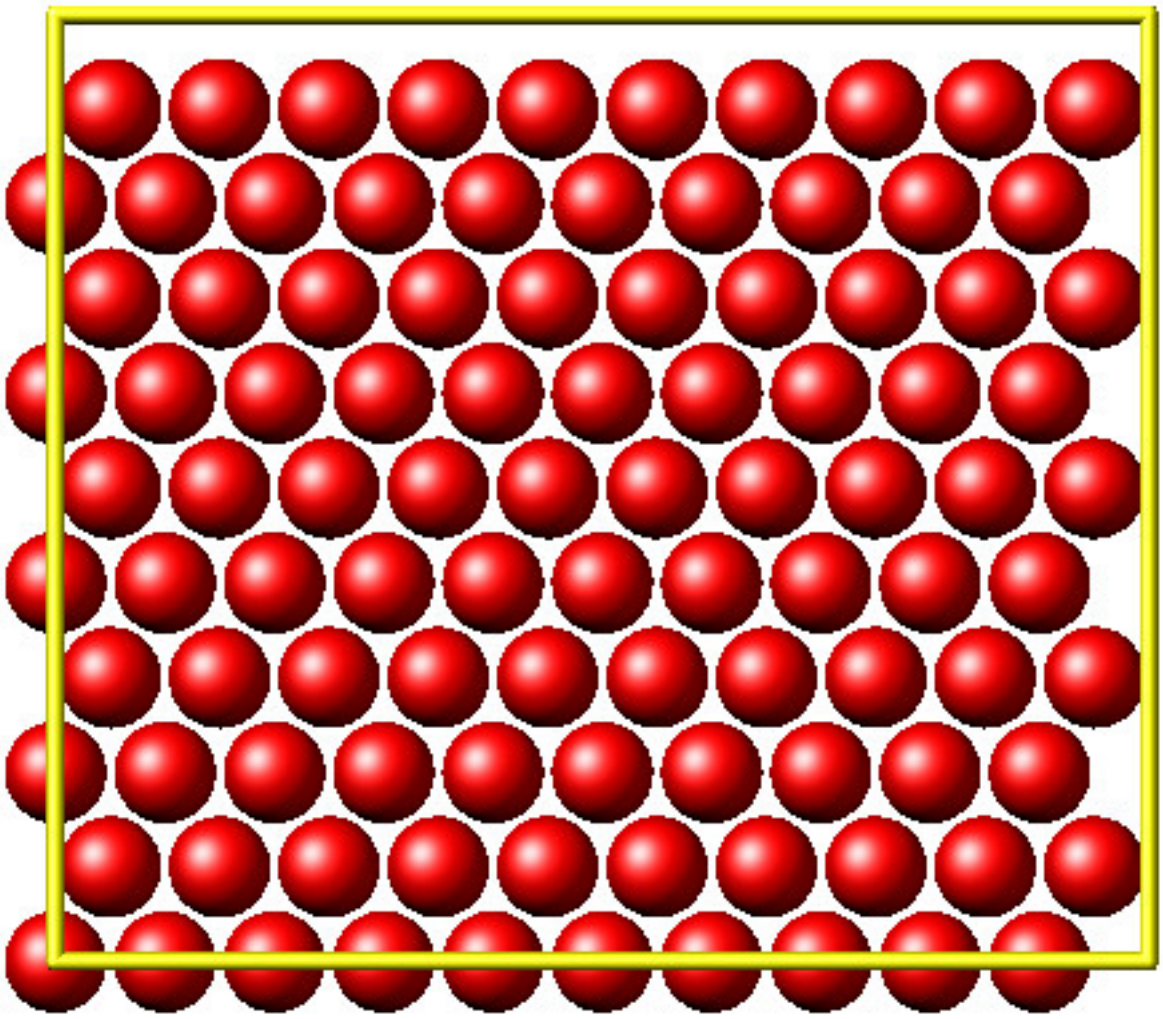
By evaluating the potential energy for each position we can verify that trajectory with the analytical formula. To compare both solutions, we plot both trajectories over each other using matplotlib, which embeds the generated plot inside the IPython notebook.




## Running a Monte Carlo relaxation

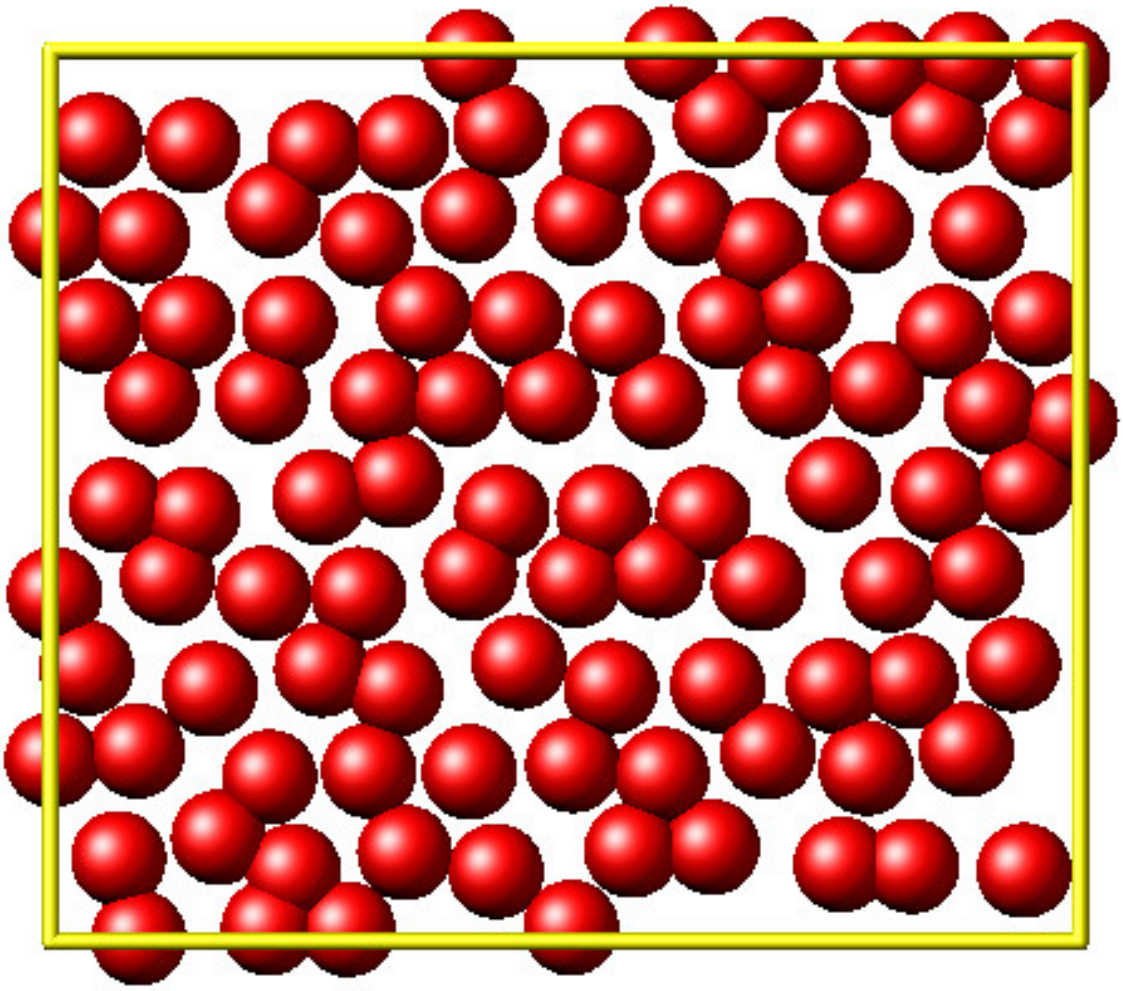
This second example shows how to use the *lammps* Python interface to create a 2D Monte Carlo Relaxation simulation, computing and plotting energy terms and even embedding video output.

Initially, a 2D system is created in a state with minimal energy.



It is then disordered by moving each atom by a random delta.

```
random.seed(27848)
deltaperturb = 0.2
x = L.numpy.extract_atom("x")
natoms = x.shape[0]

for i in range(natoms):
    dx = deltaperturb \* random.uniform(-1, 1)
    dy = deltaperturb \* random.uniform(-1, 1)
    x[i][0] += dx
    x[i][1] += dy

L.cmd.run(0, "post", "no")
```



Finally, the Monte Carlo algorithm is implemented in Python. It continuously moves random atoms by a random delta and only accepts certain moves.

```
estart = L.get_thermo("pe")
elast = estart

naccept = 0
energies = [estart]

niterations = 3000
deltamove = 0.1
kT = 0.05

for i in range(niterations):
    x = L.numpy.extract_atom("x")
    natoms = x.shape[0]
    iatom = random.randrange(0, natoms)
    current_atom = x[iatom]
```

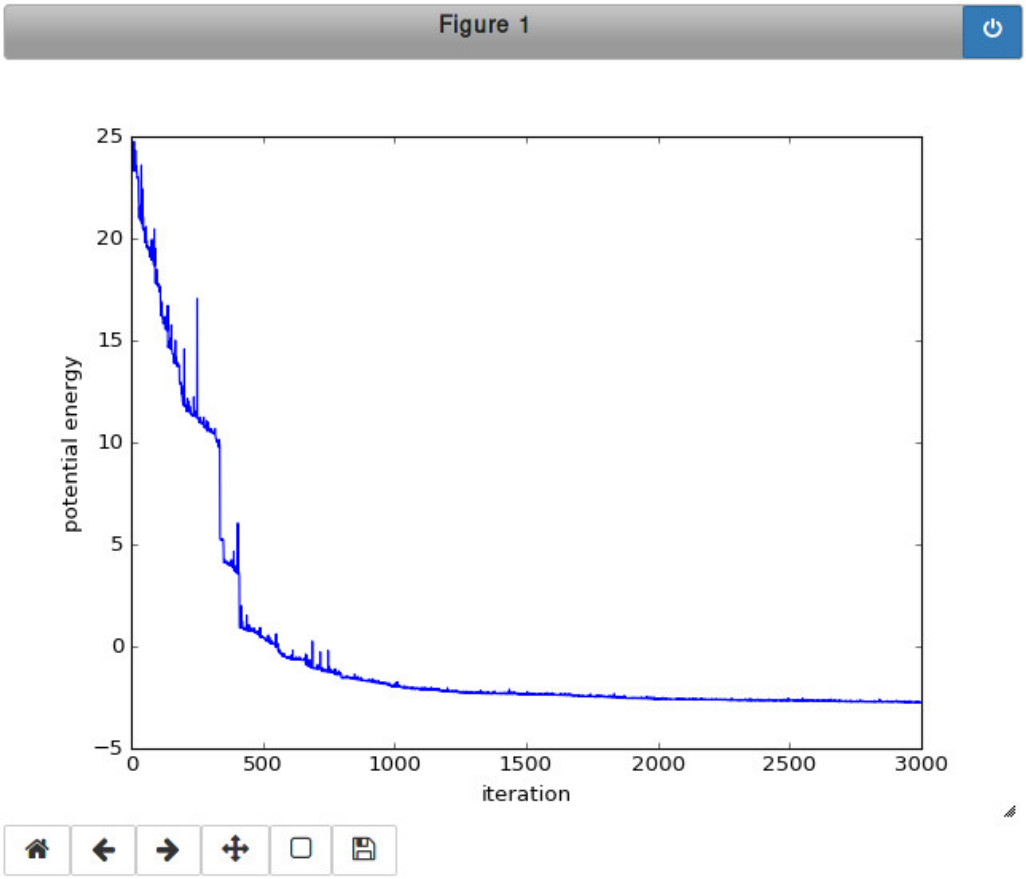
(continues on next page)

(continued from previous page)

```
    x0 = current_atom[0]
    y0 = current_atom[1]

    dx = deltamove \* random.uniform(-1, 1)
    dy = deltamove \* random.uniform(-1, 1)

    current_atom[0] = x0 + dx
    current_atom[1] = y0 + dy

    L.cmd.run(1, "pre no post no")

    e = L.get_thermo("pe")
    energies.append(e)

    if e <= elast:
        naccept += 1
        elast = e
    elif random.random() <= math.exp(natoms\*(elast-e)/kT):
        naccept += 1
        elast = e
    else:
        current_atom[0] = x0
        current_atom[1] = y0
```

The energies of each iteration are collected in a Python list and finally plotted using matplotlib.

```
In [20]: plt.xlabel('iteration')
         plt.ylabel('potential energy')
         plt.plot(energies)
```




```
Out[20]: [<matplotlib.lines.Line2D at 0x7f26a40123c8>]
```

The IPython notebook also shows how to use dump commands and embed video files inside of the IPython notebook.

### 10.6.6 Using LAMMPS on Windows 10 with WSL

**written by Richard Berger**

---

It's always been tricky for us to have LAMMPS users and developers work on Windows. We primarily develop LAMMPS to run on Linux clusters. To teach LAMMPS in workshop settings, we had to redirect Windows users to Linux Virtual Machines such as VirtualBox or Unix-like compilation with Cygwin.

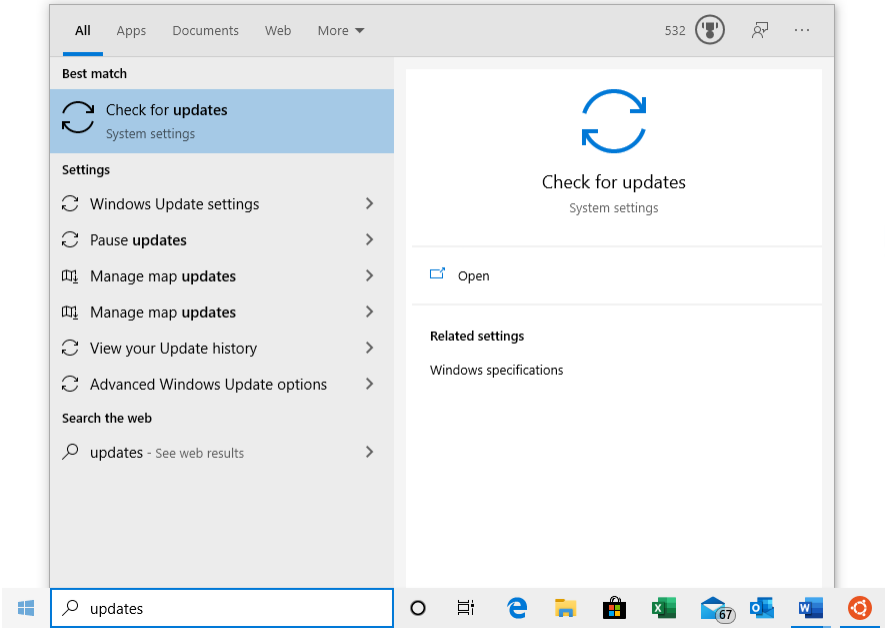
With the latest updates in Windows 10 (Version 2004, Build 19041 or higher), Microsoft has added a new way to work on Linux-based code. The Windows Subsystem for Linux (WSL). With WSL Version 2, you now get a Linux Virtual Machine that transparently integrates into Windows. All you need is to ensure you have the latest Windows updates installed and enable this new feature. Linux VMs are then easily installed using the Microsoft Store.

In this tutorial, I'll show you how to set up and compile LAMMPS for both serial and MPI usage in WSL2.

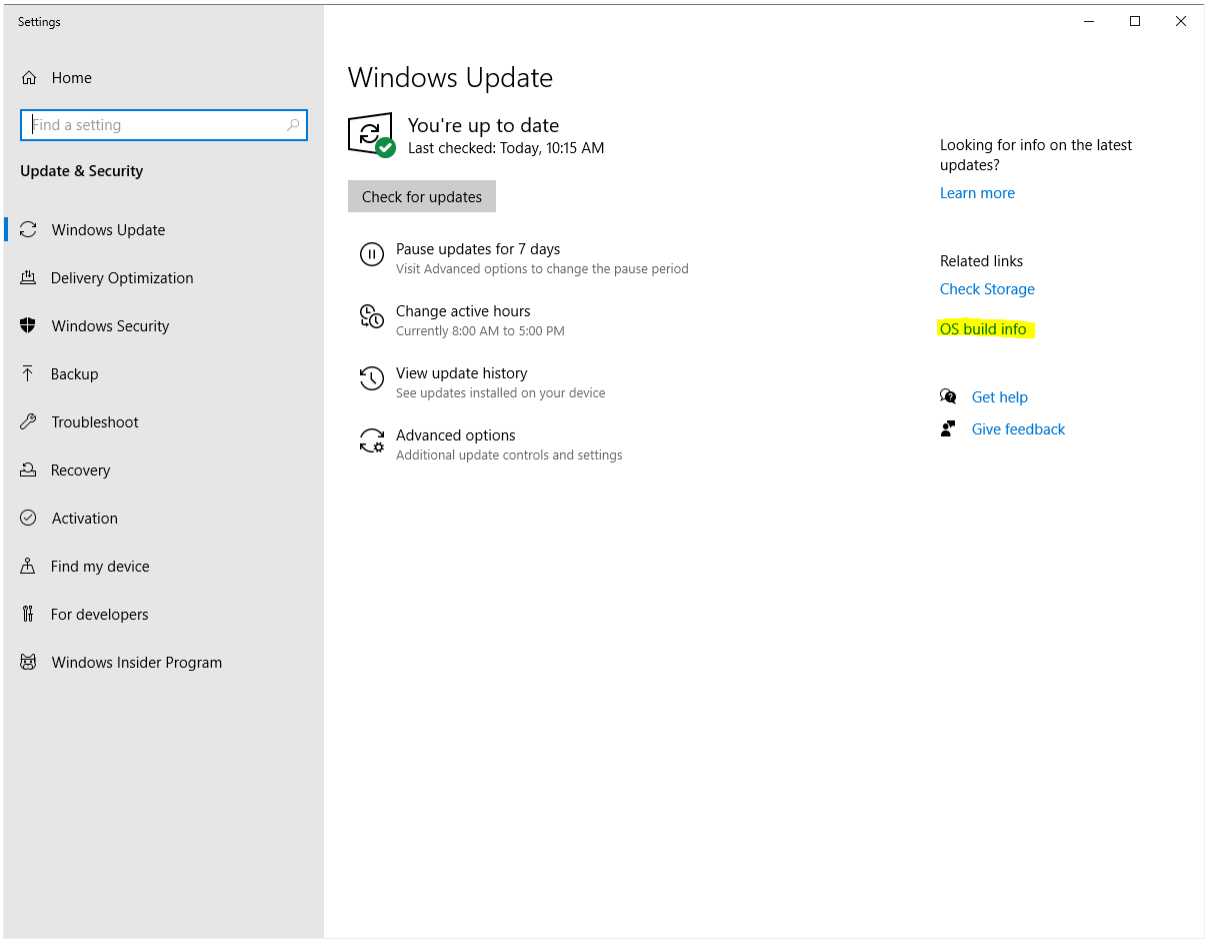
## Installation

### Upgrade to the latest Windows 10

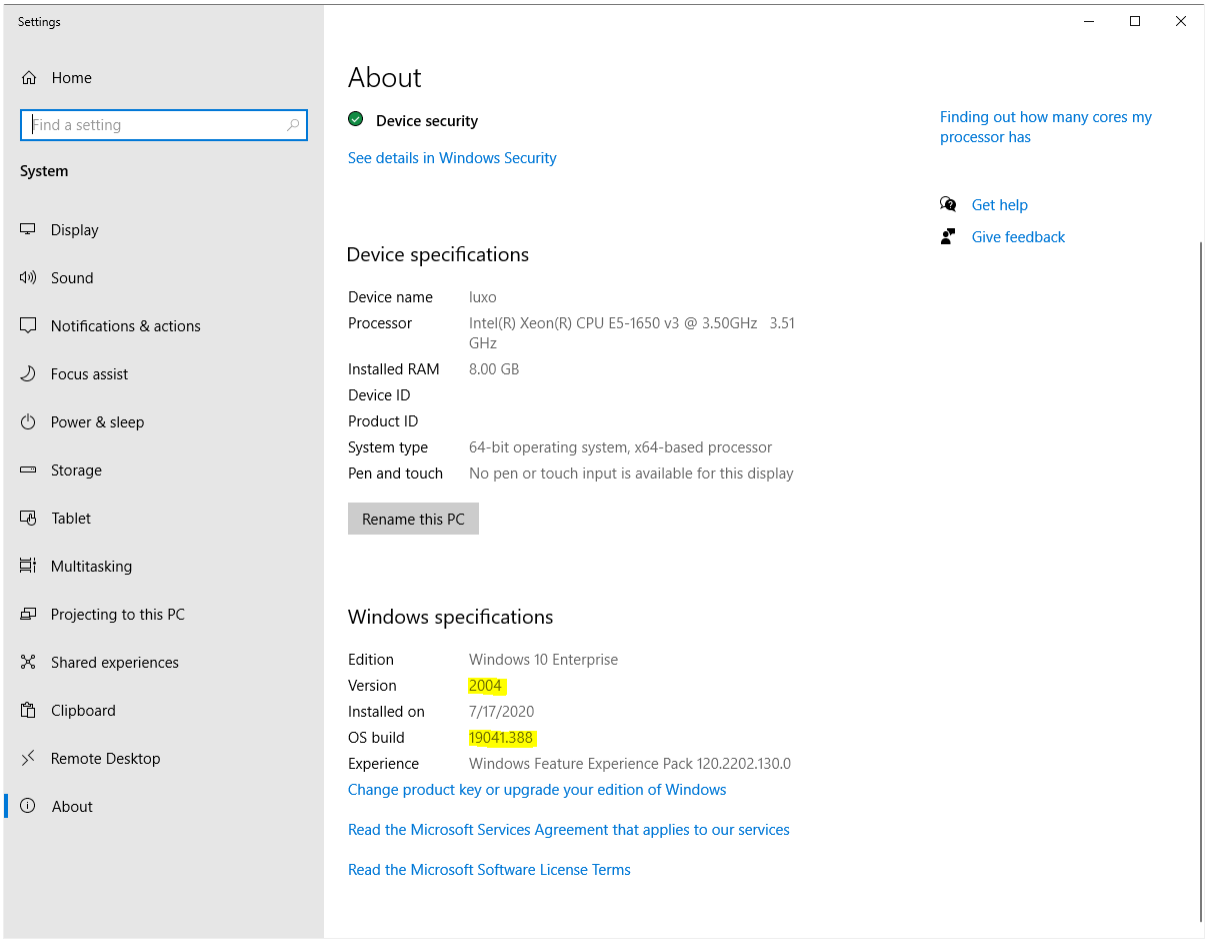
Type “Updates” in Windows Start and select “Check for Updates”.




Install all pending updates and reboot your system as many times as necessary. Continue until your Windows installation is updated.

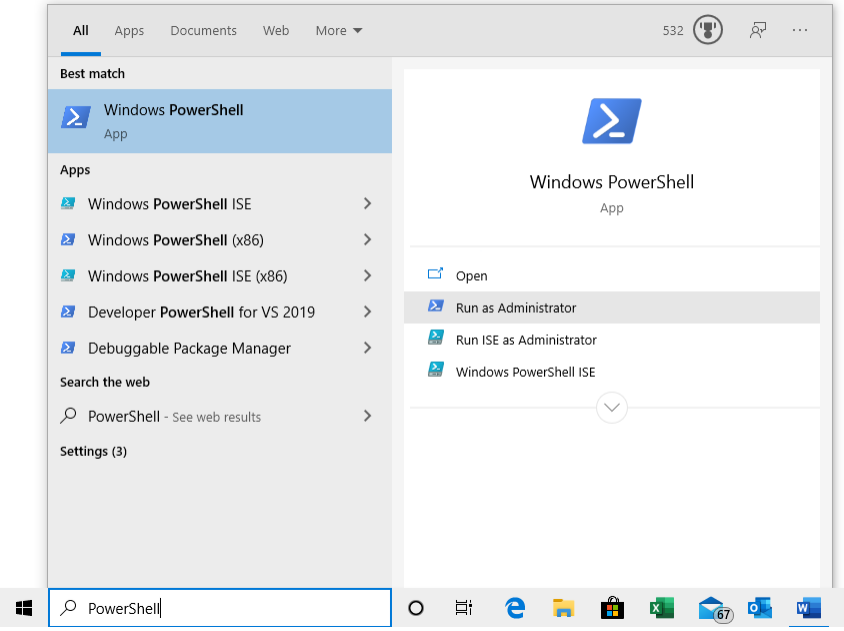



Verify your system has at least **version 2004 and build 19041 or later**. You can find this information by clicking on "OS build info".



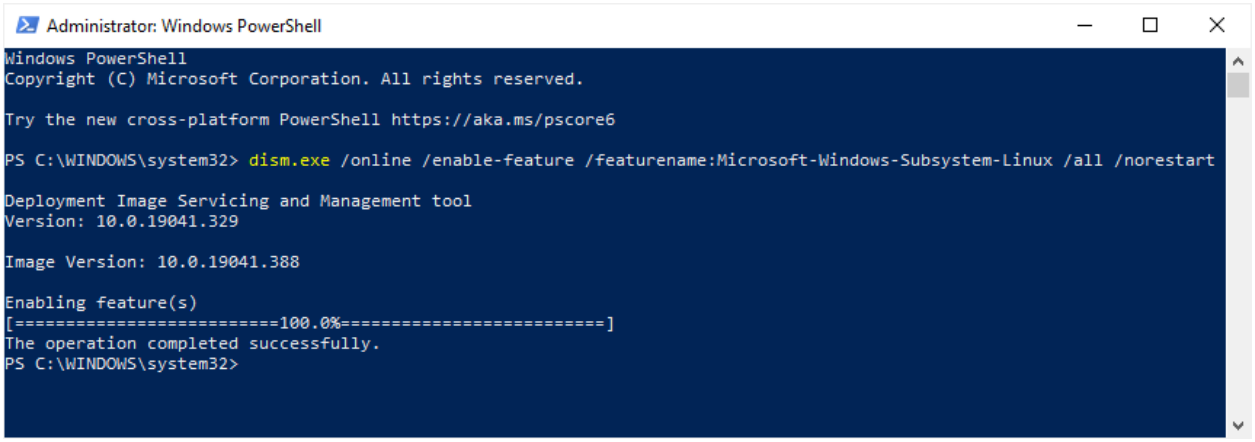

## Enable WSL

Next, we must install two additional Windows features to enable WSL support. Open a PowerShell window as an administrator. Type "PowerShell" in Windows Start and select "Run as Administrator".



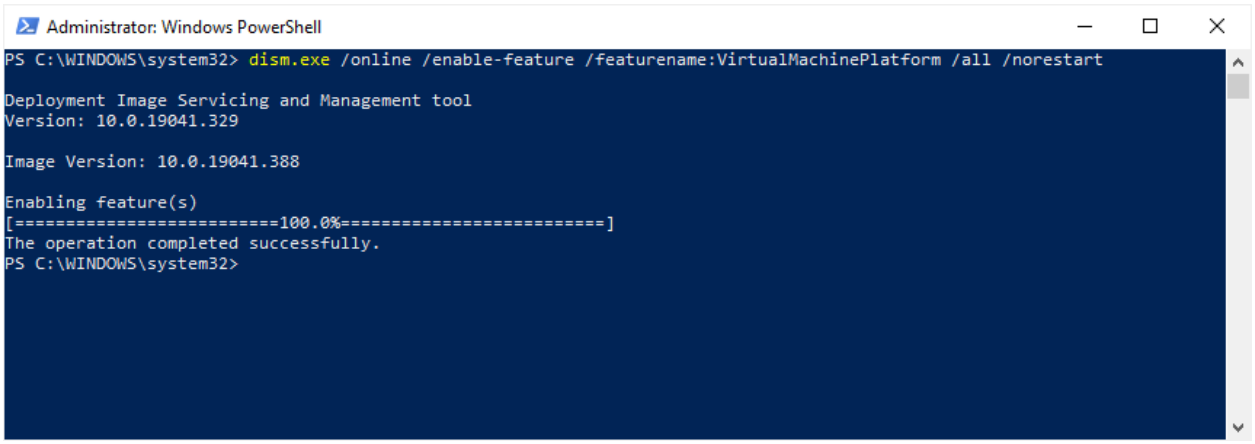

Windows will ask you for administrator access. After you accept a new command line window will appear. Type in the following command to install WSL:

```
dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /
↪norestart
```




Next, enable the VirtualMachinePlatform feature using the following command:

```
dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart
```




Finally, reboot your system.

### Update WSL kernel component

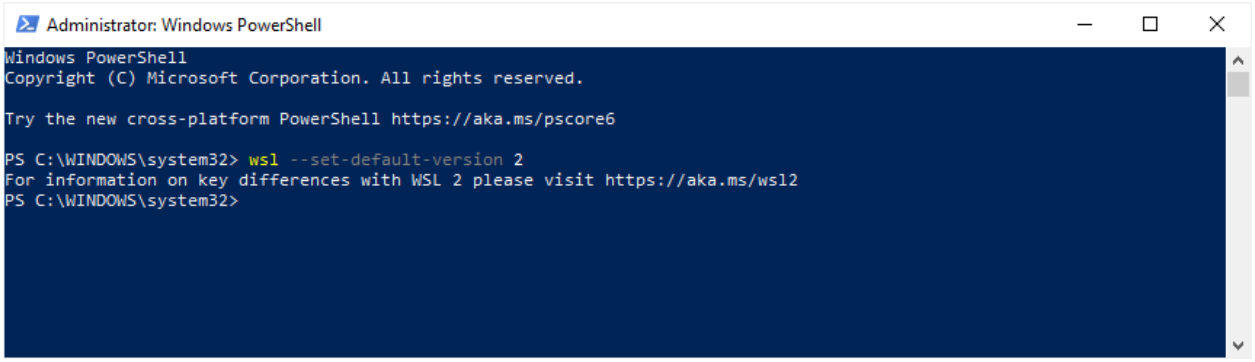
Download and install the `WSL Kernel Component Update`. Afterwards, reboot your system.

### Set WSL2 as default

Again, open PowerShell as administrator and run the following command:

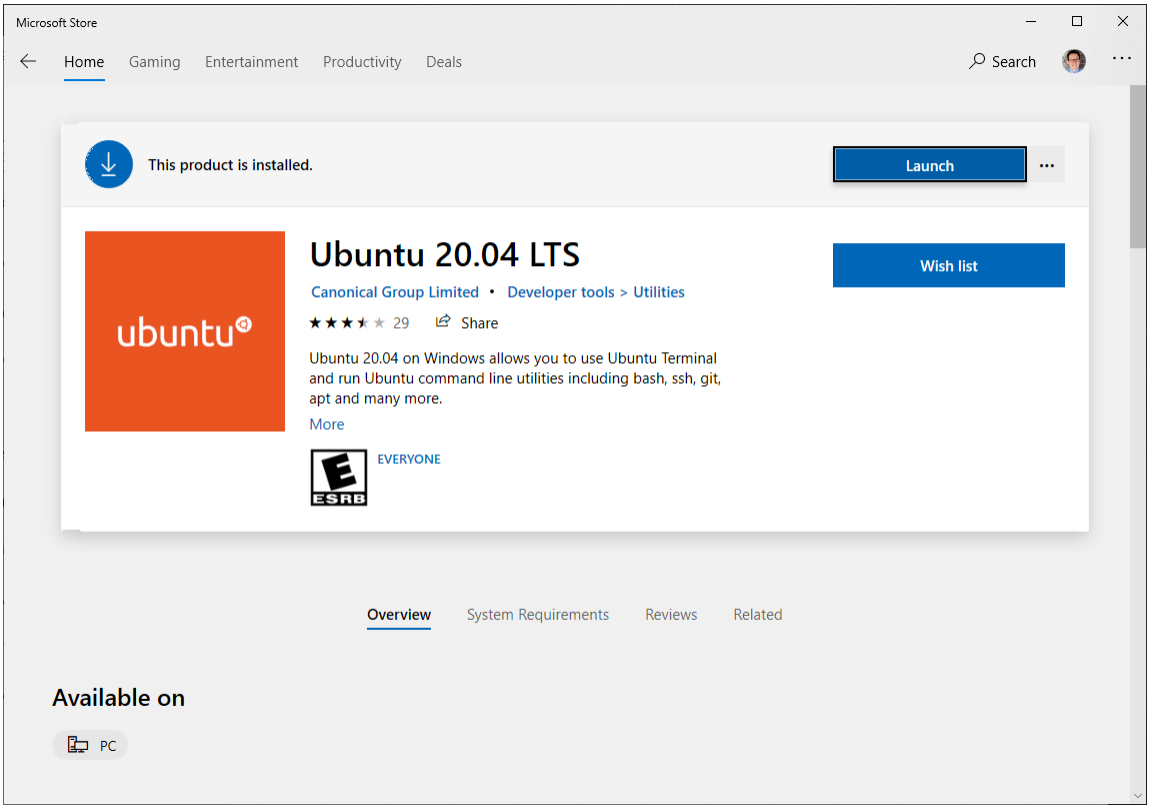
```
wsl --set-default-version 2
```

This command ensures that all future Linux installations will use WSL version 2.



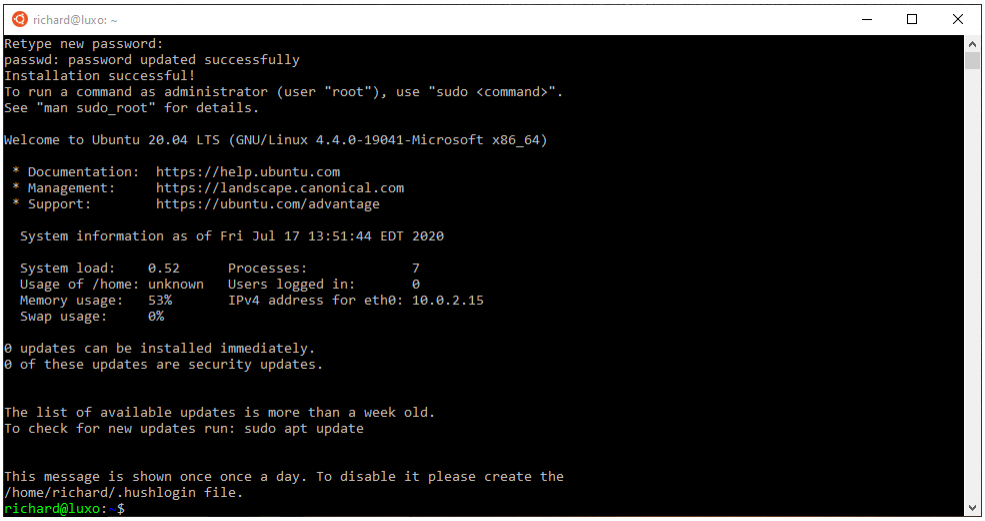

### Install a Linux Distribution

Next, we need to install a Linux distribution via the Microsoft Store. Install Ubuntu 20.04 LTS. Once installed, you can launch it like any other application from the Start Menu.




## Initial Setup

The first time you launch the Ubuntu Linux console, it will prompt you for a UNIX username and password. You will need this password to perform sudo commands later. Once completed, your Linux shell is ready for use. All your actions and commands will run as the Linux user you specified.

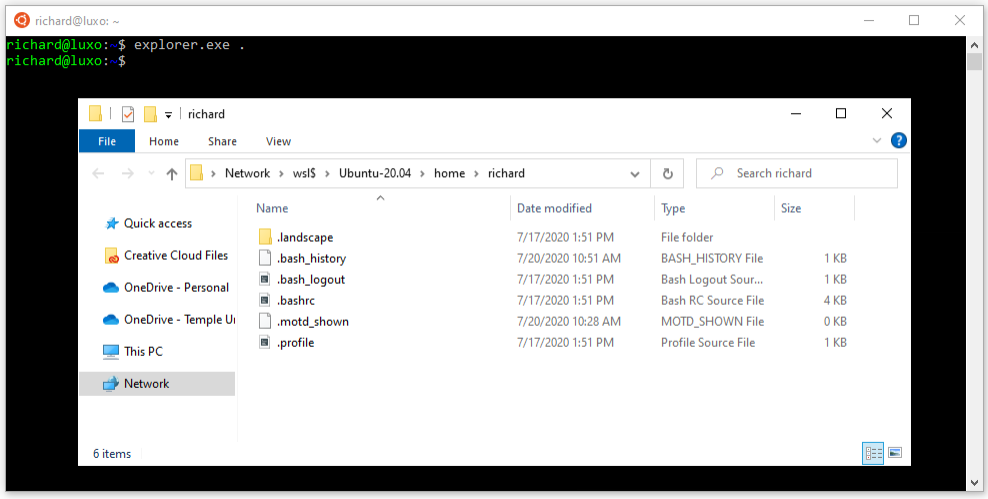


### Windows Explorer / WSL integration

Your Linux installation will have its own Linux filesystem, which contains the Ubuntu files. Your Linux user will have a regular Linux home directory in `/home/<USERNAME>`. This directory is different from your Windows User directory. Windows and Linux filesystems are connected through WSL.

All hard drives in Windows are accessible in the `/mnt` directory in Linux. E.g., WSL maps the `C` hard drive to the `/mnt/c` directory. That means you can access your Windows User directory in `/mnt/c/Users/<WINDOWS_USERNAME>`.

The Windows Explorer can also access the Linux filesystem. To illustrate this integration, open an Ubuntu console and navigate to a directory of your choice. To view this location in Windows Explorer, use the `explorer.exe .` command (do not forget the final dot!).




---

### Compiling LAMMPS

You now have a fully functioning Ubuntu installation and can follow most guides to install LAMMPS on a Linux system. Here are some of the essential steps to follow:

### Install prerequisite packages

Before we can begin, we need to download the necessary compiler tool chain and libraries to compile LAMMPS. In our Ubuntu-based Linux installation, we will use the `apt` package manager to install additional packages.

First, upgrade all existing packages using `apt update` and `apt upgrade`.

```
sudo apt update
sudo apt upgrade -y
```

Next, install the following packages with `apt install`:

```
sudo apt install -y cmake build-essential ccache gfortran openmpi-bin libopenmpi-dev \
                    libfftw3-dev libjpeg-dev libpng-dev python3-dev python3-pip \
                    python3-virtualenv libblas-dev liblapack-dev libhdf5-serial-dev \
                    hdf5-tools
```

### Download LAMMPS

Obtain a copy of the LAMMPS source code and go into it using the `cd` command.

### Option 1: Download a LAMMPS tarball using wget

```
wget https://github.com/lammps/lammps/archive/stable_3Mar2020.tar.gz
tar xvzf stable_3Mar2020.tar.gz
cd lammps
```

### Option 2: Download a LAMMPS development version from GitHub

```
git clone --depth=1 https://github.com/lammps/lammps.git
cd lammps
```

### Configure and Compile LAMMPS with CMake

A beginner-friendly way to compile LAMMPS is to use CMake. Create a `build` directory to compile LAMMPS and move into it. This directory will store the build configuration and any binaries generated during compilation.

```
mkdir build
cd build
```

There are countless ways to compile LAMMPS. It is beyond the scope of this tutorial. If you want to find out more about what can be enabled, please consult the extensive documentation.

To compile a minimal version of LAMMPS, we're going to use a preset. Presets are a way to specify a collection of CMake options using a file.

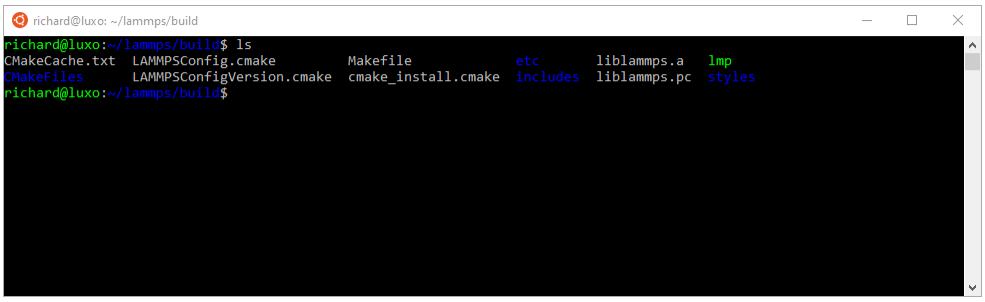
```
cmake ../cmake/presets/basic.cmake ../cmake
```

This command configures the build and generates the necessary Makefiles. To compile the binary, run the make command.

```
make -j 4
```

The `-j` option specifies how many parallel processes will perform the compilation. This option can significantly speed up compilation times. Use a number that corresponds to the number of processors in your system.

After the compilation completes successfully, you will have an executable called `lmp` in the `build` directory.




Please take note of the absolute path of your `build` directory. You will need to know the location to execute the LAMMPS binary later.

One way of getting the absolute path of the current directory is through the `$PWD` variable:

```
# prints out the current value of the PWD variable
echo $PWD
```

Let us save this value in a temporary variable `LAMMPS_BUILD_DIR` for future use:

```
LAMMPS_BUILD_DIR=$PWD
```

The full path of the LAMMPS binary then is `$LAMMPS_BUILD_DIR/lmp`.

---

### Running an example script

Now that we have a LAMMPS binary, we will run a script from the examples folder.

Switch into the `examples/melt` folder:

```
cd ../examples/melt
```

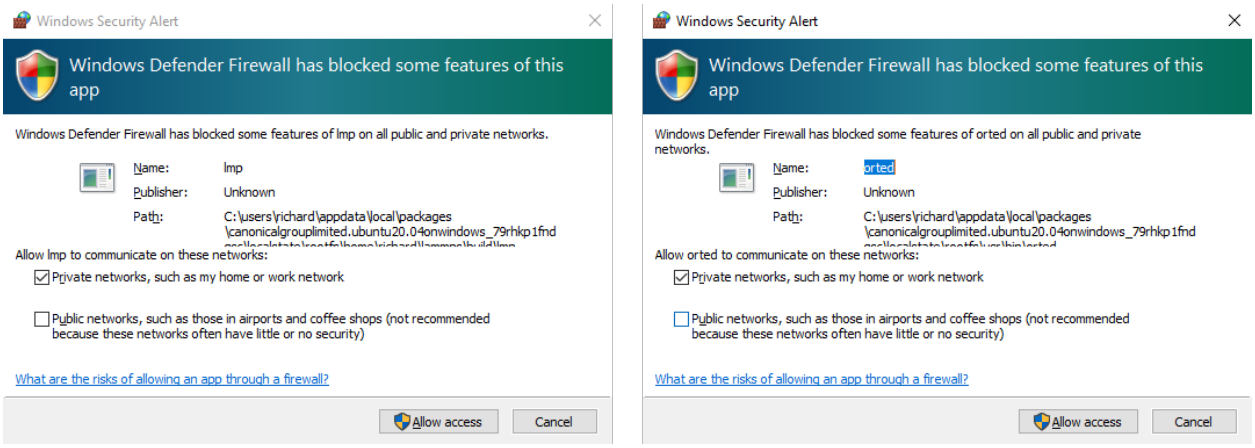
To run this example in serial, use the following command:

```
$LAMMPS_BUILD_DIR/lmp -in in.melt
```

To run the same script in parallel using MPI with 4 processes, do the following:

```
mpirun -np 4 $LAMMPS_BUILD_DIR/lmp -in in.melt
```

If you run LAMMPS for the first time, the Windows Firewall might prompt you to confirm access. LAMMPS is accessing the network stack to enable parallel computation. Allow the access.




In either serial or MPI case, LAMMPS executes and will output something similar to this:

```
LAMMPS (30 Jun 2020)
...
...
...
Total # of neighbors = 151513
Ave neighs/atom = 37.878250
Neighbor list builds = 12
Dangerous builds not checked
Total wall time: 0:00:00
```

**Congratulations! You've successfully compiled and executed LAMMPS on WSL!**

### Final steps

It is cumbersome to always specify the path of your LAMMPS binary. You can avoid this by adding the absolute path of your `build` directory to your PATH environment variable.

```
export PATH=$LAMMPS_BUILD_DIR:$PATH
```

You can then run LAMMPS input scripts like this:

```
lmp -in in.melt
```

or

```
mpirun -np 4 lmp -in in.melt
```

**Note**

The value of this `PATH` variable will disappear once you close your console window. To persist this setting edit the `$HOME/.bashrc` file using your favorite text editor and add this line:

```
export PATH=/full/path/to/your/lammps/build:$PATH
```

**Example:** If the LAMMPS executable *lmp* has the following absolute path:

```
/home/<USERNAME>/lammps/build/lmp
```

the `PATH` variable should be:

```
export PATH=/home/<USERNAME>/lammps/build:$PATH
```

Once set up, all your Ubuntu consoles will always have access to your `lmp` binary without having to specify its location.

### Conclusion

I hope this gives you good overview on how to start compiling and running LAMMPS on Windows. WSL makes preparing and running scripts on Windows a much better experience.

If you are completely new to Linux, I highly recommend investing some time in studying Linux online tutorials. E.g., tutorials about Bash Shell and Basic Unix commands (e.g., Linux Journey). Acquiring these skills will make you much more productive in this environment.

**See also**

- Windows Subsystem for Linux Documentation

CHAPTER

# ELEVEN

# EXAMPLE SCRIPTS

The LAMMPS distribution includes an examples subdirectory with many sample problems. Many are 2d models that run quickly and are straightforward to visualize, requiring at most a couple of minutes to run on a desktop machine. Each problem has an input script (in.*) and produces a log file (log.*) when it runs. Some use a data file (data.*) of initial coordinates as additional input. A few sample log file run on different machines and different numbers of processors are included in the directories to compare your answers to. E.g. a log file like log.date.crack.foo.P means the "crack" example was run on P processors of machine "foo" on that date (i.e. with that version of LAMMPS).

Many of the input files have commented-out lines for creating dump files and image files.

If you uncomment the *dump* command in the input script, a text dump file will be produced, which can be animated by various visualization programs.

If you uncomment the *dump image* command in the input script, and assuming you have built LAMMPS with a JPG library, JPG snapshot images will be produced when the simulation runs. They can be quickly post-processed into a movie using commands described on the *dump image* doc page.

Animations of many of the examples can be viewed on the Movies section of the LAMMPS website.

There are two kinds of subdirectories in the examples folder. Lower case named directories contain one or a few simple, quick-to-run problems. Upper case named directories contain up to several complex scripts that illustrate a particular kind of simulation method or model. Some of these run for longer times, e.g. to measure a particular quantity.

Lists of both kinds of directories are given below.

## 11.1 Lowercase directories

| | |
|---|---|
| airebo | polyethylene with AIREBO potential |
| amoeba | small water and bio models with AMOEBA and HIPPO potentials |
| atm | Axilrod-Teller-Muto potential example |
| balance | dynamic load balancing, 2d system |
| body | body particles, 2d system |
| bpm | simulations of solid elastic/plastic deformation and fracture |
| cmap | CMAP 5-body contributions to CHARMM force field |
| colloid | big colloid particles in a small particle solvent, 2d system |
| comb | models using the COMB potential |
| controller | use of fix controller as a thermostat |
| coreshell | core/shell model using CORESHELL package |
| crack | crack propagation in a 2d solid |

continues on next page

Table 1 – continued from previous page

| | |
|---|---|
| deposit | deposit atoms and molecules on a surface |
| dipole | point dipolar particles, 2d system |
| dreiding | methanol via Dreiding FF |
| eim | NaCl using the EIM potential |
| ellipse | ellipsoidal particles in spherical solvent, 2d system |
| fire | examples for using minimization styles fire and quickmin |
| flow | Couette and Poiseuille flow in a 2d channel |
| friction | frictional contact of spherical asperities between 2d surfaces |
| gjf | examples for Gronbech-Jensen thermostats with large time step |
| granregion | use of fix wall/region/gran as boundary on granular particles |
| grid | use of commands which overlay grids on the simulation domain |
| hugoniostat | Hugoniostat shock dynamics |
| hyper | global and local hyperdynamics of diffusion on Pt surface |
| indent | spherical indenter into a 2d solid |
| kim | use of potentials from the OpenKIM Repository |
| mc | Monte Carlo features via fix gcmc, widom and other commands |
| mdi | use of the MDI package and MolSSI MDI code coupling library |
| meam | MEAM test for SiC and shear (same as shear examples) |
| melt | rapid melt of 3d LJ system |
| mesh | create_atoms mesh command examples |
| micelle | self-assembly of small lipid-like molecules into 2d bilayers |
| min | energy minimization of 2d LJ melt |
| mliap | examples for using several bundled ML-IAP potentials |
| msst | MSST shock dynamics |
| multi | multi neighboring for systems with large interaction disparities |
| nb3b | use of non-bonded 3-body harmonic pair style |
| neb | nudged elastic band (NEB) calculation for barrier finding |
| nemd | non-equilibrium MD of 2d sheared system |
| numdiff | get forces, virial, and Born matrix from numerical differences |
| obstacle | flow around two voids in a 2d channel |
| peptide | dynamics of a small solvated peptide chain (5-mer) |
| peri | Peridynamic model of cylinder impacted by indenter |
| pour | pouring of granular particles into a 3d box, then chute flow |
| prd | parallel replica dynamics of vacancy diffusion in bulk Si |
| python | using embedded Python in a LAMMPS input script |
| qeq | use of the QEQ package for charge equilibration |
| rdf-adf | computing radial and angle distribution functions for water |
| reaxff | RDX and TATB models and more using ReaxFF |
| replicate | use of replicate command |

continues on next page

Table 1 – continued from previous page

| | |
|---|---|
| rerun | use of rerun and read_dump commands |
| rheo | RHEO simulations of fluid flows and phase transitions |
| rigid | rigid bodies modeled as independent or coupled |
| shear | sideways shear applied to 2d solid, with and without a void |
| snap | NVE dynamics for BCC tantalum crystal using SNAP potential |
| srd | stochastic rotation dynamics (SRD) particles as solvent |
| steinhardt | Steinhardt-Nelson Q_l and W_l parameters using orien-torder/atom |
| streitz | use of Streitz/Mintmire potential with charge equilibra-tion |
| stress_vcm | removing binned rigid body motion from binned stress profile |
| tad | temperature-accelerated dynamics of vacancy diffusion in bulk Si |
| tersoff | regression test input for Tersoff potential variants |
| threebody | regression test input for a variety of manybody potentials |
| tracker | track interactions in LJ melt |
| triclinic | general triclinic simulation boxes versus orthogonal boxes |
| ttm | two-temperature model examples |
| vashishta | use of the Vashishta potential |
| voronoi | Voronoi tesselation via compute voronoi/atom command |
| wall | use of reflective walls with different stochastic models |
| yaml | demonstrates use of yaml thermo and dump styles |

Here is how you can run and visualize one of the sample problems:

```
cd indent
cp ../../src/lmp_linux .          # copy LAMMPS executable to this dir
lmp_linux -in in.indent           # run the problem
```

Running the simulation produces the files *dump.indent* and *log.lammps*. You can visualize the dump file of snapshots with a variety of third-party tools highlighted on the Visualization page of the LAMMPS website.

If you uncomment the *dump image* line(s) in the input script a series of JPG images will be produced by the run (assuming you built LAMMPS with JPG support; see the *Build_settings* page for details). These can be viewed individually or turned into a movie or animated by tools like ImageMagick or QuickTime or various Windows-based tools. See the *dump image* page for more details. E.g. this Imagemagick command would create a GIF file suitable for viewing in a browser.

```
% convert -loop 1 *.jpg foo.gif
```

## 11.2 Uppercase directories

| | |
|---|---|
| ASPHERE | various aspherical particle models, using ellipsoids, rigid bodies, line/triangle particles, etc |
| COUPLE | examples of how to use LAMMPS as a library |
| DIFFUSE | compute diffusion coefficients via several methods |
| ELASTIC | compute elastic constants at zero temperature |
| ELASTIC_T | compute elastic constants at finite temperature |
| HEAT | compute thermal conductivity for LJ and water via fix ehex |
| KAPPA | compute thermal conductivity via several methods |
| LEPTON | use of fix efield/lepton |
| MC-LOOP | using LAMMPS in a Monte Carlo mode to relax the energy of a system in a input script loop |
| PACKAGES | examples for specific packages and contributed commands |
| QUANTUM | how to use LAMMPS in tandem with several quantum codes via the MDI code coupling library |
| SPIN | examples for features of the SPIN package |
| UNITS | examples that run the same simulation in lj, real, metal units |
| VISCOSITY | compute viscosity via several methods |

Nearly all of these directories have README files which give more details on how to understand and use their contents.

The PACKAGES directory has a large number of subdirectories which correspond by name to specific packages. They contain scripts that illustrate how to use the command(s) provided in those packages. Many of the subdirectories have their own README files which give further instructions. See the *Packages_details* doc page for more info on specific packages.

# Part II

# Programmer Guide

CHAPTER

ONE

# LAMMPS LIBRARY INTERFACES

As described on the *library interface to LAMMPS* page, LAMMPS can be built as a library (static or shared), so that it can be called by another code, used in a *coupled manner* with other codes, or driven through a *Python script*. The LAMMPS standalone executable itself is essentially a thin wrapper on top of the LAMMPS library, which creates a LAMMPS instance, passes the input for processing to that instance, and then exits.

Most of the APIs described below are based on C language wrapper functions in the files `src/library.h` and `src/library.cpp`, but it is also possible to use C++ directly. The basic procedure is always the same: you create one or more instances of *LAMMPS*, pass commands as strings or from files to that LAMMPS instance to execute calculations, and/or call functions that read, manipulate, and update data from the active class instances inside LAMMPS to do analysis or perform operations that are not possible with existing input script commands.

**Thread-safety**

LAMMPS was initially not conceived as a thread-safe program, but over the years changes have been applied to replace operations that collide with creating multiple LAMMPS instances from multiple-threads of the same process with thread-safe alternatives. This primarily applies to the core LAMMPS code and less so on add-on packages, especially when those packages require additional code in the *lib* folder, interface LAMMPS to Fortran libraries, or the code uses static variables (like the COLVARS package).

Another major issue to deal with is to correctly handle MPI. Creating a LAMMPS instance requires passing an MPI communicator, or it assumes the `MPI_COMM_WORLD` communicator, which spans all MPI processor ranks. When creating multiple LAMMPS object instances from different threads, this communicator has to be different for each thread or else collisions can happen. Or it has to be guaranteed, that only one thread at a time is active. MPI communicators, however, are not a problem, if LAMMPS is compiled with the MPI STUBS library, which implies that there is no MPI communication and only 1 MPI rank.

## 1.1 LAMMPS C Library API

The C library interface is the most commonly used path to manage LAMMPS instances from a compiled code and it is the basis for the *Python* and *Fortran* modules. Almost all functions of the C language API require an argument containing a "handle" in the form of a `void *` type variable, which points to the location of a LAMMPS class instance.

The `library.h` header file by default does not include the `mpi.h` header file and thus hides the *lammps_open()* function which requires the declaration of the `MPI_comm` data type. This is only a problem when the communicator that would be passed is different from `MPI_COMM_WORLD`. Otherwise calling *lammps_open_no_mpi()* will work just as well. To make *lammps_open()* available, you need to compile the code with `-DLAMMPS_LIB_MPI` or add the line `#define LAMMPS_LIB_MPI` before `#include "library.h"`.

Please note the `mpi.h` file must usually be the same (and thus the MPI library in use) for the LAMMPS code and library and the calling code. The exception is when LAMMPS was compiled in serial mode using the `STUBS` MPI library. In

that case the calling code may be compiled with a different MPI library so long as *lammps_open_no_mpi()* is called to create a LAMMPS instance. In that case each MPI rank will run LAMMPS in serial mode.

**Errors versus exceptions**

If the LAMMPS executable encounters an error condition, it will abort after printing an error message. It does so by catching the exceptions that LAMMPS could throw. For a C library interface this is usually not desirable since the calling code might lack the ability to catch such exceptions. Thus, the library functions will catch those exceptions and return from the affected functions. The error status *can be queried* and an *error message retrieved*. This is, for example used by the *LAMMPS python module* and then a suitable Python exception is thrown.

**Using the C library interface as a plugin**

Rather than including the C library interface directly including the `library.h` header file and linking to the LAMMPS (static or shared) library at compile time, you can dynamically load LAMMPS at runtime, provided you compiled LAMMPS using a shared library or DLL. The `liblammpsplugin.h` header file and the `liblammpsplugin.c` C code in the `examples/COUPLE/plugin` folder for an interface to LAMMPS that is largely identical to the regular library interface, only that it will load a LAMMPS shared library file at runtime. This can be useful for applications where the interface to LAMMPS would be an optional feature or where you would like to load different versions of the LAMMPS library (e.g. an updated one) without replacing the executable. The *LAMMPS-GUI* is an example for such an application. It has a wrapper class that supports both modes and which is used can be changed at compile time.

**Warning**

No checks are made on the arguments of the function calls of the C library interface. *All* function arguments must be non-NULL unless *explicitly* allowed, and must point to consistent and valid data. Buffers for storing returned data must be allocated to a suitable size. Passing invalid or unsuitable information will likely cause crashes or corrupt data.

---

### 1.1.1 Creating or deleting a LAMMPS object

This section documents the following functions:

- *lammps_open()*
- *lammps_open_no_mpi()*
- *lammps_open_fortran()*
- *lammps_close()*
- *lammps_mpi_init()*
- *lammps_mpi_finalize()*
- *lammps_kokkos_finalize()*
- *lammps_python_finalize()*
- *lammps_plugin_finalize()*
- *lammps_error()*

The *lammps_open()* and *lammps_open_no_mpi()* functions are used to create and initialize a LAMMPS() instance. They return a reference to this instance as a void * pointer to be used as the "handle" argument in subsequent function calls until that instance is destroyed by calling *lammps_close()*. Here is a simple example demonstrating its use:

```
#include "library.h"
#include <stdio.h>

int main(int argc, char **argv)
{
  void *handle;
  int version;
  const char *lmpargv[] = { "liblammps", "-log", "none"};
  int lmpargc = sizeof(lmpargv)/sizeof(const char *);

  /* create LAMMPS instance */
  handle = lammps_open_no_mpi(lmpargc, (char **)lmpargv, NULL);
  if (handle == NULL) {
    printf("LAMMPS initialization failed");
    lammps_mpi_finalize();
    return 1;
  }

  /* get and print numerical version code */
  version = lammps_version(handle);
  printf("LAMMPS Version: %d\n",version);

  /* delete LAMMPS instance and shut down MPI */
  lammps_close(handle);
  lammps_mpi_finalize();
  return 0;
}
```

The LAMMPS library uses the MPI library it was compiled with and will either run on all processors in the MPI_COMM_WORLD communicator or on the set of processors in the communicator passed as the comm argument of *lammps_open()*. This means the calling code can run LAMMPS on all or a subset of processors. For example, a wrapper code might decide to alternate between LAMMPS and another code, allowing them both to run on all the processors. Or it might allocate part of the processors to LAMMPS and the rest to the other code by creating a custom communicator with MPI_Comm_split() and running both codes concurrently before syncing them up periodically. Or it might instantiate multiple instances of LAMMPS to perform different calculations and either alternate between them, run them concurrently on split communicators, or run them one after the other. The *lammps_open()* function may be called multiple times for this latter purpose.

The *lammps_close()* function is used to shut down the *LAMMPS* class pointed to by the handle passed as an argument and free all its memory. This has to be called for every instance created with one of the *lammps_open()* functions. It will, however, **not** call MPI_Finalize(), since that may only be called once. See *lammps_mpi_finalize()* for an alternative to invoking MPI_Finalize() explicitly from the calling program.

---

void ***lammps_open**(int argc, char **argv, MPI_Comm comm, void **ptr)

Create instance of the *LAMMPS* class and return pointer to it.

The *lammps_open()* function creates a new *LAMMPS* class instance while passing in a list of strings as if they were *command-line arguments* for the LAMMPS executable, and an MPI communicator for LAMMPS to run under. Since the list of arguments is **exactly** as when called from the command-line, the first argument would be the name of the executable and thus is otherwise ignored. However `argc` may be set to 0 and then `argv` may be `NULL`. If MPI is not yet initialized, `MPI_Init()` will be called during creation of the LAMMPS class instance.

If for some reason the creation or initialization of the LAMMPS instance fails a null pointer is returned.

Changed in version 18Sep2020: This function now has the pointer to the created LAMMPS class instance as return value. For backward compatibility it is still possible to provide the address of a pointer variable as final argument *ptr*.

Deprecated since version 18Sep2020: The *ptr* argument will be removed in a future release of LAMMPS. It should be set to `NULL` instead.

***See also***

*lammps_open_no_mpi()*, *lammps_open_fortran()*

**Note**

This function is **only** declared when the code using the LAMMPS `library.h` include file is compiled with `-DLAMMPS_LIB_MPI`, or contains a `#define LAMMPS_LIB_MPI 1` statement before `#include "library.h"`. Otherwise you can only use the *lammps_open_no_mpi()* or *lammps_open_fortran()* functions.

**Parameters**

- **argc** – number of command-line arguments
- **argv** – list of command-line argument strings
- **comm** – MPI communicator for this *LAMMPS* instance
- **ptr** – pointer to a void pointer variable which serves as a handle; may be `NULL`

**Returns**

pointer to new *LAMMPS* instance cast to `void *`

---

void ***lammps_open_no_mpi**(int argc, char **argv, void **ptr)

Variant of `lammps_open()` that implicitly uses `MPI_COMM_WORLD`.

This function is a version of *lammps_open()*, that is missing the MPI communicator argument. It will use `MPI_COMM_WORLD` instead. The type and purpose of arguments and return value are otherwise the same.

Outside of the convenience, this function is useful, when the LAMMPS library was compiled in serial mode, but the calling code runs in parallel and the `MPI_Comm` data type of the STUBS library would not be compatible with that of the calling code.

If for some reason the creation or initialization of the LAMMPS instance fails a null pointer is returned.

Changed in version 18Sep2020: This function now has the pointer to the created LAMMPS class instance as return value. For backward compatibility it is still possible to provide the address of a pointer variable as final argument *ptr*.

Deprecated since version 18Sep2020: The *ptr* argument will be removed in a future release of LAMMPS. It should be set to `NULL` instead.

*See also*
*lammps_open()*, *lammps_open_fortran()*

**Parameters**

- **argc** – number of command-line arguments
- **argv** – list of command-line argument strings
- **ptr** – pointer to a void pointer variable which serves as a handle; may be NULL

**Returns**
pointer to new *LAMMPS* instance cast to `void *`

---

void ***lammps_open_fortran**(int argc, char **argv, int f_comm)

Variant of `lammps_open()` using a Fortran MPI communicator.

Added in version 18Sep2020.

This function is a version of *lammps_open()*, that uses an integer for the MPI communicator as the MPI Fortran interface does. It is used in the *lammps()* constructor of the LAMMPS Fortran module. Internally it converts the *f_comm* argument into a C-style MPI communicator with `MPI_Comm_f2c()` and then calls *lammps_open()*.

If for some reason the creation or initialization of the LAMMPS instance fails a null pointer is returned.

*See also*
*lammps_open_fortran()*, *lammps_open_no_mpi()*

**Parameters**

- **argc** – number of command-line arguments
- **argv** – list of command-line argument strings
- **f_comm** – Fortran style MPI communicator for this *LAMMPS* instance

**Returns**
pointer to new *LAMMPS* instance cast to `void *`

---

void **lammps_close**(void *handle)

Delete a *LAMMPS* instance created by lammps_open() or its variants.

This function deletes the LAMMPS class instance pointed to by `handle` that was created by one of the *lammps_open()* variants. It does **not** call `MPI_Finalize()` to allow creating and deleting multiple LAMMPS instances concurrently or sequentially. See *lammps_mpi_finalize()* for a function performing this operation.

**Parameters**
**handle** – pointer to a previously created *LAMMPS* instance

---

void **lammps_mpi_init**()

Ensure the MPI environment is initialized.

Added in version 18Sep2020.

The MPI standard requires that any MPI application must call `MPI_Init()` exactly once before performing any other MPI function calls. This function checks, whether MPI is already initialized and calls `MPI_Init()` in case it is not.

---

void **lammps_mpi_finalize**()

Shut down the MPI infrastructure.

Added in version 18Sep2020.

The MPI standard requires that any MPI application calls `MPI_Finalize()` before exiting. Even if a calling program does not do any MPI calls, MPI is still initialized internally to avoid errors accessing any MPI functions. This function should then be called right before exiting the program to wait until all (parallel) tasks are completed and then MPI is cleanly shut down. After calling this function no more MPI calls may be made.

***See also***

*lammps_kokkos_finalize()*, *lammps_python_finalize()*, *lammps_plugin_finalize()*

---

void **lammps_kokkos_finalize**()

Shut down the Kokkos library environment.

Added in version 2Jul2021.

The Kokkos library may only be initialized once during the execution of a process. This is done automatically the first time Kokkos functionality is used. This requires that the Kokkos environment must be explicitly shut down after any LAMMPS instance using it is closed (to release associated resources). After calling this function no Kokkos functionality may be used.

***See also***

*lammps_mpi_finalize()*, *lammps_python_finalize()*, *lammps_plugin_finalize()*

---

void **lammps_python_finalize**()

Clear the embedded Python environment

Added in version 20Sep2021.

This function resets and clears an embedded Python environment by calling the Py_Finalize() function of the embedded Python library, if enabled. This call would free up all allocated resources and release loaded shared objects.

However, this is **not** done when a LAMMPS instance is deleted because a) LAMMPS may have been used through the Python module and thus the Python interpreter is external and not embedded into LAMMPS and therefore may not be reset by LAMMPS b) some Python modules and extensions, most notably NumPy, are not compatible with being initialized multiple times, which would happen if additional LAMMPS instances using Python would be created *after* after calling Py_Finalize().

This function can be called to explicitly clear the Python environment in case it is safe to do so.

***See also***
*lammps_mpi_finalize()*, *lammps_kokkos_finalize()*, *lammps_plugin_finalize()*

---

void **lammps_plugin_finalize**()

Unload all plugins and release the corresponding DSO handles

Added in version 12Jun2025.

This function clears the list of all loaded plugins and closes the corresponding DSO handles and releases the imported executable code.

However, this is **not** done when a LAMMPS instance is deleted because plugins and their shared objects are global properties.

This function can be called to explicitly clear out all loaded plugins in case it is safe to do so.

***See also***
*lammps_mpi_finalize()*, *lammps_kokkos_finalize()*, *lammps_python_finalize()*

---

void **lammps_error**(void *handle, int error_type, const char *error_text)

Call a *LAMMPS* Error class function

Added in version 3Nov2022.

This function is a wrapper around functions in the `Error` to print an error message and then stop LAMMPS.

The *error_type* parameter selects which function to call. It is a sum of constants from `_LMP_ERROR_CONST`. If the value does not match any valid combination of constants a warning is printed and the function returns.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **error_type** – parameter to select function in the Error class
- **error_text** – error message

### 1.1.2 Executing commands

This section documents the following functions:

- *lammps_file()*
- *lammps_command()*
- *lammps_commands_list()*
- *lammps_commands_string()*
- *lammps_expand()*

---

Once a LAMMPS instance is created, there are multiple ways to "drive" a simulation. In most cases it is easiest to process single or multiple LAMMPS commands like in an input file. This can be done through reading a file or passing single commands or lists of commands or blocks of commands with the following functions.

Via these functions, the calling code can have LAMMPS act on a series of *input file commands* that are either read from a file or passed as strings. For example, this allows setup of a problem from an input script, and then running it in stages while performing other operations in between or concurrently. The caller can interleave the LAMMPS function calls with operations it performs, such as calls to extract information from or set information within LAMMPS, or calls to another code's library.

Just as with *input script parsing* comments can be included in the file or strings, and expansion of variables with `${name}` or `$(expression)` syntax is performed. Below is a short example using some of these functions.

```
/* define to make the otherwise hidden prototype for "lammps_open()" visible */
#define LAMMPS_LIB_MPI
#include "library.h"

#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  void *handle;
  int i;

  MPI_Init(&argc, &argv);
  handle = lammps_open(0, NULL, MPI_COMM_WORLD, NULL);
  lammps_file(handle,"in.sysinit");
  lammps_command(handle,"run 1000 post no");

  for (i=0; i < 100; ++i) {
    lammps_commands_string(handle,"run 100 pre no post no\n"
                                  "print 'PE = $(pe)'\n"
                                  "print 'KE = $(ke)'\n");
  }
  lammps_close(handle);
  MPI_Finalize();
  return 0;
}
```

---

void **lammps_file**(void *handle, const char *file)

Process *LAMMPS* input from a file.

This function processes commands in the file pointed to by *filename* line by line and thus functions very similar to the *include* command. The function returns when the end of the file is reached and the commands have completed.

The actual work is done by the functions *Input::file(const char *)* and *Input::file()*.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **filename** – name of a file with *LAMMPS* input

---

char *lammps_command(void *handle, const char *cmd)

Process a single *LAMMPS* input command from a string.

This function tells LAMMPS to execute the single command in the string *cmd*. The entire string is considered as command and need not have a (final) newline character. Newline characters in the body of the string, however, will be treated as part of the command and will **not** start a second command. The function *lammps_commands_string()* processes a string with multiple command-lines.

The function returns the name of the command on success or NULL when passing a string without a command.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **cmd** – string with a single *LAMMPS* command

**Returns**

string with parsed command name or NULL

---

void lammps_commands_list(void *handle, int ncmd, const char **cmds)

Process multiple *LAMMPS* input commands from list of strings.

This function processes multiple commands from a list of strings by first concatenating the individual strings in *cmds* into a single string, inserting newline characters as needed. The combined string is passed to *lammps_commands_string()* for processing.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **ncmd** – number of lines in *cmds*
- **cmds** – list of strings with *LAMMPS* commands

---

void lammps_commands_string(void *handle, const char *str)

Process a block of *LAMMPS* input commands from a single string.

This function processes a multi-line string similar to a block of commands from a file. The string may have multiple lines (separated by newline characters) and also single commands may be distributed over multiple lines with continuation characters ('&'). Those lines are combined by removing the '&' and the following newline character. After this processing the string is handed to LAMMPS for parsing and executing.

Added in version 21Nov2023: The command is now able to process long strings with triple quotes and loops using *jump SELF <label>*.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **str** – string with block of *LAMMPS* input commands

---

char *lammps_expand(void *handle, const char *line)

expand a single *LAMMPS* input line from a string.

This function tells LAMMPS to expand the string in *cmd* like it would process an input line fed to *lammps_command()* **without** executing it. The *entire* string is considered as input and need not have a (final) newline character. Newline characters in the body of the string, however, will be treated as part of the command and will **not** start a second command.

The function returns the expanded string in a new string buffer that must be freed with *lammps_free()* after use to avoid a memory leak.

***See also***

*lammps_eval()*

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **line** – string with a single *LAMMPS* input line

**Returns**

string with expanded line

### 1.1.3 System properties

This section documents the following functions:

- *lammps_get_natoms()*
- *lammps_get_thermo()*
- *lammps_last_thermo()*
- *lammps_extract_box()*
- *lammps_reset_box()*
- *lammps_memory_usage()*
- *lammps_get_mpi_comm()*
- *lammps_extract_setting()*
- *lammps_extract_global_datatype()*
- *lammps_extract_global()*
- *lammps_extract_pair_dimension()*
- *lammps_extract_pair()*
- *lammps_map_atom()*

---

The library interface allows the extraction of different kinds of information about the active simulation instance and also - in some cases - to apply modifications to it. This enables combining of a LAMMPS simulation with other processing and simulation methods computed by the calling code, or by another code that is coupled to LAMMPS via the library interface. In some cases the data returned is direct reference to the original data inside LAMMPS, cast to a void pointer. In that case the data needs to be cast to a suitable pointer for the calling program to access it, and you may need to know the correct dimensions and lengths. This also means you can directly change those value(s) from the calling program (e.g., to modify atom positions). Of course, changing values should be done with care. When accessing per-atom data,

please note that these data are the per-processor **local** data and are indexed accordingly. Per-atom data can change sizes and ordering at every neighbor list rebuild or atom sort event as atoms migrate between subdomains and processors.

```
#include "library.h"
#include <stdio.h>

int main(int argc, char **argv)
{
  void *handle;
  int i;

  handle = lammps_open_no_mpi(0, NULL, NULL);
  lammps_file(handle,"in.sysinit");
  printf("Running a simulation with %g atoms.\n",
         lammps_get_natoms(handle));

  printf(" %d local and %d ghost atoms. %d atom types\n",
         lammps_extract_setting(handle,"nlocal"),
         lammps_extract_setting(handle,"nghost"),
         lammps_extract_setting(handle,"ntypes"));

  double  *dt = (double *)lammps_extract_global(handle,"dt");
  printf("Changing timestep from %g to 0.5\n", *dt);
  *dt = 0.5;

  lammps_command(handle,"run 1000 post no");

  for (i=0; i < 10; ++i) {
    lammps_command(handle,"run 100 pre no post no");
    printf("PE = %g\nKE = %g\n",
           lammps_get_thermo(handle,"pe"),
           lammps_get_thermo(handle,"ke"));
  }
  lammps_close(handle);
  return 0;
}
```

---

double **lammps_get_natoms**(void *handle)

Return the total number of atoms in the system.

This number may be very large when running large simulations across multiple processes. Depending on compile time choices, LAMMPS may be using either 32-bit or a 64-bit integer to store this number. For portability this function returns thus a double precision floating point number, which can represent up to a 53-bit signed integer exactly ($\approx 10^{16}$).

As an alternative, you can use *lammps_extract_global()* and cast the resulting pointer to an integer pointer of the correct size and dereference it. The size of that integer (in bytes) can be queried by calling *lammps_extract_setting()* to return the size of a `bigint` integer.

Changed in version 18Sep2020: The type of the return value was changed from `int` to `double` to accommodate reporting atom counts for larger systems that would overflow a 32-bit int without having to depend on a 64-bit bit integer type definition.

**Parameters**
**handle** – pointer to a previously created *LAMMPS* instance

**Returns**
total number of atoms or 0 if value is too large

---

double **lammps_get_thermo**(void *handle, const char *keyword)

Evaluate a thermo keyword.

This function returns the current value of a *thermo keyword*. Unlike *lammps_extract_global()* it does not give access to the storage of the desired data but returns its value as a double, so it can also return information that is computed on-the-fly. Use *lammps_last_thermo()* to get access to the cached data from the last thermo output.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **keyword** – string with the name of the thermo keyword

**Returns**
value of the requested thermo property or 0.0

---

void ***lammps_last_thermo**(void *handle, const char *what, int index)

Access cached data from last thermo output

Added in version 15Jun2023.

This function provides access to cached data from the last thermo output. This differs from *lammps_get_thermo()* in that it does **not** trigger an evaluation. Instead it provides direct access to a read-only location of the last thermo output data and the corresponding keyword strings. How to handle the return value depends on the value of the *what* argument string. When accessing the data from a concurrent thread while LAMMPS is running, the cache needs to be locked first and then unlocked after the data is obtained, so that the data is not corrupted while reading in case LAMMPS wants to update it at the same time. Outside of a run, the lock/unlock calls have no effect.

| Value of *what* | Description of return value | Data type | Uses index |
|---|---|---|---|
| setup | 1 if setup is not completed and thus thermo data invalid, 0 otherwise | pointer to int | no |
| line | line number (0-based) of current line in current file or buffer | pointer to int | no |
| imagename | file name of the last *dump image* file written | pointer to 0-terminated const char array | no |
| step | timestep when the last thermo output was generated or -1 | pointer to bigint | no |
| num | number of fields in thermo output | pointer to int | no |
| keyword | column keyword for thermo output | pointer to 0-terminated const char array | yes |
| type | data type of thermo output column; see `_LMP_DATATYPE_CONST` | pointer to int | yes |
| data | actual field data for column | pointer to int, int64_t or double | yes |
| lock | acquires lock to thermo data cache | NULL pointer | no |
| unlock | releases lock to thermo data cache | NULL pointer | no |

**Note**

The *type* property points to a static location that is reassigned with every call, so the returned pointer should be recast, dereferenced, and assigned immediately. Otherwise, its value may be changed with the next invocation of the function.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **what** – string with the kind of data requested
- **index** – integer with index into data arrays, ignored for scalar data

**Returns**

pointer to location of requested data cast to void or NULL

---

void **lammps_extract_box**(void *handle, double *boxlo, double *boxhi, double *xy, double *yz, double *xz, int *pflags, int *boxflag)

Extract simulation box parameters.

This function (re-)initializes the simulation box and boundary information and then assign the designated data to the locations in the pointers passed as arguments. Any argument (except the first) may be a NULL pointer and then will not be assigned.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **boxlo** – pointer to 3 doubles where the lower box boundary is stored
- **boxhi** – pointer to 3 doubles where the upper box boundary is stored
- **xy** – pointer to a double where the xy tilt factor is stored

- **yz** – pointer to a double where the yz tilt factor is stored
- **xz** – pointer to a double where the xz tilt factor is stored
- **pflags** – pointer to 3 ints, set to 1 for periodic boundaries and 0 for non-periodic
- **boxflag** – pointer to an int, which is set to 1 if the box will be changed during a simulation by a fix and 0 if not.

---

void **lammps_reset_box**(void *handle, double *boxlo, double *boxhi, double xy, double yz, double xz)

Reset simulation box parameters.

This function sets the simulation box dimensions (upper and lower bounds and tilt factors) from the provided data and then re-initializes the box information and all derived settings. It may only be called before atoms are created.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **boxlo** – pointer to 3 doubles containing the lower box boundary
- **boxhi** – pointer to 3 doubles containing the upper box boundary
- **xy** – xy tilt factor
- **yz** – yz tilt factor
- **xz** – xz tilt factor

---

void **lammps_memory_usage**(void *handle, double *meminfo)

Get memory usage information

Added in version 18Sep2020.

This function will retrieve memory usage information for the current LAMMPS instance or process. The *meminfo* buffer will be filled with 3 different numbers (if supported by the operating system). The first is the tally (in MBytes) of all large memory allocations made by LAMMPS. This is a lower boundary of how much memory is requested and does not account for memory allocated on the stack or allocations via `new`. The second number is the current memory allocation of the current process as returned by a memory allocation reporting in the system library. The third number is the maximum amount of RAM (not swap) used by the process so far. If any of the two latter parameters is not supported by the operating system it will be set to zero.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **meminfo** – buffer with space for at least 3 double to store data in.

---

int **lammps_get_mpi_comm**(void *handle)

Return current *LAMMPS* world communicator as integer

Added in version 18Sep2020.

This will take the LAMMPS "world" communicator and convert it to an integer using `MPI_Comm_c2f()`, so it is equivalent to the corresponding MPI communicator in Fortran. This way it can be safely passed around between different programming languages. To convert it to the C language representation use `MPI_Comm_f2c()`.

If LAMMPS was compiled with MPI_STUBS, this function returns -1.

***See also***
*lammps_open_fortran()*

**Parameters**
**handle** – pointer to a previously created *LAMMPS* instance

**Returns**
Fortran representation of the *LAMMPS* world communicator

---

int **lammps_extract_setting**(void *handle, const char *keyword)

Query *LAMMPS* about global settings.

This function will retrieve or compute global properties. In contrast to *lammps_get_thermo()* this function returns an `int`. The following tables list the currently supported keyword. If a keyword is not recognized, the function returns -1. The integer sizes functions may be called without a valid LAMMPS object handle (it is ignored).

- *Integer sizes*
- *Image masks*
- *System status*
- *System sizes*
- *Neighbor list settings*
- *Atom style flags*
- *Thermo settings*

**Integer sizes**

| Keyword | Description / Return value |
|---|---|
| bigint | size of the `bigint` integer type, 4 or 8 bytes. Set at *compile time*. |
| tagint | size of the `tagint` integer type, 4 or 8 bytes. Set at *compile time*. |
| imageint | size of the `imageint` integer type, 4 or 8 bytes. Set at *compile time*. |
| MAX_GROUP | size of the bitmask for groups in bits, should be 32. Currently hard coded. |

**Image masks**

These settings are related to how LAMMPS stores and interprets periodic images. The values are used internally by the *Fortran interface* and are not likely to be useful to users.

| Keyword | Description / Return value |
|---|---|
| IMGMASK | Bit-mask used to convert image flags to a single integer |
| IMGMAX | Maximum allowed image number for a particular atom |
| IMGBITS | Bits used in image counts |
| IMG2BITS | Second bitmask used in image counts |

**System status**

| Keyword | Description / Return value |
|---|---|
| dimension | Number of dimensions: 2 or 3. See *dimension command*. |
| box_exist | 1 if the simulation box is defined, 0 if not. See *create_box command*. |
| kokkos_active | 1 if the KOKKOS package is compiled in **and** activated, 0 if not. See *KOKKOS package*. |
| kokkos_nthreads | Number of Kokkos threads per MPI process, 0 if Kokkos is not active. See *KOKKOS package*. |
| kokkos_ngpus | Number of Kokkos gpus per physical node, 0 if Kokkos is not active or no GPU support. See *KOKKOS package*. |
| nthreads | Number of requested OpenMP threads per MPI process for LAMMPS' execution |
| newton_bond | 1 if Newton's 3rd law is applied to bonded interactions, 0 if not. |
| newton_pair | 1 if Newton's 3rd law is applied to non-bonded interactions, 0 if not. |
| triclinic | 1 if the the simulation box is triclinic, 0 if orthogonal. See *change_box command*. |

**Communication status**

| Keyword | Description / Return value |
|---|---|
| universe_rank | MPI rank on LAMMPS' universe communicator (0 <= universe_rank < universe_size) |
| universe_size | Number of ranks on LAMMPS' universe communicator (world_size <= universe_size) |
| world_rank | MPI rank on LAMMPS' world communicator (0 <= world_rank < world_size, = comm->me) |
| world_size | Number of ranks on LAMMPS' world communicator (aka comm->nprocs) |
| comm_style | communication style (0 = BRICK, 1 = TILED) |
| comm_layout | communication layout (0 = LAYOUT_UNIFORM, 1 = LAYOUT_NONUNIFORM, 2 = LAYOUT_TILED) |
| comm_mode | communication mode (0 = SINGLE, 1 = MULTI) |
| ghost_velocity | whether velocities are communicated for ghost atoms (0 = no, 1 = yes) |

**System sizes**

| Keyword | Description / Return value |
|---|---|
| nlocal | number of "owned" atoms of the current MPI rank. |
| nghost | number of "ghost" atoms of the current MPI rank. |
| nall | number of all "owned" and "ghost" atoms of the current MPI rank. |
| nmax | maximum of nlocal+nghost across all MPI ranks (for per-atom data array size). |
| ntypes | number of atom types |
| nbondtypes | number of bond types |
| nangletypes | number of angle types |
| ndihedraltypes | number of dihedral types |
| nimpropertypes | number of improper types |
| bond_per_atom | size of per-atom bond data arrays |
| angle_per_atom | size of per-atom angle data arrays |
| dihedral_per_atom | size of per-atom dihedral data arrays |
| improper_per_atom | size of per-atom improper data arrays |
| maxspecial | size of per-atom special data array |
| nellipsoids | number of atoms that have ellipsoid data |
| nlines | number of atoms that have line data (see *pair style line/lj*) |
| ntris | number of atoms that have triangle data (see *pair style tri/lj*) |
| nbodies | number of atoms that have body data (see *the Body particle HowTo*) |

**Neighbor list settings**

| neigh_every | neighbor lists are rebuild every this many steps |
|---|---|
| neigh_delay | neighbor lists are rebuild delayed this many steps |
| neigh_dist_check | 0 if always rebuild, 1 rebuild after 1/2 skin |
| neigh_ago | neighbor lists were rebuilt this many steps ago |
| nbondlist | number of entries in bondlist (get list with *lammps_extract_global()*) |
| nanglelist | number of entries in anglelist (get list with *lammps_extract_global()*) |
| ndihedrallist | number of entries in dihedrallist (get list with *lammps_extract_global()*) |
| nimproperlist | number of entries in improperlist (get list with *lammps_extract_global()*) |

**Atom style flags**

| Keyword | Description / Return value |
|---|---|
| molecule_flag | 1 if the atom style includes molecular topology data. See *atom_style command*. |
| q_flag | 1 if the atom style includes point charges. See *atom_style command*. |
| mu_flag | 1 if the atom style includes point dipoles. See *atom_style command*. |
| rmass_flag | 1 if the atom style includes per-atom masses, 0 if there are per-type masses. See *atom_style command*. |
| radius_flag | 1 if the atom style includes a per-atom radius. See *atom_style command*. |
| body_flag | 1 if the atom style describes body particles. See *atom_style command*. |
| ellipsoid_flag | 1 if the atom style describes extended particles that may be ellipsoidal. See *atom_style command*. |
| line_flag | 1 if the atom style describes line particles. See *atom_style command*. |
| tri_flag | 1 if the atom style describes tri particles. See *atom_style command*. |
| omega_flag | 1 if the atom style can store per-atom rotational velocities. See *atom_style command*. |
| torque_flag | 1 if the atom style can store per-atom torques. See *atom_style command*. |
| angmom_flag | 1 if the atom style can store per-atom angular momentum. See *atom_style command*. |

**Thermo settings**

| Keyword | Description / Return value |
|---|---|
| thermo_every | The current interval of thermo output. See *thermo command*. |
| thermo_norm | 1 if the thermo output is normalized. See *thermo_modify command*. |

***See also***

*lammps_extract_global()*

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **keyword** – string with the name of the thermo keyword

**Returns**

value of the queried setting or -1 if unknown

---

int **lammps_extract_global_datatype**(void *handle, const char *name)

Get data type of internal global *LAMMPS* variables or arrays.

Added in version 18Sep2020.

This function returns an integer that encodes the data type of the global property with the specified name. See `_LMP_DATATYPE_CONST` for valid values. Callers of *lammps_extract_global()* can use this information to then decide how to cast the `void *` pointer and access the data.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance (unused)
- **name** – string with the name of the extracted property

**Returns**

integer constant encoding the data type of the property or -1 if not found.

void *__lammps_extract_global__(void *handle, const char *name)

Get pointer to internal global *LAMMPS* variables or arrays.

This function returns a pointer to the location of some global property stored in one of the constituent classes of a LAMMPS instance. The returned pointer is cast to `void *` and needs to be cast to a pointer of the type that the entity represents. The pointers returned by this function are generally persistent; therefore it is not necessary to call the function again, unless a *clear command* command is issued which wipes out and recreates the contents of the `LAMMPS` class.

Please also see `lammps_extract_setting()`, `lammps_get_thermo()`, and `lammps_extract_box()`.

The following tables list the supported names, their data types, length of the data area, and a short description. The data type can also be queried through calling `lammps_extract_global_datatype()`. The `bigint` type may be defined to be either an `int` or an `int64_t`. This is set at *compile time* of the LAMMPS library and can be queried through calling `lammps_extract_setting()`. The function `lammps_extract_global_datatype()` will directly report the "native" data type. The following tables are provided:

- *Timestep settings*
- *Simulation box settings*
- *System property settings*
- *Neighbor topology data*
- *Energy and virial tally settings*
- *Git revision and version settings*
- *Unit settings*

**Timestep settings**

| Name | Type | Length | Description |
|---|---|---|---|
| dt | double | 1 | length of the time step. See *timestep command*. |
| ntimestep | bigint | 1 | current time step number. See *reset_timestep command*. |
| atime | double | 1 | accumulated simulation time in time units. |
| atimestep | bigint | 1 | the number of the timestep when "atime" was last updated. |
| respa_levels | int | 1 | $N_{respa}$ = number of r-RESPA levels. See *run_style command*. |
| respa_dt | double | $N_{respa}$ | length of the time steps with r-RESPA. See *run_style command*. |

**Simulation box settings**

| Name | Type | Length | Description |
|---|---|---|---|
| boxlo | double | 3 | lower box boundaries in x-, y-, and z-direction; see *create_box command*. |
| boxhi | double | 3 | lower box boundaries in x-, y-, and z-direction; see *create_box command*. |
| boxxlo | double | 1 | lower box boundary in x-direction; see *create_box command*. |
| boxxhi | double | 1 | upper box boundary in x-direction; see *create_box command*. |
| boxylo | double | 1 | lower box boundary in y-direction; see *create_box command*. |
| boxyhi | double | 1 | upper box boundary in y-direction; see *create_box command*. |
| boxzlo | double | 1 | lower box boundary in z-direction; see *create_box command*. |
| boxzhi | double | 1 | upper box boundary in z-direction; see *create_box command*. |
| sublo | double | 3 | subbox lower boundaries in x-, y-, and z-direction |
| subhi | double | 3 | subbox upper boundaries in x-, y-, and z-direction |
| sublo_lambda | double | 3 | subbox lower boundaries in fractional coordinates (for triclinic cells only) |
| subhi_lambda | double | 3 | subbox upper boundaries in fractional coordinates (for triclinic cells only) |
| periodicity | int | 3 | 0 if non-periodic, 1 if periodic for x, y, and z; see *boundary command*. |
| triclinic | int | 1 | 1 if box is triclinic, 0 if orthogonal; see *change_box command*. |
| xy | double | 1 | triclinic tilt factor; see *Triclinic (non-orthogonal) simulation boxes*. |
| yz | double | 1 | triclinic tilt factor; see *Triclinic (non-orthogonal) simulation boxes*. |
| xz | double | 1 | triclinic tilt factor; see *Triclinic (non-orthogonal) simulation boxes*. |
| xlattice | double | 1 | lattice spacing in x-direction; see *lattice command*. |
| ylattice | double | 1 | lattice spacing in y-direction; see *lattice command*. |
| zlattice | double | 1 | lattice spacing in z-direction; see *lattice command*. |
| procgrid | int | 3 | processor count in x-, y-, and z- direction; see *processors command*. |

**System property settings**

| Name | Type | Length | Description |
|---|---|---|---|
| natoms | bigint | 1 | total number of atoms in the simulation. |
| nbonds | bigint | 1 | total number of bonds in the simulation. |
| nangles | bigint | 1 | total number of angles in the simulation. |
| ndihedrals | bigint | 1 | total number of dihedrals in the simulation. |
| nimpropers | bigint | 1 | total number of impropers in the simulation. |
| nlocal | int | 1 | number of "owned" atoms of the current MPI rank. |
| nghost | int | 1 | number of "ghost" atoms of the current MPI rank. |
| nmax | int | 1 | maximum of nlocal+nghost across all MPI ranks (for per-atom data array size). |
| ntypes | int | 1 | number of atom types |
| special_lj | double | 4 | special *pair weighting factors* for LJ interactions (first element is always 1.0) |
| special_coul | double | 4 | special *pair weighting factors* for Coulomb interactions (first element is always 1.0) |
| map_style | int | 1 | *atom map setting*: 0 = none, 1 = array, 2 = hash, 3 = yes |
| map_tag_max | int/bigint | 1 | largest atom ID that can be mapped to a local index (bigint with -DLAMMPS_BIGBIG) |
| sametag | int | variable | index of next local atom with the same ID in ascending order. -1 signals end. |
| sortfreq | int | 1 | frequency of atom sorting. 0 means sorting is off. |
| nextsort | bigint | 1 | timestep when atoms are sorted next |
| q_flag | int | 1 | **deprecated**. Use `lammps_extract_setting()` instead. |
| atom_style | char * | 1 | string with the current atom style. |
| pair_style | char * | 1 | string with the current pair style. |
| bond_style | char * | 1 | string with the current bond style. |
| angle_style | char * | 1 | string with the current angle style. |
| dihedral_style | char * | 1 | string with the current dihedral style. |
| improper_style | char * | 1 | string with the current improper style. |
| kspace_style | char * | 1 | string with the current KSpace style. |

**Neighbor topology data**

Get length of lists with *lammps_extract_setting()*.

| Name | Type | Length | Description |
|---|---|---|---|
| neigh_skin | double | 1 | neighbor list skin |
| neigh_cutmin | double | 1 | minimum neighbor cutoff across all type pairs |
| neigh_cutmax | double | 1 | maximum neighbor cutoff across all type pairs |
| neigh_bondlist | 2d int | nbondlist | list of bonds (atom1, atom2, type) |
| neigh_anglelist | 2d int | nanglelist | list of angles (atom1, atom2, atom3, type) |
| neigh_dihedrallist | 2d int | ndihedrallist | list of dihedrals (atom1, atom2, atom3, atom4, type) |
| neigh_improperlist | 2d int | nimproperlist | list of impropers (atom1, atom2, atom3, atom4, type) |

**Energy and virial tally settings**

| Name | Type | Length | Description |
|---|---|---|---|
| eflag_global | bigint | 1 | timestep global energy is tallied on |
| eflag_atom | bigint | 1 | timestep per-atom energy is tallied on |
| vflag_global | bigint | 1 | timestep global virial is tallied on |
| vflag_atom | bigint | 1 | timestep per-atom virial is tallied on |

**Git revision and version settings**

| Name | Type | Length | Description |
|---|---|---|---|
| git_commit | const char * | 1 | Git commit hash for the LAMMPS version. |
| git_branch | const char * | 1 | Git branch for the LAMMPS version. |
| git_descriptor | const char * | 1 | Combined descriptor for the git revision |
| lammps_version | const char * | 1 | LAMMPS version string. |

**Unit settings**

| Name | Type | Length | Description |
|---|---|---|---|
| units | char * | 1 | string with the current unit style. See *units command*. |
| boltz | double | 1 | value of the "boltz" constant. See *units command*. |
| hplanck | double | 1 | value of the "hplanck" constant. See *units command*. |
| mvv2e | double | 1 | factor to convert $\frac{1}{2}mv^2$ for a particle to the current energy unit; See *units command*. |
| ftm2v | double | 1 | (description missing) See *units command*. |
| mv2d | double | 1 | (description missing) See *units command*. |
| nktv2p | double | 1 | (description missing) See *units command*. |
| qqr2e | double | 1 | factor to convert $\frac{q_i q_j}{r}$ to energy units; See *units command*. |
| qe2f | double | 1 | (description missing) See *units command*. |
| vxmu2f | double | 1 | (description missing) See *units command*. |
| xxt2kmu | double | 1 | (description missing) See *units command*. |
| dielectric | double | 1 | value of the dielectric constant. See *dielectric command*. |
| qqrd2e | double | 1 | (description missing) See *units command*. |
| e_mass | double | 1 | (description missing) See *units command*. |
| hhmrr2e | double | 1 | (description missing) See *units command*. |
| mvh2r | double | 1 | (description missing) See *units command*. |
| angstrom | double | 1 | constant to convert current length unit to angstroms; 1.0 for reduced (aka "lj") units. See *units command*. |
| femtosecond | double | 1 | constant to convert current time unit to femtoseconds; 1.0 for reduced (aka "lj") units |
| qelectron | double | 1 | (description missing) See *units command*. |

**Warning**

Modifying the data in the location pointed to by the returned pointer may lead to inconsistent internal data and thus may cause failures or crashes or bogus simulations. In general it is thus usually better to use a LAMMPS input command that sets or changes these parameters. Those will take care of all side effects and necessary updates of settings derived from such settings. Where possible, a reference to such a command or a relevant section of the manual is given below.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **name** – string with the name of the extracted property

**Returns**

pointer (cast to `void *`) to the location of the requested property. NULL if name is not known.

---

int **lammps_extract_pair_dimension**(void *handle, const char *name)

Get data dimension of pair style data accessible via Pair::extract().

Added in version 29Aug2024.

This function returns an integer that specified the dimensionality of the data that can be extracted from the current pair style with `Pair::extract()`. Callers of *lammps_extract_pair()* can use this information to then decide how to cast the `void *` pointer and access the data.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **name** – string with the name of the extracted property

**Returns**

integer constant encoding the dimensionality of the extractable pair style property or -1 if not found.

---

void ***lammps_extract_pair**(void *handle, const char *name)

Get extract pair style data accessible via Pair::extract().

Added in version 29Aug2024.

This function returns a pointer to data available from the current pair style with `Pair::extract()`. The dimensionality of the returned pointer can be determined with *lammps_extract_pair_dimension()*.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **name** – string with the name of the extracted property

**Returns**

pointer (cast to `void *`) to the location of the requested property. NULL if name is not known.

---

int **lammps_map_atom**(void *handle, const void *id)

Map global atom ID to local atom index

Added in version 27June2024.

This function returns an integer that corresponds to the local atom index for an atom with the global atom ID *id*. The atom ID is passed as a void pointer so that it can use the same interface for either a 32-bit or 64-bit tagint. The size of the tagint can be determined using *lammps_extract_setting()*.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **id** – void pointer to the atom ID (of data type tagint, i.e. 32-bit or 64-bit integer)

**Returns**

local atom index or -1 if the atom is not found or no map exists

### 1.1.4 Per-atom properties

This section documents the following functions:

- *lammps_extract_atom_datatype()*
- *lammps_extract_atom_size()*
- *lammps_extract_atom()*

---

int **lammps_extract_atom_datatype**(void *handle, const char *name)

Get data type of a *LAMMPS* per-atom property

Added in version 18Sep2020.

This function returns an integer that encodes the data type of the per-atom property with the specified name. See _LMP_DATATYPE_CONST for valid values. Callers of *lammps_extract_atom()* can use this information to decide how to cast the `void *` pointer and access the data. In addition, *lammps_extract_atom_size()* can be used to get information about the vector or array dimensions.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **name** – string with the name of the extracted property

**Returns**

integer constant encoding the data type of the property or -1 if not found.

---

int **lammps_extract_atom_size**(void *handle, const char *name, int type)

Get dimension info of a *LAMMPS* per-atom property

Added in version 19Nov2024.

This function returns an integer with the size of the per-atom property with the specified name. This allows to accurately determine the size of the per-atom data vectors or arrays. For per-atom arrays, the *type* argument is required to return either the number of rows or the number of columns. It is ignored for per-atom vectors.

Callers of *lammps_extract_atom()* can use this information in combination with the result from *lammps_extract_atom_datatype()* to decide how to cast the `void *` pointer and access the data.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **name** – string with the name of the extracted property
- **type** – either LMP_SIZE_ROWS or LMP_SIZE_COLS if *name* refers to a per-atom array otherwise ignored

**Returns**
integer with the size of the vector or array dimension or -1

---

void ***lammps_extract_atom**(void *handle, const char *name)

Get pointer to a *LAMMPS* per-atom property.

This function returns a pointer to the location of per-atom properties (and per-atom-type properties in the case of the 'mass' keyword). Per-atom data is distributed across sub-domains and thus MPI ranks. The returned pointer is cast to `void *` and needs to be cast to a pointer of data type that the entity represents. You can use the functions *lammps_extract_atom_datatype()* and *lammps_extract_atom_size()* to determine data type, dimensions and sizes of the storage pointed to by the returned pointer.

A table with supported keywords is included in the documentation of the *Atom::extract()* function.

Changed in version 4Jul2026.

When using the KOKKOS package with a device backend, per-atom data is now synchronized from the device to the host before the pointer is returned. Previously the host copy could be out-of-date for calls not aligned with an output or end-of-run step.

**Note**

When running with the KOKKOS package and per-atom data residing on an accelerator device (e.g. a GPU), the requested data is synchronized from the device to the host before the pointer is returned, so that the host data accessed through it is current even when this function is called between output steps (for example from the LAMMPS GUI or a Python script while a run is in progress).

**Warning**

The pointers returned by this function are generally not persistent since per-atom data may be re-distributed, re-allocated, and re-ordered at every re-neighboring operation.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **name** – string with the name of the extracted property

**Returns**
pointer (cast to `void *`) to the location of the requested data or `NULL` if not found.

### 1.1.5 Computes, fixes, variables

This section documents accessing or modifying data stored by computes, fixes, or variables in LAMMPS using the following functions:

- *lammps_extract_compute()*
- *lammps_extract_fix()*
- *lammps_extract_variable_datatype()*
- *lammps_extract_variable()*
- *lammps_set_variable()*
- *lammps_set_string_variable()*
- *lammps_set_internal_variable()*
- *lammps_variable_info()*
- *lammps_eval()*
- *lammps_clearstep_compute()*
- *lammps_addstep_compute_all()*
- *lammps_addstep_compute()*


---

void ***lammps_extract_compute**(void *handle, const char *id, int style, int type)

Get pointer to data from a *LAMMPS* compute.

This function returns a pointer to the location of data provided by a *compute command* instance identified by the compute-ID. Computes may provide global, per-atom, or local data, and those may be a scalar, a vector, or an array or they may provide the information about the dimensions of the respective data. Since computes may provide multiple kinds of data, it is required to set style and type flags representing what specific data is desired. This also determines to what kind of pointer the returned pointer needs to be cast to access the data correctly. The function returns `NULL` if the compute ID is not found or the requested data is not available or current. The following table lists the available options.

| Style (see *_LMP_STYLE_CONST*) | Type (see *_LMP_TYPE_CONST*) | Returned type | Returned data |
|---|---|---|---|
| LMP_STYLE_GLOBAL | LMP_TYPE_SCALAR | `double *` | Global scalar |
| LMP_STYLE_GLOBAL | LMP_TYPE_VECTOR | `double *` | Global vector |
| LMP_STYLE_GLOBAL | LMP_TYPE_ARRAY | `double **` | Global array |
| LMP_STYLE_GLOBAL | LMP_SIZE_VECTOR | `int *` | Length of global vector |
| LMP_STYLE_GLOBAL | LMP_SIZE_ROWS | `int *` | Rows of global array |
| LMP_STYLE_GLOBAL | LMP_SIZE_COLS | `int *` | Columns of global array |
| LMP_STYLE_ATOM | LMP_TYPE_VECTOR | `double *` | Per-atom value |
| LMP_STYLE_ATOM | LMP_TYPE_ARRAY | `double **` | Per-atom vector |
| LMP_STYLE_ATOM | LMP_SIZE_COLS | `int *` | Columns in per-atom array, 0 if vector |
| LMP_STYLE_LOCAL | LMP_TYPE_VECTOR | `double *` | Local data vector |
| LMP_STYLE_LOCAL | LMP_TYPE_ARRAY | `double **` | Local data array |
| LMP_STYLE_LOCAL | LMP_SIZE_VECTOR | `int *` | Alias for LMP_SIZE_ROWS |
| LMP_STYLE_LOCAL | LMP_SIZE_ROWS | `int *` | Number of local array rows or length of vector |
| LMP_STYLE_LOCAL | LMP_SIZE_COLS | `int *` | Number of local array columns, 0 if vector |

**Note**

If the compute's data is not computed for the current step, the compute will be invoked. LAMMPS cannot easily check at that time, if it is valid to invoke a compute, so it may fail with an error. The caller has to check to avoid such an error.

**Warning**

The pointers returned by this function are generally not persistent since the computed data may be re-distributed, re-allocated, and re-ordered at every invocation. It is advisable to re-invoke this function before the data is accessed, or make a copy if the data shall be used after other LAMMPS commands have been issued.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **id** – string with ID of the compute
- **style** – constant indicating the style of data requested (global, per-atom, or local)
- **type** – constant indicating type of data (scalar, vector, or array) or size of rows or columns

**Returns**

pointer (cast to `void *`) to the location of the requested data or `NULL` if not found.

---

void ***lammps_extract_fix**(void *handle, const char *id, int style, int type, int nrow, int ncol)

Get pointer to data from a *LAMMPS* fix.

This function returns a pointer to data provided by a *fix command* instance identified by its fix-ID. Fixes may provide global, per-atom, or local data, and those may be a scalar, a vector, or an array, or they may provide the information about the dimensions of the respective data. Since individual fixes may provide multiple kinds of data, it is required to set style and type flags representing what specific data is desired. This also determines to what kind of pointer the returned pointer needs to be cast to access the data correctly. The function set the error status and returns `NULL` if the fix ID is not found or the requested data is not available.

**Accessing global data**

When requesting **global** data, the fix data can internally only be accessed one item at a time without access to the underlying pointer itself (it may also be computed on-the-fly). Thus this function allocates temporary storage for the requested data, copy the the data to it, and return a pointer to the location of the copy. Therefore the allocated storage needs to be freed with *lammps_free()* after its use to avoid a memory leak. Example:

```
double *dptr = (double *) lammps_extract_fix(handle, name, LMP_STYLE_GLOBAL, LMP_
↪TYPE_VECTOR, 0, 0);
double value = *dptr;
lammps_free((void *)dptr);
```

**Requesting rows, columns, or the entire global array**

In order to avoid the inefficient allocation and deallocation of temporary storage for single values, this functions accepts special values of -1 for the nrow and ncol arguments. For the negative values, the entire row, or column, or full array is copied to a (flat) block of storage and its pointer returned. This still is a copy and needs to be deallocated with *lammps_free()* as indicated in the note above. In case of requesting the whole array, the data is returned column by column, i.e. as d(c_0,r_0), d(c_0,r_1), ... d(c_0, c_nrow-1), d(c_1,r_0), d(c_1,r_1), ... d(c_ncol-1, r_nrow-1) for a total of nrow * ncol elements. Example use:

```
int nrows = *(int *) lammps_extract_fix(handle, name, LMP_STYLE_GLOBAL, LMP_SIZE_
↪ROWS, 0,0);
int ncols = *(int *) lammps_extract_fix(handle, name, LMP_STYLE_GLOBAL, LMP_SIZE_
↪COLS, 0,0);
double *dptr = (double *) lammps_extract_fix(handle, name, LMP_STYLE_GLOBAL, LMP_
↪TYPE_ARRAY, -1, -1);
printf("values[%d][%d] = {\n", ncols, nrows);
for (int j = 0; j < ncols; ++j) {
  printf(" { ");
  for (int i = 0; i < nrows; ++i) printf("%g, ", dvalue[j * nrows + i]);
  printf("},\n");
}
printf("};\n");
lammps_free((void *)dptr);
```

The following table lists the available options.

| Style (see *_LMP_STYLE_CONST*) | Type (see *_LMP_TYPE_CONST*) | Returned type | Returned data |
|---|---|---|---|
| LMP_STYLE_GLOBAL | LMP_TYPE_SCALAR | `double *` | Copy of global scalar |
| LMP_STYLE_GLOBAL | LMP_TYPE_VECTOR | `double *` | Copy of global vector element at index nrow |
| LMP_STYLE_GLOBAL | LMP_TYPE_ARRAY | `double *` | Copy of global array element at nrow, ncol |
| LMP_STYLE_GLOBAL | LMP_SIZE_VECTOR | `int *` | Length of global vector |
| LMP_STYLE_GLOBAL | LMP_SIZE_ROWS | `int *` | Rows in global array |
| LMP_STYLE_GLOBAL | LMP_SIZE_COLS | `int *` | Columns in global array |
| LMP_STYLE_ATOM | LMP_TYPE_VECTOR | `double *` | Per-atom value |
| LMP_STYLE_ATOM | LMP_TYPE_ARRAY | `double **` | Per-atom vector |
| LMP_STYLE_ATOM | LMP_SIZE_COLS | `int *` | Columns of per-atom array, 0 if vector |
| LMP_STYLE_LOCAL | LMP_TYPE_VECTOR | `double *` | Local data vector |
| LMP_STYLE_LOCAL | LMP_TYPE_ARRAY | `double **` | Local data array |
| LMP_STYLE_LOCAL | LMP_SIZE_ROWS | `int *` | Number of local data rows |
| LMP_STYLE_LOCAL | LMP_SIZE_COLS | `int *` | Number of local data columns |

**Note**

LAMMPS cannot easily check if it is valid to access the data, so it may fail with an error and return a NULL pointer. The caller has to avoid such an error.

**Warning**

The pointers returned by this function for per-atom or local data are generally not persistent, since the computed data may be re-distributed, re-allocated, and re-ordered at every invocation of the fix. It is thus advisable to re-invoke this function before the data is accessed, or make a copy, if the data shall be used after other LAMMPS commands have been issued.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **id** – string with ID of the fix
- **style** – constant indicating the style of data requested (global, per-atom, or local)
- **type** – constant indicating type of data (scalar, vector, or array) or size of rows or columns
- **nrow** – row index (only used for global vectors and arrays), zero-based
- **ncol** – column index (only used for global arrays), zero-based

**Returns**

pointer (cast to `void *`) to the location of the requested data or `NULL` if not found.

---

int **lammps_extract_variable_datatype**(void *handle, const char *name)

Get data type of a *LAMMPS* variable.

Added in version 3Nov2022.

This function returns an integer that encodes the data type of the variable with the specified name. See *_LMP_VAR_CONST* for valid values. Callers of *lammps_extract_variable()* can use this information to decide how to cast the `void *` pointer and access the data.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **name** – string with the name of the extracted variable

**Returns**

integer constant encoding the data type of the property or -1 if not found.

---

void ***lammps_extract_variable**(void *handle, const char *name, const char *group)

Get pointer to data from a *LAMMPS* variable.

This function returns a pointer to data from a LAMMPS *variable command* identified by its name. When the variable is either an *equal*-style compatible variable, a *vector*-style variable, or an *atom*-style variable, the variable is evaluated and the corresponding value(s) returned. Variables of style *internal* are compatible with *equal*-style variables and so are *python*-style variables, if they return a numeric value. For other variable styles, their string value is returned. The function returns `NULL` when a variable of the provided *name* is not found or of an incompatible style. The *group* argument is only used for *atom*-style variables and ignored otherwise, with one exception: for style *vector*, if *group* is "GET_VECTOR_SIZE", the returned pointer will yield the length of

the vector to be returned when dereferenced. This pointer must be deallocated after the value is read to avoid a memory leak. If *group* is set to NULL when extracting data from an *atom*-style variable, the group is assumed to be "all".

When requesting data from an *equal*-style or compatible variable this function allocates storage for a single double value, copies the returned value to it, and returns a pointer to the location of the copy. Therefore the allocated storage needs to be freed after its use to avoid a memory leak. Example:

```
double *dptr = (double *) lammps_extract_variable(handle, name, NULL);
double value = *dptr;
lammps_free((void *)dptr);
```

For *atom*-style variables, the return value is a pointer to an allocated block of storage of double of the length atom->nlocal. Since the data returned are a copy, the location will persist, but its content will not be updated in case the variable is re-evaluated. To avoid a memory leak, this pointer needs to be freed after use in the calling program.

For *vector*-style variables, the returned pointer points to actual LAMMPS data and thus it should **not** be deallocated. Its length depends on the variable, compute, or fix data used to construct the *vector*-style variable. This length can be fetched by calling this function with *group* set to a non-NULL pointer (NULL returns the vector). In that case it will return the vector length as an allocated int pointer cast to a void * pointer. That pointer can be recast and dereferenced to an integer yielding the length of the vector. This pointer must be deallocated when finished with it to avoid memory leaks. Example:

```
double *vectvals = (double *) lammps_extract_variable(handle, name, NULL);
int *intptr = (int *) lammps_extract_variable(handle, name, 1);
int vectlen = *intptr;
lammps_free((void *)intptr);
```

For other variable styles the returned pointer needs to be cast to a char pointer and it should **not** be deallocated. Example:

```
const char *cptr = (const char *) lammps_extract_variable(handle,name,NULL);
printf("The value of variable %s is %s\n", name, cptr);
```

**Note**

LAMMPS cannot easily check if it is valid to access the data referenced by the variables (e.g., computes, fixes, or thermodynamic info), so it may fail with an error. The caller has to make certain that the data is extracted only when it safe to evaluate the variable and thus an error or crash are avoided.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **name** – name of the variable
- **group** – group-ID for atom style variable or NULL or non-NULL to get vector length

**Returns**

pointer (cast to void *) to the location of the requested data or NULL if not found.

int **lammps_set_variable**(void *handle, const char *name, const char *str)

Set the value of a string-style variable.

Deprecated since version 7Feb2024.

This function assigns a new value from the string str to the string-style variable *name*. This is a way to directly change the string value of a LAMMPS variable that was previous defined with a *variable name string* command without using any LAMMPS commands to delete and redefine the variable.

Returns -1 if a variable of that name does not exist or if it is not a string-style variable, otherwise 0.

**Warning**

This function is deprecated and *lammps_set_string_variable()* should be used instead.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **name** – name of the variable
- **str** – new value of the variable

**Returns**

0 on success or -1 on failure

---

int **lammps_set_string_variable**(void *handle, const char *name, const char *str)

Set the value of a string-style variable.

Added in version 7Feb2024.

This function assigns a new value from the string str to the string-style variable *name*. This is a way to directly change the string value of a LAMMPS variable that was previous defined with a *variable name string* command without using any LAMMPS commands to delete and redefine the variable.

Returns -1 if a variable of that name does not exist or if it is not a string-style variable, otherwise 0.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **name** – name of the variable
- **str** – new value of the variable

**Returns**

0 on success or -1 on failure

---

int **lammps_set_internal_variable**(void *handle, const char *name, double value)

Set the value of an internal-style variable.

Added in version 7Feb2024.

This function assigns a new value from the floating point number *value* to the internal-style variable *name*. This is a way to directly change the numerical value of such a LAMMPS variable that was previous defined with a *variable name internal* command without using any LAMMPS commands to delete and redefine the variable.

Returns -1 if a variable of that name does not exist or is not an internal-style variable, otherwise 0.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **name** – name of the variable
- **value** – new value of the variable

**Returns**
0 on success or -1 on failure

---

int **lammps_variable_info**(void *handle, int idx, char *buf, int bufsize)

Retrieve informational string for a variable.

Added in version 21Nov2023.

This function copies a string with human readable information about a defined variable: name, style, current value(s) into the provided C-style string buffer. That is the same info as produced by the *info variables* command. The length of the buffer must be provided as *buf_size* argument. If the info exceeds the length of the buffer, it will be truncated accordingly. If the index is out of range, the function returns 0 and *buffer* is set to an empty string, otherwise 1.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **idx** – index of the variable (0 <= idx < nvar)
- **buffer** – string buffer to copy the info to
- **buf_size** – size of the provided string buffer

**Returns**
1 if successful, otherwise 0

---

double **lammps_eval**(void *handle, const char *expr)

Evaluate an immediate variable expression

Added in version 4Feb2025.

This function takes a string with an expression that can be used for *equal style variables*, evaluates it and returns the resulting (scalar) value as a floating point number.

***See also***
*lammps_expand()*

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **expr** – string with expression

**Returns**
result from expression

void **lammps_clearstep_compute**(void *handle)

Clear whether a compute has been invoked.

Added in version 4Feb2025: This function clears the invoked flag of all computes. Called everywhere that computes are used, before computes are invoked. The invoked flag is used to avoid re-invoking same compute multiple times and to flag computes that store invocation times as having been invoked

***See also***

*lammps_addstep_compute_all() lammps_addstep_compute()*

**Parameters**

**handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.

---

void **lammps_addstep_compute_all**(void *handle, void *nextstep)

Add next timestep to all computes

Added in version 4Feb2025: loop over all computes schedule next invocation for those that store invocation times called when not sure what computes will be needed on newstep do not loop only over n_timeflag, since may not be set yet

***See also***

*lammps_clearstep_compute() lammps_addstep_compute()*

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **newstep** – pointer to bigint of next timestep the compute will be invoked

---

void **lammps_addstep_compute**(void *handle, void *nextstep)

Add next timestep to compute if it has been invoked in the current timestep

Added in version 4Feb2025: loop over computes that store invocation times if its invoked flag set on this timestep, schedule next invocation called everywhere that computes are used, after computes are invoked

***See also***

*lammps_addstep_compute_all() lammps_clearstep_compute()*

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **newstep** – next timestep the compute will be invoked

---

enum LAMMPS_NS::multitype::_LMP_DATATYPE_CONST

Data type constants for extracting data from atoms, computes and fixes

This enum must be kept in sync with the corresponding enum or constants in python/lammps/constants.py, fortran/lammps.f90, tools/swig/lammps.i, src/library.h, and examples/COUPLE/plugin/liblammpsplugin.h

*Values:*

enumerator **LAMMPS_NONE**

no data type assigned (yet)

enumerator **LAMMPS_INT**

32-bit integer (array)

enumerator **LAMMPS_INT_2D**

two-dimensional 32-bit integer array

enumerator **LAMMPS_DOUBLE**

64-bit double (array)

enumerator **LAMMPS_DOUBLE_2D**

two-dimensional 64-bit double array

enumerator **LAMMPS_INT64**

64-bit integer (array)

enumerator **LAMMPS_INT64_2D**

two-dimensional 64-bit integer array

enumerator **LAMMPS_STRING**

C-String

enum **_LMP_STYLE_CONST**

Style constants for extracting data from computes and fixes.

Must be kept in sync with the equivalent constants in python/lammps/constants.py, fortran/lammps.f90, tools/swig/lammps.i, and examples/COUPLE/plugin/liblammpsplugin.h

*Values:*

enumerator **LMP_STYLE_GLOBAL**

return global data

enumerator **LMP_STYLE_ATOM**

return per-atom data

enumerator **LMP_STYLE_LOCAL**

return local data

enum **_LMP_TYPE_CONST**

Type and size constants for extracting data from computes and fixes.

Must be kept in sync with the equivalent constants in `python/lammps/constants.py`, `fortran/lammps.f90`, `tools/swig/lammps.i`, and `examples/COUPLE/plugin/liblammpsplugin.h`

*Values:*

enumerator **LMP_TYPE_SCALAR**

return scalar

enumerator **LMP_TYPE_VECTOR**

return vector

enumerator **LMP_TYPE_ARRAY**

return array

enumerator **LMP_SIZE_VECTOR**

return length of vector

enumerator **LMP_SIZE_ROWS**

return number of rows

enumerator **LMP_SIZE_COLS**

return number of columns

enum **_LMP_VAR_CONST**

Variable style constants for extracting data from variables.

Must be kept in sync with the equivalent constants in `python/lammps/constants.py`, `fortran/lammps.f90`, `tools/swig/lammps.i`, and `examples/COUPLE/plugin/liblammpsplugin.h`

*Values:*

enumerator **LMP_VAR_EQUAL**

compatible with equal-style variables

enumerator **LMP_VAR_ATOM**

compatible with atom-style variables

enumerator **LMP_VAR_VECTOR**

compatible with vector-style variables

enumerator **LMP_VAR_STRING**

return value will be a string (catch-all)

### 1.1.6 Scatter/gather operations

This section has functions which gather per-atom data from one or more processors into a contiguous global list ordered by atom ID. The same list is returned to all calling processors. It also contains functions which scatter per-atom data from a contiguous global list across the processors that own those atom IDs. It also has a create_atoms() function which can create new atoms by scattering them appropriately to owning processors in the LAMMPS spatial decomposition.

It documents the following functions:

- *lammps_gather_atoms()*
- *lammps_gather_atoms_concat()*
- *lammps_gather_atoms_subset()*
- *lammps_scatter_atoms()*
- *lammps_scatter_atoms_subset()*
- *lammps_gather_bonds()*
- *lammps_gather_angles()*
- *lammps_gather_dihedrals()*
- *lammps_gather_impropers()*
- *lammps_gather()*
- *lammps_gather_concat()*
- *lammps_gather_subset()*
- *lammps_scatter()*
- *lammps_scatter_subset()*
- *lammps_create_atoms()*
- *lammps_create_molecule()*

---

void **lammps_gather_atoms**(void *handle, const char *name, int type, int count, void *data)

Gather the named atom-based entity for all atoms across all processes, in order.

This subroutine gathers data for all atoms and stores them in a one-dimensional array allocated by the user. The data will be ordered by atom ID, which requires consecutive atom IDs (1 to *natoms*). If you need a similar array but have non-consecutive atom IDs, see *lammps_gather_atoms_concat()*; for a similar array but for a subset of atoms, see *lammps_gather_atoms_subset()*.

The *data* array will be ordered in groups of *count* values, sorted by atom ID (e.g., if *name* is *x* and *count* = 3, then *data* = x[0][0], x[0][1], x[0][2], x[1][0], x[1][1], x[1][2], x[2][0], ...); *data* must be pre-allocated by the caller to length (*count* × *natoms*), as queried by *lammps_get_natoms()*, *lammps_extract_global()*, or *lammps_extract_setting()*.

**Restrictions**

This function is not compatible with `-DLAMMPS_BIGBIG`.

Atom IDs must be defined and consecutive.

The total number of atoms must not be more than 2147483647 (max 32-bit signed int).

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **name** – desired quantity (e.g., *x* or *q*)
- **dtype** – 0 for `int` values, 1 for `double` values
- **count** – number of per-atom values (e.g., 1 for *type* or *q*, 3 for *x* or *f*); use *count* = 3 with *image* if you want a single image flag unpacked into (*x*,*y*,*z*) components.
- **data** – per-atom values packed in a 1-dimensional array of length *natoms* * *count*.

---

void **lammps_gather_atoms_concat**(void *handle, const char *name, int type, int count, void *data)

Gather the named atom-based entity for all atoms across all processes, unordered.

This subroutine gathers data for all atoms and stores them in a one-dimensional array allocated by the user. The data will be a concatenation of chunks from each processor's owned atoms, in whatever order the atoms are in on each processor. This process has no requirement that the atom IDs be consecutive. If you need the ID of each atom, you can do another *lammps_gather_atoms_concat()* call with *name* set to `id`. If you have consecutive IDs and want the data to be in order, use *lammps_gather_atoms()*; for a similar array but for a subset of atoms, use *lammps_gather_atoms_subset()*.

The *data* array will be in groups of *count* values, with *natoms* groups total, but not in order by atom ID (e.g., if *name* is *x* and *count* is 3, then *data* might be something like x[10][0], x[10][1], x[10][2], x[2][0], x[2][1], x[2][2], x[4][0], ...); *data* must be pre-allocated by the caller to length (*count* × *natoms*), as queried by *lammps_get_natoms()*, *lammps_extract_global()*, or *lammps_extract_setting()*.

**Restrictions**

This function is not compatible with `-DLAMMPS_BIGBIG`.

Atom IDs must be defined.

The total number of atoms must not be more than 2147483647 (max 32-bit signed int).

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **name** – desired quantity (e.g., *x* or *q*\ )
- **dtype** – 0 for `int` values, 1 for `double` values
- **count** – number of per-atom values (e.g., 1 for *type* or *q*, 3 for *x* or *f*); use *count* = 3 with "image" if you want single image flags unpacked into (*x*,*y*,*z*)
- **data** – per-atom values packed in a 1-dimensional array of length *natoms* * *count*.

---

void **lammps_gather_atoms_subset**(void *handle, const char *name, int type, int count, int ndata, int *ids, void *data)

Gather the named atom-based entity for a subset of atoms.

This subroutine gathers data for the requested atom IDs and stores them in a one-dimensional array allocated by the user. The data will be ordered by atom ID, but there is no requirement that the IDs be consecutive. If you wish to return a similar array for *all* the atoms, use `lammps_gather_atoms()` or `lammps_gather_atoms_concat()`.

The *data* array will be in groups of *count* values, sorted by atom ID in the same order as the array *ids* (e.g., if *name* is *x*, *count* = 3, and *ids* is {100, 57, 210}, then *data* might look like {x[100][0], x[100][1], x[100][2], x[57][0], x[57][1], x[57][2], x[210][0], ...}; *ids* must be provided by the user with length *ndata*, and *data* must be pre-allocated by the caller to length (*count* × *ndata*).

**Restrictions**

This function is not compatible with `-DLAMMPS_BIGBIG`.

Atom IDs must be defined and an *atom map must be enabled*

The total number of atoms must not be more than 2147483647 (max 32-bit signed int).

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **name** – desired quantity (e.g., *x* or *q*)
- **dtype** – 0 for `int` values, 1 for `double` values
- **count** – number of per-atom values (e.g., 1 for *type* or *q*, 3 for *x* or *f*); use *count* = 3 with "image" if you want single image flags unpacked into (*x*,*y*,*z*)
- **ndata** – number of atoms for which to return data (can be all of them)
- **ids** – list of *ndata* atom IDs for which to return data
- **data** – per-atom values packed in a 1-dimensional array of length *ndata* * *count*.

---

void **lammps_scatter_atoms**(void *handle, const char *name, int type, int count, void *data)

Scatter the named atom-based entities in *data* to all processes.

This subroutine takes data stored in a one-dimensional array supplied by the user and scatters them to all atoms on all processes. The data must be ordered by atom ID, with the requirement that the IDs be consecutive. Use `lammps_scatter_atoms_subset()` to scatter data for some (or all) atoms, unordered.

The *data* array needs to be ordered in groups of *count* values, sorted by atom ID (e.g., if *name* is *x* and *count* = 3, then *data* = {x[0][0], x[0][1], x[0][2], x[1][0], x[1][1], x[1][2], x[2][0], ...}); *data* must be of length (*count* × *natoms*).

**Restrictions**

This function is not compatible with `-DLAMMPS_BIGBIG`.

Atom IDs must be defined, must be consecutive, and an *atom map must be enabled*

The total number of atoms must not be more than 2147483647 (max 32-bit signed int).

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **name** – desired quantity (e.g., *x* or *q*)
- **dtype** – 0 for `int` values, 1 for `double` values
- **count** – number of per-atom values (e.g., 1 for *type* or *q*, 3 for *x* or *f*); use *count* = 3 with *image* if you have a single image flag packed into (*x*,*y*,*z*) components.
- **data** – per-atom values packed in a one-dimensional array of length *natoms* * *count*.

---

void **lammps_scatter_atoms_subset**(void *handle, const char *name, int type, int count, int ndata, int *ids, void *data)

Scatter the named atom-based entities in *data* from a subset of atoms to all processes.

This subroutine takes data stored in a one-dimensional array supplied by the user and scatters them to a subset of atoms on all processes. The array *data* contains data associated with atom IDs, but there is no requirement that the IDs be consecutive, as they are provided in a separate array. Use *lammps_scatter_atoms()* to scatter data for all atoms, in order.

The *data* array needs to be organized in groups of *count* values, with the groups in the same order as the array *ids*. For example, if you want *data* to be the array {x[1][0], x[1][1], x[1][2], x[100][0], x[100][1], x[100][2], x[57][0], x[57][1], x[57][2]}, then *count* = 3, *ndata* = 3, and *ids* would be {1, 100, 57}.

**Restrictions**

This function is not compatible with `-DLAMMPS_BIGBIG`.

Atom IDs must be defined and an *atom map must be enabled*

The total number of atoms must not be more than 2147483647 (max 32-bit signed int).

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **name** – desired quantity (e.g., *x* or *q*)
- **dtype** – 0 for `int` values, 1 for `double` values
- **count** – number of per-atom values (e.g., 1 for *dtype* or *q*, 3 for *x* or *f*); use *count* = 3 with "image" if you have all the image flags packed into (*xyz*)
- **ndata** – number of atoms listed in *ids* and *data* arrays
- **ids** – list of *ndata* atom IDs to scatter data to
- **data** – per-atom values packed in a 1-dimensional array of length *ndata* * *count*.

---

void **lammps_gather_bonds**(void *handle, void *data)

Gather type and constituent atom info for all bonds

Added in version 28Jul2021.

This function copies the list of all bonds into a buffer provided by the calling code. The buffer will be filled with bond type, bond atom 1, bond atom 2 for each bond. Thus the buffer has to be allocated to the dimension of 3 times the **total** number of bonds times the size of the LAMMPS "tagint" type, which is either 4 or 8 bytes depending on whether they are stored in 32-bit or 64-bit integers, respectively. This size depends on the compile time settings used when compiling the LAMMPS library and can be queried by calling *lammps_extract_setting()* with the keyword "tagint".

When running in parallel, the data buffer must be allocated on **all** MPI ranks and will be filled with the information for **all** bonds in the system.

Below is a brief C code demonstrating accessing this collected bond information.

```
#include "library.h"

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    int tagintsize;
    int64_t i, nbonds;
    void *handle, *bonds;

    handle = lammps_open_no_mpi(0, NULL, NULL);
    lammps_file(handle, "in.some_input");

    tagintsize = lammps_extract_setting(handle, "tagint");
    if (tagintsize == 4)
        nbonds = *(int32_t *)lammps_extract_global(handle, "nbonds");
     else
        nbonds = *(int64_t *)lammps_extract_global(handle, "nbonds");
    bonds = malloc(nbonds * 3 * tagintsize);

    lammps_gather_bonds(handle, bonds);

    if (lammps_extract_setting(handle, "world_rank") == 0) {
        if (tagintsize == 4) {
            int32_t *bonds_real = (int32_t *)bonds;
            for (i = 0; i < nbonds; ++i) {
                printf("bond % 4ld: type = %d, atoms: % 4d  % 4d\n",i,
                       bonds_real[3*i], bonds_real[3*i+1], bonds_real[3*i+2]);
            }
        } else {
            int64_t *bonds_real = (int64_t *)bonds;
            for (i = 0; i < nbonds; ++i) {
                printf("bond % 4ld: type = %ld, atoms: % 4ld  % 4ld\n",i,
                       bonds_real[3*i], bonds_real[3*i+1], bonds_real[3*i+2]);
            }
        }
    }

    lammps_close(handle);
    lammps_mpi_finalize();
```

(continues on next page)

(continued from previous page)

```
    free(bonds);
    return 0;
}
```

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **data** – pointer to data to copy the result to

---

void **lammps_gather_angles**(void *handle, void *data)

Gather type and constituent atom info for all angles

Added in version 8Feb2023.

This function copies the list of all angles into a buffer provided by the calling code. The buffer will be filled with angle type, angle atom 1, angle atom 2, angle atom 3 for each angle. Thus the buffer has to be allocated to the dimension of 4 times the **total** number of angles times the size of the LAMMPS "tagint" type, which is either 4 or 8 bytes depending on whether they are stored in 32-bit or 64-bit integers, respectively. This size depends on the compile time settings used when compiling the LAMMPS library and can be queried by calling *lammps_extract_setting()* with the keyword "tagint".

When running in parallel, the data buffer must be allocated on **all** MPI ranks and will be filled with the information for **all** angles in the system.

Below is a brief C code demonstrating accessing this collected angle information.

```
#include "library.h"

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    int tagintsize;
    int64_t i, nangles;
    void *handle, *angles;

    handle = lammps_open_no_mpi(0, NULL, NULL);
    lammps_file(handle, "in.some_input");

    tagintsize = lammps_extract_setting(handle, "tagint");
    if (tagintsize == 4)
        nangles = *(int32_t *)lammps_extract_global(handle, "nangles");
     else
        nangles = *(int64_t *)lammps_extract_global(handle, "nangles");
    angles = malloc(nangles * 4 * tagintsize);

    lammps_gather_angles(handle, angles);
```

(continues on next page)

(continued from previous page)

```
    if (lammps_extract_setting(handle, "world_rank") == 0) {
        if (tagintsize == 4) {
            int32_t *angles_real = (int32_t *)angles;
            for (i = 0; i < nangles; ++i) {
                printf("angle % 4ld: type = %d, atoms: % 4d  % 4d  % 4d\n",i,
                       angles_real[4*i], angles_real[4*i+1], angles_real[4*i+2],␣
↪angles_real[4*i+3]);
            }
        } else {
            int64_t *angles_real = (int64_t *)angles;
            for (i = 0; i < nangles; ++i) {
                printf("angle % 4ld: type = %ld, atoms: % 4ld  % 4ld  % 4ld\n",i,
                       angles_real[4*i], angles_real[4*i+1], angles_real[4*i+2],␣
↪angles_real[4*i+3]);
            }
        }
    }

    lammps_close(handle);
    lammps_mpi_finalize();
    free(angles);
    return 0;
}
```

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **data** – pointer to data to copy the result to

---

void **lammps_gather_dihedrals**(void *handle, void *data)

Gather type and constituent atom info for all dihedrals

Added in version 8Feb2023.

This function copies the list of all dihedrals into a buffer provided by the calling code. The buffer will be filled with dihedral type, dihedral atom 1, dihedral atom 2, dihedral atom 3, dihedral atom 4 for each dihedral. Thus the buffer has to be allocated to the dimension of 5 times the **total** number of dihedrals times the size of the LAMMPS "tagint" type, which is either 4 or 8 bytes depending on whether they are stored in 32-bit or 64-bit integers, respectively. This size depends on the compile time settings used when compiling the LAMMPS library and can be queried by calling *lammps_extract_setting()* with the keyword "tagint".

When running in parallel, the data buffer must be allocated on **all** MPI ranks and will be filled with the information for **all** dihedrals in the system.

Below is a brief C code demonstrating accessing this collected dihedral information.

```
#include "library.h"

#include <stdint.h>
#include <stdio.h>
```

(continues on next page)

(continued from previous page)

```
#include <stdlib.h>

int main(int argc, char **argv)
{
    int tagintsize;
    int64_t i, ndihedrals;
    void *handle, *dihedrals;

    handle = lammps_open_no_mpi(0, NULL, NULL);
    lammps_file(handle, "in.some_input");

    tagintsize = lammps_extract_setting(handle, "tagint");
    if (tagintsize == 4)
        ndihedrals = *(int32_t *)lammps_extract_global(handle, "ndihedrals");
     else
        ndihedrals = *(int64_t *)lammps_extract_global(handle, "ndihedrals");
    dihedrals = malloc(ndihedrals * 5 * tagintsize);

    lammps_gather_dihedrals(handle, dihedrals);

    if (lammps_extract_setting(handle, "world_rank") == 0) {
        if (tagintsize == 4) {
            int32_t *dihedrals_real = (int32_t *)dihedrals;
            for (i = 0; i < ndihedrals; ++i) {
                printf("dihedral % 4ld: type = %d, atoms: % 4d  % 4d  % 4d  % 4d\n",
↪i,
                       dihedrals_real[5*i], dihedrals_real[5*i+1], dihedrals_
↪real[5*i+2], dihedrals_real[5*i+3], dihedrals_real[5*i+4]);
            }
        } else {
            int64_t *dihedrals_real = (int64_t *)dihedrals;
            for (i = 0; i < ndihedrals; ++i) {
                printf("dihedral % 4ld: type = %ld, atoms: % 4ld  % 4ld  % 4ld  %␣
↪4ld\n",i,
                       dihedrals_real[5*i], dihedrals_real[5*i+1], dihedrals_
↪real[5*i+2], dihedrals_real[5*i+3], dihedrals_real[5*i+4]);
            }
        }
    }

    lammps_close(handle);
    lammps_mpi_finalize();
    free(dihedrals);
    return 0;
}
```

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **data** – pointer to data to copy the result to

void **lammps_gather_impropers**(void *handle, void *data)

Gather type and constituent atom info for all impropers

Added in version 8Feb2023.

This function copies the list of all impropers into a buffer provided by the calling code. The buffer will be filled with improper type, improper atom 1, improper atom 2, improper atom 3, improper atom 4 for each improper. Thus the buffer has to be allocated to the dimension of 5 times the **total** number of impropers times the size of the LAMMPS "tagint" type, which is either 4 or 8 bytes depending on whether they are stored in 32-bit or 64-bit integers, respectively. This size depends on the compile time settings used when compiling the LAMMPS library and can be queried by calling *lammps_extract_setting()* with the keyword "tagint".

When running in parallel, the data buffer must be allocated on **all** MPI ranks and will be filled with the information for **all** impropers in the system.

Below is a brief C code demonstrating accessing this collected improper information.

```
#include "library.h"

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    int tagintsize;
    int64_t i, nimpropers;
    void *handle, *impropers;

    handle = lammps_open_no_mpi(0, NULL, NULL);
    lammps_file(handle, "in.some_input");

    tagintsize = lammps_extract_setting(handle, "tagint");
    if (tagintsize == 4)
        nimpropers = *(int32_t *)lammps_extract_global(handle, "nimpropers");
     else
        nimpropers = *(int64_t *)lammps_extract_global(handle, "nimpropers");
    impropers = malloc(nimpropers * 5 * tagintsize);

    lammps_gather_impropers(handle, impropers);

    if (lammps_extract_setting(handle, "world_rank") == 0) {
        if (tagintsize == 4) {
            int32_t *impropers_real = (int32_t *)impropers;
            for (i = 0; i < nimpropers; ++i) {
                printf("improper % 4ld: type = %d, atoms: % 4d  % 4d  % 4d  % 4d\n",
↪i,
                       impropers_real[5*i], impropers_real[5*i+1], impropers_
↪real[5*i+2], impropers_real[5*i+3], impropers_real[5*i+4]);
            }
        } else {
            int64_t *impropers_real = (int64_t *)impropers;
            for (i = 0; i < nimpropers; ++i) {
                printf("improper % 4ld: type = %ld, atoms: % 4ld  % 4ld  % 4ld  %␣
```

(continues on next page)

(continued from previous page)

```
↪4ld\n",i,
                        impropers_real[5*i], impropers_real[5*i+1], impropers_
↪real[5*i+2], impropers_real[5*i+3], impropers_real[5*i+4]);
            }
        }
    }

    lammps_close(handle);
    lammps_mpi_finalize();
    free(impropers);
    return 0;
}
```

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **data** – pointer to data to copy the result to

---

void **lammps_gather**(void *handle, const char *name, int type, int count, void *data)

Gather the named per-atom, per-atom fix, per-atom compute, or fix property/atom-based entities from all processes, in order by atom ID.

This subroutine gathers data from all processes and stores them in a one-dimensional array allocated by the user. The array *data* will be ordered by atom ID, which requires consecutive IDs (1 to *natoms*). If you need a similar array but for non-consecutive atom IDs, see *lammps_gather_concat()*; for a similar array but for a subset of atoms, see *lammps_gather_subset()*.

The *data* array will be ordered in groups of *count* values, sorted by atom ID (e.g., if *name* is *x*, then *data* is {x[0][0], x[0][1], x[0][2], x[1][0], x[1][1], x[1][2], x[2][0], ...}); *data* must be pre-allocated by the caller to the correct length (*count* × *natoms*), as queried by *lammps_get_natoms()*, *lammps_extract_global()*, or *lammps_extract_setting()*.

This function will return an error if fix or compute data are requested and the fix or compute ID given does not have per-atom data.

**Restrictions**

This function is not compatible with `-DLAMMPS_BIGBIG`.

Atom IDs must be defined and must be consecutive.

The total number of atoms must not be more than 2147483647 (max 32-bit signed int).

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **name** – desired quantity (e.g., "x" or "f" for atom properties, "f_id" for per-atom fix data, "c_id" for per-atom compute data, "d_name" or "i_name" for fix property/atom vectors with *count* = 1, "d2_name" or "i2_name" for fix property/atom vectors with *count* > 1)
- **dtype** – 0 for `int` values, 1 for `double` values

- **count** – number of per-atom values (e.g., 1 for *type* or $q$, 3 for $x$ or $f$); use *count* = 3 with *image* if you want the image flags unpacked into ($x$,$y$,$z$) components.
- **data** – per-atom values packed into a one-dimensional array of length *natoms* * *count*.

---

void **lammps_gather_concat**(void *handle, const char *name, int type, int count, void *data)

Gather the named per-atom, per-atom fix, per-atom compute, or fix property/atom-based entities from all processes, unordered.

This subroutine gathers data for all atoms and stores them in a one-dimensional array allocated by the user. The data will be a concatenation of chunks from each processor's owned atoms, in whatever order the atoms are in on each processor. This process has no requirement that the atom IDs be consecutive. If you need the ID of each atom, you can do another call to either *lammps_gather_atoms_concat()* or *lammps_gather_concat()* with *name* set to `id`. If you have consecutive IDs and want the data to be in order, use *lammps_gather()*; for a similar array but for a subset of atoms, use *lammps_gather_subset()*.

The *data* array will be in groups of *count* values, with *natoms* groups total, but not in order by atom ID (e.g., if *name* is $x$ and *count* is 3, then *data* might be something like {x[10][0], x[10][1], x[10][2], x[2][0], x[2][1], x[2][2], x[4][0], ...}); *data* must be pre-allocated by the caller to length (*count* $\times$ *natoms*), as queried by *lammps_get_natoms()*, *lammps_extract_global()*, or *lammps_extract_setting()*.

**Restrictions**

This function is not compatible with `-DLAMMPS_BIGBIG`.

Atom IDs must be defined.

The total number of atoms must be less than 2147483647 (max 32-bit signed int).

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **name** – desired quantity (e.g., "x" or "f" for atom properties, "f_id" for per-atom fix data, "c_id" for per-atom compute data, "d_name" or "i_name" for fix property/atom vectors with count = 1, "d2_name" or "i2_name" for fix property/atom vectors with count > 1)
- **dtype** – 0 for `int` values, 1 for `double` values
- **count** – number of per-atom values (e.g., 1 for *type* or $q$, 3 for $x$ or $f$); use *count* = 3 with *image* if you want the image flags unpacked into ($x$,$y$,$z$) components.
- **data** – per-atom values packed into a one-dimensional array of length *natoms* * *count*.

---

void **lammps_gather_subset**(void *handle, const char *name, int type, int count, int ndata, int *ids, void *data)

Gather the named per-atom, per-atom fix, per-atom compute, or fix property/atom-based entities from all processes for a subset of atoms.

This subroutine gathers data for the requested atom IDs and stores them in a one-dimensional array allocated by the user. The data will be ordered by atom ID, but there is no requirement that the IDs be consecutive. If you wish to return a similar array for *all* the atoms, use *lammps_gather()* or *lammps_gather_concat()*.

The *data* array will be in groups of *count* values, sorted by atom ID in the same order as the array *ids* (e.g., if *name* is *x*, *count* = 3, and *ids* is {100, 57, 210}, then *data* might look like {x[100][0], x[100][1], x[100][2], x[57][0], x[57][1], x[57][2], x[210][0], . . .}); *ids* must be provided by the user with length *ndata*, and *data* must be pre-allocated by the caller to length (*count* × *ndata*).

**Restrictions**

This function is not compatible with `-DLAMMPS_BIGBIG`.

Atom IDs must be defined and an *atom map must be enabled*

The total number of atoms must not be more than 2147483647 (max 32-bit signed int).

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **name** – desired quantity (e.g., "x" or "f" for atom properties, "f_id" for per-atom fix data, "c_id" for per-atom compute data, "d_name" or "i_name" for fix property/atom vectors with *count* = 1, "d2_name" or "i2_name" for fix property/atom vectors with *count* > 1)
- **dtype** – 0 for `int` values, 1 for `double` values
- **count** – number of per-atom values (e.g., 1 for *type* or *q*, 3 for *x* or *f*); use *count* = 3 with *image* if you want the image flags unpacked into (*x*,*y*,*z*) components.
- **ndata** – number of atoms for which to return data (can be all of them)
- **ids** – list of *ndata* atom IDs for which to return data
- **data** – per-atom values packed into a one-dimensional array of length *ndata* * *count*.

---

void **lammps_scatter**(void *handle, const char *name, int type, int count, void *data)

Scatter the named per-atom, per-atom fix, per-atom compute, or fix property/atom-based entity in *data* to all processes.

This subroutine takes data stored in a one-dimensional array supplied by the user and scatters them to all atoms on all processes. The data must be ordered by atom ID, with the requirement that the IDs be consecutive. Use *lammps_scatter_subset()* to scatter data for some (or all) atoms, unordered.

The *data* array needs to be ordered in groups of *count* values, sorted by atom ID (e.g., if *name* is *x* and *count* = 3, then *data* = {x[0][0], x[0][1], x[0][2], x[1][0], x[1][1], x[1][2], x[2][0], . . .}); *data* must be of length (*count* × *natoms*).

**Restrictions**

This function is not compatible with `-DLAMMPS_BIGBIG`.

Atom IDs must be defined, must be consecutive, and an *atom map must be enabled*

The total number of atoms must not be more than 2147483647 (max 32-bit signed int).

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance

- **name** – desired quantity (e.g., "x" or "f" for atom properties, "f_id" for per-atom fix data, "c_id" for per-atom compute data, "d_name" or "i_name" for fix property/atom vectors with *count* = 1, "d2_name" or "i2_name" for fix property/atom vectors with *count* > 1)
- **dtype** – 0 for `int` values, 1 for `double` values
- **count** – number of per-atom values (e.g., 1 for *type* or *q*, 3 for *x* or *f*); use *count* = 3 with *image* if you have a single image flag packed into (*x*,*y*,*z*) components.
- **data** – per-atom values packed in a one-dimensional array of length *natoms* * *count*.

---

void **lammps_scatter_subset**(void *handle, const char *name, int type, int count, int ndata, int *ids, void *data)

Scatter the named per-atom, per-atom fix, per-atom compute, or fix property/atom-based entities in *data* from a subset of atoms to all processes.

This subroutine takes data stored in a one-dimensional array supplied by the user and scatters them to a subset of atoms on all processes. The array *data* contains data associated with atom IDs, but there is no requirement that the IDs be consecutive, as they are provided in a separate array. Use *lammps_scatter()* to scatter data for all atoms, in order.

The *data* array needs to be organized in groups of *count* values, with the groups in the same order as the array *ids*. For example, if you want *data* to be the array {x[1][0], x[1][1], x[1][2], x[100][0], x[100][1], x[100][2], x[57][0], x[57][1], x[57][2]}, then *count* = 3, *ndata* = 3, and *ids* would be {1, 100, 57}.

**Restrictions**

This function is not compatible with `-DLAMMPS_BIGBIG`.

Atom IDs must be defined and an *atom map must be enabled*

The total number of atoms must not be more than 2147483647 (max 32-bit signed int).

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **name** – desired quantity (e.g., "x" or "f" for atom properties, "f_id" for per-atom fix data, "c_id" for per-atom compute data, "d_name" or "i_name" for fix property/atom vectors with *count* = 1, "d2_name" or "i2_name" for fix property/atom vectors with *count* > 1)
- **dtype** – 0 for `int` values, 1 for `double` values
- **count** – number of per-atom values (e.g., 1 for *type* or *q*, 3 for *x* or *f*); use *count* = 3 with "image" if you want single image flags unpacked into (*x*,*y*,*z*)
- **ndata** – number of atoms listed in *ids* and *data* arrays
- **ids** – list of *ndata* atom IDs to scatter data to
- **data** – per-atom values packed in a 1-dimensional array of length *ndata* * *count*.

---

int **lammps_create_atoms**(void *handle, int n, const int *id, const int *type, const double *x, const double *v, const int *image, int bexpand)

Create N atoms from list of coordinates

The prototype for this function when compiling with `-DLAMMPS_BIGBIG` is:

```
int lammps_create_atoms(void *handle, int n, int64_t *id, int *type, double *x,␣
↪double *v, int64_t *image, int bexpand);
```

This function creates additional atoms from a given list of coordinates and a list of atom types. Additionally the atom-IDs, velocities, and image flags may be provided. If atom-IDs are not provided, they will be automatically created as a sequence following the largest existing atom-ID.

This function is useful to add atoms to a simulation or - in tandem with *lammps_reset_box()* - to restore a previously extracted and saved state of a simulation. Additional properties for the new atoms can then be assigned via the *lammps_scatter_atoms()* *lammps_extract_atom()* functions.

For non-periodic boundaries, atoms will **not** be created that have coordinates outside the box unless it is a shrink-wrap boundary and the shrinkexceed flag has been set to a non-zero value. For periodic boundaries atoms will be wrapped back into the simulation cell and its image flags adjusted accordingly, unless explicit image flags are provided.

The function returns the number of atoms created or -1 on failure (e.g., when called before as box has been created).

Coordinates and velocities have to be given in a 1d-array in the order X(1), Y(1), Z(1), X(2), Y(2), Z(2), ..., X(N), Y(N), Z(N).

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **n** – number of atoms, N, to be added to the system
- **id** – pointer to N atom IDs; `NULL` will generate IDs
- **type** – pointer to N atom types (required)
- **x** – pointer to 3N doubles with x-,y-,z- positions of the new atoms (required)
- **v** – pointer to 3N doubles with x-,y-,z- velocities of the new atoms (set to 0.0 if `NULL`)
- **image** – pointer to N imageint sets of image flags, or `NULL`
- **bexpand** – if 1, atoms outside of shrink-wrap boundaries will still be created and not dropped and the box extended

**Returns**

number of atoms created on success; -1 on failure (no box, no atom IDs, etc.)

---

void **lammps_create_molecule**(void *handle, const char *id, const char *json)

Create new molecule template from JSON data provided as C-style string

Added in version 22Jul2025.

This function creates a new molecule template similar to the *molecule command*, but uses JSON data passed as a C-style string instead of reading it from a file.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance
- **id** – molecule-ID
- **jsonstr** – molecule data in JSON format as C-style string

### 1.1.7 Neighbor list access

The following functions enable access to neighbor lists generated by LAMMPS or querying of their properties:

- *lammps_find_compute_neighlist()*
- *lammps_find_fix_neighlist()*
- *lammps_find_pair_neighlist()*
- *lammps_neighlist_num_elements()*
- *lammps_neighlist_element_neighbors()*

---

int **lammps_find_compute_neighlist**(void *handle, const char *id, int request)

Find index of a neighbor list requested by a compute

The neighbor list request from a compute is identified by the compute ID and the request ID. The request ID is typically 0, but will be

0 in case a compute has multiple neighbor list requests.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **id** – Identifier of compute instance
- **reqid** – request id to identify neighbor list in case there are multiple requests from the same compute

**Returns**

return neighbor list index if found, otherwise -1

---

int **lammps_find_fix_neighlist**(void *handle, const char *id, int request)

Find index of a neighbor list requested by a fix

The neighbor list request from a fix is identified by the fix ID and the request ID. The request ID is typically 0, but will be > 0 in case a fix has multiple neighbor list requests.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **id** – Identifier of fix instance
- **reqid** – request id to identify neighbor list in case there are multiple requests from the same fix

**Returns**

return neighbor list index if found, otherwise -1

---

int **lammps_find_pair_neighlist**(void *handle, const char *style, int exact, int nsub, int request)

Find index of a neighbor list requested by a pair style

This function determines which of the available neighbor lists for pair styles matches the given conditions. It first matches the style name. If exact is 1 the name must match exactly, if exact is 0, a regular expression or sub-string match is done. If the pair style is hybrid or hybrid/overlay the style is matched against the sub styles instead. If a the same pair style is used multiple times as a sub-style, the nsub argument must be > 0 and represents the nth instance of the sub-style (same as for the pair_coeff command, for example). In that case nsub=0 will not produce a match and this function will return -1.

The final condition to be checked is the request ID (reqid). This will normally be 0, but some pair styles request multiple neighbor lists and set the request ID to a value > 0.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **style** – String used to search for pair style instance
- **exact** – Flag to control whether style should match exactly or only a regular expression / sub-string match is applied.
- **nsub** – match nsub-th hybrid sub-style instance of the same style
- **reqid** – request id to identify neighbor list in case there are multiple requests from the same pair style instance

**Returns**

return neighbor list index if found, otherwise -1

---

int **lammps_neighlist_num_elements**(void *handle, int idx)

Return the number of entries in the neighbor list with given index

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **idx** – neighbor list index

**Returns**

return number of entries in neighbor list, -1 if idx is not a valid index

---

void **lammps_neighlist_element_neighbors**(void *handle, int idx, int element, int *iatom, int *numneigh, int **neighbors)

Return atom local index, number of neighbors, and array of neighbor local atom indices of neighbor list entry

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **idx** – index of this neighbor list in the list of all neighbor lists
- **element** – index of this neighbor list entry
- **iatom** – **[out]** local atom index (i.e. in the range [0, nlocal + nghost), -1 if invalid idx or element value
- **numneigh** – **[out]** number of neighbors of atom iatom or 0
- **neighbors** – **[out]** pointer to array of neighbor atom local indices or NULL

### 1.1.8 Configuration information

This section documents the following functions:

- *lammps_version()*
- *lammps_get_os_info()*
- *lammps_config_has_mpi_support()*
- *lammps_config_has_omp_support()*
- *lammps_config_has_gzip_support()*
- *lammps_config_has_png_support()*
- *lammps_config_has_jpeg_support()*
- *lammps_config_has_ffmpeg_support()*
- *lammps_config_has_exceptions()*
- *lammps_config_has_package()*
- *lammps_config_package_count()*
- *lammps_config_package_name()*
- *lammps_config_accelerator()*
- *lammps_has_gpu_device()*
- lammps_gpu_device_info()
- *lammps_has_style()*
- *lammps_style_count()*
- *lammps_style_name()*
- *lammps_has_id()*
- *lammps_id_count()*
- *lammps_id_name()*

---

These library functions can be used to query the LAMMPS library for compile time settings and included packages and styles. This enables programs that use the library interface to determine whether the linked LAMMPS library is compatible with the requirements of the application without crashing during the LAMMPS functions (e.g. due to missing pair styles from packages) or to choose between different options (e.g. whether to use `lj/cut`, `lj/cut/opt`, `lj/cut/omp` or `lj/cut/intel`). Most of the functions can be called directly without first creating a LAMMPS instance. While crashes within LAMMPS may be recovered from by enabling *exceptions*, avoiding them proactively is a safer approach.

Listing 1: Example for using configuration settings functions

```
#include "library.h"
#include <stdio.h>

int main(int argc, char **argv)
{
  void *handle;
```

(continues on next page)

(continued from previous page)

```
  handle = lammps_open_no_mpi(0, NULL, NULL);
  lammps_file(handle, "in.missing");
  if (lammps_has_error(handle)) {
    char errmsg[256];
    int errtype;
    errtype = lammps_get_last_error_message(handle, errmsg, 256);
    fprintf(stderr, "LAMMPS failed with error: %s\n", errmsg);
    return 1;
  }
  /* write compressed dump file depending on available of options */
  if (lammps_has_style(handle, "dump", "atom/zstd")) {
    lammps_command(handle, "dump d1 all atom/zstd 100 dump.zst");
  } else if (lammps_has_style(handle, "dump", "atom/gz")) {
    lammps_command(handle, "dump d1 all atom/gz 100 dump.gz");
  } else if (lammps_config_has_gzip_support()) {
    lammps_command(handle, "dump d1 all atom 100 dump.gz");
  } else {
    lammps_command(handle, "dump d1 all atom 100 dump");
  }
  lammps_close(handle);
  return 0;
}
```

---

int **lammps_version**(void *handle)

Get numerical representation of the *LAMMPS* version date.

The *lammps_version()* function returns an integer representing the version of the LAMMPS code in the format YYYYMMDD. This can be used to implement backward compatibility in software using the LAMMPS library interface. The specific format guarantees, that this version number is growing with every new LAMMPS release.

**Parameters**
**handle** – pointer to a previously created *LAMMPS* instance

**Returns**
an integer representing the version data in the format YYYYMMDD

---

void **lammps_get_os_info**(char *buffer, int buf_size)

Get operating system and architecture information

Added in version 9Oct2020.

The *lammps_get_os_info()* function can be used to retrieve detailed information about the hosting operating system and compiler/runtime.

A suitable buffer for a C-style string has to be provided and its length. The assembled text will be truncated to not overflow this buffer. The string is typically a few hundred bytes long.

**Parameters**

- **buffer** – string buffer to copy the information to
- **buf_size** – size of the provided string buffer

---

int **lammps_config_has_mpi_support**()

This function is used to query whether *LAMMPS* was compiled with a real MPI library or in serial. For the real MPI library it reports the size of the MPI communicator in bytes (4 or 8), which allows to check for compatibility with a hosting code.

**Returns**
0 when compiled with MPI STUBS, otherwise the MPI_Comm size in bytes

---

int **lammps_config_has_omp_support**()

This function is used to query whether *LAMMPS* was compiled with OpenMP enabled.

Added in version 10Sep2025.

***See also***
*lammps_config_has_mpi_support()*

**Returns**
1 when compiled with OpenMP enabled, otherwise 0

---

int **lammps_config_has_gzip_support**()

Check if the *LAMMPS* library supports reading or writing compressed files via a pipe to gzip or similar compression programs

Several LAMMPS commands (e.g., *read_data command*, *write_data command*, *dump styles atom, custom, and xyz*) support reading and writing compressed files via creating a pipe to the `gzip` program. This function checks whether this feature was *enabled at compile time*. It does **not** check whether``gzip`` or any other supported compression programs themselves are installed and usable.

**Returns**
1 if yes, otherwise 0

---

int **lammps_config_has_png_support**()

Check if the *LAMMPS* library supports writing PNG format images

The LAMMPS *dump style image* supports writing multiple image file formats. Most of them, however, need support from an external library, and using that has to be *enabled at compile time*. This function checks whether support for the PNG image file format is available in the current LAMMPS library.

**Returns**
1 if yes, otherwise 0

---

int **lammps_config_has_jpeg_support**()

Check if the *LAMMPS* library supports writing JPEG format images

The LAMMPS *dump style image* supports writing multiple image file formats. Most of them, however, need support from an external library, and using that has to be *enabled at compile time*. This function checks whether support for the JPEG image file format is available in the current LAMMPS library.

**Returns**
1 if yes, otherwise 0

---

int **lammps_config_has_ffmpeg_support**()

Check if the *LAMMPS* library supports creating movie files via a pipe to ffmpeg

The LAMMPS *dump style movie* supports generating movies from images on-the-fly via creating a pipe to the ffmpeg program. This function checks whether this feature was *enabled at compile time*. It does **not** check whether the `ffmpeg` itself is installed and usable.

**Returns**
1 if yes, otherwise 0

---

int **lammps_config_has_exceptions**()

Check whether *LAMMPS* errors will throw C++ exceptions.

Deprecated since version 21Nov2023: LAMMPS has now exceptions always enabled, so this function will now always return 1 and can be removed from applications using the library interface.

In case of an error, LAMMPS will either abort or throw a C++ exception. The latter has to be *enabled at compile time*. This function checks if exceptions were enabled.

When using the library interface with C++ exceptions enabled, the library interface functions will "catch" them and the error status can then be checked by calling *lammps_has_error()* and the most recent error message can be retrieved via *lammps_get_last_error_message()*. This can allow to restart a calculation or delete and recreate the LAMMPS instance when C++ exceptions are enabled. One application of using exceptions this way is the *LAMMPS-GUI*. If C++ exceptions are disabled and an error happens during a call to LAMMPS, the application will terminate.

**Returns**
1 if yes, otherwise 0

---

int **lammps_config_has_package**(const char*)

Check whether a specific package has been included in *LAMMPS*

This function checks whether the LAMMPS library in use includes the specific *LAMMPS package* provided as argument.

**Parameters**
**name** – string with the name of the package

**Returns**
1 if included, 0 if not.

---

int **lammps_config_package_count**()

Count the number of installed packages in the *LAMMPS* library.

This function counts how many *LAMMPS packages* are included in the LAMMPS library in use.

**Returns**
number of packages included

---

int **lammps_config_package_name**(int, char*, int)

Get the name of a package in the list of installed packages in the *LAMMPS* library.

This function copies the name of the package with the index *idx* into the provided C-style string buffer. The length of the buffer must be provided as *buf_size* argument. If the name of the package exceeds the length of the buffer, it will be truncated accordingly. If the index is out of range, the function returns 0 and *buffer* is set to an empty string, otherwise 1;

**Parameters**

- **idx** – index of the package in the list of included packages (0 <= idx < package count)
- **buffer** – string buffer to copy the name of the package to
- **buf_size** – size of the provided string buffer

**Returns**
1 if successful, otherwise 0

---

int **lammps_config_accelerator**(const char*, const char*, const char*)

Check for compile time settings in accelerator packages included in *LAMMPS*.

This function checks availability of compile time settings of included *accelerator packages* in LAMMPS. Supported packages names are "GPU", "KOKKOS", "INTEL", and "OPENMP". Supported categories are "api" with possible settings "cuda", "hip", "phi", "pthreads", "opencl", "openmp", and "serial", and "precision" with possible settings "double", "mixed", and "single". If the combination of package, category, and setting is available, the function returns 1, otherwise 0.

**Parameters**

- **package** – string with the name of the accelerator package
- **category** – string with the category name of the setting
- **setting** – string with the name of the specific setting

**Returns**
1 if available, 0 if not.

---

int **lammps_has_gpu_device**()

Check for presence of a viable GPU package device

Added in version 14May2021.

The *lammps_has_gpu_device()* function checks at runtime if an accelerator device is present that can be used with the *GPU package*. If at least one suitable device is present the function will return 1, otherwise 0.

More detailed information about the available device or devices can be obtained by calling the *lammps_get_gpu_device_info()* function.

**Returns**
1 if viable device is available, 0 if not.

---

void **lammps_get_gpu_device_info**(char *buffer, int buf_size)

Get GPU package device information

Added in version 14May2021.

The *lammps_get_gpu_device_info()* function can be used to retrieve detailed information about any accelerator devices that are viable for use with the *GPU package*. It will produce a string that is equivalent to the output of the `nvc_get_device` or `ocl_get_device` or `hip_get_device` tools that are compiled alongside LAMMPS if the GPU package is enabled.

A suitable buffer for a C-style string has to be provided and its length. The assembled text will be truncated to not overflow this buffer. This string can be several kilobytes long, if multiple devices are present.

**Parameters**

- **buffer** – string buffer to copy the information to
- **buf_size** – size of the provided string buffer

---

int **lammps_has_style**(void*, const char*, const char*)

Check if a specific style has been included in *LAMMPS*

This function checks if the LAMMPS library in use includes the specific *style* of a specific *category* provided as an argument. Valid categories are: *atom*, *integrate*, *minimize*, *pair*, *bond*, *angle*, *dihedral*, *improper*, *kspace*, *compute*, *fix*, *region*, *dump*, and *command*.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **category** – category of the style
- **name** – name of the style

**Returns**
1 if included, 0 if not.

---

int **lammps_style_count**(void*, const char*)

Count the number of styles of category in the *LAMMPS* library.

This function counts how many styles in the provided *category* are included in the LAMMPS library in use. Please see *lammps_has_style()* for a list of valid categories.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **category** – category of styles

**Returns**
number of styles in category

---

int **lammps_style_name**(void*, const char*, int, char*, int)

Look up the name of a style by index in the list of style of a given category in the *LAMMPS* library.

This function copies the name of the *category* style with the index *idx* into the provided C-style string buffer. The length of the buffer must be provided as *buf_size* argument. If the name of the style exceeds the length of the buffer, it will be truncated accordingly. If the index is out of range, the function returns 0 and *buffer* is set to an empty string, otherwise 1.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **category** – category of styles
- **idx** – index of the style in the list of *category* styles (0 <= idx < style count)
- **buffer** – string buffer to copy the name of the style to
- **buf_size** – size of the provided string buffer

**Returns**

1 if successful, otherwise 0

---

int **lammps_has_id**(void*, const char*, const char*)

Check if a specific ID exists in the current *LAMMPS* instance

Added in version 9Oct2020.

This function checks if the current LAMMPS instance a *category* ID of the given *name* exists. Valid categories are: *compute*, *dump*, *fix*, *group*, *molecule*, *region*, and *variable*.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **category** – category of the id
- **name** – name of the id

**Returns**

1 if included, 0 if not.

---

int **lammps_id_count**(void*, const char*)

Count the number of IDs of a category.

Added in version 9Oct2020.

This function counts how many IDs in the provided *category* are defined in the current LAMMPS instance. Please see *lammps_has_id()* for a list of valid categories.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **category** – category of IDs

**Returns**
number of IDs in category

---

int **lammps_id_name**(void*, const char*, int, char*, int)

Look up the name of an ID by index in the list of IDs of a given category.

Added in version 9Oct2020.

This function copies the name of the *category* ID with the index *idx* into the provided C-style string buffer. The length of the buffer must be provided as *buf_size* argument. If the name of the style exceeds the length of the buffer, it will be truncated accordingly. If the index is out of range, the function returns 0 and *buffer* is set to an empty string, otherwise 1.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **category** – category of IDs
- **idx** – index of the ID in the list of *category* styles (0 <= idx < count)
- **buffer** – string buffer to copy the name of the style to
- **buf_size** – size of the provided string buffer

**Returns**
1 if successful, otherwise 0

### 1.1.9 Utility functions

To simplify some tasks, the library interface contains these utility functions. They do not directly call the LAMMPS library.

- *lammps_encode_image_flags()*
- *lammps_decode_image_flags()*
- *lammps_set_fix_external_callback()*
- *lammps_fix_external_get_force()*
- *lammps_fix_external_set_energy_global()*
- *lammps_fix_external_set_energy_peratom()*
- *lammps_fix_external_set_virial_global()*
- *lammps_fix_external_set_virial_peratom()*
- *lammps_fix_external_set_vector_length()*
- *lammps_fix_external_set_vector()*
- *lammps_flush_buffers()*
- *lammps_free()*
- *lammps_is_running()*
- *lammps_force_timeout()*
- *lammps_has_error()*

- *lammps_get_last_error_message()*
- *lammps_set_show_error()*
- *lammps_python_api_version()*

The *lammps_free()* function is a clean-up function to free memory that the library had allocated previously via other function calls. Look for notes in the descriptions of the individual commands where such memory buffers were allocated that require the use of *lammps_free()*.

---

int **lammps_encode_image_flags**(int ix, int iy, int iz)

Encode three integer image flags into a single imageint.

The prototype for this function when compiling with `-DLAMMPS_BIGBIG` is:

```
int64_t lammps_encode_image_flags(int ix, int iy, int iz);
```

This function performs the bit-shift, addition, and bit-wise OR operations necessary to combine the values of three integers representing the image flags in x-, y-, and z-direction. Unless LAMMPS is compiled with -DLAMMPS_BIGBIG, those integers are limited 10-bit signed integers [-512, 511]. Otherwise the return type changes from `int` to `int64_t` and the valid range for the individual image flags becomes [-1048576,1048575], i.e. that of a 21-bit signed integer. There is no check on whether the arguments conform to these requirements.

**Parameters**

- **ix** – image flag value in x
- **iy** – image flag value in y
- **iz** – image flag value in z

**Returns**

encoded image flag integer

---

void **lammps_decode_image_flags**(int image, int *flags)

Decode a single image flag integer into three regular integers

The prototype for this function when compiling with `-DLAMMPS_BIGBIG` is:

```
void lammps_decode_image_flags(int64_t image, int *flags);
```

This function does the reverse operation of *lammps_encode_image_flags()* and takes an image flag integer does the bit-shift and bit-masking operations to decode it and stores the resulting three regular integers into the buffer pointed to by *flags*.

**Parameters**

- **image** – encoded image flag integer
- **flags** – pointer to storage where the decoded image flags are stored.

---

void **lammps_set_fix_external_callback**(void *handle, const char *id, FixExternalFnPtr funcptr, void *ptr)

Set up the callback function for a fix external instance with the given ID.

Fix *external* allows programs that are running LAMMPS through its library interface to modify certain LAMMPS properties on specific timesteps, similar to the way other fixes do.

This function sets the callback function for use with the "pf/callback" mode. The function has to have C language bindings with the prototype:

```
void func(void *ptr, bigint timestep, int nlocal, tagint *ids, double **x, double␣
↪**fexternal);
```

The argument *ptr* to this function will be stored in fix external and the passed as the first argument calling the callback function *func()*. This would usually be a pointer to the active LAMMPS instance, i.e. the same pointer as the *handle* argument. This would be needed to call functions that set the global or per-atom energy or virial contributions from within the callback function.

The callback mechanism is one of the two modes of how forces and can be applied to a simulation with the help of fix external. The alternative is the array mode where you call *lammps_fix_external_get_force()*.

Please see the documentation for *fix external* for more information about how to use the fix and how to couple it with an external code.

Changed in version 28Jul2021.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **id** – fix ID of fix external instance
- **funcptr** – pointer to callback function
- **ptr** – pointer to object in calling code, passed to callback function as first argument

---

double ****lammps_fix_external_get_force**(void *handle, const char *id)

Get pointer to the force array storage in a fix external instance with the given ID.

Added in version 28Jul2021.

Fix *external* allows programs that are running LAMMPS through its library interface to add or modify certain LAMMPS properties on specific timesteps, similar to the way other fixes do.

This function provides access to the per-atom force storage in a fix external instance with the given fix-ID to be added to the individual atoms when using the "pf/array" mode. The *fexternal* array can be accessed like other "native" per-atom arrays accessible via the *lammps_extract_atom()* function. Please note that the array stores holds the forces for *local* atoms for each MPI ranks, in the order determined by the neighbor list build. Because the underlying data structures can change as well as the order of atom as they migrate between MPI processes because of the domain decomposition parallelization, this function should be always called immediately before the forces are going to be set to get an up-to-date pointer. You can use, for example, *lammps_extract_setting()* to obtain the number of local atoms *nlocal* and then assume the dimensions of the returned force array as `double force[nlocal][3]`.

This is an alternative to the callback mechanism in fix external set up by *lammps_set_fix_external_callback()*. The main difference is that this mechanism can be used

when forces are be pre-computed and the control alternates between LAMMPS and the external code, while the callback mechanism can call the external code to compute the force when the fix is triggered and needs them.

Please see the documentation for *fix external* for more information about how to use the fix and how to couple it with an external code.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **id** – fix ID of fix external instance

**Returns**

a pointer to the per-atom force array allocated by the fix

---

void **lammps_fix_external_set_energy_global**(void *handle, const char *id, double eng)

Set the global energy contribution for a fix external instance with the given ID.

Added in version 28Jul2021.

This is a companion function to *lammps_set_fix_external_callback()* and *lammps_fix_external_get_force()* to also set the contribution to the global energy from the external code. The value of the *eng* argument will be stored in the fix and applied on the current and all following timesteps until changed by another call to this function. The energy is in energy units as determined by the current *units* settings and is the **total** energy of the contribution. Thus when running in parallel all MPI processes have to call this function with the **same** value and this will be returned as scalar property of the fix external instance when accessed in LAMMPS input commands or from variables.

Please see the documentation for *fix external* for more information about how to use the fix and how to couple it with an external code.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **id** – fix ID of fix external instance
- **eng** – total energy to be added to the global energy

---

void **lammps_fix_external_set_energy_peratom**(void *handle, const char *id, double *eng)

Set the per-atom energy contribution for a fix external instance with the given ID.

Added in version 28Jul2021.

This is a companion function to *lammps_set_fix_external_callback()* to set the per-atom energy contribution due to the fix from the external code as part of the callback function. For this to work, the handle to the LAMMPS object must be passed as the *ptr* argument when registering the callback function.

Please see the documentation for *fix external* for more information about how to use the fix and how to couple it with an external code.

> **Note**
>
> This function is fully independent from *lammps_fix_external_set_energy_global()* and will **NOT** add any contributions to the global energy tally and **NOT** check whether the sum of the contributions added here are consistent with the global added energy.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **id** – fix ID of fix external instance
- **eng** – pointer to array of length nlocal with the energy to be added to the per-atom energy

---

void **lammps_fix_external_set_virial_global**(void *handle, const char *id, double *virial)

Set the global virial contribution for a fix external instance with the given ID.

Added in version 28Jul2021.

This is a companion function to *lammps_set_fix_external_callback()* and *lammps_fix_external_get_force()* to set the contribution to the global virial from the external code.

The 6 values of the *virial* array will be stored in the fix and applied on the current and all following timesteps until changed by another call to this function. The components of the virial need to be stored in the order: *xx*, *yy*, *zz*, *xy*, *xz*, *yz*. In LAMMPS the virial is stored internally as *stress*volume* in units of *pressure*volume* as determined by the current *units* settings and is the **total** contribution. Thus when running in parallel all MPI processes have to call this function with the **same** value and this will then be added by fix external.

Please see the documentation for *fix external* for more information about how to use the fix and how to couple it with an external code.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **id** – fix ID of fix external instance
- **virial** – the 6 global stress tensor components to be added to the global virial

---

void **lammps_fix_external_set_virial_peratom**(void *handle, const char *id, double **virial)

Set the per-atom virial contribution for a fix external instance with the given ID.

Added in version 28Jul2021.

This is a companion function to *lammps_set_fix_external_callback()* to set the per-atom virial contribution due to the fix from the external code as part of the callback function. For this to work, the handle to the LAMMPS object must be passed as the *ptr* argument when registering the callback function.

The order and units of the per-atom stress tensor elements are the same as for the global virial. The code in fix external assumes the dimensions of the per-atom virial array is `double virial[nlocal][6]`.

Please see the documentation for *fix external* for more information about how to use the fix and how to couple it with an external code.

> **Note**
>
> This function is fully independent from *lammps_fix_external_set_virial_global()* and will **NOT** add any contributions to the global virial tally and **NOT** check whether the sum of the contributions added here are consistent with the global added virial.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **id** – fix ID of fix external instance
- **virial** – a list of nlocal entries with the 6 per-atom stress tensor components to be added to the per-atom virial

---

void **lammps_fix_external_set_vector_length**(void *handle, const char *id, int len)

Set the vector length for a global vector stored with fix external for analysis

Added in version 28Jul2021.

This is a companion function to *lammps_set_fix_external_callback()* and *lammps_fix_external_get_force()* to set the length of a global vector of properties that will be stored with the fix via *lammps_fix_external_set_vector()*.

This function needs to be called **before** a call to *lammps_fix_external_set_vector()* and **before** a run or minimize command. When running in parallel it must be called from **all** MPI processes and with the same length parameter.

Please see the documentation for *fix external* for more information about how to use the fix and how to couple it with an external code.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **id** – fix ID of fix external instance
- **len** – length of the global vector to be stored with the fix

---

void **lammps_fix_external_set_vector**(void *handle, const char *id, int idx, double val)

Store a global vector value for a fix external instance with the given ID.

Added in version 28Jul2021.

This is a companion function to *lammps_set_fix_external_callback()* and *lammps_fix_external_get_force()* to set the values of a global vector of properties that will be stored with the fix. And can be accessed from within LAMMPS input commands (e.g., fix ave/time or variables) when used in a vector context.

This function needs to be called **after** a call to *lammps_fix_external_set_vector_length()* and the and **before** a run or minimize command. When running in parallel it must be called from **all** MPI processes and with the **same** index and value parameters. The value is assumed to be extensive.

Please see the documentation for *fix external* for more information about how to use the fix and how to couple it with an external code.

**Note**

The index in the *idx* parameter is 1-based, i.e. the first element is set with idx = 1 and the last element of the vector with idx = N, where N is the value of the *len* parameter of the call to *lammps_fix_external_set_vector_length()*.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.
- **id** – fix ID of fix external instance
- **idx** – 1-based index of in global vector
- **val** – value to be stored in global vector

---

void **lammps_flush_buffers**(void *ptr)

Flush output buffers

This function can be used to flush buffered output to be written to screen and logfile pointers to simplify capturing output from LAMMPS library calls.

**Parameters**

**handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.

---

void **lammps_free**(void *ptr)

Free memory buffer allocated by *LAMMPS*.

Some of the LAMMPS C library interface functions return data as pointer to a buffer that has been allocated by LAMMPS or the library interface. This function can be used to delete those in order to avoid memory leaks.

**Parameters**

**ptr** – pointer to data allocated by *LAMMPS*

---

int **lammps_is_running**(void *handle)

Check if *LAMMPS* is currently inside a run or minimization

This function can be used from signal handlers or multi-threaded applications to determine if the *LAMMPS* instance is currently active.

**Parameters**

**handle** – pointer to a previously created *LAMMPS* instance cast to `void *`.

**Returns**

0 if idle or >0 if active

---

void **lammps_force_timeout**(void *handle)

Force a timeout to stop an ongoing run cleanly.

This function can be used from signal handlers or multi-threaded applications to cleanly terminate an ongoing run.

**Parameters**
**handle** – pointer to a previously created *LAMMPS* instance cast to `void *`

---

int **lammps_has_error**(void *handle)

Check if there is a (new) error message available

This function can be used to query if an error inside of LAMMPS has thrown a *C++ exception*.

> **Note**
>
> Changed in version 2Aug2023.
>
> The *handle* pointer may be `NULL` for this function, as would be the case when a call to create a LAMMPS instance has failed. Then this function will not check the error status inside the LAMMPS instance, but instead would check the global error buffer of the library interface.

**Parameters**
**handle** – pointer to a previously created *LAMMPS* instance cast to `void *` or NULL

**Returns**
0 on no error, 1 on error.

---

int **lammps_get_last_error_message**(void *handle, char *buffer, int buf_size)

Copy the last error message into the provided buffer

This function can be used to retrieve the error message that was set in the event of an error inside of LAMMPS which resulted in a *C++ exception*. A suitable buffer for a C-style string has to be provided and its length. If the internally stored error message is longer, it will be truncated accordingly. If the buffer is a NULL pointer, then nothing will be copied. The return value of the function corresponds to the kind of error: a "1" indicates an error that occurred on all MPI ranks and is often recoverable, while a "2" indicates an abort that would happen only in a single MPI rank and thus may not be recoverable, as other MPI ranks may be waiting on the failing MPI ranks to send messages.

> **Note**
>
> Changed in version 2Aug2023.
>
> The *handle* pointer may be `NULL` for this function, as would be the case when a call to create a LAMMPS instance has failed. Then this function will not check the error buffer inside the LAMMPS instance, but instead would check the global error buffer of the library interface.
>
> Changed in version 21Nov2023.
>
> The *buffer* pointer may be `NULL`. This will clear any error status without copying the error message.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *` or NULL.
- **buffer** – string buffer to copy the error message to, may be NULL
- **buf_size** – size of the provided string buffer

**Returns**
1 when all ranks had the error, 2 on a single rank error.

---

int **lammps_set_show_error**(void *handle, const int flag)

Enable or disable direct printing of error messages

Added in version 2Apr2025.

This function can be used to stop LAMMPS from printing error messages *before* LAMMPS throws a *C++ exception*. This is so it may be left to the code calling the library interface whether to check for them, and retrieve and print error messages using the library interface functions *lammps_has_error()* and *lammps_get_last_error_message()*. The function returns the previous setting so that one can easily override the setting temporarily and restore it afterwards.

**Parameters**

- **handle** – pointer to a previously created *LAMMPS* instance cast to `void *` or NULL
- **flag** – enable (not 0) or disable (0) printing error messages before throwing exception

**Returns**
previous setting of the flag

---

int **lammps_python_api_version**()

Return API version of embedded Python interpreter

Added in version 3Nov2022.

This function is used by the ML-IAP python code (mliappy) to verify the API version of the embedded python interpreter of the PYTHON package. It returns -1 if the PYTHON package is not enabled.

**Returns**
PYTHON_API_VERSION constant of the python interpreter or -1

### 1.1.10 Extending the C API

The functionality of the LAMMPS library interface has historically been motivated by the needs of its users. Functions have been added or expanded as they were needed and used. Contributions to the interface are always welcome. However with a refactoring of the library interface and its documentation that started in Spring 2020, there are now a few requirements for including new changes or extensions.

- New functions should be orthogonal to existing ones and not implement functionality that can already be achieved with the existing APIs.
- All changes and additions should be documented with Doxygen style comments and references to those functions added to the corresponding files in the `doc/src` folder.
- If possible, new unit tests to test those new features should be added.

- New features should also be implemented and documented not just for the C interface, but also the Python and Fortran interfaces.
- All additions should work and be compatible with `-DLAMMPS_BIGBIG`, `-DLAMMPS_SMALLBIG` as well as when compiling with and without MPI support.
- The `library.h` file should be kept compatible to C code at a level similar to C89. Its interfaces may not reference any custom data types (e.g. `bigint`, `tagint`, and so on) that are only known inside of LAMMPS; instead `int` and `int64_t` should be used.
- only use C style comments, not C++ style

Please note that these are not **strict** requirements, but the LAMMPS developers very much appreciate, if they are followed and can assist with implementing what is missing. It helps maintaining the code base and keeping it consistent.

---

## 1.2 LAMMPS Python API

The LAMMPS Python module enables calling the LAMMPS C library API from Python by dynamically loading functions in the LAMMPS shared library through the Python ctypes module. Because of the dynamic loading, it is **required** that LAMMPS is compiled in *"shared" mode*. The Python interface is object-oriented, but otherwise tries to be very similar to the C library API. More information on this is in the *Use Python with LAMMPS* section of the manual. Use of the LAMMPS Python module is described in *The lammps Python module*.

---

## 1.3 LAMMPS Fortran API

The LAMMPS Fortran module is a wrapper around calling functions from the LAMMPS C library API. This is done using the ISO_C_BINDING feature in Fortran 2003. The interface is object-oriented but otherwise tries to be very similar to the C library API and the basic Python module.

### 1.3.1 The *LIBLAMMPS* Fortran Module

The *LIBLAMMPS* module provides an interface to call LAMMPS from Fortran. It is based on the LAMMPS C library interface and requires a fully Fortran 2003-compatible compiler to be compiled. It is designed to be self-contained and not require any support functions written in C, C++, or Fortran other than those in the C library interface and the LAMMPS Fortran module itself.

While C libraries have a defined binary interface (ABI) and can thus be used from multiple compiler versions from different vendors as long as they are compatible with the hosting operating system, the same is not true for Fortran programs. Thus, the LAMMPS Fortran module needs to be compiled alongside the code using it from the source code in `fortran/lammps.f90` *and* with the same compiler used to build the rest of the Fortran code that interfaces to LAMMPS. When linking, you also need to *link to the LAMMPS library*. A typical command for a simple program using the Fortran interface would be:

```
mpifort -o testlib.x lammps.f90 testlib.f90 -L. -llammps
```

Please note that the MPI compiler wrapper is only required when the calling the library *from* an MPI-parallelized program. Otherwise, using the plain Fortran compiler (gfortran, ifort, flang, etc.) will suffice, since there are no direct references to MPI library features, definitions and subroutine calls; MPI communicators are referred to by their integer index representation as required by the Fortran MPI interface. It may be necessary to link to additional libraries, depending on how LAMMPS was configured and whether the LAMMPS library *was compiled as a static or dynamic library*.

If the LAMMPS library itself has been compiled with MPI support, the resulting executable will be able to run LAMMPS in parallel with `mpirun`, `mpiexec`, or equivalent. This may be either on the "world" communicator or a sub-communicator created by the calling Fortran code. If, on the other hand, the LAMMPS library has been compiled **without** MPI support, each LAMMPS instance will run independently using just one processor.

Please also note that the order of the source files matters: the `lammps.f90` file needs to be compiled first, since it provides the *LIBLAMMPS* module that would need to be imported by the calling Fortran code in order to uses the Fortran interface. A working example can be found together with equivalent examples in C and C++ in the `examples/COUPLE/simple` folder of the LAMMPS distribution.

**Fortran compiler compatibility**

A fully Fortran 2003 compatible Fortran compiler is required. This means that currently only GNU Fortran 9 and later are compatible and thus the default compilers of Red Hat or CentOS 7 and Ubuntu 18.04 LTS and not compatible. Either newer compilers need to be installed or the Linux updated.

### 1.3.2 Creating or deleting a LAMMPS object

With the Fortran interface, the creation of a *LAMMPS* instance is included in the constructor for creating the *lammps()* derived type. To import the definition of that type and its type-bound procedures, you need to add a `USE LIBLAMMPS` statement. Internally, it will call either *lammps_open_fortran()* or *lammps_open_no_mpi()* from the C library API to create the class instance. All arguments are optional and *lammps_mpi_init()* will be called automatically if it is needed. Similarly, optional calls to *lammps_mpi_finalize()*, *lammps_kokkos_finalize()*, *lammps_python_finalize()*, and *lammps_plugin_finalize()* are integrated into the *close()* function and triggered with the optional logical argument set to `.TRUE.`. Here is a simple example:

```
PROGRAM testlib
  USE LIBLAMMPS                 ! include the LAMMPS library interface
  IMPLICIT NONE
  TYPE(lammps) :: lmp           ! derived type to hold LAMMPS instance
  CHARACTER(LEN=12), PARAMETER :: args(3) = &
      [ CHARACTER(LEN=12) :: 'liblammps', '-log', 'none' ]

  ! create a LAMMPS instance (and initialize MPI)
  lmp = lammps(args)
  ! get and print numerical version code
  PRINT*, 'LAMMPS Version: ', lmp%version()
  ! delete LAMMPS instance (and shutdown MPI)
  CALL lmp%close(.TRUE.)
END PROGRAM testlib
```

It is also possible to pass command-line flags from Fortran to C/C++ and thus make the resulting executable behave similarly to the standalone executable (it will ignore the *-in/-i* flag, though). This allows using the command-line to configure accelerator and suffix settings, configure screen and logfile output, or to set index style variables from the command-line and more. Here is a correspondingly adapted version of the previous example:

```
PROGRAM testlib2
  USE LIBLAMMPS                 ! include the LAMMPS library interface
  IMPLICIT NONE
  TYPE(lammps) :: lmp           ! derived type to hold LAMMPS instance
  CHARACTER(LEN=128), ALLOCATABLE :: command_args(:)
```

(continues on next page)

(continued from previous page)

```
  INTEGER :: i, argc

  ! copy command-line flags to `command_args()`
  argc = COMMAND_ARGUMENT_COUNT()
  ALLOCATE(command_args(0:argc))
  DO i=0, argc
    CALL GET_COMMAND_ARGUMENT(i, command_args(i))
  END DO

  ! create a LAMMPS instance (and initialize MPI)
  lmp = lammps(command_args)
  ! get and print numerical version code
  PRINT*, 'Program name:   ', command_args(0)
  PRINT*, 'LAMMPS Version: ', lmp%version()
  ! delete LAMMPS instance (and shuts down MPI)
  CALL lmp%close(.TRUE.)
  DEALLOCATE(command_args)
END PROGRAM testlib2
```

### 1.3.3 Executing LAMMPS commands

Once a LAMMPS instance is created, it is possible to "drive" the LAMMPS simulation by telling LAMMPS to read commands from a file or to pass individual or multiple commands from strings or lists of strings. This is done similarly to how it is implemented in the *C library interface*. Before handing off the calls to the C library interface, the corresponding Fortran versions of the calls (*file()*, *command()*, *commands_list()*, and *commands_string()*) have to make copies of the strings passed as arguments so that they can be modified to be compatible with the requirements of strings in C without affecting the original strings. Those copies are automatically deleted after the functions return. Below is a small demonstration of the uses of the different functions.

```
PROGRAM testcmd
  USE LIBLAMMPS
  TYPE(lammps) :: lmp
  CHARACTER(LEN=512) :: cmds
  CHARACTER(LEN=40), ALLOCATABLE :: cmdlist(:)
  CHARACTER(LEN=10) :: trimmed
  INTEGER :: i

  lmp = lammps()
  CALL lmp%file('in.melt')
  CALL lmp%command('variable zpos index 1.0')
  ! define 10 groups of 10 atoms each
  ALLOCATE(cmdlist(10))
  DO i=1, 10
    WRITE(trimmed,'(I10)') 10*i
    WRITE(cmdlist(i),'(A,I1,A,I10,A,A)')       &
        'group g', i-1, ' id ', 10*(i-1)+1, ':', ADJUSTL(trimmed)
  END DO
  CALL lmp%commands_list(cmdlist)
  ! run multiple commands from multi-line string
  cmds = 'clear' // NEW_LINE('A') //                        &
```

(continues on next page)

(continued from previous page)

```
      'region  box block 0 2 0 2 0 2' // NEW_LINE('A') //  &
      'create_box 1 box' // NEW_LINE('A') //               &
      'create_atoms 1 single 1.0 1.0 ${zpos}'
  CALL lmp%commands_string(cmds)
  CALL lmp%close(.TRUE.)
END PROGRAM testcmd
```

### 1.3.4 Accessing system properties

The C library interface allows the *extraction of different kinds of information* about the active simulation instance and also—in some cases—to apply modifications to it, and the Fortran interface provides access to the same data using Fortran-style, C-interoperable data types. In some cases, the Fortran library interface makes pointers to internal LAMMPS data structures accessible; when accessing them through the library interfaces, special care is needed to avoid data corruption and crashes. Please see the documentation of the individual type-bound procedures for details.

Below is an example demonstrating some of the possible uses.

```
PROGRAM testprop
  USE LIBLAMMPS
  USE, INTRINSIC :: ISO_C_BINDING, ONLY : c_double, c_int64_t, c_int
  USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY : OUTPUT_UNIT
  TYPE(lammps) :: lmp
  INTEGER(KIND=c_int64_t), POINTER :: natoms, ntimestep, bval
  REAL(KIND=c_double), POINTER :: dt, dval
  INTEGER(KIND=c_int), POINTER :: nfield, typ, ival
  INTEGER(KIND=c_int) :: i
  CHARACTER(LEN=11) :: key
  REAL(KIND=c_double) :: pe, ke

  lmp = lammps()
  CALL lmp%file('in.sysinit')
  natoms = lmp%extract_global('natoms')
  WRITE(OUTPUT_UNIT,'(A,I0,A)') 'Running a simulation with ', natoms, ' atoms'
  WRITE(OUTPUT_UNIT,'(I0,A,I0,A,I0,A)') lmp%extract_setting('nlocal'), &
      ' local and ', lmp%extract_setting('nghost'), ' ghost atoms. ', &
      lmp%extract_setting('ntypes'), ' atom types'

  CALL lmp%command('run 2 post no')

  ntimestep = lmp%last_thermo('step', 0)
  nfield = lmp%last_thermo('num', 0)
  WRITE(OUTPUT_UNIT,'(A,I0,A,I0)') 'Last thermo output on step: ', ntimestep, &
      ',  number of fields: ', nfield
  DO i=1, nfield
      key = lmp%last_thermo('keyword',i)
      typ = lmp%last_thermo('type',i)
      IF (typ == lmp%dtype%i32) THEN
          ival = lmp%last_thermo('data',i)
          WRITE(OUTPUT_UNIT,*) key, ':', ival
      ELSE IF (typ == lmp%dtype%i64) THEN
          bval = lmp%last_thermo('data',i)
```

(continues on next page)

(continued from previous page)

```
          WRITE(OUTPUT_UNIT,*) key, ':', bval
       ELSE IF (typ == lmp%dtype%r64) THEN
          dval = lmp%last_thermo('data',i)
          WRITE(OUTPUT_UNIT,*) key, ':', dval
       END IF
    END DO

    dt = lmp%extract_global('dt')
    ntimestep = lmp%extract_global('ntimestep')
    WRITE(OUTPUT_UNIT,'(A,I0,A,F4.1,A)') 'At step: ', ntimestep, &
        '  Changing timestep from', dt, ' to 0.5'
    dt = 0.5_c_double
    CALL lmp%command('run 2 post no')

    WRITE(OUTPUT_UNIT,'(A,I0)') 'At step: ', ntimestep
    pe = lmp%get_thermo('pe')
    ke = lmp%get_thermo('ke')
    WRITE(OUTPUT_UNIT,*) 'PE = ', pe
    WRITE(OUTPUT_UNIT,*) 'KE = ', ke

    CALL lmp%close(.TRUE.)
END PROGRAM testprop
```

### 1.3.5 The *LIBLAMMPS* module API

Below are the detailed descriptions of definitions and interfaces of the contents of the *LIBLAMMPS* Fortran interface to LAMMPS.

**type lammps**

Derived type that is the general class of the Fortran interface. It holds a reference to the *LAMMPS* class instance to which any of the included calls are forwarded.

**Type fields**

- **% handle** *[c_ptr]* :: reference to the LAMMPS class
- **% style** *[type(lammps_style)]* :: derived type to access lammps style constants
- **% type** *[type(lammps_type)]* :: derived type to access lammps type constants
- **% dtype** *[type(lammps_dtype)]* :: derived type to access lammps data type constants
- **% close** *[subroutine]* :: *close()*
- **% error** *[subroutine]* :: *error()*
- **% file** *[subroutine]* :: *file()*
- **% command** *[subroutine]* :: *command()*
- **% commands_list** *[subroutine]* :: *commands_list()*
- **% commands_string** *[subroutine]* :: *commands_string()*
- **% get_natoms** *[function]* :: *get_natoms()*
- **% get_thermo** *[function]* :: *get_thermo()*

- **% last_thermo** *[function]* :: *last_thermo()*
- **% extract_box** *[subroutine]* :: *extract_box()*
- **% reset_box** *[subroutine]* :: *reset_box()*
- **% memory_usage** *[subroutine]* :: *memory_usage()*
- **% get_mpi_comm** *[function]* :: *get_mpi_comm()*
- **% extract_setting** *[function]* :: *extract_setting()*
- **% extract_global** *[function]* :: *extract_global()*
- **% map_atom** *[function]* :: map_atom()
- **% extract_atom** *[function]* :: *extract_atom()*
- **% extract_compute** *[function]* :: *extract_compute()*
- **% extract_fix** *[function]* :: *extract_fix()*
- **% extract_variable** *[function]* :: *extract_variable()*
- **% set_variable** *[subroutine]* :: *set_variable()*
- **% set_string_variable** *[subroutine]* :: set_set_string_variable()
- **% set_internal_variable** *[subroutine]* :: *set_internal_variable()*
- **% eval** *[function]* :: *eval()*
- **% clearstep_compute** *[subroutine]* :: *clearstep_compute()*
- **% addstep_compute** *[subroutine]* :: *addstep_compute()*
- **% addstep_compute_all** *[subroutine]* :: *addstep_compute_all()*
- **% gather_atoms** *[subroutine]* :: *gather_atoms()*
- **% gather_atoms_concat** *[subroutine]* :: *gather_atoms_concat()*
- **% gather_atoms_subset** *[subroutine]* :: *gather_atoms_subset()*
- **% scatter_atoms** *[subroutine]* :: *scatter_atoms()*
- **% scatter_atoms_subset** *[subroutine]* :: *scatter_atoms_subset()*
- **% gather_bonds** *[subroutine]* :: *gather_bonds()*
- **% gather_angles** *[subroutine]* :: *gather_angles()*
- **% gather_dihedrals** *[subroutine]* :: *gather_dihedrals()*
- **% gather_impropers** *[subroutine]* :: *gather_impropers()*
- **% gather** *[subroutine]* :: *gather()*
- **% gather_concat** *[subroutine]* :: *gather_concat()*
- **% gather_subset** *[subroutine]* :: *gather_subset()*
- **% scatter** *[subroutine]* :: *scatter()*
- **% scatter_subset** *[subroutine]* :: *scatter_subset()*
- **% create_atoms** *[subroutine]* :: *create_atoms()*
- **% find_pair_neighlist** *[function]* :: *find_pair_neighlist()*
- **% find_fix_neighlist** *[function]* :: *find_fix_neighlist()*

- % **find_compute_neighlist** *[function]* :: *find_compute_neighlist()*
- % **neighlist_num_elements** *[function]* :: *neighlist_num_elements()*
- % **neighlist_element_neighbors** *[subroutine]* :: *neighlist_element_neighbors()*
- % **version** *[function]* :: *version()*
- % **get_os_info** *[subroutine]* :: *get_os_info()*
- % **config_has_mpi_support** *[function]* :: *config_has_mpi_support()*
- % **config_has_omp_support** *[function]* :: *config_has_omp_support()*
- % **config_has_gzip_support** *[function]* :: *config_has_gzip_support()*
- % **config_has_png_support** *[function]* :: *config_has_png_support()*
- % **config_has_jpeg_support** *[function]* :: *config_has_jpeg_support()*
- % **config_has_ffmpeg_support** *[function]* :: *config_has_ffmpeg_support()*
- % **config_has_exceptions** *[function]* :: *config_has_exceptions()*
- % **config_has_package** *[function]* :: *config_has_package()*
- % **config_package_count** *[function]* :: *config_package_count()*
- % **config_package_name** *[function]* :: *config_package_name()*
- % **installed_packages** *[subroutine]* :: *installed_packages()*
- % **config_accelerator** *[function]* :: *config_accelerator()*
- % **has_gpu_device** *[function]* :: *has_gpu_device()*
- % **get_gpu_device_info** *[subroutine]* :: *get_gpu_device_info()*
- % **has_style** *[function]* :: *has_style()*
- % **style_count** *[function]* :: *style_count()*
- % **style_name** *[function]* :: *style_name()*
- % **has_id** *[function]* :: *has_id()*
- % **id_count** *[function]* :: *id_count()*
- % **id_name** *[subroutine]* :: *id_name()*
- % **plugin_count** *[subroutine]* :: *plugin_count()*
- % **plugin_name** :: *plugin_name()*
- % **encode_image_flags** *[function]* :: *encode_image_flags()*
- % **decode_image_flags** *[subroutine]* :: *decode_image_flags()*
- % **set_fix_external_callback** *[subroutine]* :: *set_fix_external_callback()*
- % **fix_external_get_force** *[function]* :: *fix_external_get_force()*
- % **fix_external_set_energy_global** *[subroutine]* :: *fix_external_set_energy_global()*
- % **fix_external_set_virial_global** *[subroutine]* :: *fix_external_set_virial_global()*
- % **fix_external_set_energy_peratom** *[subroutine]* :: *fix_external_set_energy_peratom()*

- **% fix_external_set_virial_peratom** *[subroutine]* :: fix_external_set_virial_peratom()
- **% fix_external_set_vector_length** *[subroutine]* :: *fix_external_set_vector_length()*
- **% fix_external_set_vector** *[subroutine]* :: *fix_external_set_vector()*
- **% flush_buffers** *[subroutine]* :: *flush_buffers()*
- **% is_running** *[function]* :: *is_running()*
- **% force_timeout** *[subroutine]* :: *force_timeout()*
- **% has_error** *[function]* :: *has_error()*
- **% get_last_error_message** *[subroutine]* :: *get_last_error_message()*

---

**function lammps**(*[args][,comm]*)

This is the constructor for the Fortran class and will forward the arguments to a call to either *lammps_open_fortran()* or *lammps_open_no_mpi()*. If the LAMMPS library has been compiled with MPI support, it will also initialize MPI, if it has not already been initialized before.

The *args* argument with the list of command-line parameters is optional and so it the *comm* argument with the MPI communicator. If *comm* is not provided, MPI_COMM_WORLD is assumed. For more details please see the documentation of *lammps_open()*.

**Options**

- **args** *[character(len=*),dimension(:),optional]* :: arguments as list of strings
- **comm** *[integer,optional]* :: MPI communicator

**Call to**

*lammps_open_fortran() lammps_open_no_mpi()*

**Return**

**lammps** :: an instance of the *lammps* derived type

**Note**

The MPI_F08 module, which defines Fortran 2008 bindings for MPI, is not directly supported by this interface due to the complexities of supporting both the MPI_F08 and MPI modules at the same time. However, you should be able to use the MPI_VAL member of the MPI_comm derived type to access the integer value of the communicator, such as in

```
PROGRAM testmpi
  USE LIBLAMMPS
  USE MPI_F08
  TYPE(lammps) :: lmp
  lmp = lammps(comm=MPI_COMM_SELF%MPI_VAL)
END PROGRAM testmpi
```

**type lammps_style**

This derived type is there to provide a convenient interface for the style constants used with *extract_compute()*, *extract_fix()*, and *extract_variable()*. Assuming your LAMMPS instance is called lmp, these constants will be lmp%style%global, lmp%style%atom, and lmp%style%local. These values are identical to the values described in *_LMP_STYLE_CONST* for the C library interface.

**Type fields**

- **% global** *[integer(c_int)]* :: used to request global data
- **% atom** *[integer(c_int)]* :: used to request per-atom data
- **% local** *[integer(c_int)]* :: used to request local data

**type lammps_type**

This derived type is there to provide a convenient interface for the type constants used with *extract_compute()*, *extract_fix()*, and *extract_variable()*. Assuming your LAMMPS instance is called `lmp`, these constants will be `lmp%type%scalar`, `lmp%type%vector`, and `lmp%type%array`. These values are identical to the values described in *_LMP_TYPE_CONST* for the C library interface.

**Type fields**

- **% scalar** *[integer(c_int)]* :: used to request scalars
- **% vector** *[integer(c_int)]* :: used to request vectors
- **% array** *[integer(c_int)]* :: used to request arrays (matrices)

### Procedures Bound to the *lammps* Derived Type

**subroutine close**([*finalize*])

This method will close down the LAMMPS instance through calling *lammps_close()*. If the *finalize* argument is present and has a value of `.TRUE.`, then this subroutine also calls *lammps_kokkos_finalize()*, *lammps_mpi_finalize()*, *lammps_python_finalize()*, and *lammps_plugin_finalize()*.

**Options**

**finalize** *[logical,optional]* :: shut down the MPI environment of the LAMMPS library if `.TRUE.`.

**Call to**

*lammps_close()* *lammps_mpi_finalize()* *lammps_kokkos_finalize()* *lammps_python_finalize()* *lammps_plugin_finalize()*

---

**subroutine error**(*error_type*, *error_text*)

This method is a wrapper around the *lammps_error()* function and will dispatch an error through the LAMMPS Error class.

Added in version 3Nov2022.

**Parameters**

- **error_type** *[integer(c_int)]* :: constant to select which Error class function to call
- **error_text** *[character(len=*)]* :: error message

**Call to**

*lammps_error()*

---

**subroutine file**(*filename*)

This method will call *lammps_file()* to have LAMMPS read and process commands from a file.

**Parameters**

**filename** *[character(len=*)]* :: name of file with LAMMPS commands

**Call to**

*lammps_file()*

---

**subroutine command**(*cmd*)

This method will call *lammps_command()* to have LAMMPS execute a single command.

**Parameters**
**cmd** *[character(len=*)]* :: single LAMMPS command

**Call to**
*lammps_command()*

---

**subroutine commands_list**(*cmds*)

This method will call *lammps_commands_list()* to have LAMMPS execute a list of input lines.

**Parameters**
**cmd** *[character(len=*),dimension(:)]* :: list of LAMMPS input lines

**Call to**
*lammps_commands_list()*

---

**subroutine commands_string**(*str*)

This method will call *lammps_commands_string()* to have LAMMPS execute a block of commands from a string.

**Parameters**
**str** *[character(len=*)]* :: LAMMPS input in string

**Call to**
*lammps_commands_string()*

---

**function get_natoms**()

This function will call *lammps_get_natoms()* and return the number of atoms in the system.

**Call to**
*lammps_get_natoms()*

**Return**
**natoms** *[real(c_double)]* :: number of atoms

**Note**

If you would prefer to get the number of atoms in its native format (i.e., as a 32- or 64-bit integer, depending on how LAMMPS was compiled), this can be extracted with *extract_global()*.

---

**function get_thermo**(*name*)

This function will call *lammps_get_thermo()* and return the value of the corresponding thermodynamic keyword.

Added in version 3Nov2022.

**Parameters**
**name** *[character(len=*)]* :: string with the name of the thermo keyword

**Call to**
*lammps_get_thermo()*

**Return**
**value** *[real(c_double)]* :: value of the requested thermo property or *0.0_c_double*

---

**function last_thermo**(*what*, *index*)

This function will call *lammps_last_thermo()* and returns either a string or a pointer to a cached copy of LAMMPS last thermodynamic output, depending on the data requested through *what*. Note that *index* uses 1-based indexing to access thermo output columns.

Added in version 15Jun2023.

Note that this function actually does not return a value, but rather associates the pointer on the left side of the assignment to point to internal LAMMPS data (with the exception of string data, which are copied and returned as ordinary Fortran strings). Pointers must be of the correct data type to point to said data (typically `INTEGER(c_int)`, `INTEGER(c_int64_t)`, or `REAL(c_double)`). The pointer being associated with LAMMPS data is type-checked at run-time via an overloaded assignment operator. The pointers returned by this function point to temporary, read-only data that may be overwritten at any time, so their target values need to be copied to local storage if they are supposed to persist.

For example,

```
PROGRAM thermo
  USE LIBLAMMPS
  USE, INTRINSIC :: ISO_C_BINDING, ONLY : c_double, c_int64_t, c_int
  TYPE(lammps) :: lmp
  INTEGER(KIND=c_int64_t), POINTER :: ntimestep, bval
  REAL(KIND=c_double), POINTER :: dval
  INTEGER(KIND=c_int), POINTER :: nfield, typ, ival
  INTEGER(KIND=c_int) :: i
  CHARACTER(LEN=11) :: key

  lmp = lammps()
  CALL lmp%file('in.sysinit')

  ntimestep = lmp%last_thermo('step', 0)
  nfield = lmp%last_thermo('num', 0)
  PRINT*, 'Last thermo output on step: ', ntimestep, '  Number of fields: ', nfield
  DO i=1, nfield
      key = lmp%last_thermo('keyword',i)
      typ = lmp%last_thermo('type',i)
      IF (typ == lmp%dtype%i32) THEN
          ival = lmp%last_thermo('data',i)
          PRINT*, key, ':', ival
      ELSE IF (typ == lmp%dtype%i64) THEN
          bval = lmp%last_thermo('data',i)
          PRINT*, key, ':', bval
      ELSE IF (typ == lmp%dtype%r64) THEN
          dval = lmp%last_thermo('data',i)
          PRINT*, key, ':', dval
      END IF
  END DO
  CALL lmp%close(.TRUE.)
END PROGRAM thermo
```

would extract the last timestep where thermo output was done and the number of columns it printed. Then it loops over the columns to print out column header keywords and the corresponding data.

**Note**

If *last_thermo()* returns a string, the string must have a length greater than or equal to the length of the string (not including the terminal NULL character) that LAMMPS returns. If the variable's length is too short, the string will be truncated. As usual in Fortran, strings are padded with spaces at the end. If you use an allocatable string, the string **must be allocated** prior to calling this function.

**Parameters**

- **what** *[character(len=*)]* :: string with the name of the thermo keyword
- **index** *[integer(c_int)]* :: 1-based column index

**Call to**

*lammps_last_thermo()*

**Return**

**pointer** *[polymorphic]* :: pointer to LAMMPS data. The left-hand side of the assignment should be either a string (if expecting string data) or a C-compatible pointer (e.g., `INTEGER(c_int), POINTER :: nlocal`) to the extracted property.

**Warning**

Modifying the data in the location pointed to by the returned pointer may lead to inconsistent internal data and thus may cause failures, crashes, or bogus simulations. In general, it is much better to use a LAMMPS input command that sets or changes these parameters. Using an input command will take care of all side effects and necessary updates of settings derived from such settings.

---

**subroutine extract_box**(*[boxlo][, boxhi][, xy][, yz][, xz][, pflags][, boxflag]*)

This subroutine will call *lammps_extract_box()*. All parameters are optional, though obviously at least one should be present. The parameters *pflags* and *boxflag* are stored in LAMMPS as integers, but should be declared as LOGICAL variables when calling from Fortran.

Added in version 3Nov2022.

**Options**

- **boxlo** *[real(c_double),dimension(3),optional]* :: vector in which to store lower-bounds of simulation box
- **boxhi** *[real(c_double),dimension(3),optional]* :: vector in which to store upper-bounds of simulation box
- **xy** *[real(c_double),optional]* :: variable in which to store *xy* tilt factor
- **yz** *[real(c_double),optional]* :: variable in which to store *yz* tilt factor
- **xz** *[real(c_double),optional]* :: variable in which to store *xz* tilt factor
- **pflags** *[logical,dimension(3),optional]* :: vector in which to store periodicity flags (`.TRUE.` means periodic in that dimension)

- **boxflag** *[logical,optional]* :: variable in which to store boolean denoting whether the box will change during a simulation (`.TRUE.` means box will change)

**Call to**

*lammps_extract_box()*

**Note**

Note that a frequent use case of this function is to extract only one or more of the options rather than all seven. For example, assuming "lmp" represents a properly-initialized LAMMPS instance, the following code will extract the periodic box settings into the variable "periodic":

```
! code to start up
LOGICAL :: periodic(3)
! code to initialize LAMMPS / run things / etc.
CALL lmp%extract_box(pflags = periodic)
```

---

**subroutine reset_box**(*boxlo*, *boxhi*, *xy*, *yz*, *xz*)

This subroutine will call *lammps_reset_box()*. All parameters are required.

Added in version 3Nov2022.

**Parameters**

- **boxlo** *[real(c_double),dimension(3)]* :: vector of three doubles containing the lower box boundary
- **boxhi** *[real(c_double),dimension(3)]* :: vector of three doubles containing the upper box boundary
- **xy** *[real(c_double)]* :: *x*–*y* tilt factor
- **yz** *[real(c_double)]* :: *y*–*z* tilt factor
- **xz** *[real(c_double)]* :: *x*–*z* tilt factor

**Call to**

*lammps_reset_box()*

---

**subroutine memory_usage**(*meminfo*)

This subroutine will call *lammps_memory_usage()* and store the result in the three-element array *meminfo*.

Added in version 3Nov2022.

**Parameters**

**meminfo** *[real(c_double),dimension(3)]* :: vector of three doubles in which to store memory usage data

**Call to**

*lammps_memory_usage()*

---

**function get_mpi_comm()**

This function returns a Fortran representation of the LAMMPS "world" communicator.

Added in version 3Nov2022.

**Call to**

*lammps_get_mpi_comm()*

**Return**

**comm** *[integer]* :: Fortran integer equivalent to the MPI communicator LAMMPS is using

**Note**

The C library interface currently returns type `int` instead of type `MPI_Fint`, which is the C type corresponding to Fortran `INTEGER` types of the default kind. On most compilers, these are the same anyway, but this interface exchanges values this way to avoid warning messages.

**Note**

The `MPI_F08` module, which defines Fortran 2008 bindings for MPI, is not directly supported by this function. However, you should be able to convert between the two using the *MPI_VAL* member of the communicator. For example,

```
USE MPI_F08
USE LIBLAMMPS
TYPE(lammps) :: lmp
TYPE(MPI_Comm) :: comm
! ... [commands to set up LAMMPS/etc.]
comm%MPI_VAL = lmp%get_mpi_comm()
```

should assign an `MPI_F08` communicator properly.

---

**function extract_setting(***keyword***)**

Query LAMMPS about global settings. See the documentation for the *lammps_extract_setting()* function from the C library.

Added in version 3Nov2022.

**Parameters**

**keyword** *[character(len=*)]* :: string containing the name of the thermo keyword

**Call to**

*lammps_extract_setting()*

**Return**

**setting** *[integer(c_int)]* :: value of the queried setting or −1 if unknown

---

**function extract_global(***name***)**

This function calls *lammps_extract_global()* and returns either a string or a pointer to internal global LAMMPS data, depending on the data requested through *name*.

Added in version 3Nov2022.

Note that this function actually does not return a value, but rather associates the pointer on the left side of the assignment to point to internal LAMMPS data (with the exception of string data, which are copied and returned as ordinary Fortran strings). Pointers must be of the correct data type to point to said data (typically `INTEGER(c_int)`, `INTEGER(c_int64_t)`, or `REAL(c_double)`) and have compatible kind and rank. The pointer being associated with LAMMPS data is type-, kind-, and rank-checked at run-time via an overloaded assignment operator. The pointers returned by this function are generally persistent; therefore it is not necessary to call the function again unless a *clear command* command has been issued, which wipes out and recreates the contents of the *LAMMPS* class.

For example,

```
PROGRAM demo
  USE, INTRINSIC :: ISO_C_BINDING, ONLY : c_int64_t, c_int, c_double
  USE LIBLAMMPS
  TYPE(lammps) :: lmp
  INTEGER(c_int), POINTER :: nlocal => NULL()
  INTEGER(c_int64_t), POINTER :: ntimestep => NULL()
  REAL(c_double), POINTER :: dt => NULL()
  CHARACTER(LEN=10) :: units
  lmp = lammps()
  ! other commands
  nlocal = lmp%extract_global('nlocal')
  ntimestep = lmp%extract_global('ntimestep')
  dt = lmp%extract_global('dt')
  units = lmp%extract_global('units')
  ! more commands
  lmp.close(.TRUE.)
END PROGRAM demo
```

would extract the number of atoms on this processor, the current time step, the size of the current time step, and the units being used into the variables *nlocal*, *ntimestep*, *dt*, and *units*, respectively.

**Note**

If *extract_global()* returns a string, the string must have a length greater than or equal to the length of the string (not including the terminal `NULL` character) that LAMMPS returns. If the variable's length is too short, the string will be truncated. As usual in Fortran, strings are padded with spaces at the end. If you use an allocatable string, the string **must be allocated** prior to calling this function, but you can automatically reallocate it to the correct length after the function returns, viz.,

```
PROGRAM test
  USE LIBLAMMPS
  TYPE(lammps) :: lmp
  CHARACTER(LEN=:), ALLOCATABLE :: str
  lmp = lammps()
  CALL lmp%command('units metal')
  ALLOCATE(CHARACTER(LEN=80) :: str)
  str = lmp%extract_global('units')
  str = TRIM(str) ! re-allocates to length len_trim(str) here
  PRINT*, LEN(str), LEN_TRIM(str)
END PROGRAM test
```

will print the number 5 (the length of the word "metal") twice.

**Parameters**
**name** *[character(len=*)]* :: string with the name of the property to extract

**Call to**
*lammps_extract_global()*

**Return**
**pointer** *[polymorphic]* :: pointer to LAMMPS data. The left-hand side of the assignment should be either a string (if expecting string data) or a C-compatible pointer (e.g., `INTEGER(c_int), POINTER :: nlocal`) to the extracted property. If expecting vector data, the pointer should have dimension ":".

**Warning**

Modifying the data in the location pointed to by the returned pointer may lead to inconsistent internal data and thus may cause failures, crashes, or bogus simulations. In general, it is much better to use a LAMMPS input command that sets or changes these parameters. Using an input command will take care of all side effects and necessary updates of settings derived from such settings.

---

`function extract_atom(`*name*`)`

This function calls *lammps_extract_atom()* and returns a pointer to LAMMPS data tied to the `Atom` class, depending on the data requested through *name*.

Added in version 3Nov2022.

Note that this function actually does not return a pointer, but rather associates the pointer on the left side of the assignment to point to internal LAMMPS data. Pointers must be of the correct type, kind, and rank (e.g., `INTEGER(c_int), DIMENSION(:)` for "type", "mask", or "id"; `INTEGER(c_int64_t), DIMENSION(:)` for "id" if LAMMPS was compiled with the `-DLAMMPS_BIGBIG` flag; `REAL(c_double), DIMENSION(:,:)` for "x", "v", or "f"; and so forth). The pointer being associated with LAMMPS data is type-, kind-, and rank-checked at run-time.

**Parameters**
**name** *[character(len=*)]* :: string with the name of the property to extract

**Call to**
*lammps_extract_atom()*

**Return**
**pointer** *[polymorphic]* :: pointer to LAMMPS data. The left-hand side of the assignment should be a C-interoperable pointer of appropriate kind and rank (e.g., `INTEGER(c_int), POINTER :: mask(:)`) to the extracted property. If expecting vector data, the pointer should have dimension ":"; if expecting matrix data, the pointer should have dimension ":,:".

**Warning**

Pointers returned by this function are generally not persistent, as per-atom data may be redistributed, reallocated, and reordered at every re-neighboring operation. It is advisable to re-bind pointers using *extract_atom()* between runs.

**Array index order**

Two-dimensional arrays returned from *extract_atom()* will be **transposed** from equivalent arrays in C, and they will be indexed from 1 instead of 0. For example, in C,

```
void *lmp;
double **x;
/* more code to setup, etc. */
x = lammps_extract_atom(lmp, "x");
printf("%f\n", x[5][1]);
```

will print the *y*-coordinate of the sixth atom on this processor. Conversely,

```
TYPE(lammps) :: lmp
REAL(c_double), DIMENSION(:,:), POINTER :: x => NULL()
! more code to setup, etc.
x = lmp%extract_atom("x")
PRINT '(f0.6)', x(2,6)
```

will print the *y*-coordinate of the sixth atom on this processor (note the transposition of the two indices). This is not a choice, but rather a consequence of the different conventions adopted by the Fortran and C standards decades ago: in C, the block of data

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
```

interpreted as a $4 \times 4$ matrix would be

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix},$$

that is, in row-major order. In Fortran, the same block of data is interpreted in column-major order, namely,

$$\begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix}.$$

This difference in interpretation of the same block of data by the two languages means, in effect, that matrices from C or C++ will be transposed when interpreted in Fortran.

**Note**

If you would like the indices to start at 0 instead of 1 (which follows typical notation in C and C++, but not Fortran), you can create another pointer and associate it thus:

```
REAL(c_double), DIMENSION(:,:), POINTER :: x, x0
x = lmp%extract_atom("x")
x0(0:,0:) => x
```

The above would cause the dimensions of *x* to be (1:3, 1:nmax) and those of *x0* to be (0:2, 0:nmax−1).

**function extract_compute**(*id*, *style*, *type*)

This function calls *lammps_extract_compute()* and returns a pointer to LAMMPS data tied to the Compute class, specifically data provided by the compute identified by *id*. Computes may provide global, per-atom, or local data, and those data may be a scalar, a vector, or an array. Since computes may provide multiple kinds of data, the user is required to specify which set of data is to be returned through the *style* and *type* variables.

Added in version 3Nov2022.

Note that this function actually does not return a value, but rather associates the pointer on the left side of the assignment to point to internal LAMMPS data. Pointers must be of the correct data type to point to said data (i.e., REAL(c_double)) and have compatible rank. The pointer being associated with LAMMPS data is type-, kind-, and rank-checked at run-time via an overloaded assignment operator.

For example,

```
TYPE(lammps) :: lmp
REAL(c_double), DIMENSION(:), POINTER :: COM
! code to setup, create atoms, etc.
CALL lmp%command('compute COM all com')
COM = lmp%extract_compute('COM', lmp%style%global, lmp%style%type)
```

will bind the variable *COM* to the center of mass of the atoms created in your simulation. The vector in this case has length 3; the length (or, in the case of array data, the number of rows and columns) is determined for you based on data from the Compute class.

**Array index order**

Two-dimensional arrays returned from *extract_compute()* will be **transposed** from equivalent arrays in C, and they will be indexed from 1 instead of 0. See the note at *extract_atom()* for further details.

The following combinations are possible (assuming lmp is the name of your LAMMPS instance):

| Style | Type | Type to assign to | Returned data |
|---|---|---|---|
| lmp%style%global | lmp%type%scalar | REAL(c_double), POINTER | Global scalar |
| lmp%style%global | lmp%type%vector | REAL(c_double), DIMENSION(:), POINTER | Global vector |
| lmp%style%global | lmp%type%array | REAL(c_double), DIMENSION(:,:), POINTER | Global array |
| lmp%style%atom | lmp%type%vector | REAL(c_double), DIMENSION(:), POINTER | Per-atom vector |
| lmp%style%atom | lmp%type%array | REAL(c_double), DIMENSION(:,:), POINTER | Per-atom array |
| lmp%style%local | lmp%type%vector | REAL(c_double), DIMENSION(:), POINTER | Local vector |
| lmp%style%local | lmp%type%array | REAL(c_double), DIMENSION(:,:), POINTER | Local array |

**Parameters**

- **id** *[character(len=*)]* :: compute ID from which to extract data
- **style** *[integer(c_int)]* :: value indicating the style of data to extract (global, per-atom, or local)
- **type** *[integer(c_int)]* :: value indicating the type of data to extract (scalar, vector, or array)

**Call to**
*lammps_extract_compute()*

**Return**
**pointer** *[polymorphic]* :: pointer to LAMMPS data. The left-hand side of the assignment should be a C-compatible pointer (e.g., `REAL(c_double), POINTER :: x`) to the extracted property. If expecting vector data, the pointer should have dimension ":"; if expecting array (matrix) data, the pointer should have dimension ":,:".

**Note**

If the compute's data are not already computed for the current step, the compute will be invoked. LAMMPS cannot easily check at that time if it is valid to invoke a compute, so it may fail with an error. The caller has to check to avoid such an error.

**Warning**

The pointers returned by this function are generally not persistent, since the computed data may be re-distributed, re-allocated, and re-ordered at every invocation. It is advisable to re-invoke this function before the data are accessed or make a copy if the data are to be used after other LAMMPS commands have been issued. Do **not** modify the data returned by this function.

---

`function extract_fix`(*id, style, type[, nrow][, ncol]*)

This function calls *lammps_extract_fix()* and returns a pointer to LAMMPS data tied to the `Fix` class, specifically data provided by the fix identified by *id*. Fixes may provide global, per-atom, or local data, and those data may be a scalar, a vector, or an array. Since many fixes provide multiple kinds of data, the user is required to specify which set of data is to be returned through the *style* and *type* variables.

Added in version 3Nov2022.

Global data are calculated at the time they are requested and are only available element-by-element. As such, the user is expected to provide the *nrow* variable to specify which element of a global vector or the *nrow* and *ncol* variables to specify which element of a global array the user wishes LAMMPS to return. The *ncol* variable is optional for global scalar or vector data, and both *nrow* and *ncol* are optional when a global scalar is requested, as well as when per-atom or local data are requested. The following combinations are possible (assuming `lmp` is the name of your LAMMPS instance):

| Style | Type | nrow | ncol | Type to assign to | Returned data |
|---|---|---|---|---|---|
| lmp%style%global | lmp%type%scalar | Ignored | Ignored | REAL(c_double) | Global scalar |
| lmp%style%global | lmp%type%vector | Required | Ignored | REAL(c_double) | Element of global vector |
| lmp%style%global | lmp%type%array | Required | Required | REAL(c_double) | Element of global array |
| lmp%style%atom | lmp%type%scalar | | | | (not allowed) |
| lmp%style%atom | lmp%type%vector | Ignored | Ignored | REAL(c_double), DIMENSION(:), POINTER | Per-atom vector |
| lmp%style%atom | lmp%type%array | Ignored | Ignored | REAL(c_double), DIMENSION(:,:), POINTER | Per-atom array |
| lmp%style%local | lmp%type%scalar | | | | (not allowed) |
| lmp%style%local | lmp%type%vector | Ignored | Ignored | REAL(c_double), DIMENSION(:), POINTER | Per-atom vector |
| lmp%style%local | lmp%type%array | Ignored | Ignored | REAL(c_double), DIMENSION(:,:), POINTER | Per-atom array |

In the case of global data, this function returns a value of type `REAL(c_double)`. For per-atom or local data, this function does not return a value but instead associates the pointer on the left side of the assignment to point to internal LAMMPS data. Pointers must be of the correct type and kind to point to said data (i.e., `REAL(c_double)`) and have compatible rank. The pointer being associated with LAMMPS data is type-, kind-, and rank-checked at run-time via an overloaded assignment operator.

For example,

```
TYPE(lammps) :: lmp
REAL(c_double) :: dr, dx, dy, dz
! more code to set up, etc.
lmp%command('fix george all recenter 2 2 2')
! more code
dr = lmp%extract_fix("george", lmp%style%global, lmp%style%scalar)
dx = lmp%extract_fix("george", lmp%style%global, lmp%style%vector, 1)
dy = lmp%extract_fix("george", lmp%style%global, lmp%style%vector, 2)
dz = lmp%extract_fix("george", lmp%style%global, lmp%style%vector, 3)
```

will extract the global scalar calculated by *fix recenter* into the variable *dr* and the three elements of the global vector calculated by fix recenter into the variables *dx*, *dy*, and *dz*, respectively.

If asked for per-atom or local data, *extract_fix()* returns a pointer to actual LAMMPS data. The pointer returned will have the appropriate size to match the internal data, and will be type/kind/rank-checked at the time of the assignment. For example,

```
TYPE(lammps) :: lmp
REAL(c_double), DIMENSION(:), POINTER :: r
! more code to set up, etc.
lmp%command('fix state all store/state 0 x y z')
! more code
r = lmp%extract_fix('state', lmp%style%atom, lmp%type%array)
```

will bind the pointer *r* to internal LAMMPS data representing the per-atom array computed by *fix store/state* when three inputs are specified. Similarly,

```
TYPE(lammps) :: lmp
REAL(c_double), DIMENSION(:), POINTER :: x
! more code to set up, etc.
lmp%command('fix state all store/state 0 x')
! more code
x = lmp%extract_fix('state', lmp%style%atom, lmp%type%vector)
```

will associate the pointer *x* with internal LAMMPS data corresponding to the per-atom vector computed by *fix store/state* when only one input is specified. Similar examples with `lmp%style%atom` replaced by `lmp%style%local` will extract local data from fixes that define local vectors and/or arrays.

**Warning**

The pointers returned by this function for per-atom or local data are generally not persistent, since the computed data may be redistributed, reallocated, and reordered at every invocation of the fix. It is thus advisable to re-invoke this function before the data are accessed or to make a copy if the data are to be used after other LAMMPS commands have been issued.

**Note**

LAMMPS cannot easily check if it is valid to access the data, so it may fail with an error. The caller has to avoid such an error.

**Parameters**

- **id** *[character(len=*)]* :: string with the name of the fix from which to extract data
- **style** *[integer(c_int)]* :: value indicating the style of data to extract (global, per-atom, or local)
- **type** *[integer(c_int)]* :: value indicating the type of data to extract (scalar, vector, or array)
- **nrow** *[integer(c_int)]* :: row index (used only for global vectors and arrays)
- **ncol** *[integer(c_int)]* :: column index (only used for global arrays)

**Call to**

*lammps_extract_fix()*

**Return**

**data** *[polymorphic]* :: LAMMPS data (for global data) or a pointer to LAMMPS data (for per-atom or local data). The left-hand side of the assignment should be of type `REAL(c_double)` and have appropriate rank (i.e., `DIMENSION(:)` if expecting per-atom or local vector data and `DIMENSION(:,:)` if expecting per-atom or local array data). If expecting local or per-atom data, it should have the `POINTER` attribute, but if expecting global data, it should be an ordinary (non-`POINTER`) variable.

**Array index order**

Two-dimensional global, per-atom, or local array data from *extract_fix()* will be **transposed** from equivalent arrays in C (or in the ordinary LAMMPS interface accessed through thermodynamic output), and they

will be indexed from 1, not 0. This is true even for global data, which are returned as scalars—this is done primarily so the interface is consistent, as there is no choice but to transpose the indices for per-atom or local array data. See the similar note under *extract_atom()* for further details.

---

**function extract_variable**(*name*[, *group* ])

This function calls *lammps_extract_variable()* and returns a scalar, vector, or string containing the value of the variable identified by *name*. When the variable is an *equal*-style variable (or one compatible with that style such as *internal*), the variable is evaluated and the corresponding value returned. When the variable is an *atom*-style variable, the variable is evaluated and a vector of values is returned. With all other variables, a string is returned. The *group* argument is only used for *atom* style variables and is ignored otherwise. If *group* is absent for *atom*-style variables, the group is assumed to be "all".

Added in version 3Nov2022.

This function returns the values of the variables, not pointers to them. Vectors pointing to *atom*-style variables should be of type `REAL(c_double)`, be of rank 1 (i.e., `DIMENSION(:)`), and have the `ALLOCATABLE` attribute.

**Note**

Unlike the C library interface, the Fortran interface does not require you to deallocate memory when you are through; this is done for you, behind the scenes.

For example,

```
TYPE(lammps) :: lmp
REAL(c_double) :: area
! more code to set up, etc.
lmp%command('variable A equal lx*ly')
! more code
area = lmp%extract_variable("A")
```

will extract the *x*–*y* cross-sectional area of the simulation into the variable *area*.

**Parameters**

**name** *[character(len=*)]* :: variable name to evaluate

**Options**

**group** *[character(len=*),optional]* :: group for which to extract per-atom data (if absent, use "all")

**Call to**

*lammps_extract_variable()*

**Return**

**data** *[polymorphic]* :: scalar of type `REAL(c_double)` (for *equal*-style variables and others that are *equal*-compatible), vector of type `REAL(c_double), DIMENSION(:), ALLOCATABLE` for *atom*- or *vector*-style variables, or `CHARACTER(LEN=*)` for *string*-style and compatible variables. Strings whose length is too short to hold the result will be truncated. Allocatable strings must be allocated before this function is called; see note at *extract_global()* regarding allocatable strings. Allocatable arrays (for *atom*- and *vector*-style data) will be reallocated on assignment.

**Note**

LAMMPS cannot easily check if it is valid to access the data referenced by the variables (e.g., computes, fixes, or thermodynamic info), so it may fail with an error. The caller has to make certain that the data are extracted only when it is safe to evaluate the variable and thus an error and crash are avoided.

---

**subroutine set_variable**(*name*, *str*)

Set the value of a string-style variable.

Deprecated since version 7Feb2024.

This function assigns a new value from the string *str* to the string-style variable *name*. If *name* does not exist or is not a string-style variable, an error is generated.

**Warning**

This subroutine is deprecated and *set_string_variable()* should be used instead.

**Parameters**

- **name** *[character(len=*)]* :: name of the variable
- **str** *[character(len=*)]* :: new value to assign to the variable

**Call to**

*lammps_set_variable()*

---

**subroutine set_string_variable**(*name*, *str*)

Set the value of a string-style variable.

Added in version 7Feb2024.

This function assigns a new value from the string *str* to the string-style variable *name*. If *name* does not exist or is not a string-style variable, an error is generated.

**Parameters**

- **name** *[character(len=*)]* :: name of the variable
- **str** *[character(len=*)]* :: new value to assign to the variable

**Call to**

*lammps_set_string_variable()*

---

**subroutine set_internal_variable**(*name*, *val*)

Set the value of a internal-style variable.

Added in version 7Feb2024.

This function assigns a new value from the floating-point number *val* to the internal-style variable *name*. If *name* does not exist or is not an internal-style variable, an error is generated.

**Parameters**

- **name** *[character(len=*)]* :: name of the variable
- **val** *[real(c_double)]* :: new value to assign to the variable

**Call to**
*lammps_set_internal_variable()*

---

`function eval(expr)`

This function is a wrapper around *lammps_eval()* that takes a LAMMPS equal style variable string, evaluates it and returns the resulting scalar value as a floating-point number.

Added in version 4Feb2025.

**Parameters**
**expr** *[character(len=*)]* :: string to be evaluated

**Call to**
*lammps_eval()*

**Return**
**value** *[real(c_double)]* :: result of the evaluated string

---

`subroutine clearstep_compute()`

Clear whether a compute has been invoked

Added in version 4Feb2025.

**Call to**
*lammps_clearstep_compute()*

---

`subroutine addstep_compute(nextstep)`

Add timestep to list of future compute invocations if the compute has been invoked on the current timestep

Added in version 4Feb2025.

overloaded for 32-bit and 64-bit integer arguments

**Parameters**
**nextstep** *[integer(kind=8 or kind=4)]* :: next timestep

**Call to**
*lammps_addstep_compute()*

---

`subroutine addstep_compute_all(nextstep)`

Add timestep to list of future compute invocations

Added in version 4Feb2025.

overloaded for 32-bit and 64-bit integer arguments

**Parameters**
**nextstep** *[integer(kind=8 or kind=4)]* :: next timestep

**Call to**
*lammps_addstep_compute_all()*

---

**subroutine gather_atoms**(*name*, *count*, *data*)

This function calls *lammps_gather_atoms()* to gather the named atom-based entity for all atoms on all processors and return it in the vector *data*. The vector *data* will be ordered by atom ID, which requires consecutive atom IDs (1 to *natoms*).

Added in version 3Nov2022.

If you need a similar array but have non-consecutive atom IDs, see *gather_atoms_concat()*; for a similar array but for a subset of atoms, see *gather_atoms_subset()*.

The *data* array will be ordered in groups of *count* values, sorted by atom ID (e.g., if *name* is *x* and *count* = 3, then *data* = [*x*(1,1), *x*(2,1), *x*(3,1), *x*(1,2), *x*(2,2), *x*(3,2), *x*(1,3), ...]); *data* must be ALLOCATABLE and will be allocated to length (*count* × *natoms*), as queried by *get_natoms()*.

This function is not compatible with -DLAMMPS_BIGBIG.

**Parameters**

- **name** *[character(len=*)]* :: desired quantity (e.g., *x* or *mask*)
- **count** *[integer(c_int)]* :: number of per-atom values you expect per atom (e.g., 1 for *type*, *mask*, or *charge*; 3 for *x*, *v*, or *f*). Use *count* = 3 with *image* if you want a single image flag unpacked into *x*/*y*/*z* components.
- **data** *[polymorphic,dimension(:),allocatable]* :: array into which to store the data. Array *must* have the ALLOCATABLE attribute and be of rank 1 (i.e., DIMENSION(:)). If this array is already allocated, it will be reallocated to fit the length of the incoming data. It should have type INTEGER(c_int) if expecting integer data and REAL(c_double) if expecting floating-point data.

**Call to**

*lammps_gather_atoms()*

**Note**

If you want data from this function to be accessible as a two-dimensional array, you can declare a rank-2 pointer and reassign it, like so:

```
USE, INTRINSIC :: ISO_C_BINDING
USE LIBLAMMPS
TYPE(lammps) :: lmp
REAL(c_double), DIMENSION(:), ALLOCATABLE, TARGET :: xdata
REAL(c_double), DIMENSION(:,:), POINTER :: x
! other code to set up, etc.
CALL lmp%gather_atoms('x',3,xdata)
x(1:3,1:size(xdata)/3) => xdata
```

You can then access the *y*-component of atom 3 with x(2,3). See the note about array index order at *extract_atom()*.

---

**subroutine gather_atoms_concat**(*name*, *count*, *data*)

This function calls *lammps_gather_atoms_concat()* to gather the named atom-based entity for all atoms on all processors and return it in the vector *data*.

Added in version 3Nov2022.

The vector *data* will not be ordered by atom ID, and there is no restriction on the IDs being consecutive. If you need the IDs, you can do another *gather_atoms_concat()* with *name* set to id.

If you need a similar array but have consecutive atom IDs, see *gather_atoms()*; for a similar array but for a subset of atoms, see *gather_atoms_subset()*.

This function is not compatible with -DLAMMPS_BIGBIG.

**Parameters**

- **name** *[character(len=*)]* :: desired quantity (e.g., *x* or *mask*)
- **count** *[integer(c_int)]* :: number of per-atom values you expect per atom (e.g., 1 for *type*, *mask*, or *charge*; 3 for *x*, *v*, or *f*). Use *count* = 3 with *image* if you want a single image flag unpacked into *x*/*y*/*z* components.
- **data** *[polymorphic,dimension(:),allocatable]* :: array into which to store the data. Array *must* have the ALLOCATABLE attribute and be of rank 1 (i.e., DIMENSION(:)). If this array is already allocated, it will be reallocated to fit the length of the incoming data. It should have type INTEGER(c_int) if expecting integer data and REAL(c_double) if expecting floating-point data.

**Call to**

*lammps_gather_atoms_concat()*

---

**subroutine gather_atoms_subset**(*name*, *count*, *ids*, *data*)

This function calls *lammps_gather_atoms_subset()* to gather the named atom-based entity for the atoms in the array *ids* from all processors and return it in the vector *data*.

Added in version 3Nov2022.

This subroutine gathers data for the requested atom IDs and stores them in a one-dimensional allocatable array. The data will be ordered by atom ID, but there is no requirement that the IDs be consecutive. If you wish to return a similar array for *all* the atoms, use *gather_atoms()* or *gather_atoms_concat()*.

The *data* array will be in groups of *count* values, sorted by atom ID in the same order as the array *ids* (e.g., if *name* is *x*, *count* = 3, and *ids* is [100, 57, 210], then *data* might look like [*x*(1,100), *x*(2,100), *x*(3,100), *x*(1,57), *x*(2,57), *x*(3,57), *x*(1,210), . . .]; *ids* must be provided by the user, and *data* must be of rank 1 (i.e., DIMENSION(:)) and have the ALLOCATABLE attribute.

This function is not compatible with -DLAMMPS_BIGBIG.

**Parameters**

- **name** *[character(len=*)]* :: desired quantity (e.g., *x* or *mask*)
- **count** *[integer(c_int)]* :: number of per-atom values you expect per atom (e.g., 1 for *type*, *mask*, or *charge*; 3 for *x*, *v*, or *f*). Use *count* = 3 with *image* if you want a single image flag unpacked into *x*/*y*/*z* components.
- **ids** *[integer(c_int),dimension(:)]* :: atom IDs corresponding to the atoms to be gathered
- **data** *[polymorphic,dimension(:),allocatable]* :: array into which to store the data. Array *must* have the ALLOCATABLE attribute and be of rank 1 (i.e., DIMENSION(:)). If this array is already allocated, it will be reallocated to fit the length of the incoming data. It should have type INTEGER(c_int) if expecting integer data and REAL(c_double) if expecting floating-point data.

**Call to**

*lammps_gather_atoms_subset()*

---

`subroutine scatter_atoms`(*name*, *data*)

This function calls *lammps_scatter_atoms()* to scatter the named atom-based entities in *data* to all processors.

Added in version 3Nov2022.

This subroutine takes data stored in a one-dimensional array supplied by the user and scatters them to all atoms on all processors. The data must be ordered by atom ID, with the requirement that the IDs be consecutive. Use *scatter_atoms_subset()* to scatter data for some (or all) atoms, in any order.

The *data* array needs to be ordered in groups of *count* values, sorted by atom ID (e.g., if *name* is *x* and *count* = 3, then *data* = [*x*(1,1), *x*(2,1), *x*(3,1), *x*(1,2), *x*(2,2), *x*(3,2), *x*(1,3), . . .]); *data* must be of length *natoms* or 3**natoms*.

**Parameters**

- **name** *[character(len=*)]* :: quantity to be scattered (e.g., *x* or *charge*)
- **data** *[polymorphic,dimension(:)]* :: per-atom values packed in a one-dimensional array containing the data to be scattered. This array must have length *natoms* (e.g., for *type* or *charge*) or length *natoms* × 3 (e.g., for *x* or *f*). The array *data* must be rank 1 (i.e., `DIMENSION(:)`) and be of type `INTEGER(c_int)` (e.g., for *mask* or *type*) or of type `REAL(c_double)` (e.g., for *x* or *charge* or *f*).

**Call to**

*lammps_scatter_atoms()*

---

`subroutine scatter_atoms_subset`(*name*, *ids*, *data*)

This function calls *lammps_scatter_atoms_subset()* to scatter the named atom-based entities in *data* to all processors.

Added in version 3Nov2022.

This subroutine takes data stored in a one-dimensional array supplied by the user and scatters them to a subset of atoms on all processors. The array *data* contains data associated with atom IDs, but there is no requirement that the IDs be consecutive, as they are provided in a separate array, *ids*. Use *scatter_atoms()* to scatter data for all atoms, in order.

The *data* array needs to be organized in groups of 1 or 3 values, depending on which quantity is being scattered, with the groups in the same order as the array *ids*. For example, if you want *data* to be the array [*x*(1,1), *x*(2,1), *x*(3,1), *x*(1,100), *x*(2,100), *x*(3,100), *x*(1,57), *x*(2,57), *x*(3,57)], then *ids* would be [1, 100, 57] and *name* would be *x*.

**Parameters**

- **name** *[character(len=*)]* :: quantity to be scattered (e.g., *x* or *charge*)
- **ids** *[integer(c_int),dimension(:)]* :: atom IDs corresponding to the atoms being scattered
- **data** *[polymorphic,dimension(:)]* :: per-atom values packed into a one-dimensional array containing the data to be scattered. This array must have either the same length as *ids* (for *mask*, *type*, etc.) or three times its length (for *x*, *f*, etc.); the array must be rank 1 and be of type `INTEGER(c_int)` (e.g., for *mask* or *type*) or of type `REAL(c_double)` (e.g., for *charge*, *x*, or *f*).

**Call to**

*lammps_scatter_atoms_subset()*

---

**subroutine gather_bonds**(*data*)

Gather type and constituent atom information for all bonds.

Added in version 3Nov2022.

This function copies the list of all bonds into an allocatable array. The array will be filled with (bond type, bond atom 1, bond atom 2) for each bond. The array is allocated to the right length (i.e., three times the number of bonds). The array *data* must be of the same type as the LAMMPS `tagint` type, which is equivalent to either `INTEGER(c_int)` or `INTEGER(c_int64_t)`, depending on whether `-DLAMMPS_BIGBIG` was used when LAMMPS was built. If the supplied array does not match, an error will result at run-time.

An example of how to use this routine is below:

```
PROGRAM bonds
  USE, INTRINSIC :: ISO_C_BINDING, ONLY : c_int
  USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY : OUTPUT_UNIT
  USE LIBLAMMPS
  IMPLICIT NONE
  INTEGER(c_int), DIMENSION(:), ALLOCATABLE, TARGET :: bonds
  INTEGER(c_int), DIMENSION(:,:), POINTER :: bonds_array
  TYPE(lammps) :: lmp
  INTEGER :: i
  ! other commands to initialize LAMMPS, create bonds, etc.
  CALL lmp%gather_bonds(bonds)
  bonds_array(1:3, 1:SIZE(bonds)/3) => bonds
  DO i = 1, SIZE(bonds)/3
    WRITE(OUTPUT_UNIT,'(A,1X,I4,A,I4,1X,I4)') 'bond', bonds_array(1,i), &
      '; type = ', bonds_array(2,i), bonds_array(3,i)
  END DO
END PROGRAM bonds
```

**Parameters**

**data** *[integer(kind=*),allocatable]* :: array into which to copy the result. *The `KIND` parameter is either `c_int` or, if LAMMPS was compiled with `-DLAMMPS_BIGBIG`, kind `c_int64_t`.

**Call to**

*lammps_gather_bonds()*

---

**subroutine gather_angles**(*data*)

Gather type and constituent atom information for all angles.

Added in version 8Feb2023.

This function copies the list of all angles into an allocatable array. The array will be filled with (angle type, angle atom 1, angle atom 2, angle atom 3) for each angle. The array is allocated to the right length (i.e., four times the number of angles). The array *data* must be of the same type as the LAMMPS `tagint` type, which is equivalent to either `INTEGER(c_int)` or `INTEGER(c_int64_t)`, depending on whether `-DLAMMPS_BIGBIG` was used when LAMMPS was built. If the supplied array does not match, an error will result at run-time.

An example of how to use this routine is below:

```
PROGRAM angles
  USE, INTRINSIC :: ISO_C_BINDING, ONLY : c_int
  USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY : OUTPUT_UNIT
```

(continues on next page)

(continued from previous page)

```
  USE LIBLAMMPS
  IMPLICIT NONE
  INTEGER(c_int), DIMENSION(:), ALLOCATABLE, TARGET :: angles
  INTEGER(c_int), DIMENSION(:,:), POINTER :: angles_array
  TYPE(lammps) :: lmp
  INTEGER :: i
  ! other commands to initialize LAMMPS, create angles, etc.
  CALL lmp%gather_angles(angles)
  angles_array(1:4, 1:SIZE(angles)/4) => angles
  DO i = 1, SIZE(angles)/4
    WRITE(OUTPUT_UNIT,'(A,1X,I4,A,I4,1X,I4,1X,I4)') 'angle', angles_array(1,i), &
      '; type = ', angles_array(2,i), angles_array(3,i), angles_array(4,i)
  END DO
END PROGRAM angles
```

**Parameters**

**data** *[integer(kind=*),allocatable]* :: array into which to copy the result. *The `KIND` parameter is either `c_int` or, if LAMMPS was compiled with `-DLAMMPS_BIGBIG`, kind `c_int64_t`.

**Call to**

*lammps_gather_angles()*

---

**subroutine gather**(*self*, *name*, *count*, *data*)

Gather the named per-atom, per-atom fix, per-atom compute, or fix property/atom-based entities from all processes, in order by atom ID.

Added in version 22Dec2022.

This subroutine gathers data from all processes and stores them in a one-dimensional allocatable array. The array *data* will be ordered by atom ID, which requires consecutive IDs (1 to *natoms*). If you need a similar array but for non-consecutive atom IDs, see *lammps_gather_concat()*; for a similar array but for a subset of atoms, see *lammps_gather_subset()*.

The *data* array will be ordered in groups of *count* values, sorted by atom ID (e.g., if *name* is *x*, then *data* is [x(1,1), x(2,1), x(3,1), x(1,2), x(2,2), x(3,2), x(1,3), . . .]); *data* must be `ALLOCATABLE` and will be allocated to length (*count* × *natoms*), as queried by *get_natoms()*.

This function will return an error if fix or compute data are requested and the fix or compute ID given does not have per-atom data. See the note about re-interpreting the vector as a matrix at *gather_atoms()*.

This function is not compatible with `-DLAMMPS_BIGBIG`.

**Parameters**

- **name** *[character(len=*)]* :: desired quantity (e.g., "x" or "mask" for atom properties, "f_id" for per-atom fix data, "c_id" for per-atom compute data, "d_name" or "i_name" for fix property/atom vectors with *count* = 1, "d2_name" or "i2_name" for fix property/atom vectors with *count* > 1)
- **count** *[integer(c_int)]* :: number of per-atom values (e.g., 1 for *type* or *charge*, 3 for *x* or *f*); use *count* = 3 with *image* if you want the image flags unpacked into (*x*,*y*,*z*) components.
- **data** *[real(c_double),dimension(:),allocatable]* :: array into which to store the data. Array *must* have the `ALLOCATABLE` attribute and be of rank 1 (i.e., `DIMENSION(:)`). If this array is already allocated, it will be reallocated to fit the length of the incoming data.

**Call to**
*lammps_gather()*

---

**subroutine gather_dihedrals**(*data*)

Gather type and constituent atom information for all dihedrals.

Added in version 8Feb2023.

This function copies the list of all dihedrals into an allocatable array. The array will be filled with (dihedral type, dihedral atom 1, dihedral atom 2, dihedral atom 3, dihedral atom 4) for each dihedral. The array is allocated to the right length (i.e., five times the number of dihedrals). The array *data* must be of the same type as the LAMMPS `tagint` type, which is equivalent to either `INTEGER(c_int)` or `INTEGER(c_int64_t)`, depending on whether `-DLAMMPS_BIGBIG` was used when LAMMPS was built. If the supplied array does not match, an error will result at run-time.

An example of how to use this routine is below:

```
PROGRAM dihedrals
  USE, INTRINSIC :: ISO_C_BINDING, ONLY : c_int
  USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY : OUTPUT_UNIT
  USE LIBLAMMPS
  IMPLICIT NONE
  INTEGER(c_int), DIMENSION(:), ALLOCATABLE, TARGET :: dihedrals
  INTEGER(c_int), DIMENSION(:,:), POINTER :: dihedrals_array
  TYPE(lammps) :: lmp
  INTEGER :: i
  ! other commands to initialize LAMMPS, create dihedrals, etc.
  CALL lmp%gather_dihedrals(dihedrals)
  dihedrals_array(1:5, 1:SIZE(dihedrals)/5) => dihedrals
  DO i = 1, SIZE(dihedrals)/5
    WRITE(OUTPUT_UNIT,'(A,1X,I4,A,I4,1X,I4,1X,I4,1X,I4)') 'dihedral', dihedrals_
↪array(1,i), &
      '; type = ', dihedrals_array(2,i), dihedrals_array(3,i), dihedrals_array(4,i),
↪ dihedrals_array(5,i)
  END DO
END PROGRAM dihedrals
```

**Parameters**
**data** *[integer(kind=*),allocatable]* :: array into which to copy the result. *The `KIND` parameter is either `c_int` or, if LAMMPS was compiled with `-DLAMMPS_BIGBIG`, kind `c_int64_t`.

**Call to**
*lammps_gather_dihedrals()*

---

**subroutine gather_impropers**(*data*)

Gather type and constituent atom information for all impropers.

Added in version 8Feb2023.

This function copies the list of all impropers into an allocatable array. The array will be filled with (improper type, improper atom 1, improper atom 2, improper atom 3, improper atom 4) for each improper. The array is allocated to the right length (i.e., five times the number of impropers). The array *data* must be of the same type as the LAMMPS `tagint` type, which is equivalent to either `INTEGER(c_int)` or `INTEGER(c_int64_t)`,

depending on whether -DLAMMPS_BIGBIG was used when LAMMPS was built. If the supplied array does not match, an error will result at run-time.

An example of how to use this routine is below:

```
PROGRAM impropers
  USE, INTRINSIC :: ISO_C_BINDING, ONLY : c_int
  USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY : OUTPUT_UNIT
  USE LIBLAMMPS
  IMPLICIT NONE
  INTEGER(c_int), DIMENSION(:), ALLOCATABLE, TARGET :: impropers
  INTEGER(c_int), DIMENSION(:,:), POINTER :: impropers_array
  TYPE(lammps) :: lmp
  INTEGER :: i
  ! other commands to initialize LAMMPS, create impropers, etc.
  CALL lmp%gather_impropers(impropers)
  impropers_array(1:5, 1:SIZE(impropers)/5) => impropers
  DO i = 1, SIZE(impropers)/5
    WRITE(OUTPUT_UNIT,'(A,1X,I4,A,I4,1X,I4,1X,I4,1X,I4)') 'improper', impropers_
↪array(1,i), &
      '; type = ', impropers_array(2,i), impropers_array(3,i), impropers_array(4,i),
↪ impropers_array(5,i)
  END DO
END PROGRAM impropers
```

**Parameters**

**data** *[integer(kind=*),allocatable]* :: array into which to copy the result. *The KIND parameter is either c_int or, if LAMMPS was compiled with -DLAMMPS_BIGBIG, kind c_int64_t.

**Call to**

*lammps_gather_impropers()*

---

**subroutine gather_concat**(*self*, *name*, *count*, *data*)

Gather the named per-atom, per-atom fix, per-atom compute, or fix property/atom-based entities from all processes, unordered.

Added in version 22Dec2022.

This subroutine gathers data for all atoms and stores them in a one-dimensional allocatable array. The data will be a concatenation of chunks from each processor's owned atoms, in whatever order the atoms are in on each processor. This process has no requirement that the atom IDs be consecutive. If you need the ID of each atom, you can do another call to either *gather_atoms_concat()* or *gather_concat()* with *name* set to id. If you have consecutive IDs and want the data to be in order, use *gather()*; for a similar array but for a subset of atoms, use *gather_subset()*.

The *data* array will be in groups of *count* values, with *natoms* groups total, but not in order by atom ID (e.g., if *name* is *x* and *count* is 3, then *data* might be something like [x(1,11), x(2,11), x(3,11), x(1,3), x(2,3), x(3,3), x(1,5), ...]); *data* must be ALLOCATABLE and will be allocated to length (*count* $\times$ *natoms*), as queried by *get_natoms()*.

This function is not compatible with -DLAMMPS_BIGBIG.

**Parameters**

- **name** *[character(len=*)]* :: desired quantity (e.g., "x" or "mask" for atom properties, "f_id" for per-atom fix data, "c_id" for per-atom compute data, "d_name" or "i_name" for fix property/atom vectors with *count* = 1, "d2_name" or "i2_name" for fix property/atom vectors with *count* $> 1$)
- **count** *[integer(c_int)]* :: number of per-atom values you expect per atom (e.g., 1 for *type*, *mask*, or *charge*; 3 for *x*, *v*, or *f*). Use *count* = 3 with *image* if you want a single image flag unpacked into *x*/*y*/*z* components.
- **data** *[polymorphic,dimension(:),allocatable]* :: array into which to store the data. Array *must* have the `ALLOCATABLE` attribute and be of rank 1 (i.e., `DIMENSION(:)`). If this array is already allocated, it will be reallocated to fit the length of the incoming data. It should have type `INTEGER(c_int)` if expecting integer data and `REAL(c_double)` if expecting floating-point data.

**Call to**

*lammps_gather_concat()*

---

`subroutine gather_subset`(*name*, *count*, *ids*, *data*)

Gather the named per-atom, per-atom fix, per-atom compute, or fix property/atom-based entities from all processes for a subset of atoms.

Added in version 22Dec2022.

This subroutine gathers data for the requested atom IDs and stores them in a one-dimensional allocatable array. The data will be ordered by atom ID, but there is no requirement that the IDs be consecutive. If you wish to return a similar array for *all* the atoms, use *gather()* or *gather_concat()*.

The *data* array will be in groups of *count* values, sorted by atom ID in the same order as the array *ids* (e.g., if *name* is *x*, *count* = 3, and *ids* is [100, 57, 210], then *data* might look like [*x*(1,100), *x*(2,100), *x*(3,100), *x*(1,57), *x*(2,57), *x*(3,57), *x*(1,210), . . .]); *ids* must be provided by the user, and *data* must have the `ALLOCATABLE` attribute and be of rank 1 (i.e., `DIMENSION(:)`). If *data* is already allocated, it will be reallocated to fit the length of the incoming data.

This function is not compatible with `-DLAMMPS_BIGBIG`.

**Parameters**

- **name** *[character(len=*)]* :: quantity to be scattered
- **ids** *[integer(c_int),dimension(:)]* :: atom IDs corresponding to the atoms being scattered (e.g., "x" or "f" for atom properties, "f_id" for per-atom fix data, "c_id" for per-atom compute data, "d_name" or "i_name" for fix property/atom vectors with *count* = 1, "d2_name" or "i2_name" for fix property/atom vectors with *count* $> 1$)
- **count** *[integer(c_int)]* :: number of per-atom values you expect per atom (e.g., 1 for *type*, *mask*, or *charge*; 3 for *x*, *v*, or *f*). Use *count* = 3 with *image* if you want a single image flag unpacked into *x*/*y*/*z* components.
- **data** *[polymorphic,dimension(:),allocatable]* :: per-atom values packed into a one-dimensional array containing the data to be scattered. This array must have the `ALLOCATABLE` attribute and will be allocated either to the same length as *ids* (for *mask*, *type*, etc.) or to three times its length (for *x*, *f*, etc.); the array must be rank 1 and be of type `INTEGER(c_int)` (e.g., for *mask* or *type*) or of type `REAL(c_double)` (e.g., for *charge*, *x*, or *f*).

**Call to**

*lammps_gather_subset()*

---

**subroutine scatter**(*name*, *data*)

This function calls *lammps_scatter()* to scatter the named per-atom, per-atom fix, per-atom compute, or fix property/atom-based entity in *data* to all processes.

Added in version 22Dec2022.

This subroutine takes data stored in a one-dimensional array supplied by the user and scatters them to all atoms on all processes. The data must be ordered by atom ID, with the requirement that the IDs be consecutive. Use *scatter_subset()* to scatter data for some (or all) atoms, unordered.

The *data* array needs to be ordered in groups of *count* values, sorted by atom ID (e.g., if *name* is *x* and *count* = 3, then *data* = [*x*(1,1), *x*(2,1), *x*(3,1), *x*(1,2), *x*(2,2), *x*(3,2), *x*(1,3), . . .]); *data* must be of length (*count* × *natoms*).

This function is not compatible with `-DLAMMPS_BIGBIG`.

**Parameters**

- **name** *[character(len=*)]* :: desired quantity (e.g., "x" or "f" for atom properties, "f_id" for per-atom fix data, "c_id" for per-atom compute data, "d_name" or "i_name" for fix property/atom vectors with *count* = 1, "d2_name" or "i2_name" for fix property/atom vectors with *count* > 1)
- **data** *[polymorphic,dimension(:)]* :: per-atom values packed in a one-dimensional array; *data* should be of type `INTEGER(c_int)` or `REAL(c_double)`, depending on the type of data being scattered, and be of rank 1 (i.e., `DIMENSION(:)`).

**Call to**

*lammps_scatter()*

---

**subroutine scatter_subset**(*name*, *ids*, *data*)

This function calls *lammps_scatter_subset()* to scatter the named per-atom, per-atom fix, per-atom compute, or fix property/atom-based entities in *data* from a subset of atoms to all processes.

Added in version 22Dec2022.

This subroutine takes data stored in a one-dimensional array supplied by the user and scatters them to a subset of atoms on all processes. The array *data* contains data associated with atom IDs, but there is no requirement that the IDs be consecutive, as they are provided in a separate array. Use *scatter()* to scatter data for all atoms, in order.

The *data* array needs to be organized in groups of *count* values, with the groups in the same order as the array *ids*. For example, if you want *data* to be the array [x(1,1), x(2,1), x(3,1), x(1,100), x(2,100), x(3,100), x(1,57), x(2,57), x(3,57)], then *count* = 3 and *ids* = [1, 100, 57].

This function is not compatible with `-DLAMMPS_BIGBIG`.

**Parameters**

- **name** *[character(len=*)]* :: desired quantity (e.g., "x" or "mask" for atom properties, "f_id" for per-atom fix data, "c_id" for per-atom compute data, "d_name" or "i_name" for fix property/atom vectors with *count* = 1, "d2_name" or "i2_name" for fix property/atom vectors with *count* > 1)
- **ids** *[integer(c_int)]* :: list of atom IDs to scatter data for
- **data** *[polymorphic ,dimension(:)]* :: per-atom values packed in a one-dimensional array of length *size(ids)* * *count*.

**Call to**

*lammps_scatter_subset()*

**function create_atoms**(*[id,] type, x, [v,] [image,] [bexpand]*)

This method calls *lammps_create_atoms()* to create additional atoms from a given list of coordinates and a list of atom types. Additionally, the atom IDs, velocities, and image flags may be provided.

Added in version 3Nov2022.

**Parameters**

- **type** *[integer(c_int),dimension(N)]* :: vector of $N$ atom types (required/see note below)
- **x** *[real(c_double),dimension(3N)]* :: vector of $3N$ $x/y/z$ positions of the new atoms, arranged as $[x_1, y_1, z_1, x_2, y_2, \ldots]$ (required/see note below)

**Options**

- **id** *[integer(kind=*),dimension(N),optional]* :: vector of $N$ atom IDs; if absent, LAMMPS will generate them for you. *The KIND parameter should be c_int unless LAMMPS was compiled with -DLAMMPS_BIGBIG, in which case it should be c_int64_t.
- **v** *[real(c_double),dimension(3N),optional]* :: vector of $3N$ $x/y/z$ velocities of the new atoms, arranged as $[v_{1,x}, v_{1,y}, v_{1,z}, v_{2,x}, \ldots]$; if absent, they will be set to zero
- **image** *[integer(kind=*),dimension(N),optional]* :: vector of $N$ image flags; if absent, they are set to zero. *The KIND parameter should be c_int unless LAMMPS was compiled with -DLAMMPS_BIGBIG, in which case it should be c_int64_t. See note below.
- **bexpand** *[logical,optional]* :: if .TRUE., atoms outside of shrink-wrap boundaries will be created, not dropped, and the box dimensions will be extended. Default is .FALSE.

**Return**

**atoms** *[integer(c_int)]* :: number of created atoms

**Call to**

*lammps_create_atoms()*

**Note**

The *type* and *x* arguments are required, but they are declared OPTIONAL in the module because making them mandatory would require *id* to be present as well. To have LAMMPS generate the ids for you, use a call something like

```
lmp%create_atoms(type=new_types, x=new_xs)
```

**Note**

When LAMMPS has been compiled with -DLAMMPS_BIGBIG, it is not possible to include the *image* parameter but omit the *id* parameter. Either *id* must be present, or both *id* and *image* must be absent. This is required because having all arguments be optional in both generic functions creates an ambiguous interface. This limitation does not exist if LAMMPS was not compiled with -DLAMMPS_BIGBIG.

**subroutine create_molecule**(*id*, *jsonstr*)

Add molecule template from string with JSON data

Added in version 22Jul2025.

**Parameters**

- **id** *[character(len=*)]* :: desired molecule-ID
- **jsonstr** *[character(len=*)]* :: string with JSON data defining the molecule template

**Call to**

*lammps_create_molecule()*

---

`function find_pair_neighlist`(*style[, exact][, nsub][, reqid]*)

Find index of a neighbor list requested by a pair style.

Added in version 3Nov2022.

This function determines which of the available neighbor lists for pair styles matches the given conditions. It first matches the style name. If *exact* is `.TRUE.`, the name must match exactly; if `.FALSE.`, a regular expression or sub-string match is done. If the pair style is *hybrid* or *hybrid/overlay*, the style is matched against the sub-styles instead. If the same pair style is used multiple times as a sub-style, the *nsub* argument must be $> 0$; this argument represents the *n*th instance of the sub-style (same as for the *pair_coeff* command, for example). In that case, *nsub* = 0 will not produce a match, and the function will return $-1$.

The final condition to be checked is the request ID (*reqid*). This will usually be zero, but some pair styles request multiple neighbor lists and set the request ID to a value greater than zero.

**Parameters**

**style** *[character(len=*)]* :: String used to search for pair style instance.

**Options**

- **exact** *[logical,optional]* :: Flag to control whether style should match exactly or only a regular expression/sub-string match is applied. Default: `.TRUE.`.
- **nsub** *[integer(c_int),optional]* :: Match *nsub*th hybrid sub-style instance of the same style. Default: 0.
- **reqid** *[integer(c_int),optional]* :: Request ID to identify the neighbor list in case there are multiple requests from the same pair style instance. Default: 0.

**Call to**

*lammps_find_pair_neighlist()*

**Return**

**index** *[integer(c_int)]* :: Neighbor list index if found, otherwise $-1$.

---

`function find_fix_neighlist`(*id*[, *reqid* ])

Find index of a neighbor list requested by a fix.

Added in version 3Nov2022.

The neighbor list request from a fix is identified by the fix ID and the request ID. The request ID is typically zero, but will be $> 0$ for fixes with multiple neighbor list requests.

**Parameters**

**id** *[character(len=*)]* :: Identifier of fix instance

**Options**

**reqid** *[integer(c_int),optional]* :: request ID to identify the neighbor list in cases in which there are multiple requests from the same fix. Default: 0.

**Call to**
*lammps_find_fix_neighlist()*

**Return**
**index** *[integer(c_int)]* :: neighbor list index if found, otherwise −1

---

`function find_compute_neighlist(`*id*`[`, *reqid* `])`

Find index of a neighbor list requested by a compute.

Added in version 3Nov2022.

The neighbor list request from a compute is identified by the compute ID and the request ID. The request ID is typically zero, but will be $> 0$ in case a compute has multiple neighbor list requests.

**Parameters**
**id** *[character(len=*)]* :: Identifier of compute instance.

**Options**
**reqid** *[integer(c_int),optional]* :: request ID to identify the neighbor list in cases in which there are multiple requests from the same compute. Default: 0.

**Call to**
*lammps_find_compute_neighlist()*

**Return**
**index** *[integer(c_int)]* :: neighbor list index if found, otherwise −1.

---

`function neighlist_num_elements(`*idx*`)`

Return the number of entries in the neighbor list with the given index.

Added in version 3Nov2022.

**Parameters**
**idx** *[integer(c_int)]* :: neighbor list index

**Call to**
*lammps_neighlist_num_elements() lammps_neighlist_num_elements()*

**Return**
**inum** *[integer(c_int)]* :: number of entries in neighbor list, or −1 if *idx* is not a valid index.

---

`subroutine neighlist_element_neighbors(`*idx*, *element*, *iatom*, *neighbors*`)`

Return atom local index, number of neighbors, and array of neighbor local atom indices of a neighbor list entry.

Added in version 3Nov2022.

**Parameters**

- **idx** *[integer(c_int)]* :: index of this neighbor list in the list of all neighbor lists
- **element** *[integer(c_int)]* :: index of this neighbor list entry
- **iatom** *[integer(c_int)]* :: local atom index (i.e., in the range [1,nlocal+nghost]; -1 if invalid or element value
- **neighbors** *[integer(c_int),dimension(:),pointer]* :: pointer to an array of neighboring atom local indices

**Call to**
*lammps_neighlist_element_neighbors()*

---

function version()

This method returns the numeric LAMMPS version like *lammps_version()* does.

**Call to**
*lammps_version()*

**Return**
**version** *[integer]* :: LAMMPS version

---

subroutine get_os_info(*buffer*)

This function can be used to retrieve detailed information about the hosting operating system and compiler/runtime environment.

Added in version 3Nov2022.

A suitable buffer has to be provided. The assembled text will be truncated so as not to overflow this buffer. The string is typically a few hundred bytes long.

**Parameters**
**buffer** *[character(len=*)]* :: string that will house the information.

**Call to**
*lammps_get_os_info()*

---

function config_has_mpi_support()

This function is used to query whether LAMMPS was compiled with a real MPI library or in serial.

Added in version 3Nov2022.

**Call to**
*lammps_config_has_mpi_support()*

**Return**
**has_mpi** *[logical]* :: `.FALSE.` when compiled with STUBS, `.TRUE.` if complied with MPI.

---

function config_has_omp_support()

This function is used to query whether LAMMPS was compiled with OpenMP enabled.

Added in version 10Sep2025.

**Call to**
*lammps_config_has_omp_support()*

**Return**
**has_omp** *[logical]* :: `.TRUE.` when compiled with OpenMP enabled, `.FALSE.` if not.

---

function config_has_gzip_support()

Check if the LAMMPS library supports reading or writing compressed files via a pipe to gzip or similar compression programs.

Added in version 3Nov2022.

Several LAMMPS commands (e.g., *read_data command*, *write_data command*, *dump styles atom, custom, and xyz*) support reading and writing compressed files via creating a pipe to the `gzip` program. This function checks whether this feature was *enabled at compile time*. It does **not** check whether `gzip` or any other supported compression programs themselves are installed and usable.

**Call to**
*lammps_config_has_gzip_support()*

**Return**
**has_gzip** *[logical]*

---

function config_has_png_support()

Check if the LAMMPS library supports writing PNG format images.

Added in version 3Nov2022.

The LAMMPS *dump style image* supports writing multiple image file formats. Most of them, however, need support from an external library, and using that has to be *enabled at compile time*. This function checks whether support for the PNG image file format is available in the current LAMMPS library.

**Call to**
*lammps_config_has_png_support()*

**Return**
**has_png** *[logical]*

---

function config_has_jpeg_support()

Check if the LAMMPS library supports writing JPEG format images.

Added in version 3Nov2022.

The LAMMPS *dump style image* supports writing multiple image file formats. Most of them, however, need support from an external library, and using that has to be *enabled at compile time*. This function checks whether support for the JPEG image file format is available in the current LAMMPS library.

**Call to**
*lammps_config_has_jpeg_support()*

**Return**
**has_jpeg** *[logical]*

---

function config_has_ffmpeg_support()

Check if the LAMMPS library supports creating movie files via a pipe to ffmpeg.

Added in version 3Nov2022.

The LAMMPS *dump style movie* supports generating movies from images on-the-fly via creating a pipe to the ffmpeg program. This function checks whether this feature was *enabled at compile time*. It does **not** check whether the `ffmpeg` itself is installed and usable.

**Call to**
*lammps_config_has_ffmpeg_support()*

**Return**
**has_ffmpeg** *[logical]*

---

`function config_has_exceptions()`

Check whether LAMMPS errors will throw C++ exceptions.

Added in version 3Nov2022.

When using the library interface, the library interface functions will "catch" exceptions, and then the error status can be checked by calling *has_error()*. The most recent error message can be retrieved via *get_last_error_message()*. This allows to restart a calculation or delete and recreate the LAMMPS instance when a C++ exception occurs. One application of using exceptions this way is LAMMPS-GUI

**Call to**
*lammps_config_has_exceptions()*

**Return**
**has_exceptions** *[logical]*

---

`function config_has_package(`*name*`)`

Check whether a specific package has been included in LAMMPS

Added in version 3Nov2022.

This function checks whether the LAMMPS library in use includes the specific *LAMMPS package* provided as argument.

**Call to**
*lammps_config_has_package()*

**Return**
**has_package** *[logical]*

---

`function config_package_count()`

Count the number of installed packages in the LAMMPS library.

Added in version 3Nov2022.

This function counts how many *LAMMPS packages* are included in the LAMMPS library in use. It directly calls the C library function *lammps_config_package_count()*.

**Call to**
*lammps_config_package_count()*

**Return**
**npackages** *[integer(c_int)]* :: number of packages installed

---

**subroutine config_package_name**(*idx*, *buffer*)

Get the name of a package in the list of installed packages in the LAMMPS library.

Added in version 3Nov2022.

This subroutine copies the name of the package with the index *idx* into the provided string *buffer*. If the name of the package exceeds the length of the buffer, it will be truncated accordingly. If the index is out of range, *buffer* is set to an empty string.

**Parameters**

- **idx** *[integer(c_int)]* :: index of the package in the list of included packages ($0 \leq idx <$ package count)
- **buffer** *[character(len=*)]* :: string to hold the name of the package

**Call to**

*lammps_config_package_name()*

---

**subroutine installed_packages**(*package*[, *length* ])

Obtain a list of the names of enabled packages in the LAMMPS shared library and store it in *package*.

Added in version 3Nov2022.

This function is analogous to the *installed_packages* function in the Python API. The optional argument *length* sets the length of each string in the vector *package* (default: 31).

**Parameters**

**package** *[character(len=:),dimension(:),allocatable]* :: list of packages; *must* have the `ALLOCATABLE` attribute and be of rank 1 (i.e., `DIMENSION(:)`) with allocatable length.

**Options**

**length** *[integer,optional]* :: length of each string in the list. Default: 31.

**Call to**

*lammps_config_package_count() lammps_config_package_name()*

---

**function config_accelerator**(*package*, *category*, *setting*)

This function calls *lammps_config_accelerator()* to check the availability of compile time settings of included *accelerator packages* in LAMMPS.

Added in version 3Nov2022.

Supported packages names are “GPU”, “KOKKOS”, “INTEL”, and “OPENMP”. Supported categories are “api” with possible settings “cuda”, “hip”, “phi”, “pthreads”, “opencl”, “openmp”, and “serial”; and “precision” with possible settings “double”, “mixed”, and “single”.

**Parameters**

- **package** *[character(len=*)]* :: string with the name of the accelerator package
- **category** *[character(len=*)]* :: string with the name of the setting
- **setting** *[character(len=*)]* :: string with the name of the specific setting

**Call to**

*lammps_config_accelerator()*

**Return**

**available** *[logical]* :: `.TRUE.` if the combination of package, category, and setting is available, otherwise `.FALSE.`.

---

**function has_gpu_device()**

Checks for the presence of a viable GPU package device.

Added in version 3Nov2022.

This function calls *lammps_has_gpu_device()*, which checks at runtime whether an accelerator device is present that can be used with the *GPU package*.

More detailed information about the available device or devices can be obtained by calling the *get_gpu_device_info()* subroutine.

**Call to**
*lammps_has_gpu_device()*

**Return**
**available** *[logical]* :: `.TRUE.` if a viable device is available, `.FALSE.` if not.

---

**subroutine get_gpu_device_info(*buffer*)**

Get GPU package device information.

Added in version 3Nov2022.

Calls *lammps_get_gpu_device_info()* to retrieve detailed information about any accelerator devices that are viable for use with the *GPU package*. It will fill *buffer* with a string that is equivalent to the output of the `nvc_get_device` or `ocl_get_device` or `hip_get_device` tools that are compiled alongside LAMMPS if the GPU package is enabled.

A suitable-length Fortran string has to be provided. The assembled text will be truncated so as not to overflow this buffer. This string can be several kilobytes long if multiple devices are present.

**Parameters**
**buffer** *[character(len=*)]* :: string into which to copy the information.

**Call to**
*lammps_get_gpu_device_info()*

---

**function has_style(*category*, *name*)**

Check whether a specific style has been included in LAMMPS.

Added in version 3Nov2022.

This function calls *lammps_has_style()* to check whether the LAMMPS library in use includes the specific style *name* associated with a specific *category* provided as arguments. Please see *lammps_has_style()* for a list of valid categories.

**Parameters**

- **category** *[character(len=*)]* :: category of the style
- **name** *[character(len=*)]* :: name of the style

**Call to**
*lammps_has_style()*

**Return**
**has_style** *[logical]* :: `.TRUE.` if included, `.FALSE.` if not.

---

function style_count(*category*)

Count the number of styles of *category* in the LAMMPS library.

Added in version 3Nov2022.

This function counts how many styles in the provided *category* are included in the LAMMPS library currently in use. Please see *lammps_has_style()* for a list of valid categories.

**Parameters**
**category** *[character(len=*)]* :: category of styles to count

**Call to**
*lammps_style_count()*

**Return**
**count** *[integer(c_int)]* :: number of styles in *category*

---

subroutine style_name(*category*, *idx*, *buffer*)

Look up the name of a style by index in the list of styles of a given category in the LAMMPS library.

Added in version 3Nov2022.

This function calls *lammps_style_name()* and copies the name of the *category* style with index *idx* into the provided string *buffer*. The length of *buffer* must be long enough to contain the name of the style; if it is too short, the name will be truncated accordingly. If *idx* is out of range, *buffer* will be the empty string and a warning will be issued.

**Parameters**

- **category** *[character(len=*)]* :: category of styles
- **idx** *[integer(c_int)]* :: index of the style in the list of *category* styles ($1 \leq idx \leq$ style count)
- **buffer** *[character(len*)]* :: string buffer to copy the name of the style into

**Call to**
*lammps_style_name()*

---

function has_id(*category*, *name*)

This function checks if the current LAMMPS instance a *category* ID of the given *name* exists. Valid categories are: *compute*, *dump*, *fix*, *group*, *molecule*, *region*, and *variable*.

Added in version 3Nov2022.

**Parameters**

- **category** *[character(len=*)]* :: category of the ID
- **name** *[character(len=*)]* :: name of the ID

**Call to**
*lammps_has_id()*

**Return**
**has_id** *[logical]* :: `.TRUE.` if *category* style *name* exists, `.FALSE.` if not.

---

function id_count(*category*)

This function counts how many IDs in the provided *category* are defined in the current LAMMPS instance. Please see *has_id()* for a list of valid categories.

Added in version 3Nov2022.

**Parameters**
**category** *[character(len=*)]* :: category of the ID

**Call to**
*lammps_id_count()*

**Return**
**count** *[integer(c_int)]* :: number of IDs in *category*

---

subroutine id_name(*category*, *idx*, *buffer*)

Look up the name of an ID by index in the list of IDs of a given category.

Added in version 3Nov2022.

This function copies the name of the *category* ID with the index *idx* into the provided string *buffer*. The length of the buffer must be long enough to hold the string; if the name of the style exceeds the length of the buffer, it will be truncated accordingly. If *buffer* is ALLOCATABLE, it must be allocated *before* the function is called. If *idx* is out of range, *buffer* is set to an empty string and a warning is issued.

**Parameters**

- **category** *[character(len=*)]* :: category of IDs
- **idx** *[integer(c_int)]* :: index of the ID in the list of *category* styles ($0 \leq idx < count$)
- **buffer** *[character(len=*)]* :: string into which to copy the name of the style

**Call to**
*lammps_id_name()*

---

function plugin_count()

This function counts the number of loaded plugins.

Added in version 3Nov2022.

**Call to**
lammps_plugin_count()

**Return**
**n** *[integer(c_int)]* :: number of loaded plugins

---

subroutine plugin_name(*idx*, *stylebuf*, *namebuf*)

Look up the style and name of a plugin by its index in the list of plugins.

Added in version 3Nov2022.

This function copies the name of the *style* plugin with the index *idx* into the provided C-style string buffer. The length of the buffer must be provided as *buf_size* argument. If the name of the style exceeds the length of the buffer, it will be truncated accordingly. If the index is out of range, both strings are set to the empty string and a warning is printed.

**Parameters**

- **idx** *[integer(c_int)]* :: index of the plugin in the list all or *style* plugins
- **stylebuf** *[character(len=*)]* :: string into which to copy the style of the plugin
- **namebuf** *[character(len=*)]* :: string into which to copy the style of the plugin

**Call to**
`lammps_plugin_name()`

---

`function encode_image_flags`(*ix*, *iy*, *iz*)

Encodes three integer image flags into a single imageint.

Added in version 3Nov2022.

This function performs the bit-shift, addition, and bit-wise OR operations necessary to combine the values of three integers representing the image flags in the *x*-, *y*-, and *z*-directions. Unless LAMMPS is compiled with `-DLAMMPS_BIGBIG`, those integers are limited to 10-bit signed integers $[-512, 512)$. If `-DLAMMPS_BIGBIG` was used when compiling, then the return value is of kind `c_int64_t` instead of kind `c_int`, and the valid range for the individual image flags becomes $[-1048576, 1048575)$ (i.e., the range of a 21-bit signed integer). There is no check on whether the arguments conform to these requirements; values out of range will simply be wrapped back into the interval.

**Parameters**

- **ix** *[integer(c_int)]* :: image flag in *x*-direction
- **iy** *[integer(c_int)]* :: image flag in *y*-direction
- **iz** *[integer(c_int)]* :: image flag in *z*-direction

**Return**
**imageint** *[integer(kind=*)]* :: encoded image flag. *The `KIND` parameter is `c_int` unless LAMMPS was built with `-DLAMMPS_BIGBIG`, in which case it is `c_int64_t`.

**Note**

The fact that the programmer does not know the `KIND` parameter of the return value until compile time means that it is impossible to define an interface that works for both sizes of `imageint`. One side effect of this is that you must assign the return value of this function to a variable; it cannot be used as the argument to another function or as part of an array constructor. For example,

```
my_images = [lmp%encode_image_flags(0,0,0), lmp%encode_image_flags(1,0,0)]
```

will *not* work; instead, do something like

```
my_images(1) = lmp%encode_image_flags(0,0,0)
my_images(2) = lmp%encode_image_flags(1,0,0)
```

---

`subroutine decode_image_flags`(*image*, *flags*)

This function does the reverse operation of *encode_image_flags()*: it takes the image flag and performs the bit-shift and bit-masking operations to decode it and stores the resulting three integers into the array *flags*.

Added in version 3Nov2022.

**Parameters**

- **image** *[integer(kind=*)]* :: encoded image flag. *The `KIND` parameter is either `c_int` or, if LAMMPS was compiled with `-DLAMMPS_BIGBIG`, `c_int64_t`. Kind compatibility is checked at run-time.
- **flags** *[integer(c_int),dimension(3)]* :: three-element vector where the decoded image flags will be stored.

---

**subroutine set_fix_external_callback**(*id*, *callback*, *caller*)

Set the callback function for a *fix external* instance with the given ID.

Added in version 22Dec2022.

Fix *external* allows programs that are running LAMMPS through its library interface to modify certain LAMMPS properties on specific time steps, similar to the way other fixes do.

This subroutine sets the callback function for use with the "pf/callback" mode. The function should have Fortran language bindings with the following interface, which depends on how LAMMPS was compiled:

```
ABSTRACT INTERFACE
  SUBROUTINE external_callback(caller, timestep, ids, x, fexternal)
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : c_int, c_double, c_int64_t
    CLASS(*), INTENT(INOUT) :: caller
    INTEGER(c_bigint), INTENT(IN) :: timestep
    INTEGER(c_tagint), DIMENSION(:), INTENT(IN) :: ids
    REAL(c_double), DIMENSION(:,:), INTENT(IN) :: x
    REAL(c_double), DIMENSION(:,:), INTENT(OUT) :: fexternal
  END SUBROUTINE external_callback
END INTERFACE
```

where `c_bigint` is `c_int64_t` and `c_tagint` is `c_int64_t` if `-DLAMMPS_BIGBIG` was used and `c_int` otherwise.

The argument *caller* to *set_fix_external_callback()* is unlimited polymorphic (i.e., it can be any Fortran object you want to pass to the calling function) and will be available as the first argument to the callback function. It can be your LAMMPS instance, which you might need if the callback function needs access to the library interface. The argument must be a scalar; to pass non-scalar data, wrap those data in a derived type and pass an instance of the derived type to *caller*.

The array *ids* is an array of length *nlocal* (as accessed from the `Atom` class or through *extract_global()*). The arrays *x* and *fexternal* are 3 × *nlocal* arrays; these are transposed from what they would look like in C (see note about array index order at *extract_atom()*).

The callback mechanism is one of two ways that forces can be applied to a simulation with the help of *fix external*. The alternative is *array* mode, where one calls *fix_external_get_force()*.

Please see the documentation for *fix external* for more information about how to use the fix and couple it with external programs.

**Parameters**

- **id** *[character(len=*)]* :: ID of *fix external* instance
- **callback** *[external]* :: subroutine *fix external* should call
- **caller** *[class(*),optional]* :: object you wish to pass to the callback procedure (must be a scalar; see note)

**Call to**

*lammps_set_fix_external_callback()*

**Note**

The interface for your callback function must match types precisely with the abstract interface block given above. **The compiler probably will not be able to check this for you.** In particular, the first argument ("caller") must be of type `CLASS(*)` or you will probably get a segmentation fault or at least a misinterpretation of whatever is in memory there. You can resolve the object using the `SELECT TYPE` construct. An example callback function (assuming LAMMPS was compiled with `-DLAMMPS_SMALLBIG`) that applies something akin to Hooke's Law (with each atom having a different $k$ value) is shown below.

```
MODULE stuff
  USE, INTRINSIC :: ISO_C_BINDING, ONLY : c_int, c_double, c_int64_t
  USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY : error_unit
  IMPLICIT NONE

  TYPE shield
     REAL(c_double), DIMENSION(:,:), ALLOCATABLE :: k
     ! assume k gets allocated to dimension(3,nlocal) at some point
     ! and assigned values
  END TYPE shield

  SUBROUTINE my_callback(caller, timestep, ids, x, fexternal)
    CLASS(*), INTENT(INOUT) :: caller
    INTEGER(c_int), INTENT(IN) :: timestep
    INTEGER(c_int64_t), INTENT(IN) :: ids
    REAL(c_double), INTENT(IN) :: x(:,:)
    REAL(c_double), INTENT(OUT) :: fexternal(:,:)

    SELECT TYPE (caller)
      TYPE IS (shield)
         fexternal = - caller%k * x
      CLASS DEFAULT
         WRITE(error_unit,*) 'UH OH...'
    END SELECT
  END SUBROUTINE my_callback
END MODULE stuff

! then, when assigning the callback function, do this:
PROGRAM example
  USE LIBLAMMPS
  USE stuff
  TYPE(lammps) :: lmp
  TYPE(shield) :: my_shield
  lmp = lammps()
  CALL lmp%command('fix ext all external pf/callback 1 1')
  CALL lmp%set_fix_external_callback('ext', my_callback, my_shield)
END PROGRAM example
```

**function fix_external_get_force**(*id*)

Get pointer to the force array storage in a fix external instance with the given ID.

Added in version 22Dec2022.

Fix *external* allows programs that are running LAMMPS through its library interfaces to add or modify certain LAMMPS properties on specific time steps, similar to the way other fixes do.

This function provides access to the per-atom force storage in a fix external instance with the given fix-ID to be added to the individual atoms when using the "pf/array" mode. The *fexternal* array can be accessed like other "native" per-atom arrays accessible via the *extract_atom()* function. Please note that the array stores the forces for *local* atoms for each MPI rank, in the order determined by the neighbor list build. Because the underlying data structures can change as well as the order of atom as they migrate between MPI processes because of the domain decomposition parallelization, this function should be always called immediately before the forces are going to be set to get an up-to-date pointer. You can use, for example, *extract_setting()* to obtain the number of local atoms *nlocal* and then assume the dimensions of the returned force array as `REAL(c_double) :: force(3,nlocal)`.

This function is an alternative to the callback mechanism in fix external set up by *set_fix_external_callback()*. The main difference is that this mechanism can be used when forces are to be pre-computed and the control alternates between LAMMPS and the external driver, while the callback mechanism can call an external subroutine to compute the force when the fix is triggered and needs them.

Please see the documentation for *fix external* for more information about how to use the fix and how to couple it with an external program.

**Parameters**
**id** *[character(len=*)]* :: ID of *fix external* instance

**Call to**
*lammps_fix_external_get_force()*

**Return**
**fexternal** *[real(c_double),dimension(3,nlocal)]* :: pointer to the per-atom force array allocated by the fix

---

`subroutine fix_external_set_energy_global(`*id*, *eng*`)`

Set the global energy contribution for a *fix external* instance with the given ID.

Added in version 22Dec2022.

This is a companion function to *set_fix_external_callback()* and *fix_external_get_force()* that also sets the contribution to the global energy from the external program. The value of the *eng* argument will be stored in the fix and applied on the current and all following time steps until changed by another call to this function. The energy is in energy units as determined by the current *units* settings and is the **total** energy of the contribution. Thus, when running in parallel, all MPI processes have to call this function with the **same** value, and this will be returned as a scalar property of the fix external instance when accessed in LAMMPS input commands or from variables.

Please see the documentation for *fix external* for more information about how to use the fix and how to couple it with an external program.

**Parameters**

- **id** *[character(len=*)]* :: fix ID of fix external instance
- **eng** *[real(c_double)]* :: total energy to be added to the global energy

**Call to**
*lammps_fix_external_set_energy_global()*

---

**subroutine fix_external_set_virial_global**(*id*, *virial*)

Set the global virial contribution for a fix external instance with the given ID.

Added in version 22Dec2022.

This is a companion function to *set_fix_external_callback()* and *fix_external_get_force()* to set the contribution to the global virial from an external program.

The six values of the *virial* array will be stored in the fix and applied on the current and all following time steps until changed by another call to this function. The components of the virial need to be stored in the following order: *xx*, *yy*, *zz*, *xy*, *xz*, *yz*. In LAMMPS, the virial is stored internally as *stress*volume* in units of *pressure*volume* as determined by the current *units* settings and is the **total** contribution. Thus, when running in parallel, all MPI processes have to call this function with the **same** value, and this will then be added by fix external.

Please see the documentation for *fix external* for more information about how to use the fix and how to couple it with an external code.

**Parameters**

- **id** *[character(len=*)]* :: fix ID of fix external instance
- **virial** *[real(c_double),dimension(6)]* :: the six global stress tensor components to be added to the global virial

**Call to**

*lammps_fix_external_set_virial_global()*

---

**subroutine fix_external_set_energy_peratom**(*id*, *eng*)

Set the per-atom energy contribution for a fix external instance with the given ID.

Added in version 22Dec2022.

This is a companion function to *set_fix_external_callback()* to set the per-atom energy contribution due to the fix from the external program as part of the callback function. For this to work, the LAMMPS object must be passed as part of the *caller* argument when registering the callback function, or the callback function must otherwise have access to the LAMMPS object, such as through a module-based pointer.

**Note**

This function is fully independent from *fix_external_set_energy_global()* and will **NOT** add any contributions to the global energy tally and will **NOT** check whether the sum of the contributions added here are consistent with the global added energy.

Please see the documentation for *fix external* for more information about how to use the fix and how to couple it with an external code.

**Parameters**

- **id** *[character(len=*)]* :: fix ID of the fix external instance
- **eng** *[real(c_double),dimension(:)]* :: array of length *nlocal* containing the energy to add to the per-atom energy

**Call to**

*lammps_fix_external_set_energy_peratom()*

---

**subroutine set_fix_external_set_virial_peratom**(*id*, *virial*)

This is a companion function to *set_fix_external_callback()* to set the per-atom virial contribution due to the fix from the external program as part of the callback function. For this to work, the LAMMPS object must be passed as the *caller* argument when registering the callback function.

Added in version 22Dec2022.

> **Note**
>
> This function is fully independent from *fix_external_set_virial_global()* and will **NOT** add any contributions to the global virial tally and **NOT** check whether the sum of the contributions added here are consistent with the global added virial.

The order and units of the per-atom stress tensor elements are the same as for the global virial. The type and dimensions of the per-atom virial array must be `REAL(c_double), DIMENSION(6,nlocal)`.

Please see the documentation for *fix external* for more information about how to use the fix and how to couple it with an external program.

**Parameters**

- **id** *[character(len=*)]* :: fix ID of fix external instance
- **virial** *[real(c_double),dimension(:,:)]* :: an array of 6 × *nlocal* components to be added to the per-atom virial

**Call to**

`lammps_set_virial_peratom()`

---

**subroutine fix_external_set_vector_length**(*id*, *length*)

Set the vector length for a global vector stored with fix external for analysis.

Added in version 22Dec2022.

This is a companion function to *set_fix_external_callback()* and *fix_external_get_force()* to set the length of a global vector of properties that will be stored with the fix via *fix_external_set_vector()*.

This function needs to be called **before** a call to *fix_external_set_vector()* and **before** a run or minimize command. When running in parallel, it must be called from **all** MPI processes with the same length argument.

Please see the documentation for *fix external* for more information about how to use the fix and how to couple it with an external program.

**Parameters**

- **id** *[character(len=*)]* :: fix ID of fix external instance
- **length** *[integer(c_int)]* :: length of the global vector to be stored with the fix

**Call to**

*lammps_fix_external_set_vector_length()*

---

**subroutine fix_external_set_vector**(*id*, *idx*, *val*)

Store a global vector value for a fix external instance with the given ID.

Added in version 22Dec2022.

This is a companion function to *set_fix_external_callback()* and *fix_external_get_force()* to set the values of a global vector of properties that will be stored with the fix and can be accessed from within LAMMPS input commands (e.g., fix ave/time or variables) when used in a vector context.

This function needs to be called **after** a call to *fix_external_set_vector_length()* and **before** a run or minimize command. When running in parallel, it must be called from **all** MPI processes with the **same** *idx* and *val* parameters. The variable *val* is assumed to be extensive.

> **Note**
>
> The index in the *idx* parameter is 1-based (i.e., the first element is set with *idx* = 1, and the last element of the vector with *idx* = *N*, where *N* is the value of the *length* parameter of the call to *fix_external_set_vector_length()*).

Please see the documentation for *fix external* for more information about how to use the fix and how to couple it with an external code.

**Parameters**

- **id** *[character(len=*)]* :: ID of fix external instance
- **idx** *[integer(c_int)]* :: 1-based index in global vector
- **val** *[integer(c_int)]* :: value to be stored in global vector at index *idx*

**Call to**

*lammps_fix_external_set_vector()*

---

`subroutine flush_buffers()`

This function calls *lammps_flush_buffers()*, which flushes buffered output to be written to screen and logfile. This can simplify capturing output from LAMMPS library calls.

Added in version 3Nov2022.

**Call to**

*lammps_flush_buffers()*

---

`function is_running()`

Check if LAMMPS is currently inside a run or minimization.

Added in version 3Nov2022.

This function can be used from signal handlers or multi-threaded applications to determine if the LAMMPS instance is currently active.

**Call to**

*lammps_is_running()*

**Return**

**is_running** *[logical]* :: `.FALSE.` if idle or `.TRUE.` if active

---

**subroutine force_timeout()**

Force a timeout to stop an ongoing run cleanly.

Added in version 3Nov2022.

This function can be used from signal handlers or multi-threaded applications to terminate an ongoing run cleanly.

**Call to**
*lammps_force_timeout()*

---

**function has_error()**

Check if there is a (new) error message available.

Added in version 3Nov2022.

This function can be used to query if an error inside of LAMMPS has thrown a *C++ exception*.

**Call to**
*lammps_has_error()*

**Return**
**has_error** *[logical]* :: `.TRUE.` if there is an error.

---

**subroutine get_last_error_message(***buffer*[, *status* ])

Copy the last error message into the provided buffer.

Added in version 3Nov2022.

This function can be used to retrieve the error message that was set in the event of an error inside of LAMMPS that resulted in a *C++ exception*. A suitable buffer for a string has to be provided. If the internally-stored error message is longer than the string, it will be truncated accordingly. The optional argument *status* indicates the kind of error: a "1" indicates an error that occurred on all MPI ranks and is often recoverable, while a "2" indicates an abort that would happen only in a single MPI rank and thus may not be recoverable, as other MPI ranks may be waiting on the failing MPI rank(s) to send messages.

**Parameters**
**buffer** *[character(len=*)]* :: string buffer to copy the error message into

**Options**
**status** *[integer(c_int),optional]* :: 1 when all ranks had the error, 2 on a single-rank error.

**Call to**
*lammps_get_last_error_message()*

---

## 1.4 LAMMPS C++ API

It is also possible to invoke the LAMMPS C++ API directly in your code. It lacks some of the convenience of the C library API, but it allows more direct access to simulation data and thus more low-level manipulations. The following links provide some examples and references to the C++ API.

### 1.4.1 Using the C++ API directly

Using the C++ classes of the LAMMPS library is lacking some of the convenience of the C library API, but it allows a more direct access to simulation data and thus more low-level manipulations and tighter integration of LAMMPS into another code. While for the complete C library API is provided in the `library.h` header file, for using the C++ API it is required to include the individual header files defining the individual classes in use. Typically the name of the class and the name of the header follow some simple rule. Examples are given below.

### 1.4.2 Creating or deleting a LAMMPS object

When using the LAMMPS library interfaces, the core task is to create an instance of the *LAMMPS_NS::LAMMPS* class. In C++ this can be done directly through the `new` operator. All further operations are then initiated through calling member functions of some of the components of the LAMMPS class or accessing their data members. The destruction of the LAMMPS instance is correspondingly initiated by using the `delete` operator. Here is a simple example:

```
#include "lammps.h"

#include <mpi.h>
#include <iostream>

int main(int argc, char **argv)
{
    LAMMPS_NS::LAMMPS *lmp;
    // custom argument vector for LAMMPS library
    const char *lmpargv[] {"liblammps", "-log", "none"};
    int lmpargc = sizeof(lmpargv)/sizeof(const char *);

    // explicitly initialize MPI
    MPI_Init(&argc, &argv);

    // create LAMMPS instance
    lmp = new LAMMPS_NS::LAMMPS(lmpargc, (char **)lmpargv, MPI_COMM_WORLD);
    // output numerical version string
    std::cout << "LAMMPS version ID: " << lmp->num_ver << std::endl;
    // delete LAMMPS instance
    delete lmp;

    // stop MPI environment
    MPI_Finalize();
    return 0;
}
```

This minimal example only requires to include the `lammps.h` header file since it only accesses a non-pointer member of the LAMMPS class.

### 1.4.3 Executing LAMMPS commands

Once a LAMMPS instance is created by your C++ code, you need to set up a simulation and that is most conveniently done by "driving" it through issuing commands like you would do when running a LAMMPS simulation from an input script. Processing of input in LAMMPS is handled by the *Input* class an instance of which is a member of the *LAMMPS* class. You have two options: reading commands from a file, or executing a single command from a string. See below for a small example:

```
#include "lammps.h"
#include "input.h"
#include <mpi.h>

using namespace LAMMPS_NS;

int main(int argc, char **argv)
{
    const char *lmpargv[] {"liblammps", "-log", "none"};
    int lmpargc = sizeof(lmpargv)/sizeof(const char *);

    MPI_Init(&argc, &argv);
    LAMMPS *lmp = new LAMMPS(lmpargc, (char **)lmpargv, MPI_COMM_WORLD);
    lmp->input->file("in.melt");
    lmp->input->one("run 100 post no");
    delete lmp;
    return 0;
}
```

CHAPTER

TWO

# USE PYTHON WITH LAMMPS

These pages describe various ways that LAMMPS and Python can be used together.

## 2.1 Overview

The LAMMPS distribution includes a `python` directory with the Python code needed to run LAMMPS from Python. The `python/lammps` package contains *the "lammps" Python module* that wraps the LAMMPS C-library interface. This module makes it is possible to do the following either from a Python script, or interactively from a Python prompt:

- create one or more instances of LAMMPS
- invoke LAMMPS commands or read them from an input script
- run LAMMPS incrementally
- extract LAMMPS results
- and modify internal LAMMPS data structures.

From a Python script you can do this in serial or in parallel. Running Python interactively in parallel does not generally work, unless you have a version of Python that extends Python to enable multiple instances of Python to read what you type.

To do all of this, you must build LAMMPS in *"shared" mode* and make certain that your Python interpreter can find the `lammps` Python package and the LAMMPS shared library file.

The Python wrapper for LAMMPS uses the ctypes package in Python, which auto-generates the interface code needed between Python and a set of C-style library functions. Ctypes has been part of the standard Python distribution since version 2.5. You can check which version of Python you have by simply typing "python" at a shell prompt. Below is an example output for Python version 3.8.5.

```
$ python
Python 3.8.5 (default, Aug 12 2020, 00:00:00)
[GCC 10.2.1 20200723 (Red Hat 10.2.1-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

**Warning**

The options described in this section of the manual for using Python with LAMMPS support only Python 3.6 or later. For use with Python 2.x you will need to use an older LAMMPS version like 29 Aug 2024 or older. If you notice Python code in the LAMMPS distribution that is not compatible with Python 3, please contact the LAMMPS developers or submit and issue on GitHub

LAMMPS can work together with Python in two ways. First, Python can wrap LAMMPS through the its *library interface*, so that a Python script can create one or more instances of LAMMPS and launch one or more simulations. In Python terms, this is referred to as "extending" Python with a LAMMPS module.

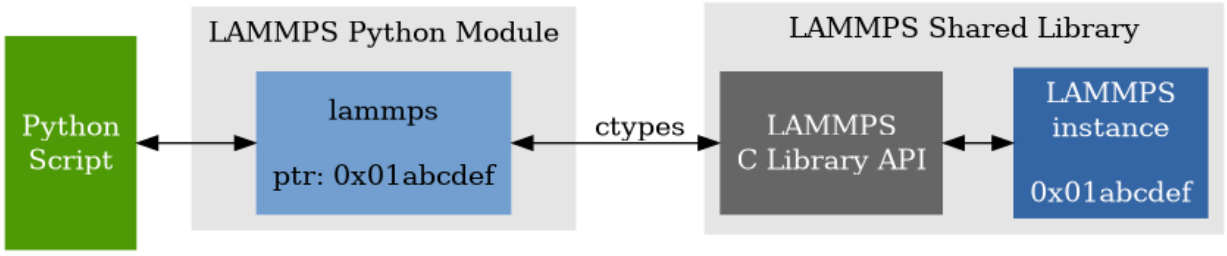



Fig. 1: Launching LAMMPS via Python

Second, LAMMPS can use the Python interpreter, so that a LAMMPS input script or styles can invoke Python code directly, and pass information back-and-forth between the input script and Python functions you write. This Python code can also call back to LAMMPS to query or change its attributes through the LAMMPS Python module mentioned above. In Python terms, this is called "embedding" Python into LAMMPS. When used in this mode, Python can perform script operations that the simple LAMMPS input script syntax can not.

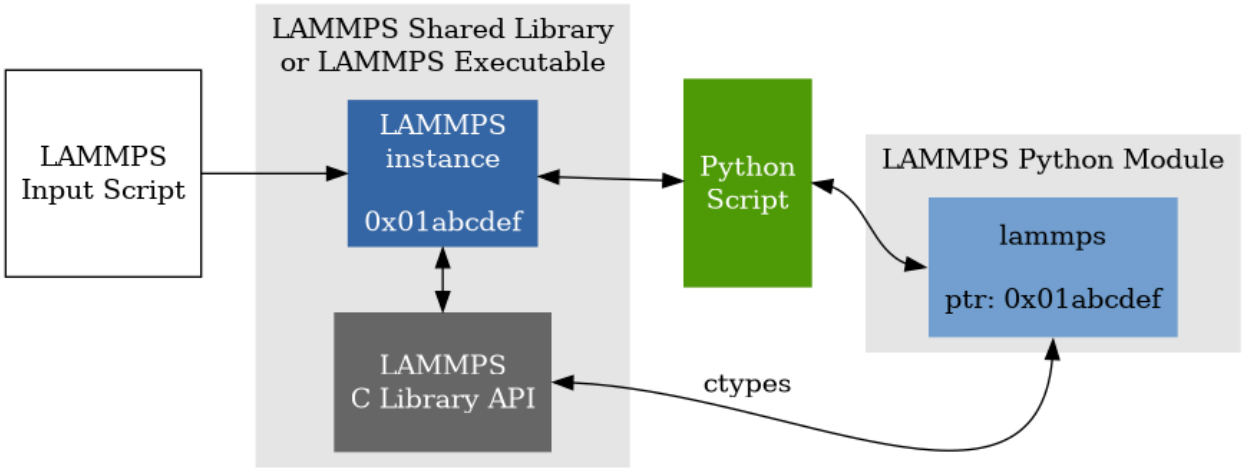



Fig. 2: Calling Python code from LAMMPS

## 2.2 Installation

The LAMMPS Python module enables calling the *LAMMPS C library API* from Python by dynamically loading functions in the LAMMPS shared library through the Python ctypes module. Because of the dynamic loading, it is required that LAMMPS is compiled in *"shared" mode*.

Changed in version 2Apr2025.

LAMMPS currently only supports Python version 3.6 or later.

Two components are necessary for Python to be able to invoke LAMMPS code:

- The LAMMPS Python Package (`lammps`) from the `python` folder

- The LAMMPS Shared Library (`liblammps.so`, `liblammps.dylib` or `liblammps.dll`) from the folder where you compiled LAMMPS.

### 2.2.1 Installing the LAMMPS Python Module and Shared Library

Making LAMMPS usable within Python and vice versa requires putting the LAMMPS Python package (`lammps`) into a location where the Python interpreter can find it and installing the LAMMPS shared library into a folder that the dynamic loader searches or inside of the installed `lammps` package folder. There are multiple ways to achieve this.

1. Install both components into a Python `site-packages` folder, either system-wide or in the corresponding user-specific folder. This way no additional environment variables need to be set, but the shared library is otherwise not accessible.
2. Do an installation into a virtual environment.
3. Leave the files where they are in the source/development tree and adjust some environment variables.

**Python package**

Compile LAMMPS with either *CMake* or the *traditional make* procedure in *shared mode*. After compilation has finished, type (in the compilation folder):

```
make install-python
```

This will try to build a so-called (binary) wheel file, a compressed binary python package and then install it with the python package manager 'pip'. Installation will be attempted into a system-wide `site-packages` folder and if that fails into the corresponding folder in the user's home directory. For a system-wide installation you usually would have to gain superuser privilege first, e.g. though `sudo`

| File | Location | Notes |
|---|---|---|
| LAMMPS Python package | • `$HOME/.local/lib/pythonX.Y/site-packages/lammps` | `X.Y` depends on the installed Python version |
| LAMMPS shared library | • `$HOME/.local/lib/pythonX.Y/site-packages/lammps` | `X.Y` depends on the installed Python version |

For a system-wide installation those folders would then become.

| File | Location | Notes |
|---|---|---|
| LAMMPS Python package | • `/usr/lib/pythonX.Y/site-packages/lammps` | `X.Y` depends on the installed Python version |
| LAMMPS shared library | • `/usr/lib/pythonX.Y/site-packages/lammps` | `X.Y` depends on the installed Python version |

No environment variables need to be set for those, as those folders are searched by default by Python or the LAMMPS Python package.

Changed in version 24Mar2022.

**Note**

If there is an existing installation of the LAMMPS python module, `make install-python` will try to update it. However, that will fail if the older version of the module was installed by LAMMPS versions until 17Feb2022. Those were using the distutils package, which does not create a "manifest" that allows a clean uninstall. The `make install-python` command will always produce a lammps-<version>-<python>-<abi>-<os>-<arch>.whl file (the 'wheel'). And this file can be later installed directly with `python -m pip install <wheel file>.whl` without having to type `make install-python` again and repeating the build step, too.

For the traditional make process you can override the python version to version x.y when calling `make` with `PYTHON=pythonX.Y`. For a CMake based compilation this choice has to be made during the CMake configuration step.

If the default settings of `make install-python` are not what you want, you can invoke `install.py` from the `python` directory manually as

```
python3 install.py -p <python package> -l <shared library> -v <version.h file> [-
↪n] [-f]
```

- The `-p` flag argument is the full path to the `python/lammps` folder to be installed,
- the `-l` flag argument is the full path to the LAMMPS shared library file to be installed,
- the `-v` flag argument is the full path to the `src/version.h` file
- the optional `-n` flag instructs the script to only build a wheel file but not attempt to install it (default is to try installing),
- the optional `-w` flag argument is the path to a folder where to store the resulting wheel file (default is the current folder)
- and the optional `-f` argument instructs the script to force installation even if pip would otherwise refuse installation with an *error about externally managed environments*. The Python developers recommend to not augment a Python installation with custom packages, both at the user and the system level, and advise to use virtual environments instead. Some recent Linux distributions enforce that recommendation by default.

Example command line for building only the wheel after building LAMMPS with `cmake` in the folder `build`:

```
python3 python/install.py -n -p python/lammps -l build/liblammps.so -v src/
↪version.h -w build
```

**Virtual environment**

A virtual environment is a minimal Python installation inside of a folder. It allows isolating and customizing a Python environment that is mostly independent from a user or system installation. For the core Python environment, it uses symbolic links to the system installation and thus it can be set up quickly and will not take up much disk space. This gives you the flexibility to install (newer/different) versions of Python packages that would potentially conflict with already installed system packages. It also does not requite any superuser privileges. See PEP 405: Python Virtual Environments for more information.

To create a virtual environment in the folder `$HOME/myenv`, use the venv module as follows.

```
# create virtual environment in folder $HOME/myenv
python3 -m venv $HOME/myenv
```

To activate the virtual environment type:

```
source $HOME/myenv/bin/activate
```

This has to be done every time you log in or open a new terminal window and after you turn off the virtual environment with the `deactivate` command.

When using CMake to build LAMMPS, you need to set `CMAKE_INSTALL_PREFIX` to the value of the `$VIRTUAL_ENV` environment variable during the configuration step. For the traditional make procedure, no additional steps are needed. After compiling LAMMPS you can do a "Python package only" installation with `make install-python` and the LAMMPS Python package and the shared library file are installed into the following locations:

| File | Location | Notes |
|---|---|---|
| LAMMPS Python Module | • `$VIRTUAL_ENV/lib/pythonX.Y/site-packages/lammps` | `X.Y` depends on the installed Python version |
| LAMMPS shared library | • `$VIRTUAL_ENV/lib/pythonX.Y/site-packages/lammps` | `X.Y` depends on the installed Python version |

**In place usage**

You can also *compile LAMMPS* as usual in *"shared" mode* leave the shared library and Python package inside the source/compilation folders. Instead of copying the files where they can be found, you need to set the environment variables `PYTHONPATH` (for the Python package) and `LD_LIBRARY_PATH` (or `DYLD_LIBRARY_PATH` on macOS

For Bourne shells (bash, ksh and similar) the commands are:

```
export PYTHONPATH=${PYTHONPATH}:${HOME}/lammps/python
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${HOME}/lammps/src
```

For the C-shells like csh or tcsh the commands are:

```
setenv PYTHONPATH ${PYTHONPATH}:${HOME}/lammps/python
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:${HOME}/lammps/src
```

On macOS you may also need to set `DYLD_LIBRARY_PATH` accordingly. You can make those changes permanent by editing your `$HOME/.bashrc` or `$HOME/.login` files, respectively.

**Note**

The `PYTHONPATH` needs to point to the parent folder that contains the `lammps` package!

In case you run into an "externally-managed-environment" error when trying to install the LAMMPS Python module, please refer to *corresponding paragraph* in the Python HOWTO page to learn about options for handling this error.

To verify if LAMMPS can be successfully started from Python, start the Python interpreter, load the `lammps` Python module and create a LAMMPS instance. This should not generate an error message and produce output similar to the following:

```
$ python
Python 3.8.5 (default, Sep  5 2020, 10:50:12)
[GCC 10.2.0] on linux
```

(continues on next page)

(continued from previous page)

```
Type "help", "copyright", "credits" or "license" for more information.
>>> import lammps
>>> lmp = lammps.lammps()
LAMMPS (18 Sep 2020)
using 1 OpenMP thread(s) per MPI task
>>>
```

**Note**

Unless you opted for "In place use", you will have to rerun the installation any time you recompile LAMMPS to ensure the latest Python package and shared library are installed and used.

**Note**

If you want Python to be able to load different versions of the LAMMPS shared library with different settings, you will need to manually copy the files under different names (e.g. `liblammps_mpi.so` or `liblammps_gpu.so`) into the appropriate folder as indicated above. You can then select the desired library through the *name* argument of the LAMMPS object constructor (see *Creating or deleting a LAMMPS object*).

### 2.2.2 Extending Python to run in parallel

If you wish to run LAMMPS in parallel from Python, you need to extend your Python with an interface to MPI. This also allows you to make MPI calls directly from Python in your script, if you desire.

We have tested this with MPI for Python (aka mpi4py) and you will find installation instruction for it below.

Installation of mpi4py (version 4.0.1 as of Feb 2025) can be done as follows:

- Via `pip` into a local user folder with:

```
python3 -m pip install --user mpi4py
```

- Via `dnf` into a system folder for RedHat/Fedora systems:

```
# for use with OpenMPI
sudo dnf install python3-mpi4py-openmpi
# for use with MPICH
sudo dnf install python3-mpi4py-mpich
```

- Via `pip` into a virtual environment (see above):

```
$ source $HOME/myenv/activate
(myenv)$ python -m pip install mpi4py
```

- Via `pip` into a system folder (not recommended):

```
sudo python3 -m pip install mpi4py
```

For more detailed installation instructions and additional options, please see the mpi4py installation page.

To use `mpi4py` and LAMMPS in parallel from Python, you **must** make certain that **both** are using the **same** implementation and version of MPI library. If you only have one MPI library installed on your system this is not an issue, but it can be if you have multiple MPI installations (e.g. on an HPC cluster to be selected through environment modules).

Your LAMMPS build is explicit about which MPI it is using, since it is either detected during CMake configuration or in the traditional make build system you specify the details in your low-level `src/MAKE/Makefile.foo` file. The installation process of `mpi4py` uses the `mpicc` command to find information about the MPI it uses to build against. And it tries to load "libmpi.so" from the `LD_LIBRARY_PATH`. This may or may not find the MPI library that LAMMPS is using. If you have problems running both mpi4py and LAMMPS together, this is an issue you may need to address, e.g. by loading the module for different MPI installation so that mpi4py finds the right one.

If you have successfully installed mpi4py, you should be able to run Python and type

```
from mpi4py import MPI
```

without error. You should also be able to run Python in parallel on a simple test script

```
mpirun -np 4 python3 test.py
```

where `test.py` contains the lines

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
print("Proc %d out of %d procs" % (comm.Get_rank(),comm.Get_size()))
```

and see one line of output for each processor you run on. Please note that the order of the lines is not deterministic

```
$ mpirun -np 4 python3 test.py
Proc 0 out of 4 procs
Proc 1 out of 4 procs
Proc 2 out of 4 procs
Proc 3 out of 4 procs
```

## 2.3 Run LAMMPS from Python

After compiling the LAMMPS shared library and making it ready to use, you can now write and run Python scripts that import the LAMMPS Python module and launch LAMMPS simulations through Python scripts. The following pages take you through the various steps necessary, what functionality is available and give some examples how to use it.

---

### 2.3.1 Running LAMMPS and Python in serial

To run a LAMMPS input in serial, type these lines into Python interactively from the `bench` directory:

```
from lammps import lammps
lmp = lammps()
lmp.file("in.lj")
```

Or put the same lines in the file `test.py` and run it as

```
python3 test.py
```

Either way, you should see the results of running the `in.lj` benchmark on a single processor appear on the screen, the same as if you had typed something like:

```
lmp_serial -in in.lj
```

### 2.3.2 Running LAMMPS and Python in parallel with MPI

To run LAMMPS in parallel, assuming you have installed the mpi4py package as discussed *Extending Python to run in parallel*, create a `test.py` file containing these lines:

```
from mpi4py import MPI
from lammps import lammps
lmp = lammps()
lmp.file("in.lj")
me = MPI.COMM_WORLD.Get_rank()
nprocs = MPI.COMM_WORLD.Get_size()
print("Proc %d out of %d procs has" % (me,nprocs),lmp)
MPI.Finalize()
```

You can run the script in parallel as:

```
mpirun -np 4 python3 test.py
```

and you should see the same output as if you had typed

```
mpirun -np 4 lmp_mpi -in in.lj
```

Note that without the mpi4py specific lines from `test.py`

```
from lammps import lammps
lmp = lammps()
lmp.file("in.lj")
```

running the script with `mpirun` on $P$ processors would lead to $P$ independent simulations to run parallel, each with a single processor. Therefore, if you use the mpi4py lines and you see multiple LAMMPS single processor outputs, mpi4py is not working correctly.

Also note that once you import the mpi4py module, mpi4py initializes MPI for you, and you can use MPI calls directly in your Python script, as described in the mpi4py documentation. The last line of your Python script should be `MPI.finalize()`, to ensure MPI is shut down correctly.

### 2.3.3 Running Python scripts

Note that any Python script (not just for LAMMPS) can be invoked in one of several ways:

```
python script.py
python -i script.py
./script.py
```

The last command requires that the first line of the script be something like this:

```
#!/usr/bin/python
```

or

```
#!/usr/bin/env python
```

where the path in the first case needs to point to where you have Python installed (the second option is workaround for when this may change), and that you have made the script file executable:

```
chmod +x script.py
```

Without the `-i` flag, Python will exit when the script finishes. With the `-i` flag, you will be left in the Python interpreter when the script finishes, so you can type subsequent commands. As mentioned above, you can only run Python interactively when running Python on a single processor, not in parallel.

---

### 2.3.4 Creating or deleting a LAMMPS object

With the Python interface the creation of a *LAMMPS* instance is included in the constructor for the *lammps* class. Internally it will call either *lammps_open()* or *lammps_open_no_mpi()* from the C library API to create the class instance.

All arguments are optional. The *name* argument allows loading a LAMMPS shared library that is named `liblammps_machine.so` instead of the default name of `liblammps.so`. In most cases the latter will be installed or used. The *ptr* argument is for use of the *lammps* module from inside a LAMMPS instance, e.g. with the *python* command, where a pointer to the already existing *LAMMPS* class instance can be passed to the Python class and used instead of creating a new instance. The *comm* argument may be used in combination with the mpi4py module to pass an MPI communicator to LAMMPS and thus it is possible to run the Python module like the library interface on a subset of the MPI ranks after splitting the communicator.

Here is a simple example using the LAMMPS Python interface:

```
from lammps import lammps

# NOTE: argv[0] is set by the lammps class constructor
args = ["-log", "none"]

# create LAMMPS instance
lmp = lammps(cmdargs=args)

# get and print numerical version code
print("LAMMPS Version: ", lmp.version())

# explicitly close and delete LAMMPS instance (optional)
lmp.close()
```

Same as with the *C library API*, this will use the `MPI_COMM_WORLD` communicator for the MPI library that LAMMPS was compiled with.

The *lmp.close()* call is optional since the LAMMPS class instance will also be deleted automatically during the *lammps* class destructor. Instead of *lmp.close()* it is also possible to call *lmp.finalize()*; this will destruct the LAMMPS instance, but also finalized and release the MPI and/or Kokkos environment if enabled and active.

Note that you can create multiple LAMMPS objects in your Python script, and coordinate and run multiple simulations, e.g.

```
from lammps import lammps
lmp1 = lammps()
lmp2 = lammps()
lmp1.file("in.file1")
lmp2.file("in.file2")
```

### 2.3.5 Executing commands

Once an instance of the *lammps* class is created, there are multiple ways to "feed" it commands. In a way that is not very different from running a LAMMPS input script, except that Python has many more facilities for structured programming than the LAMMPS input script syntax. Furthermore it is possible to "compute" what the next LAMMPS command should be.

Same as in the equivalent *C library functions*, commands can be read from a file, a single string, a list of strings and a block of commands in a single multi-line string. They are processed under the same boundary conditions as the C library counterparts. The example below demonstrates the use of `lammps.file()`, `lammps.command()`, `lammps.commands_list()`, and `lammps.commands_string()`:

```
from lammps import lammps
lmp = lammps()

# read commands from file 'in.melt'
lmp.file('in.melt')

# issue a single command
lmp.command('variable zpos index 1.0')

# create 10 groups with 10 atoms each
cmds = [f"group g{i} id {10*i+1}:{10*(i+1)}" for i in range(10)]
lmp.commands_list(cmds)

# run commands from a multi-line string
block = """
clear
region  box block 0 2 0 2 0 2
create_box 1 box
create_atoms 1 single 1.0 1.0 ${zpos}
"""
lmp.commands_string(block)
```

For convenience, the *lammps* class also provides a command wrapper `cmd` that turns any LAMMPS command into a regular function call.

For instance, the following LAMMPS command

```
region box block 0 10 0 5 -0.5 0.5
```

would normally be executed with the following Python code:

```
from lammps import lammps

lmp = lammps()
lmp.command("region box block 0 10 0 5 -0.5 0.5")
```

With the `cmd` wrapper, any LAMMPS command can be split up into arbitrary parts. These parts are then passed to a member function with the name of the *command*. For the *region* command that means the `region()` method can be called. The arguments of the command can be passed as one string, or individually.

```
from lammps import lammps

L = lammps()
```

(continues on next page)

(continued from previous page)

```
# pass command parameters as one string
L.cmd.region("box block 0 10 0 5 -0.5 0.5")

# OR pass them individually
L.cmd.region("box block", 0, 10, 0, 5, -0.5, 0.5)
```

In the latter example, all parameters except the first are Python floating-point literals. The member function takes the entire parameter list and transparently merges it to a single command string.

The benefit of this approach is avoiding redundant command calls and easier parameterization. With *command*, *commands_list*, and *commands_string* the parameterization needed to be done manually by creating formatted command strings.

```
lmp.command("region box block %f %f %f %f %f %f" % (xlo, xhi, ylo, yhi, zlo, zhi))
```

In contrast, methods of the *cmd* wrapper accept parameters directly and will convert them automatically to a final command string.

```
L.cmd.region("box block", xlo, xhi, ylo, yhi, zlo, zhi)
```

**Note**

When running in IPython you can use Tab-completion after `L.cmd.` to see all available LAMMPS commands.

Using these facilities, the previous example shown above can be rewritten as follows:

```
from lammps import lammps
L = lammps()

# read commands from file 'in.melt'
L.file('in.melt')

# issue a single command
L.cmd.variable('zpos', 'index', 1.0)

# create 10 groups with 10 atoms each
for i in range(10):
   L.cmd.group(f"g{i}", "id", f"{10*i+1}:{10*(i+1)}")

L.cmd.clear()
L.cmd.region("box block", 0, 2, 0, 2, 0, 2)
L.cmd.create_box(1, "box")
L.cmd.create_atoms(1, "single", 1.0, 1.0, "${zpos}")
```

### 2.3.6 System properties

Similar to what is described in *System properties*, the instances of `lammps` can be used to extract different kinds of information about the active LAMMPS instance and also to modify some of it.

In some cases the data returned is a direct reference to the original data inside LAMMPS cast to `ctypes` pointers. Where possible, the wrappers will determine the `ctypes` data type and cast pointers accordingly. If `numpy` is installed

arrays can also be extracted as numpy arrays, which will access the C arrays directly and have the correct dimensions to protect against invalid accesses.

**Warning**

When accessing per-atom data, please note that this data is the per-processor local data and indexed accordingly. These arrays can change sizes and order at every neighbor list rebuild and atom sort event as atoms are migrating between subdomains.

```
from lammps import lammps

lmp = lammps()
lmp.file("in.sysinit")

natoms = lmp.get_natoms()
print(f"running simulation with {natoms} atoms")

lmp.command("run 1000 post no");

for i in range(10):
   lmp.command("run 100 pre no post no")
   pe = lmp.get_thermo("pe")
   ke = lmp.get_thermo("ke")
   print(f"PE = {pe}\nKE = {ke}")

lmp.close()
```

**Methods**:

- *version()*: return the numerical version id, e.g. LAMMPS 2 Sep 2015 -> 20150902
- *get_thermo()*: return current value of a thermo keyword
- *last_thermo()*: return a dictionary of the last thermodynamic output
- *get_natoms()*: total # of atoms as int
- *reset_box()*: reset the simulation box size
- *extract_setting()*: return a global setting
- *extract_global()*: extract a global quantity
- *extract_box()*: extract box info
- *create_atoms()*: create N atoms with IDs, types, x, v, and image flags

**Properties**:

- *last_thermo_step*: the last timestep thermodynamic output was computed

### 2.3.7 Per-atom properties

Similar to what is described in *Per-atom properties*, the instances of *lammps* can be used to extract atom quantities and modify some of them.

In some cases the data returned is a direct reference to the original data inside LAMMPS cast to `ctypes` pointers. Where possible, the wrappers will determine the `ctypes` data type and cast pointers accordingly. If `numpy` is installed

arrays can also be extracted as numpy arrays, which will access the C arrays directly and have the correct dimensions to protect against invalid accesses.

**Warning**

When accessing per-atom data, please note that this data is the per-processor local data and indexed accordingly. These arrays can change sizes and order at every neighbor list rebuild and atom sort event as atoms are migrating between subdomains.

```
from lammps import lammps

lmp = lammps()
lmp.file("in.sysinit")


# Read/Write access via ctypes
nlocal = lmp.extract_global("nlocal")
x = lmp.extract_atom("x")

for i in range(nlocal):
   print("(x,y,z) = (", x[i][0], x[i][1], x[i][2], ")")

# Read/Write access via NumPy arrays
atom_id = L.numpy.extract_atom("id")
atom_type = L.numpy.extract_atom("type")
x = L.numpy.extract_atom("x")
v = L.numpy.extract_atom("v")
f = L.numpy.extract_atom("f")

# set position in 2D simulation
x[0] = (1.0, 0.0)

# set position in 3D simulation
x[0] = (1.0, 0.0, 1.)

lmp.close()
```

**Methods**:

- *extract_atom()*: extract a per-atom quantity

**Numpy Methods**:

- *numpy.extract_atom()*: extract a per-atom quantity as numpy array

### 2.3.8 Compute, fixes, variables

This section documents accessing or modifying data from objects like computes, fixes, or variables in LAMMPS using the *lammps* module.

For *lammps.extract_compute()* and *lammps.extract_fix()*, the global, per-atom, or local data calculated by the compute or fix can be accessed. What is returned depends on whether the compute or fix calculates a scalar or vector or array. For a scalar, a single double value is returned. If the compute or fix calculates a vector or array, a pointer to the internal LAMMPS data is returned, which you can use via normal Python subscripting.

The one exception is that for a fix that calculates a global vector or array, a single double value from the vector or array is returned, indexed by I (vector) or I and J (array). I,J are zero-based indices. See the *Howto output* page for a discussion of global, per-atom, and local data, and of scalar, vector, and array data types. See the doc pages for individual *computes* and *fixes* for a description of what they calculate and store.

For *lammps.extract_variable()*, an *equal-style or atom-style variable* is evaluated and its result returned.

For equal-style variables a single `c_double` value is returned and the group argument is ignored. For atom-style variables, a vector of `c_double` is returned, one value per atom, which you can use via normal Python subscripting. The values will be zero for atoms not in the specified group.

*lammps.numpy.extract_compute()*, *lammps.numpy.extract_fix()*, and *lammps.numpy.extract_variable()* are equivalent NumPy implementations that return NumPy arrays instead of `ctypes` pointers.

The *lammps.set_variable()* method sets an existing string-style variable to a new string value, so that subsequent LAMMPS commands can access the variable.

**Methods**:

- *lammps.extract_compute()*: extract value(s) from a compute
- *lammps.extract_fix()*: extract value(s) from a fix
- *lammps.extract_variable()*: extract value(s) from a variable
- *lammps.set_variable()*: set existing named string-style variable to value

**NumPy Methods**:

- *lammps.numpy.extract_compute()*: extract value(s) from a compute, return arrays as numpy arrays
- *lammps.numpy.extract_fix()*: extract value(s) from a fix, return arrays as numpy arrays
- *lammps.numpy.extract_variable()*: extract value(s) from a variable, return arrays as numpy arrays

### 2.3.9 Scatter/gather operations

```
data = lmp.gather_atoms(name,dtype,count)  # return per-atom property of all atoms␣
↪gathered into data, ordered by atom ID
                                           # name = "x", "q", "type", etc
data = lmp.gather_atoms_concat(name,dtype,count)  # ditto, but concatenated atom values␣
↪from each proc (unordered)
data = lmp.gather_atoms_subset(name,dtype,count,ndata,ids)  # ditto, but for subset of␣
↪Ndata atoms with IDs

lmp.scatter_atoms(name,dtype,count,data)   # scatter per-atom property to all atoms from␣
↪data, ordered by atom ID
                                           # name = "x", "q", "type", etc
                                           # count = # of per-atom values, 1 or 3, etc

lmp.scatter_atoms_subset(name,dtype,count,ndata,ids,data)  # ditto, but for subset of␣
↪Ndata atoms with IDs
```

The gather methods collect peratom info of the requested type (atom coords, atom types, forces, etc) from all processors, and returns the same vector of values to each calling processor. The scatter functions do the inverse. They distribute a vector of peratom values, passed by all calling processors, to individual atoms, which may be owned by different processors.

The *dtype* parameter is 0 for `int` values and 1 for `double` values. The *count* parameter is 1 for per-atom vectors like "type" or "q" and 3 for per-atom arrays like "x", "v", "f". Use *count* = 3 with name = "image" if you want the single integer storing the image flags unpacked into 3 components ("x", "y", and "z").

Note that the data returned by the gather methods, e.g. `gather_atoms("x")`, is different from the data structure returned by *extract_atom("x")* in four ways. (1) `gather_atoms()` returns a vector which you index as x[i]; `extract_atom()` returns an array which you index as x[i][j]. (2) `gather_atoms()` orders the atoms by atom ID while `extract_atom()` does not. (3) `gather_atoms()` returns a list of all atoms in the simulation; `extract_atoms()` returns just the atoms local to each processor. (4) Finally, the `gather_atoms()` data structure is a copy of the atom coords stored internally in LAMMPS, whereas `extract_atom()` returns an array that effectively points directly to the internal data. This means you can change values inside LAMMPS from Python by assigning a new values to the `extract_atom()` array. To do this with the `gather_atoms()` vector, you need to change values in the vector, then invoke the `scatter_atoms("x")` method.

For the scatter methods, the array of coordinates passed to must be a ctypes vector of ints or doubles, allocated and initialized something like this:

```
from ctypes import c_double
natoms = lmp.get_natoms()
n3 = 3*natoms
x = (n3*c_double)()
x[0] = x coord of atom with ID 1
x[1] = y coord of atom with ID 1
x[2] = z coord of atom with ID 1
x[3] = x coord of atom with ID 2
...
x[n3-1] = z coord of atom with ID natoms
lmp.scatter_atoms("x", 1, 3, x)
```

The coordinates can also be provided as arguments to the initializer of x:

```
from ctypes import c_double
natoms = 2
n3 = 3*natoms
# init in constructor
x = (n3*c_double)(0.0, 0.0, 0.0, 1.0, 1.0, 1.0)
lmp.scatter_atoms("x", 1, 3, x)
# or using a list
coords = [1.0, 2.0, 3.0, -3.0, -2.0, -1.0]
x = (c_double*len(coords))(*coords)
```

Alternatively, you can just change values in the vector returned by the gather methods, since they are also ctypes vectors.

### 2.3.10 Neighbor list access

Access to neighbor lists is handled through a couple of wrapper classes that allows one to treat it like either a python list or a NumPy array. The access procedure is similar to that of the C-library interface: use one of the "find" functions to look up the index of the neighbor list in the global table of neighbor lists and then get access to the neighbor list data. The code sample below demonstrates reading the neighbor list data using the NumPy access method.

```
from lammps import lammps
import numpy as np

lmp = lammps()
lmp.commands_string("""
```

(continues on next page)

(continued from previous page)

```
region box block -2 2 -2 2 -2 2
lattice fcc 1.0
create_box 1 box
create_atoms 1 box
mass 1 1.0
pair_style lj/cut 2.5
pair_coeff 1 1 1.0 1.0
run 0 post no""")

# look up the neighbor list
nlidx = lmp.find_pair_neighlist('lj/cut')
nl = lmp.numpy.get_neighlist(nlidx)
tags = lmp.extract_atom('id')
print("half neighbor list with {} entries".format(nl.size))
# print neighbor list contents
for i in range(0,nl.size):
    idx, nlist  = nl.get(i)
    print("\natom {} with ID {} has {} neighbors:".format(idx,tags[idx],nlist.size))
    if nlist.size > 0:
        for n in np.nditer(nlist):
            print("  atom {} with ID {}".format(n,tags[n]))
```

Another example for extracting a full neighbor list without evaluating a potential is shown below.

```
from lammps import lammps
import numpy as np

lmp = lammps()
lmp.commands_string("""
newton off
region box block -2 2 -2 2 -2 2
lattice fcc 1.0
create_box 1 box
create_atoms 1 box
mass 1 1.0
pair_style zero 1.0 full
pair_coeff * *
run 0 post no""")

# look up the neighbor list
nlidx = lmp.find_pair_neighlist('zero')
nl = lmp.numpy.get_neighlist(nlidx)
tags = lmp.extract_atom('id')
print("full neighbor list with {} entries".format(nl.size))
# print neighbor list contents
for i in range(0,nl.size):
    idx, nlist  = nl.get(i)
    print("\natom {} with ID {} has {} neighbors:".format(idx,tags[idx],nlist.size))
    if nlist.size > 0:
        for n in np.nditer(nlist):
            pass
            print("  atom {} with ID {}".format(n,tags[n]))
```

**Methods:**

- *lammps.get_neighlist()*: Get neighbor list for given index
- lammps.get_neighlist_size(): Get number of elements in neighbor list
- lammps.get_neighlist_element_neighbors(): Get element in neighbor list and its neighbors
- *lammps.find_pair_neighlist()*: Find neighbor list of pair style
- *lammps.find_fix_neighlist()*: Find neighbor list of fix style
- *lammps.find_compute_neighlist()*: Find neighbor list of compute style

**NumPy Methods:**

- *lammps.numpy.get_neighlist()*: Get neighbor list for given index, which uses NumPy arrays for its element neighbor arrays
- *lammps.numpy.get_neighlist_element_neighbors()*: Get element in neighbor list and its neighbors (as a numpy array)

### 2.3.11 Configuration information

The following methods can be used to query the LAMMPS library about compile time settings and included packages and styles.

Listing 1: Example for using configuration settings functions

```
from lammps import lammps

lmp = lammps()

try:
    lmp.file("in.missing")
except Exception as e:
    print("LAMMPS failed with error:", e)

# write compressed dump file depending on available of options

if lmp.has_style("dump", "atom/zstd"):
    lmp.command("dump d1 all atom/zstd 100 dump.zst")
elif lmp.has_style("dump", "atom/gz"):
    lmp.command("dump d1 all atom/gz 100 dump.gz")
elif lmp.has_gzip_support():
    lmp.command("dump d1 all atom 100 dump.gz")
else:
    lmp.command("dump d1 all atom 100 dump")
```

**Methods:**

- *lammps.has_mpi_support*
- *lammps.has_exceptions*
- *lammps.has_gzip_support*
- *lammps.has_png_support*
- *lammps.has_jpeg_support*

- *lammps.has_ffmpeg_support*
- *lammps.installed_packages*
- *lammps.get_accelerator_config*
- *lammps.has_style()*
- *lammps.available_styles()*

## 2.4 The `lammps` Python module

The LAMMPS Python interface is implemented as a module called *lammps* which is defined in the `lammps` package in the `python` folder of the LAMMPS source code distribution. After compilation of LAMMPS, the module can be installed into a Python system folder or a user folder with `make install-python`. Components of the module can then loaded into a Python session with the `import` command.

**Warning**

Alternative interfaces such as `PyLammps` and `IPyLammps` classes have been deprecated and will be removed in a future version of LAMMPS.

**Version check**

The *lammps* module stores the version number of the LAMMPS version it is installed from. When initializing the *lammps* class, this version is checked to be the same as the result from *lammps.version()*, the version of the LAMMPS shared library that the module interfaces to. If the they are not the same an AttributeError exception is raised since a mismatch of versions (e.g. due to incorrect use of the `LD_LIBRARY_PATH` or `PYTHONPATH` environment variables can lead to crashes or data corruption and otherwise incorrect behavior.

LAMMPS module global members:

`lammps.__version__`

Numerical representation of the LAMMPS version this module was taken from. Has the same format as the result of *lammps.version()*.

`lammps.get_version_number()`

Extract LAMMPS version string and convert to number

---

### 2.4.1 The `lammps` class API

The *lammps* class is the core of the LAMMPS Python interface. It is a wrapper around the *LAMMPS C library API* using the Python ctypes module and a shared library compiled from the LAMMPS sources code. The individual methods in this class try to closely follow the corresponding C functions. The handle argument that needs to be passed to the C functions is stored internally in the class and automatically added when calling the C library functions. Below is a detailed documentation of the API.

**class** `lammps.lammps`(*name=''*, *cmdargs=None*, *ptr=None*, *comm=None*)

Create an instance of the LAMMPS Python class.

This is a Python wrapper class that exposes the LAMMPS C-library interface to Python. It either requires that LAMMPS has been compiled as shared library which is then dynamically loaded via the ctypes Python module or that this module called from a Python function that is called from a Python interpreter embedded into a

LAMMPS executable, for example through the *python invoke* command. When the class is instantiated it calls the *lammps_open()* function of the LAMMPS C-library interface, which in turn will create an instance of the *LAMMPS* C++ class. The handle to this C++ class is stored internally and automatically passed to the calls to the C library interface.

**Parameters**

- **name** (*string*) – "machine" name of the shared LAMMPS library ("mpi" loads `liblammps_mpi.so`, "" loads `liblammps.so`)
- **cmdargs** (*list*) – list of command line arguments to be passed to the *lammps_open()* function. The executable name is automatically added.
- **ptr** (*pointer*) – pointer to a LAMMPS C++ class instance when called from an embedded Python interpreter. None means load symbols from shared library.
- **comm** (*MPI_Comm*) – MPI communicator (as provided by mpi4py). `None` means use `MPI_COMM_WORLD` implicitly.

**property numpy**

Return object to access numpy versions of API

It provides alternative implementations of API functions that return numpy arrays instead of ctypes pointers. If numpy is not installed, accessing this property will lead to an ImportError.

**Returns**
instance of numpy wrapper object

**Return type**
*numpy_wrapper*

**property cmd**

Return object that acts as LAMMPS command wrapper

It provides alternative to *lammps.command()* to call LAMMPS commands as if they were regular Python functions and enables auto-complete in interactive Python sessions.

```
from lammps import lammps

# melt example
L = lammps()
L.cmd.units("lj")
L.cmd.atom_style("atomic")
L.cmd.lattice("fcc", 0.8442)
L.cmd.region("box block", 0, 10, 0, 10, 0, 10)
L.cmd.create_box(1, "box")
L.cmd.create_atoms(1, "box")
L.cmd.mass(1, 1.0)
L.cmd.velocity("all create", 3.0, 87287, "loop geom")
L.cmd.pair_style("lj/cut", 2.5)
L.cmd.pair_coeff(1, 1, 1.0, 1.0, 2.5)
L.cmd.neighbor(0.3, "bin")
L.cmd.neigh_modify(every=20, delay=0, check=False)
L.cmd.fix(1, "all nve")
L.cmd.thermo(50)
L.cmd.run(250)
```

**Returns**
instance of command_wrapper object

**Return type**
command_wrapper

**property ipython**

Return object to access ipython extensions

Adds commands for visualization in IPython and Jupyter Notebooks.

**Returns**
instance of ipython wrapper object

**Return type**
*ipython.wrapper*

**close**()

Explicitly delete a LAMMPS instance through the C-library interface.

This is a wrapper around the *lammps_close()* function of the C-library interface.

**finalize**()

Shut down the MPI communication and Kokkos environment (if active) through the library interface by calling *lammps_mpi_finalize()*, *lammps_kokkos_finalize()*, *lammps_python_finalize()*, and *lammps_plugin_finalize()*

You cannot create or use any LAMMPS instances after this function is called unless LAMMPS was compiled without MPI and without Kokkos support.

**error**(*error_type*, *error_text*)

Forward error to the LAMMPS Error class.

Added in version 3Nov2022.

This is a wrapper around the *lammps_error()* function of the C-library interface.

**Parameters**

- **error_type** (*int*)
- **error_text** (*string*)

**version**()

Return a numerical representation of the LAMMPS version in use.

This is a wrapper around the *lammps_version()* function of the C-library interface.

**Returns**
version number

**Return type**
int

**get_os_info**()

Return a string with information about the OS and compiler runtime

This is a wrapper around the *lammps_get_os_info()* function of the C-library interface.

**Returns**
OS info string

**Return type**
string

**get_mpi_comm**()

Get the MPI communicator in use by the current LAMMPS instance

This is a wrapper around the *lammps_get_mpi_comm()* function of the C-library interface. It will return `None` if either the LAMMPS library was compiled without MPI support or the mpi4py Python module is not available.

**Returns**
MPI communicator

**Return type**
MPI_Comm

**expand**(*line*)

Expand a single LAMMPS string like an input line

This is a wrapper around the *lammps_expand()* function of the C-library interface.

**Parameters**
**cmd** (*string*) – a single lammps line

**Returns**
expanded string

**Return type**
string

**file**(*path*)

Read LAMMPS commands from a file.

This is a wrapper around the *lammps_file()* function of the C-library interface. It will open the file with the name/path *file* and process the LAMMPS commands line by line until the end. The function will return when the end of the file is reached.

**Parameters**
**path** (*string*) – Name of the file/path with LAMMPS commands

**command**(*cmd*)

Process a single LAMMPS input command from a string.

This is a wrapper around the *lammps_command()* function of the C-library interface.

**Parameters**
**cmd** (*string*) – a single lammps command

**commands_list**(*cmdlist*)

Process multiple LAMMPS input commands from a list of strings.

This is a wrapper around the *lammps_commands_list()* function of the C-library interface.

**Parameters**
**cmdlist** (*list of strings*) – a single lammps command

**commands_string**(*multicmd*)

Process a block of LAMMPS input commands from a string.

This is a wrapper around the *lammps_commands_string()* function of the C-library interface.

**Parameters**
**multicmd** (*string*) – text block of lammps commands

get_natoms()

Get the total number of atoms in the LAMMPS instance.

Will be precise up to 53-bit signed integer due to the underlying *lammps_get_natoms()* function returning a double.

**Returns**
number of atoms

**Return type**
int

extract_box()

Extract simulation box parameters

This is a wrapper around the *lammps_extract_box()* function of the C-library interface. Unlike in the C function, the result is returned as a list.

**Returns**
list of the extracted data: boxlo, boxhi, xy, yz, xz, periodicity, box_change

**Return type**
[ 3*double, 3*double, double, double, 3*int, int]

reset_box(*boxlo*, *boxhi*, *xy*, *yz*, *xz*)

Reset simulation box parameters

This is a wrapper around the *lammps_reset_box()* function of the C-library interface.

**Parameters**

- **boxlo** (*list of 3 floating point numbers*) – new lower box boundaries
- **boxhi** (*list of 3 floating point numbers*) – new upper box boundaries
- **xy** (*float*) – xy tilt factor
- **yz** (*float*) – yz tilt factor
- **xz** (*float*) – xz tilt factor

get_thermo(*name*)

Get current value of a thermo keyword

This is a wrapper around the *lammps_get_thermo()* function of the C-library interface.

**Parameters**
**name** (*string*) – name of thermo keyword

**Returns**
value of thermo keyword

**Return type**
double or None

property last_thermo_step

Get the last timestep where thermodynamic data was computed

**Returns**
the timestep or a negative number if there has not been any thermo output yet

**Return type**
int

`last_thermo`()

Get a dictionary of the last thermodynamic output

This is a wrapper around the *lammps_last_thermo()* function of the C-library interface. It collects the cached thermo data from the last timestep into a dictionary. The return value is None, if there has not been any thermo output yet.

**Returns**
a dictionary containing the last computed thermo output values

**Return type**
dict or None

`extract_setting`(*name*)

Query LAMMPS about global settings that can be expressed as an integer.

This is a wrapper around the *lammps_extract_setting()* function of the C-library interface. Its documentation includes a list of the supported keywords.

**Parameters**
**name** (*string*) – name of the setting

**Returns**
value of the setting

**Return type**
int

`extract_global_datatype`(*name*)

Retrieve global property datatype from LAMMPS

This is a wrapper around the *lammps_extract_global_datatype()* function of the C-library interface. Its documentation includes a list of the supported keywords. This function returns `None` if the keyword is not recognized. Otherwise it will return a positive integer value that corresponds to one of the *data type* constants defined in the *lammps* module.

**Parameters**
**name** (*string*) – name of the property

**Returns**
data type of global property, see *Data Types*

**Return type**
int

`extract_global`(*name*, *dtype=None*)

Query LAMMPS about global settings of different types.

This is a wrapper around the *lammps_extract_global()* function of the C-library interface. Since there are no pointers in Python, this method will - unlike the C function - return the value or a list of values. The *lammps_extract_global()* documentation includes a list of the supported keywords and their data types. Since Python needs to know the data type to be able to interpret the result, by default, this function will try to auto-detect the data type by asking the library. You can also force a specific data type. For that purpose the *lammps* module contains *data type* constants. This function returns `None` if either the keyword is not recognized, or an invalid data type constant is used.

**Parameters**

- **name** (*string*) – name of the property
- **dtype** (*int, optional*) – data type of the returned data (see *Data Types*)

**Returns**
value of the property or list of values or None

**Return type**
int, float, list, or NoneType

`extract_pair_dimension`(*name*)

Retrieve pair style property dimensionality from LAMMPS

Added in version 29Aug2024.

This is a wrapper around the *lammps_extract_pair_dimension()* function of the C-library interface. The list of supported keywords depends on the pair style. This function returns `None` if the keyword is not recognized.

**Parameters**
**name** (*string*) – name of the property

**Returns**
dimensionality of the extractable data (typically 0, 1, or 2)

**Return type**
int

`extract_pair`(*name*)

Extract pair style data from LAMMPS.

Added in version 29Aug2024.

This is a wrapper around the *lammps_extract_pair()* function of the C-library interface. Since there are no pointers in Python, this method will - unlike the C function - return the value or a list of values. Since Python needs to know the dimensionality to be able to interpret the result, this function will detect the dimensionality by asking the library. This function returns `None` if the keyword is not recognized.

**Parameters**
**name** (*string*) – name of the property

**Returns**
value of the property or list of values or None

**Return type**
float, list of float, list of list of floats, or NoneType

`map_atom`(*atomid*)

Map a global atom ID (aka tag) to the local atom index

This is a wrapper around the *lammps_map_atom()* function of the C-library interface.

**Parameters**
**atomid** (*int*) – atom ID

**Returns**
local index

**Return type**
int

`extract_atom_datatype`(*name*)

Retrieve per-atom property datatype from LAMMPS

This is a wrapper around the *lammps_extract_atom_datatype()* function of the C-library interface. Its documentation includes a list of the supported keywords. This function returns `None` if the keyword is

not recognized. Otherwise it will return an integer value that corresponds to one of the *data type* constants defined in the `lammps` module.

**Parameters**
**name** (*string*) – name of the property

**Returns**
data type of per-atom property (see *Data Types*)

**Return type**
int

**extract_atom_size**(*name*, *dtype*)

Retrieve per-atom property dimensions from LAMMPS

This is a wrapper around the `lammps_extract_atom_size()` function of the C-library interface. Its documentation includes a list of the supported keywords. This function returns `None` if the keyword is not recognized. Otherwise it will return an integer value with the size of the per-atom vector or array. If *name* corresponds to a per-atom array, the *dtype* keyword must be either LMP_SIZE_ROWS or LMP_SIZE_COLS from the *type* constants defined in the `lammps` module. The return value is the requested size. If *name* corresponds to a per-atom vector the *dtype* keyword is ignored.

**Parameters**

- **name** (*string*) – name of the property
- **type** (*int*) – either LMP_SIZE_ROWS or LMP_SIZE_COLS for arrays, otherwise ignored

**Returns**
data type of per-atom property (see *Data Types*)

**Return type**
int

**extract_atom**(*name*, *dtype=None*)

Retrieve per-atom properties from LAMMPS

This is a wrapper around the `lammps_extract_atom()` function of the C-library interface. Its documentation includes a list of the supported keywords and their data types. Since Python needs to know the data type to be able to interpret the result, by default, this function will try to auto-detect the data type by asking the library. You can also force a specific data type by setting `dtype` to one of the *data type* constants defined in the `lammps` module. This function returns `None` if either the keyword is not recognized, or an invalid data type constant is used.

**Note**

While the returned vectors or arrays of per-atom data are dimensioned for the range [0:nmax] - as is the underlying storage - the data is usually only valid for the range of [0:nlocal], unless the property of interest is also updated for ghost atoms. In some cases, this depends on a LAMMPS setting, see for example *comm_modify vel yes*. The actual size can be determined by calling py:meth:*extract_atom_size() <lammps.lammps.extract_atom_size>*.

**Parameters**

- **name** (*string*) – name of the property
- **dtype** (*int, optional*) – data type of the returned data (see *Data Types*)

**Returns**
requested data or None

**Return type**
ctypes.POINTER(ctypes.c_int32), ctypes.POINTER(ctypes.POINTER(ctypes.c_int32)), ctypes.POINTER(ctypes.c_int64), ctypes.POINTER(ctypes.POINTER(ctypes.c_int64)), ctypes.POINTER(ctypes.c_double), ctypes.POINTER(ctypes.POINTER(ctypes.c_double)), or NoneType

**extract_compute**(*cid*, *cstyle*, *ctype*)

Retrieve data from a LAMMPS compute

This is a wrapper around the *lammps_extract_compute()* function of the C-library interface. This function returns None if either the compute id is not recognized, or an invalid combination of *cstyle* and *ctype* constants is used. The names and functionality of the constants are the same as for the corresponding C-library function. For requests to return a scalar or a size, the value is returned, otherwise a pointer.

**Parameters**

- **cid** (*string*) – compute ID
- **cstyle** (*int*) – style of the data retrieve (global, atom, or local), see *Style Constants*
- **ctype** (*int*) – type or size of the returned data (scalar, vector, or array), see *Type Constants*

**Returns**
requested data as scalar, pointer to 1d or 2d double array, or None

**Return type**
c_double, ctypes.POINTER(c_double), ctypes.POINTER(ctypes.POINTER(c_double)), or NoneType

**extract_fix**(*fid*, *fstyle*, *ftype*, *nrow=0*, *ncol=0*)

Retrieve data from a LAMMPS fix

This is a wrapper around the *lammps_extract_fix()* function of the C-library interface. This function returns None if either the fix id is not recognized, or an invalid combination of *fstyle* and *ftype* constants is used. The names and functionality of the constants are the same as for the corresponding C-library function. For requests to return a scalar or a size, the value is returned, also when accessing global vectors or arrays, otherwise a pointer.

**Note**

When requesting global data, the fix data can only be accessed one item at a time without access to the whole vector or array. Thus this function will always return a scalar. To access vector or array elements the "nrow" and "ncol" arguments need to be set accordingly (they default to 0).

**Parameters**

- **fid** (*string*) – fix ID
- **fstyle** (*int*) – style of the data retrieve (global, atom, or local), see *Style Constants*
- **ftype** (*int*) – type or size of the returned data (scalar, vector, or array), see *Type Constants*
- **nrow** (*int*) – index of global vector element or row index of global array element
- **ncol** (*int*) – column index of global array element

**Returns**
requested data or None

**Return type**
c_double, ctypes.POINTER(c_double), ctypes.POINTER(ctypes.POINTER(c_double)), or NoneType

`extract_variable`(*name*, *group=None*, *vartype=None*)

Evaluate a LAMMPS variable and return its data

This function is a wrapper around the function *lammps_extract_variable()* of the C library interface, evaluates variable name and returns a copy of the computed data. The memory temporarily allocated by the C-interface is deleted after the data is copied to a Python variable or list. The variable must be either an equal-style (or equivalent) variable or an atom-style variable. The variable type can be provided as the `vartype` parameter, which may be one of several constants: `LMP_VAR_EQUAL`, `LMP_VAR_ATOM`, `LMP_VAR_VECTOR`, or `LMP_VAR_STRING`. If omitted or `None`, LAMMPS will determine its value for you based on a call to *lammps_extract_variable_datatype()* from the C library interface. The group parameter is only used for atom-style variables and defaults to the group "all".

**Parameters**

- **name** (*string*) – name of the variable to execute
- **group** (*string, only for atom-style variables*) – name of group for atom-style variable
- **vartype** (*int*) – type of variable, see *Variable Type Constants*

**Returns**
the requested data

**Return type**
c_double, (c_double), or NoneType

`clearstep_compute`()

Call 'lammps_clearstep_compute()' from Python

`addstep_compute`(*nextstep*)

Call 'lammps_addstep_compute()' from Python

`addstep_compute_all`(*nextstep*)

Call 'lammps_addstep_compute_all()' from Python

`flush_buffers`()

Flush output buffers

This is a wrapper around the *lammps_flush_buffers()* function of the C-library interface.

`set_variable`(*name*, *value*)

Set a new value for a LAMMPS string style variable

Deprecated since version 7Feb2024.

This is a wrapper around the *lammps_set_variable()* function of the C-library interface.

**Parameters**

- **name** (*string*) – name of the variable
- **value** (*any. will be converted to a string*) – new variable value

**Returns**
either 0 on success or -1 on failure

**Return type**
int

`set_string_variable`(*name*, *value*)

Set a new value for a LAMMPS string style variable

Added in version 7Feb2024.

This is a wrapper around the *lammps_set_string_variable()* function of the C-library interface.

**Parameters**

- **name** (*string*) – name of the variable
- **value** (*any. will be converted to a string*) – new variable value

**Returns**
either 0 on success or -1 on failure

**Return type**
int

`set_internal_variable`(*name*, *value*)

Set a new value for a LAMMPS internal style variable

Added in version 7Feb2024.

This is a wrapper around the *lammps_set_internal_variable()* function of the C-library interface.

**Parameters**

- **name** (*string*) – name of the variable
- **value** (*float or compatible. will be converted to float*) – new variable value

**Returns**
either 0 on success or -1 on failure

**Return type**
int

`eval`(*expr*)

Evaluate a LAMMPS immediate variable expression

Added in version 4Feb2025.

This function is a wrapper around the function *lammps_eval()* of the C library interface. It evaluates and expression like in immediate variables and returns the value.

**Parameters**
**expr** – immediate variable expression

**Returns**
the result of the evaluation

**Return type**
c_double

`gather_bonds`()

Retrieve global list of bonds

Added in version 28Jul2021.

This is a wrapper around the *lammps_gather_bonds()* function of the C-library interface.

This function returns a tuple with the number of bonds and a flat list of ctypes integer values with the bond type, bond atom1, bond atom2 for each bond.

**Returns**
a tuple with the number of bonds and a list of c_int or c_long

**Return type**
(int, 3*nbonds*c_tagint)

`gather_angles()`

Retrieve global list of angles

Added in version 8Feb2023.

This is a wrapper around the *lammps_gather_angles()* function of the C-library interface.

This function returns a tuple with the number of angles and a flat list of ctypes integer values with the angle type, angle atom1, angle atom2, angle atom3 for each angle.

**Returns**
a tuple with the number of angles and a list of c_int or c_long

**Return type**
(int, 4*nangles*c_tagint)

`gather_dihedrals()`

Retrieve global list of dihedrals

Added in version 8Feb2023.

This is a wrapper around the *lammps_gather_dihedrals()* function of the C-library interface.

This function returns a tuple with the number of dihedrals and a flat list of ctypes integer values with the dihedral type, dihedral atom1, dihedral atom2, dihedral atom3, dihedral atom4 for each dihedral.

**Returns**
a tuple with the number of dihedrals and a list of c_int or c_long

**Return type**
(int, 5*ndihedrals*c_tagint)

`gather_impropers()`

Retrieve global list of impropers

Added in version 8Feb2023.

This is a wrapper around the *lammps_gather_impropers()* function of the C-library interface.

This function returns a tuple with the number of impropers and a flat list of ctypes integer values with the improper type, improper atom1, improper atom2, improper atom3, improper atom4 for each improper.

**Returns**
a tuple with the number of impropers and a list of c_int or c_long

**Return type**
(int, 5*nimpropers*c_tagint)

`encode_image_flags`(*ix*, *iy*, *iz*)

convert 3 integers with image flags for x-, y-, and z-direction into a single integer like it is used internally in LAMMPS

This method is a wrapper around the *lammps_encode_image_flags()* function of library interface.

**Parameters**

- **ix** (*int*) – x-direction image flag
- **iy** (*int*) – y-direction image flag
- **iz** (*int*) – z-direction image flag

**Returns**
encoded image flags

**Return type**
lammps.c_imageint

**decode_image_flags**(*image*)

Convert encoded image flag integer into list of three regular integers.

This method is a wrapper around the *lammps_decode_image_flags()* function of library interface.

**Parameters**
**image** (*lammps.c_imageint*) – encoded image flags

**Returns**
list of three image flags in x-, y-, and z- direction

**Return type**
list of 3 int

**create_atoms**(*n*, *atomid*, *atype*, *x*, *v=None*, *image=None*, *shrinkexceed=False*)

Create N atoms from list of coordinates and properties

This function is a wrapper around the *lammps_create_atoms()* function of the C-library interface, and the behavior is similar except that the *v*, *image*, and *shrinkexceed* arguments are optional and default to *None*, *None*, and *False*, respectively. With *None* being equivalent to a `NULL` pointer in C.

The lists of coordinates, types, atom IDs, velocities, image flags can be provided in any format that may be converted into the required internal data types. Also the list may contain more than *N* entries, but not fewer. In the latter case, the function will return without attempting to create atoms. You may use the *encode_image_flags* method to properly combine three integers with image flags into a single integer.

**Parameters**

- **n** (*int*) – number of atoms for which data is provided
- **atomid** (*list of lammps.tagint*) – list of atom IDs with at least n elements or None
- **atype** (*list of int*) – list of atom types
- **x** (*list of float*) – list of coordinates for x-, y-, and z (flat list of 3n entries)
- **v** (*list of float*) – list of velocities for x-, y-, and z (flat list of 3n entries) or None (optional)
- **image** (*list of lammps.imageint*) – list of encoded image flags (optional)
- **shrinkexceed** (*bool*) – whether to expand shrink-wrap boundaries if atoms are outside the box (optional)

**Returns**
number of atoms created. 0 if insufficient or invalid data

**Return type**
int

**create_molecule**(*molid*, *jsonstr*)

Create new molecule template from string with JSON data

Added in version 22Jul2025.

This is a wrapper around the *lammps_create_molecule()* function of the library interface.

**Parameters**

- **molid** – molecule-id of the new molecule template
- **jsonstr** (*string*) – JSON data defining a new molecule template

**property has_mpi_support**

Report whether the LAMMPS shared library was compiled with a real MPI library or in serial.

This is a wrapper around the *lammps_config_has_mpi_support()* function of the library interface.

**Returns**
False when compiled with MPI STUBS, otherwise True

**Return type**
bool

**property has_omp_support**

Report whether the LAMMPS shared library was compiled with OpenMP enabled.

This is a wrapper around the *lammps_config_has_omp_support()* function of the library interface.

**Returns**
True when compiled with OpenMP enabled, otherwise False

**Return type**
bool

**property is_running**

Report whether being called from a function during a run or a minimization

Added in version 9Oct2020.

Various LAMMPS commands must not be called during an ongoing run or minimization. This property allows to check for that. This is a wrapper around the *lammps_is_running()* function of the library interface.

**Returns**
True when called during a run otherwise false

**Return type**
bool

**set_show_error**(*flag*)

Enable or disable direct printing of error messages in C++ code

Added in version 2Apr2025.

This function allows to enable or disable printing of error message directly in the C++ code. Disabling the printing avoids printing error messages twice when detecting and re-throwing them in Python code.

This is a wrapper around the *lammps_set_show_error()* function of the library interface.

**Parameters**
**flag** (*int*) – enable (1) or disable (0) printing of error message

**Returns**
previous setting of the flag

**Return type**
int

`force_timeout()`

Trigger an immediate timeout, i.e. a "soft stop" of a run.

Added in version 9Oct2020.

This function allows to cleanly stop an ongoing run or minimization at the next loop iteration. This is a wrapper around the *lammps_force_timeout()* function of the library interface.

`property has_exceptions`

Report whether the LAMMPS shared library was compiled with C++ exceptions handling enabled

This is a wrapper around the *lammps_config_has_exceptions()* function of the library interface.

**Returns**
state of C++ exception support

**Return type**
bool

`property has_gzip_support`

Report whether the LAMMPS shared library was compiled with support for reading and writing compressed files through `gzip`.

This is a wrapper around the *lammps_config_has_gzip_support()* function of the library interface.

**Returns**
state of gzip support

**Return type**
bool

`property has_png_support`

Report whether the LAMMPS shared library was compiled with support for writing images in PNG format.

This is a wrapper around the *lammps_config_has_png_support()* function of the library interface.

**Returns**
state of PNG support

**Return type**
bool

`property has_jpeg_support`

Report whether the LAMMPS shared library was compiled with support for writing images in JPEG format.

This is a wrapper around the *lammps_config_has_jpeg_support()* function of the library interface.

**Returns**
state of JPEG support

**Return type**
bool

`property has_ffmpeg_support`

State of support for writing movies with `ffmpeg` in the LAMMPS shared library

This is a wrapper around the *lammps_config_has_ffmpeg_support()* function of the library interface.

**Returns**
state of ffmpeg support

**Return type**
bool

**property has_curl_support**

Report whether the LAMMPS shared library was compiled with support for downloading files through libcurl.

This is a wrapper around the `lammps_config_has_curl_support()` function of the library interface.

**Returns**
state of CURL support

**Return type**
bool

**has_package**(*name*)

Report if the named package has been enabled in the LAMMPS shared library.

Added in version 3Nov2022.

This is a wrapper around the *lammps_config_has_package()* function of the library interface.

**Parameters**
**name** (*string*) – name of the package

**Returns**
state of package availability

**Return type**
bool

**property accelerator_config**

Return table with available accelerator configuration settings.

This is a wrapper around the *lammps_config_accelerator()* function of the library interface which loops over all known packages and categories and returns enabled features as a nested dictionary with all enabled settings as list of strings.

**Returns**
nested dictionary with all known enabled settings as list of strings

**Return type**
dictionary

**property has_gpu_device**

Availability of GPU package compatible device

This is a wrapper around the *lammps_has_gpu_device()* function of the C library interface.

**Returns**
True if a GPU package compatible device is present, otherwise False

**Return type**
bool

**get_gpu_device_info**()

Return a string with detailed information about any devices that are usable by the GPU package.

This is a wrapper around the *lammps_get_gpu_device_info()* function of the C-library interface.

**Returns**
GPU device info string

**Return type**
string

**property installed_packages**

List of the names of enabled packages in the LAMMPS shared library

This is a wrapper around the functions *lammps_config_package_count()* and *lammps_config_package_name()* of the library interface.

:return

**has_style**(*category*, *name*)

Returns whether a given style name is available in a given category

This is a wrapper around the function *lammps_has_style()* of the library interface.

**Parameters**

- **category** (*string*) – name of category
- **name** (*string*) – name of the style

**Returns**
true if style is available in given category

**Return type**
bool

**available_styles**(*category*)

Returns a list of styles available for a given category

This is a wrapper around the functions *lammps_style_count()* and *lammps_style_name()* of the library interface.

**Parameters**
**category** (*string*) – name of category

**Returns**
list of style names in given category

**Return type**
list

**has_id**(*category*, *name*)

Returns whether a given ID name is available in a given category

Added in version 9Oct2020.

This is a wrapper around the function *lammps_has_id()* of the library interface.

**Parameters**

- **category** (*string*) – name of category
- **name** (*string*) – name of the ID

**Returns**
true if ID is available in given category

**Return type**
bool

**available_ids**(*category*)

Returns a list of IDs available for a given category

Added in version 9Oct2020.

This is a wrapper around the functions *lammps_id_count()* and *lammps_id_name()* of the library interface.

Changed in version 22Jul2025.

This function has a different behavior for the "group" category: rather than only listing the available groups, it will return a full list with LMP_MAX_GROUP elements. This is because the list may have "holes" when groups are deleted. The returned list has either the name of the group or "None" for empty entries. This way, the value of 1 << idx is the groupbit that can be compared to the per-atom "mask" property to determine if an atom is member of a group.

**Parameters**
**category** (*string*) – name of category

**Returns**
list of id names in given category

**Return type**
list

**available_plugins**(*category=None*)

Returns a list of plugins available for a given category

Added in version 10Mar2021.

This is a wrapper around the functions lammps_plugin_count() and lammps_plugin_name() of the library interface.

**Returns**
list of style/name pairs of loaded plugins

**Return type**
list

**set_fix_external_callback**(*fix_id*, *callback*, *caller=None*)

Set the callback function for a fix external instance with a given fix ID.

Optionally also set a reference to the calling object.

This is a wrapper around the *lammps_set_fix_external_callback()* function of the C-library interface. However this is set up to call a Python function with the following arguments.

- object is the value of the "caller" argument
- ntimestep is the current timestep
- nlocal is the number of local atoms on the current MPI process
- tag is a 1d NumPy array of integers representing the atom IDs of the local atoms
- x is a 2d NumPy array of doubles of the coordinates of the local atoms
- f is a 2d NumPy array of doubles of the forces on the local atoms that will be added

Changed in version 28Jul2021.

**Parameters**

- **fix_id** – Fix-ID of a fix external instance
- **callback** – Python function that will be called from fix external

- **caller** – reference to some object passed to the callback function

**Type**
string

**Type**
function

**Type**
object, optional

**fix_external_get_force**(*fix_id*)

Get access to the array with per-atom forces of a fix external instance with a given fix ID.

Added in version 28Jul2021.

This is a wrapper around the *lammps_fix_external_get_force()* function of the C-library interface.

**Parameters**
**fix_id** – Fix-ID of a fix external instance

**Type**
string

**Returns**
requested data

**Return type**
ctypes.POINTER(ctypes.POINTER(ctypes.double))

**fix_external_set_energy_global**(*fix_id*, *eng*)

Set the global energy contribution for a fix external instance with the given ID.

Added in version 28Jul2021.

This is a wrapper around the *lammps_fix_external_set_energy_global()* function of the C-library interface.

**Parameters**

- **fix_id** – Fix-ID of a fix external instance
- **eng** – potential energy value to be added by fix external

**Type**
string

**Type**
float

**fix_external_set_virial_global**(*fix_id*, *virial*)

Set the global virial contribution for a fix external instance with the given ID.

Added in version 28Jul2021.

This is a wrapper around the *lammps_fix_external_set_virial_global()* function of the C-library interface.

**Parameters**

- **fix_id** – Fix-ID of a fix external instance
- **eng** – list of 6 floating point numbers with the virial to be added by fix external

**Type**
string

**Type**
float

**fix_external_set_energy_peratom**(*fix_id*, *eatom*)

Set the per-atom energy contribution for a fix external instance with the given ID.

Added in version 28Jul2021.

This is a wrapper around the *lammps_fix_external_set_energy_peratom()* function of the C-library interface.

**Parameters**

- **fix_id** – Fix-ID of a fix external instance
- **eatom** – list of potential energy values for local atoms to be added by fix external

**Type**
string

**Type**
float

**fix_external_set_virial_peratom**(*fix_id*, *vatom*)

Set the per-atom virial contribution for a fix external instance with the given ID.

Added in version 28Jul2021.

This is a wrapper around the *lammps_fix_external_set_virial_peratom()* function of the C-library interface.

**Parameters**

- **fix_id** – Fix-ID of a fix external instance
- **vatom** – list of natoms lists with 6 floating point numbers to be added by fix external

**Type**
string

**Type**
float

**fix_external_set_vector_length**(*fix_id*, *length*)

Set the vector length for a global vector stored with fix external for analysis

Added in version 28Jul2021.

This is a wrapper around the *lammps_fix_external_set_vector_length()* function of the C-library interface.

**Parameters**

- **fix_id** – Fix-ID of a fix external instance
- **length** – length of the global vector

**Type**
string

**Type**
int

**fix_external_set_vector**(*fix_id*, *idx*, *val*)

Store a global vector value for a fix external instance with the given ID.

Added in version 28Jul2021.

This is a wrapper around the *lammps_fix_external_set_vector()* function of the C-library interface.

**Parameters**

- **fix_id** – Fix-ID of a fix external instance
- **idx** – 1-based index of the value in the global vector
- **val** – value to be stored in the global vector

**Type**
string

**Type**
int

**Type**
float

**get_neighlist**(*idx*)

Returns an instance of *NeighList* which wraps access to the neighbor list with the given index

See *lammps.numpy.get_neighlist()* if you want to use NumPy arrays instead of `c_int` pointers.

**Parameters**
**idx** (*int*) – index of neighbor list

**Returns**
an instance of *NeighList* wrapping access to neighbor list data

**Return type**
*NeighList*

**get_neighlist_size**(*idx*)

Return the number of elements in neighbor list with the given index

**Parameters**
**idx** (*int*) – neighbor list index

**Returns**
number of elements in neighbor list with index idx

**Return type**
int

**get_neighlist_element_neighbors**(*idx*, *element*)

Return data of neighbor list entry

**Parameters**

- **element** (*int*) – neighbor list index
- **element** – neighbor list element index

**Returns**
tuple with atom local index, number of neighbors and array of neighbor local atom indices

**Return type**
(int, int, POINTER(c_int))

find_pair_neighlist(*style*, *exact=True*, *nsub=0*, *reqid=0*)

Find neighbor list index of pair style neighbor list

Search for a neighbor list requested by a pair style instance that matches "style". If exact is True, the pair style name must match exactly. If exact is False, the pair style name is matched against "style" as regular expression or sub-string. If the pair style is a hybrid pair style, the style is instead matched against the hybrid sub-styles. If the same pair style is used as sub-style multiple types, you must set nsub to a value n > 0 which indicates the nth instance of that sub-style to be used (same as for the pair_coeff command). The default value of 0 will fail to match in that case.

Once the pair style instance has been identified, it may have requested multiple neighbor lists. Those are uniquely identified by a request ID > 0 as set by the pair style. Otherwise the request ID is 0.

**Parameters**

- **style** (*string*) – name of pair style that should be searched for
- **exact** (*bool, optional*) – controls whether style should match exactly or only must be contained in pair style name, defaults to True
- **nsub** (*int, optional*) – match nsub-th hybrid sub-style, defaults to 0
- **reqid** (*int, optional*) – list request id, > 0 in case there are more than one, defaults to 0

**Returns**
neighbor list index if found, otherwise -1

**Return type**
int

find_fix_neighlist(*fixid*, *reqid=0*)

Find neighbor list index of fix neighbor list

The fix instance requesting the neighbor list is uniquely identified by the fix ID. In case the fix has requested multiple neighbor lists, those are uniquely identified by a request ID > 0 as set by the fix. Otherwise the request ID is 0 (the default).

**Parameters**

- **fixid** (*string*) – name of fix
- **reqid** (*int, optional*) – id of neighbor list request, in case there are more than one request, defaults to 0

**Returns**
neighbor list index if found, otherwise -1

**Return type**
int

find_compute_neighlist(*computeid*, *reqid=0*)

Find neighbor list index of compute neighbor list

The compute instance requesting the neighbor list is uniquely identified by the compute ID. In case the compute has requested multiple neighbor lists, those are uniquely identified by a request ID > 0 as set by the compute. Otherwise the request ID is 0 (the default).

**Parameters**

- **computeid** (*string*) – name of compute
- **reqid** (*int, optional*) – index of neighbor list request, in case there are more than one request, defaults to 0

**Returns**
neighbor list index if found, otherwise -1

**Return type**
int

**class** lammps.numpy_wrapper.**numpy_wrapper**(*lmp*)

lammps API NumPy Wrapper

This is a wrapper class that provides additional methods on top of an existing *lammps* instance. The methods transform raw ctypes pointers into NumPy arrays, which give direct access to the original data while protecting against out-of-bounds accesses.

There is no need to explicitly instantiate this class. Each instance of *lammps* has a `numpy` property that returns an instance.

**Parameters**
**lmp** (*lammps*) – instance of the *lammps* class

**extract_atom**(*name*, *dtype=None*, *nelem=None*, *dim=None*)

Retrieve per-atom properties from LAMMPS as NumPy arrays

This is a wrapper around the `lammps.extract_atom()` method. It behaves the same as the original method, but returns NumPy arrays instead of `ctypes` pointers.

**Note**

The returned vectors or arrays of per-atom data are dimensioned according to the return value of `lammps.extract_atom_size()`. Except for the "mass" property, the underlying storage will always be dimensioned for the range [0:nmax]. The actual usable data may be only in the range [0:nlocal] or [0:nlocal][0:dim]. Whether there is valid data in the range [nlocal:nlocal+nghost] or [nlocal:local+nghost][0:dim] depends on whether the property of interest is also updated for ghost atoms. Also the value of *dim* depends on the value of *name*. By using the optional *nelem* and *dim* parameters the dimensions of the returned NumPy array can be overridden. There is no check whether the number of elements chosen is valid.

**Parameters**

- **name** (*string*) – name of the property
- **dtype** (*int, optional*) – type of the returned data (see *Data Types*)
- **nelem** (*int, optional*) – number of elements in array
- **dim** (*int, optional*) – dimension of each element

**Returns**
requested data as NumPy array with direct access to C data or None

**Return type**
numpy.array or NoneType

**extract_compute**(*cid*, *cstyle*, *ctype*)

Retrieve data from a LAMMPS compute

This is a wrapper around the *lammps.extract_compute()* method. It behaves the same as the original method, but returns NumPy arrays instead of `ctypes` pointers.

**Parameters**

- **cid** (*string*) – compute ID
- **cstyle** (*int*) – style of the data retrieve (global, atom, or local), see *Style Constants*
- **ctype** (*int*) – type of the returned data (scalar, vector, or array), see *Type Constants*

**Returns**
requested data either as float, as NumPy array with direct access to C data, or None

**Return type**
float, numpy.array, or NoneType

**extract_fix**(*fid*, *fstyle*, *ftype*, *nrow=0*, *ncol=0*)

Retrieve data from a LAMMPS fix

This is a wrapper around the *lammps.extract_fix()* method. It behaves the same as the original method, but returns NumPy arrays instead of `ctypes` pointers.

> **Note**
>
> When requesting global data, the fix data can only be accessed one item at a time without access to the whole vector or array. Thus this function will always return a scalar. To access vector or array elements the "nrow" and "ncol" arguments need to be set accordingly (they default to 0).

**Parameters**

- **fid** (*string*) – fix ID
- **fstyle** (*int*) – style of the data retrieve (global, atom, or local), see *Style Constants*
- **ftype** (*int*) – type or size of the returned data (scalar, vector, or array), see *Type Constants*
- **nrow** (*int*) – index of global vector element or row index of global array element
- **ncol** (*int*) – column index of global array element

**Returns**
requested data

**Return type**
integer or double value, pointer to 1d or 2d double array or None

**extract_variable**(*name*, *group=None*, *vartype=0*)

Evaluate a LAMMPS variable and return its data

This function is a wrapper around the function *lammps.extract_variable()* method. It behaves the same as the original method, but returns NumPy arrays instead of `ctypes` pointers.

**Parameters**

- **name** (*string*) – name of the variable to execute
- **group** (*string*) – name of group for atom-style variable (ignored for equal-style variables)
- **vartype** (*int*) – type of variable, see *Variable Type Constants*

**Returns**
the requested data or None

**Return type**
c_double, numpy.array, or NoneType

`gather_bonds`()

Retrieve global list of bonds as a NumPy array

Added in version 28Jul2021.

This is a wrapper around *lammps.gather_bonds()*. It behaves the same as the original method, but returns a NumPy array instead of a `ctypes` list.

**Returns**
the requested data as a 2d-integer numpy array

**Return type**
numpy.array(nbonds,3)

`gather_angles`()

Retrieve global list of angles as a NumPy array

Added in version 8Feb2023.

This is a wrapper around *lammps.gather_angles()*. It behaves the same as the original method, but returns a NumPy array instead of a `ctypes` list.

**Returns**
the requested data as a 2d-integer numpy array

**Return type**
numpy.array(nangles,4)

`gather_dihedrals`()

Retrieve global list of dihedrals as a NumPy array

Added in version 8Feb2023.

This is a wrapper around *lammps.gather_dihedrals()*. It behaves the same as the original method, but returns a NumPy array instead of a `ctypes` list.

**Returns**
the requested data as a 2d-integer numpy array

**Return type**
numpy.array(ndihedrals,5)

`gather_impropers`()

Retrieve global list of impropers as a NumPy array

Added in version 8Feb2023.

This is a wrapper around *lammps.gather_impropers()*. It behaves the same as the original method, but returns a NumPy array instead of a `ctypes` list.

**Returns**
the requested data as a 2d-integer numpy array

**Return type**
numpy.array(nimpropers,5)

`fix_external_get_force`(*fix_id*)

Get access to the array with per-atom forces of a fix external instance with a given fix ID.

Changed in version 28Jul2021.

This function is a wrapper around the *lammps.fix_external_get_force()* method. It behaves the same as the original method, but returns a NumPy array instead of a `ctypes` pointer.

**Parameters**
**fix_id** – Fix-ID of a fix external instance

**Type**
string

**Returns**
requested data

**Return type**
numpy.array

**fix_external_set_energy_peratom**(*fix_id*, *eatom*)

Set the per-atom energy contribution for a fix external instance with the given ID.

Added in version 28Jul2021.

This function is an alternative to *lammps.fix_external_set_energy_peratom()* method. It behaves the same as the original method, but accepts a NumPy array instead of a list as argument.

**Parameters**

- **fix_id** – Fix-ID of a fix external instance
- **eatom** – per-atom potential energy

**Type**
string

**Type**
numpy.array

**fix_external_set_virial_peratom**(*fix_id*, *vatom*)

Set the per-atom virial contribution for a fix external instance with the given ID.

Added in version 28Jul2021.

This function is an alternative to *lammps.fix_external_set_virial_peratom()* method. It behaves the same as the original method, but accepts a NumPy array instead of a list as argument.

**Parameters**

- **fix_id** – Fix-ID of a fix external instance
- **eatom** – per-atom potential energy

**Type**
string

**Type**
numpy.array

**get_neighlist**(*idx*)

Returns an instance of *NumPyNeighList* which wraps access to the neighbor list with the given index

**Parameters**
**idx** (*int*) – index of neighbor list

**Returns**
an instance of *NumPyNeighList* wrapping access to neighbor list data

**Return type**
*NumPyNeighList*

**get_neighlist_element_neighbors**(*idx*, *element*)

Return data of neighbor list entry

This function is a wrapper around the function *lammps.get_neighlist_element_neighbors()* method. It behaves the same as the original method, but returns a NumPy array containing the neighbors instead of a `ctypes` pointer.

**Parameters**

- **element** (*int*) – neighbor list index
- **element** – neighbor list element index

**Returns**
tuple with atom local index and numpy array of neighbor local atom indices

**Return type**
(int, numpy.array)

**iarray**(*c_int_type*, *raw_ptr*, *nelem*, *dim=1*)

Convert ctypes pointer to an array of integers into a corresponding numpy array

This will cast the raw pointer into a 1-d or 2-d numpy array and set its shape

**Parameters**

- **c_int_type** – type of integer (c_int32 or c_int64)
- **raw_ptr** – ctypes pointer to the array data
- **nelem** – length of the leading dimension
- **dim** – length of the second dimension

**Type**
ctypes type

**Type**
POINTER(c_int_type)

**Type**
integer

**Type**
integer, optional

**Returns**
ctypes array converted to numpy style array with proper shape

**Return type**
numpy.array

**darray**(*raw_ptr*, *nelem*, *dim=1*)

Convert ctypes pointer to an array of doubles into a corresponding numpy array

This will cast the raw pointer into a 1-d or 2-d numpy array and set its shape

**Parameters**

- **raw_ptr** – ctypes pointer to the array data
- **nelem** – length of the leading dimension
- **dim** – length of the second dimension

**Type**
POINTER(c_double)

**Type**
integer

**Type**
integer, optional

**Returns**
ctypes array converted to numpy style array with proper shape

**Return type**
numpy.array

**class** lammps.ipython.**wrapper**(*lmp*)

lammps API IPython Wrapper

This is a wrapper class that provides additional methods on top of an existing *lammps* instance. It provides additional methods that allow create and/or embed visualizations created by native LAMMPS commands.

There is no need to explicitly instantiate this class. Each instance of *lammps* has a `ipython` property that returns an instance.

**Parameters**
**lmp** (*lammps*) – instance of the *lammps* class

**image**(*filename='snapshot.png'*, *group='all'*, *color='type'*, *diameter='type'*, *size=None*, *view=None*, *center=None*, *up=None*, *zoom=1.0*, *background_color='white'*)

Generate image using write_dump command and display it

See *dump image* for more information.

**Parameters**

- **filename** (*string*) – Name of the image file that should be generated. The extension determines whether it is PNG or JPEG
- **group** (*string*) – the group of atoms write_image should use
- **color** (*string*) – name of property used to determine color
- **diameter** (*string*) – name of property used to determine atom diameter
- **size** (*tuple (width, height)*) – dimensions of image
- **view** (*tuple (theta, phi)*) – view parameters
- **center** (*tuple (flag, center_x, center_y, center_z)*) – center parameters
- **up** (*tuple (up_x, up_y, up_z)*) – vector pointing to up direction
- **zoom** (*float*) – zoom factor
- **background_color** (*string*) – background color of scene

**Returns**
Image instance used to display image in notebook

**Return type**
IPython.core.display.Image

**video**(*filename*)

Load video from file

Can be used to visualize videos from *dump movie*.

**Parameters**
**filename** (*string*) – Path to video file

**Returns**
HTML Video Tag used by notebook to embed a video

**Return type**
IPython.display.HTML

---

### 2.4.2 Additional components of the lammps module

The *lammps* module additionally contains several constants and the *NeighList* class:

#### Data Types

**LAMMPS_INT, LAMMPS_INT_2D, LAMMPS_DOUBLE, LAMMPS_DOUBLE_2D, LAMMPS_INT64, LAMMPS_INT64_2D, LAMMPS_STRING**

Constants in the *lammps* module to indicate how to cast data when the C library function returns a void pointer. Used in *lammps.extract_global()* and *lammps.extract_atom()*. See _LMP_DATATYPE_CONST for the equivalent constants in the C library interface.

#### Style Constants

**LMP_STYLE_GLOBAL, LMP_STYLE_ATOM, LMP_STYLE_LOCAL**

Constants in the *lammps* module to select what style of data to request from computes or fixes. See *_LMP_STYLE_CONST* for the equivalent constants in the C library interface. Used in *lammps.extract_compute()*, *lammps.extract_fix()*, and their NumPy variants *lammps.numpy.extract_compute()* and *lammps.numpy.extract_fix()*.

#### Type Constants

**LMP_TYPE_SCALAR, LMP_TYPE_VECTOR, LMP_TYPE_ARRAY, LMP_SIZE_VECTOR, LMP_SIZE_ROWS, LMP_SIZE_COLS**

Constants in the *lammps* module to select what type of data to request from computes or fixes. See *_LMP_TYPE_CONST* for the equivalent constants in the C library interface. Used in *lammps.extract_compute()*, *lammps.extract_fix()*, and their NumPy variants *lammps.numpy.extract_compute()* and *lammps.numpy.extract_fix()*.

#### Variable Type Constants

**LMP_VAR_EQUAL, LMP_VAR_ATOM**

Constants in the *lammps* module to select what type of variable to query when calling *lammps.extract_variable()*. See also: *variable command*.

#### Classes representing internal objects

**class** lammps.**NeighList**(*lmp*, *idx*)

This is a wrapper class that exposes the contents of a neighbor list.

It can be used like a regular Python list. Each element is a tuple of:

- the atom local index
- its number of neighbors
- and a pointer to an c_int array containing local atom indices of its neighbors

Internally it uses the lower-level LAMMPS C-library interface.

**Parameters**

- **lmp** (lammps) – reference to instance of *lammps*
- **idx** (*int*) – neighbor list index

**property size**

**Returns**
number of elements in neighbor list

**get**(*element*)

Access a specific neighbor list entry. "element" must be a number from 0 to the size-1 of the list

**Returns**
tuple with atom local index, number of neighbors and ctypes pointer to neighbor's local atom indices

**Return type**
(int, int, ctypes.POINTER(c_int))

**find**(*iatom*)

Find the neighbor list for a specific (local) atom iatom. If there is no list for iatom, (-1, None) is returned.

**Returns**
tuple with number of neighbors and ctypes pointer to neighbor's local atom indices

**Return type**
(int, ctypes.POINTER(c_int))

**class** lammps.numpy_wrapper.**NumPyNeighList**(*lmp*, *idx*)

This is a wrapper class that exposes the contents of a neighbor list.

It can be used like a regular Python list. Each element is a tuple of:

- the atom local index
- a NumPy array containing the local atom indices of its neighbors

Internally it uses the lower-level LAMMPS C-library interface.

**Parameters**

- **lmp** (lammps) – reference to instance of *lammps*
- **idx** (*int*) – neighbor list index

**get**(*element*)

Access a specific neighbor list entry. "element" must be a number from 0 to the size-1 of the list

**Returns**
tuple with atom local index, numpy array of neighbor local atom indices

**Return type**
(int, numpy.array)

**find**(*iatom*)

Find the neighbor list for a specific (local) atom iatom. If there is no list for iatom, None is returned.

**Returns**
numpy array of neighbor local atom indices

**Return type**
numpy.array or None

## 2.5 Extending the Python interface

As noted previously, most of the `lammps` Python class methods correspond one-to-one with the functions in the LAMMPS library interface in `src/library.cpp` and `library.h`. This means you can extend the Python wrapper by following these steps:

- Add a new interface function to `src/library.cpp` and `src/library.h`.
- Rebuild LAMMPS as a shared library.
- Add a wrapper method to `python/lammps/core.py` for this interface function.
- Define the corresponding `argtypes` list and `restype` in the `lammps.__init__()` function.
- Re-install the shared library and the python module, if needed
- You should now be able to invoke the new interface function from a Python script.

## 2.6 Calling Python from LAMMPS

LAMMPS has several commands which can be used to invoke Python code directly from an input script:

- *python*
- *python-style variables*
- *equal-style and atom-style variables with formulas containing Python function wrappers*
- *fix python/invoke*
- *pair_style python*

The *python* command can be used to define and execute a Python function that you write the code for. The Python function can also be assigned to a LAMMPS python-style variable via the *variable* command. Each time the variable is evaluated, either in the LAMMPS input script itself, or by another LAMMPS command that uses the variable, this will trigger the Python function to be invoked.

The Python function can also be referenced in the formula used to define an *equal-style or atom-style variable*, using the syntax for a *Python function wrapper*. This make it easy to pass LAMMPS-related arguments to the Python function, as well as to invoke it whenever the equal- or atom-style variable is evaluated. For an atom-style variable it means the Python function can be invoked once per atom, using per-atom properties as arguments to the function.

The Python code for the function can be included directly in the input script or in an auxiliary file. The function can have arguments which are mapped to LAMMPS variables (also defined in the input script) and it can return a value to a LAMMPS variable. This is thus a mechanism for your input script to pass information to a piece of Python code, ask Python to execute the code, and return information to your input script.

Note that a Python function can be arbitrarily complex. It can import other Python modules, instantiate Python classes, call other Python functions, etc. The Python code that you provide can contain more code than the single function. It can contain other functions or Python classes, as well as global variables or other mechanisms for storing state between calls from LAMMPS to the function.

The Python function you provide can consist of "pure" Python code that only performs operations provided by standard Python. However, the Python function can also "call back" to LAMMPS through its Python-wrapped library interface, in the manner described in the *Python run* doc page. This means it can issue LAMMPS input script commands or query and set internal LAMMPS state. As an example, this can be useful in an input script to create a more complex loop with branching logic, than can be created using the simple looping and branching logic enabled by the *next* and *if* commands.

See the *python* page and the *variable* doc page for its python-style variables for more info, including examples of Python code you can write for both pure Python operations and callbacks to LAMMPS.

## 2.7 Output Readers

The Python package contains the `lammps.formats` module, which provides classes to post-process some of the output files generated by LAMMPS.

Output formats for LAMMPS python module Written by Richard Berger <richard.berger@outlook.com> and Axel Kohlmeyer <akohlmey@gmail.com>

**class** lammps.formats.**LogFile**(*filename*)

Reads LAMMPS log files and extracts the thermo information

It supports the line, multi, and yaml thermo output styles.

**Parameters**
**filename** (*str*) – path to log file

**Variables**

- **runs** – List of LAMMPS runs in log file. Each run is a dictionary with thermo fields as keys, storing the values over time
- **errors** – List of error lines in log file

**class** lammps.formats.**AvgChunkFile**(*filename*)

Reads files generated by fix ave/chunk

**Parameters**
**filename** (*str*) – path to ave/chunk file

**Variables**

- **timesteps** – List of timesteps stored in file
- **total_count** – total count over time
- **chunks** – List of chunks. Each chunk is a dictionary containing its ID, the coordinates, and the averaged quantities

## 2.8 Example Python scripts

The `python/examples` directory has Python scripts which show how Python can run LAMMPS, grab data, change it, and put it back into LAMMPS.

These are the Python scripts included as demos in the `python/examples` directory of the LAMMPS distribution, to illustrate the kinds of things that are possible when Python wraps LAMMPS. If you create your own scripts, send them to us and we can include them in the LAMMPS distribution.

| | |
|---|---|
| `trivial.py` | read/run a LAMMPS input script through Python |
| `demo.py` | invoke various LAMMPS library interface routines |
| `simple.py` | run in parallel, similar to `examples/COUPLE/simple/simple.cpp` |
| `split.py` | same as `simple.py` but running in parallel on a subset of procs |
| `gui.py` | GUI go/stop/temperature-slider to control LAMMPS |
| `plot.py` | real-time temperature plot with GnuPlot via Pizza.py |
| `viz_TOOL.py` | real-time viz via some viz package |
| `vizplotgui_TOOL.py` | combination of `viz_TOOL.py` and `plot.py` and `gui.py` |

For the `viz_TOOL.py` and `vizplotgui_TOOL.py` commands, replace `TOOL` with `gl` or `atomeye` or `pymol` or `vmd`, depending on what visualization package you have installed.

Note that for GL, you need to be able to run the Pizza.py GL tool, which is included in the pizza subdirectory. See the Pizza.py doc pages for more info:

- https://lammps.github.io/pizza/

Note that for AtomEye, you need version 3, and there is a line in the scripts that specifies the path and name of the executable. See the AtomEye web pages for more details:

- http://li.mit.edu/Archive/Graphics/A/
- http://li.mit.edu/Archive/Graphics/A3/A3.html

The latter link is to AtomEye 3 which has the scripting capability needed by these Python scripts.

Note that for PyMol, you need to have built and installed the open-source version of PyMol in your Python, so that you can import it from a Python script. See the PyMol web pages for more details:

- https://www.pymol.org
- https://github.com/schrodinger/pymol-open-source

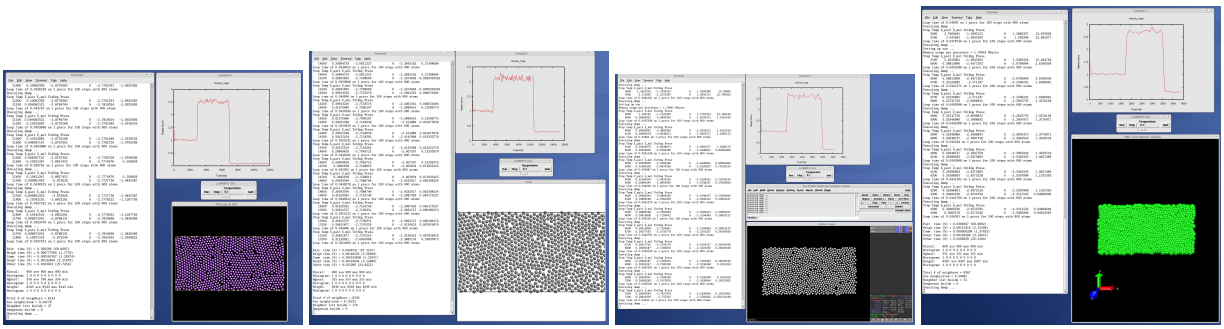
The latter link is to the open-source version.

Note that for VMD, you need a fairly current version (1.8.7 works for me) and there are some lines in the `pizza/vmd.py` script for 4 PIZZA variables that have to match the VMD installation on your system.

---

See the `python/README` file for instructions on how to run them and the source code for individual scripts for comments about what they do.

Here are screenshots of the `vizplotgui_tool.py` script in action for different visualization package options:



## 2.9 Using LAMMPS in IPython notebooks and Jupyter

If the LAMMPS Python package is installed for the same Python interpreter as IPython, you can use LAMMPS directly inside of an IPython notebook inside of Jupyter. Jupyter is a powerful integrated development environment (IDE) for many dynamic languages like Python, Julia and others, which operates inside of any web browser. Besides auto-completion and syntax highlighting it allows you to create formatted documents using Markup, mathematical formulas, graphics and animations intermixed with executable Python code. It is a great format for tutorials and showcasing your latest research.

The easiest way to install it is via `pip` from https://pypi.org/:

```
pip install --user jupyter
```

To launch an instance of Jupyter simply run the following command inside your Python environment:

```
jupyter notebook
```

### 2.9.1 Interactive Python Examples

Examples of IPython notebooks can be found in the `python/examples/ipython` subdirectory. They require LAMMPS to be compiled as shared library with PYTHON, PNG, JPEG and FFMPEG support.

To open these notebooks launch `jupyter notebook index.ipynb` inside this directory. The opened file provides an overview of the available examples.

- Example 1: Using LAMMPS with Python (`simple.ipynb`)
- Example 2: Analyzing LAMMPS thermodynamic data (`thermo.ipynb`)
- Example 3: Working with Per-Atom Data (`atoms.ipynb`)
- Example 4: Working with LAMMPS variables (`variables.ipynb`)
- Example 5: Validating a dihedral potential (`dihedrals/dihedral.ipynb`)
- Example 6: Running a Monte Carlo relaxation (`montecarlo/mc.ipynb`)

**Note**

Typically clicking a link in Jupyter will open a new tab, which might be blocked by your pop-up blocker.

## 2.10 Handling LAMMPS errors

LAMMPS and the LAMMPS library are compiled with *C++ exception support* to provide a better error handling experience. LAMMPS errors trigger throwing a C++ exception. These exceptions allow capturing errors on the C++ side and rethrowing them on the Python side. This way LAMMPS errors can be handled through the Python exception handling mechanism.

```
from lammps import lammps, MPIAbortException

lmp = lammps()

try:
   # LAMMPS will normally terminate itself and the running process if an error
   # occurs. This would kill the Python interpreter.  The library wrapper will
   # detect that an error has occurred and throw a Python exception

   lmp.command('unknown')
except MPIAbortException as ae:
   # Single MPI process got killed. This would normally be handled by an MPI abort
   pass
except Exception as e:
   # All (MPI) processes have reached this error
   pass
```

**Warning**

Capturing a LAMMPS exception in Python can still mean that the current LAMMPS process is in an illegal state and must be terminated. It is advised to save your data and terminate the Python instance as quickly as possible when running in parallel with MPI.

## 2.11 Troubleshooting

### 2.11.1 Testing if Python can launch LAMMPS

To test if LAMMPS is callable from Python, launch Python interactively and type:

```
>>> from lammps import lammps
>>> lmp = lammps()
```

If you get no errors, you're ready to use LAMMPS from Python. If the second command fails, the most common error to see is

```
OSError: Could not load LAMMPS dynamic library
```

which means Python was unable to load the LAMMPS shared library. This typically occurs if the system can't find the LAMMPS shared library or one of the auxiliary shared libraries it depends on, or if something about the library is incompatible with your Python. The error message should give you an indication of what went wrong.

If your shared library uses a suffix, such as `liblammps_mpi.so`, change the constructor call as follows (see *Creating or deleting a LAMMPS object* for more details):

```
>>> lmp = lammps(name='mpi')
```

You can also test the load directly in Python as follows, without first importing from the `lammps` module:

```
>>> from ctypes import CDLL
>>> CDLL("liblammps.so")
```

If an error occurs, carefully go through the steps in *Installing the LAMMPS Python Module and Shared Library* and on the *Build_basics* page about building a shared library.

If you are not familiar with Python, it is a powerful scripting and programming language which can do almost everything that compiled languages like C, C++, or Fortran can do in fewer lines of code. It also comes with a large collection of add-on modules for many purposes (either bundled or easily installed from Python code repositories). The major drawback is slower execution speed of the script code compared to compiled programming languages. But when the script code is interfaced to optimized compiled code, performance can be on par with a standalone executable, for as long as the scripting is restricted to high-level operations. Thus Python is also convenient to use as a "glue" language to "drive" a program through its library interface, or to hook multiple pieces of software together, such as a simulation code and a visualization tool, or to run a coupled multi-scale or multi-physics model.

See the *Coupling LAMMPS to other codes* page for more ideas about coupling LAMMPS to other codes. See the *LAMMPS Library Interfaces* page for a description of the LAMMPS library interfaces. That interface is exposed to Python either when calling LAMMPS from Python or when calling Python from a LAMMPS input script and then calling back to LAMMPS from Python code. The C-library interface is designed to be easy to add functionality to, thus the Python interface to LAMMPS is easy to extend as well.

If you create interesting Python scripts that run LAMMPS or interesting Python functions that can be called from a LAMMPS input script, that you think would be generally useful, please post them as a pull request to our GitHub site, and they can be added to the LAMMPS distribution or web page.

CHAPTER

# THREE

# MODIFYING & EXTENDING LAMMPS

LAMMPS has a modular design, so that it is easy to modify or extend with new functionality. In fact, about 95% of its source code is optional. The following pages give basic instructions on adding new features to LAMMPS. More in-depth explanations and documentation of individual functions and classes are given in *Information for Developers*.

If you add a new feature to LAMMPS and think it will be of general interest to other users, we encourage you to submit it for inclusion in LAMMPS. This process is explained in the following three pages:

- *how to prepare and submit your code*
- *requirements for submissions*
- *style guidelines*

A summary description of various types of styles in LAMMPS follows. A discussion of implementing specific styles from scratch is given in *writing new styles*.

## 3.1 Overview

The best way to add a new feature to LAMMPS is to find a similar feature and look at the corresponding source and header files to figure out what it does. You will need some knowledge of C++ to understand the high-level structure of LAMMPS and its class organization. Functions (class methods) that do actual computations are mostly written in C-style code and operate on simple C-style data structures (vectors and arrays). A high-level overview of the programming style choices in LAMMPS is *given elsewhere*.

Most of the new features described on the *Modify* doc page require you to write a new C++ derived class (excluding exceptions described below, this can often be done by making small edits to existing files). Creating a new class requires 2 files, a source code file (*.cpp) and a header file (*.h). The derived class must provide certain methods to work as a new option. Depending on how different your new feature is compared to existing features, you can either derive from the base class itself, or from a derived class that already exists. Enabling LAMMPS to invoke the new class is as simple as putting the two source files in the src directory and re-building LAMMPS.

The advantage of C++ and its object-orientation is that all the code and variables needed to define the new feature are in the 2 files you write. Thus, it should not make the rest of LAMMPS more complex or cause bugs through unwanted side effects.

Here is a concrete example. Suppose you write 2 files `pair_foo.cpp` and `pair_foo.h` that define a new class `PairFoo` which computes pairwise potentials described in the classic 1997 *paper* by Foo, *et al*. If you wish to invoke those potentials in a LAMMPS input script with a command like:

```
pair_style foo 0.1 3.5
```

then your `pair_foo.h` file should be structured as follows:

```
#ifdef PAIR_CLASS
// clang-format off
PairStyle(foo,PairFoo);
#else
// clang-format on
...
(class definition for PairFoo)
...
#endif
```

where "foo" is the style keyword in the pair_style command, and `PairFoo` is the class name defined in your `pair_foo.cpp` and `pair_foo.h` files.

When you re-build LAMMPS, your new pairwise potential becomes part of the executable and can be invoked with a pair_style command like the example above. Arguments like 0.1 and 3.5 can be defined and processed by your new class.

As illustrated by this example, many features referred to in the LAMMPS documentation are called a "style" of a particular command.

The *Modify page* lists all the common styles in LAMMPS, and discusses the header file for the base class that these styles derive from. Public variables in that file can be used and set by the derived classes, and may also be used by the base class. Sometimes they are also accessed by the rest of LAMMPS. Pure functions, which means functions declared as virtual in the base class header file and which are also set to 0, are functions you **must** implement in your new derived class to give it the functionality LAMMPS expects. Virtual functions that are not set to 0 are functions you may override or not. Those are usually defined with an empty function body.

Additionally, new output options can be added directly to the thermo.cpp, dump_custom.cpp, and variable.cpp files. These are also listed on the *Modify page*.

Here are additional guidelines for modifying LAMMPS and adding new functionality:

- Think about whether what you want to do would be better as a pre- or post-processing step. Many computations are more easily and more quickly done that way.
- Do not try to do anything within the timestepping of a run that is not parallel. For example, do not accumulate a bunch of data on a single processor and analyze it. That would run the risk of seriously degrading the parallel efficiency.
- If your new feature reads arguments or writes output, make sure you follow the unit conventions discussed by the *units* command.

---

**(Foo)** Foo, Morefoo, and Maxfoo, J of Classic Potentials, 75, 345 (1997).

## 3.2 Submitting new features for inclusion in LAMMPS

We encourage LAMMPS users to submit new features they write for LAMMPS to be included in the LAMMPS distribution and thus become easily accessible to all LAMMPS users. The LAMMPS source code is managed with git and public development is hosted on GitHub. You can monitor the repository to be notified of releases, follow the ongoing development, and comment on topics of interest to you.

This section contains general information regarding the preparation and submission of new features to LAMMPS. If you are new to development in LAMMPS, we recommend you read one of the tutorials on developing a new *pair style* or *fix style* which provide a friendly introduction to what LAMMPS development entails and common vocabulary used on this section.

### 3.2.1 Communication with the LAMMPS developers

For any larger modifications or programming project, you are encouraged to contact the LAMMPS developers ahead of time to discuss implementation strategies. That will make it easier to integrate your contribution and typically results in less work for everyone involved. You are also encouraged to search through the list of open issues on GitHub and submit a new issue for a planned feature, to avoid duplicating work (and possibly being scooped).

For informal communication with the LAMMPS developers, you may ask to join the LAMMPS developers on Slack. This slack work space is by invitation only. For access, please send an e-mail to `slack@lammps.org` explaining what part of LAMMPS you are working on. Only discussions related to LAMMPS development are tolerated in that work space, so this is **NOT** for people looking for help with compiling, installing, or using LAMMPS. Please post a message to the LAMMPS forum for those purposes.

### 3.2.2 Time and effort required

How quickly your contribution will be integrated can vary widely. It depends largely on how much effort is required by the LAMMPS developers to integrate and test it, if any and what kind of changes to the core code are required, how quickly you can address them, and how much interest the contribution is to the larger LAMMPS community. This process can be streamlined by following the *requirements* and *style guidelines*. A small, modular, well written contribution may be integrated within hours, but a complex change that requires a re-design of a core functionality in LAMMPS can take months before inclusion (though this is rare).

### 3.2.3 Submission procedure

All changes to LAMMPS (including those from LAMMPS developers) are integrated via pull requests on GitHub and cannot be merged without passing the automated testing and an approving review by a LAMMPS core developer. Before submitting your contribution, you should therefore first ensure that your added or modified code compiles and works correctly with the latest development version of LAMMPS and contains all bug fixes from it.

Once you have prepared everything, see the *LAMMPS GitHub Tutorial* page for instructions on how to submit your changes or new files through a GitHub pull request. If you are unable or unwilling to submit via GitHub yourself, you may also send patch files or full files to the LAMMPS developers and ask them to submit a pull request on GitHub on your behalf. If this is the case, create a gzipped tar file of all new or changed files or a corresponding patch file using 'diff -u' or 'diff -c' format and compress it with gzip. Please only use gzip compression, as this works well and is available on all platforms. This mode of submission may delay the integration as it depends more on the LAMMPS developers.

### 3.2.4 External contributions

If you prefer to do so, you can also develop and support your add-on feature **without** having it included in the LAMMPS distribution, for example as a download from a website of your own. See the External LAMMPS packages and tools page of the LAMMPS website for examples of groups that do this. We are happy to advertise your package and website from that page. Simply email the developers with info about your package, and we will post it there. We recommend naming external packages USER-<name> so they can be easily distinguished from packages in the LAMMPS distribution which do not have the USER- prefix.

### 3.2.5 Location of files: individual files and packages

We rarely accept new styles in the core src folder. Thus, please review the list of *available Packages* to see if your contribution should be added to one of them. It should fit into the general purpose of that package. If it does not fit well, it may be added to one of the EXTRA- packages or the MISC package.

However, if your project includes many related features that are not covered by one of the existing packages or is dependent on a library (bundled or external), it is best to create a new package with its own directory (with a name like FOO). In addition to your new files, the directory should contain a README text file containing your name and contact information and a brief description of what your new package does.

### 3.2.6 Changes to core LAMMPS files

If designed correctly, most additions do not require any changes to the core code of LAMMPS; they are simply add-on files that are compiled with the rest of LAMMPS. To make those styles work, you may need some trivial changes to the core code. An example of a trivial change is making a parent-class method "virtual" when you derive a new child class from it. If your features involve more substantive changes to the core LAMMPS files, it is particularly encouraged that you communicate with the LAMMPS developers early in development.

## 3.3 Requirements for contributions to LAMMPS

The following is a summary of the current requirements and recommendations for including contributed source code or documentation into the LAMMPS software distribution.

### 3.3.1 Motivation

The LAMMPS developers are committed to provide a software package that is versatile, reliable, high-quality, efficient, portable, and easy to maintain and modify. Achieving all of these goals is challenging since a large part of LAMMPS consists of contributed code from many different authors who may not be professionally trained programmers or familiar with the idiosyncrasies of maintaining a large software package. In addition, changes that interfere with the parallel efficiency of the core code must be avoided. As LAMMPS continues to grow and more features and functionality are added, it is necessary to follow established guidelines when accepting new contributions while also working at the same time to improve the existing code.

The following requirements and recommendations are provided as a guide. They indicate which individual requirements are strict, and which represent a preference and thus are negotiable or optional. Please feel free to contact the LAMMPS core developers in case you need additional explanations or clarifications, or you need assistance in implementing the (strict) requirements for your contributions. Requirements include:

- *Licensing requirements* (strict)
- *Integration testing* (strict)
- *Documentation* (strict)
- *Programming language standards* (strict)
- *Build system* (strict)
- *Command or style names* (strict)
- *Programming style requirements* (varied)
- *Examples* (preferred)
- *Error or warning messages and explanations* (preferred)
- *Citation reminder* (optional)
- *Testing* (optional)

### 3.3.2 Licensing requirements (strict)

Contributing authors agree when submitting a pull request that their contributions can be distributed under the LAMMPS license conditions. This is the GNU public license in version 2 (not 3 or later) for the publicly distributed versions, e.g. on the LAMMPS homepage or on GitHub. We also have a version of LAMMPS under LGPL 2.1 terms which is available on request; this will usually be the latest available or a previous stable version with a few LGPL 2.1 incompatible files removed. More details are found on the *LAMMPS open-source license page*.

Your new source files should have the LAMMPS copyright and GPL notice, followed by your name and email address at the top, like other user-contributed LAMMPS source files.

Contributions may be under a different license as long as that license does not conflict with the aforementioned terms. Contributions that use code with a conflicting license can be split into two parts:

1. the core parts (i.e. parts that must be in the *src* tree) that are licensed under compatible terms and bundled with the LAMMPS sources
2. an external library that must be downloaded and compiled (either separately or as part of the LAMMPS compilation)

Please note, that this split licensing mode may complicate including the contribution in binary packages.

### 3.3.3 Integration testing (strict)

Where possible we use available continuous integration tools to search for common programming mistakes, portability limitations, incompatible formatting, and undesired side effects. Contributed code must pass the automated tests on GitHub before it can be merged with the LAMMPS distribution. These tests compile LAMMPS in a variety of environments and settings and run the bundled unit tests. At the discretion of the LAMMPS developer managing the pull request, additional tests may be activated that test for "side effects" on running a collection of input decks and create consistent results. The translation of the documentation to HTML and PDF is also tested.

This means that contributed source code **must** compile with the most current version of LAMMPS with `-DLAMMPS_BIGBIG` in addition to the default setting of `-DLAMMPS_SMALLBIG`. The code needs to work correctly in both cases, and also in serial and parallel using MPI.

Some "disruptive" changes may break tests and require updates to the testing tools or scripts or tests themselves. This is rare. If in doubt, contact the LAMMPS developer that is assigned to the pull request.

### 3.3.4 Documentation (strict)

Contributions that add new styles or commands or augment existing ones must include the corresponding new or modified documentation in ReStructuredText format (.rst files in the `doc/src/` folder). The documentation should be written in American English and the .rst file must only use ASCII characters, so it can be cleanly translated to PDF files (via sphinx and PDFLaTeX). Special characters may be included via embedded math expression typeset in a LaTeX subset.

When adding new commands, they need to be integrated into the sphinx documentation system, and the corresponding command tables and lists updated. When translating the documentation into html files there should be no warnings. When adding a new package, some lists describing packages must also be updated as well as a package specific description added. Likewise, if necessary, some package specific build instructions should be included.

A detailed, step-by-step guide for what a documentation page must contain, how to register a new style in the command tables and overview lists, how to document accelerated variants, and which consistency checks to run is given on the *Writing or updating documentation* page.

As appropriate, the text files with the documentation can include inline mathematical expressions or figures (see `doc/JPG` for examples). Additional PDF files with further details may also be included; see `doc/PDF` for examples. The page should also include literature citations as appropriate; see the bottom of `doc/fix_nh.rst` for examples and the earlier part of the same file for how to format the cite itself. Citation labels must be unique across **all** .rst files. The "Restrictions" section of the page should indicate if your command is only available if LAMMPS is built with the appropriate package. See other command doc files for examples of how to do this.

Please run at least "make html" and "make spelling" from within the doc/src directory, and carefully inspect and proofread the resulting HTML format doc page before submitting your code. Upon submission of a pull request, checks for error free completion of the HTML and PDF build will be performed and also a spell check, a check for correct anchors and labels, and a check for completeness of references to all styles in their corresponding tables and lists is run. In case the spell check reports false positives, they can be added to the file `doc/utils/sphinx-config/false_positives.txt`

Contributions that add or modify the library interface or "public" APIs from the C++ code or the Fortran module must include suitable doxygen comments in the source and corresponding changes to the documentation sources for the "Programmer Guide" guide section of the LAMMPS manual.

If your feature requires some more complex steps and explanations to be used correctly or some external or bundled tools or scripts, we recommend that you also contribute a *Howto document* providing some more background information and some tutorial material. This can also be used to provide more in-depth explanations of models that require use of multiple commands.

As a rule-of-thumb, the more clear and self-explanatory you make your documentation, README files and examples, and the easier you make it for people to get started, the more likely it is that users will try out your new feature.

#### Programming language standards (strict)

The core of LAMMPS is written in C++17 in a style that can be mostly described as "C with classes". Advanced C++ features like operator overloading or excessive use of templates are avoided with the intent to keep the code readable to programmers that have limited C++ programming experience. C++ constructs are acceptable when they help improve the readability and reliability of the code, e.g. when using the *std::string* class instead of manipulating pointers and calling the string functions of the C library or when using *auto* to avoid redundant data type specifications. In addition, a collection of convenient *utility functions and classes* for recurring tasks and a collection of *platform neutral functions* for improved portability are provided. Containers from C++ standard library have to be used with caution, but *std::vector* can be useful and is compatible with efficient communication via MPI. Contributions with code requiring more recent C++ standards are only accepted as packages with the post C++17 standard code confined to the package so that it is optional.

Included Fortran code has to be compatible with the Fortran 2003 standard. Since not all platforms supported by LAMMPS provide good support for compiling Fortran files, it should be considered to rewrite these parts as C++ code, if possible, and thus allow for a wider adoption of the contribution. As of January 2023, all previously included Fortran code for the LAMMPS executable has been replaced by equivalent C++ code.

Python code currently must be compatible with Python 3.6. If a later version or Python is required, it needs to be documented.

Compatibility with older programming language standards is very important to maintain portability and availability of LAMMPS on many platforms. This applies especially to HPC cluster environments, which tend to be running older software stacks and where LAMMPS users may be required to use those older tools for access to advanced hardware features or not have the option to install newer compilers or libraries.

### 3.3.5 Build system (strict)

LAMMPS currently supports two build systems: one that is based on *traditional Makefiles* and one that is based on *CMake*. As of fall 2025, it is no longer required to support the traditional *traditional GNU make build system*. New packages may choose to only support *building with CMake*. Additions to existing packages must follow the requirements set by that package.

For a single pair of header and implementation files that are an independent feature, it is usually only required to add them to `src/.gitignore`.

For traditional make, if your contributed files or package depend on other LAMMPS style files or packages also being installed (e.g. because your file is a derived class from the other LAMMPS class), then an `Install.sh` file is also needed to check for those dependencies and modifications to `src/Depend.sh` to trigger the checks. See other README and Install.sh files in other directories as examples.

Similarly, for CMake support, changes may need to be made to `cmake/CMakeLists.txt`, some of the files in `cmake/presets`, and possibly a file with specific instructions needs to be added to `cmake/Modules/Packages/`. Please check out how this is handled for existing packages and ask the LAMMPS developers if you need assistance.

### 3.3.6 Command or style names, file names, and keywords (strict)

All user-visible command or style names should be all lower case and should only use letters, numbers, or forward slashes. They should be descriptive and initialisms should be avoided unless they are well established (e.g. lj for Lennard-Jones). For a compute style "some/name" the source files must be called `compute_some_name.h` and `compute_some_name.cpp`. The "include guard" in the header file would then be `LMP_COMPUTE_SOME_NAME_H` and the class name `ComputeSomeName`.

### 3.3.7 Programming style requirements (varied)

To maintain source code consistency across contributions from many people, there are various programming style requirements for contributions to LAMMPS. Some of these requirements are strict and must be followed, while others are only preferred and thus may be skipped. An in-depth discussion of the style guidelines is provided in the *programming style doc page*.

### 3.3.8 Examples (preferred)

For many new features, it is preferred that example scripts (simple, small, fast to complete on 1 CPU) are included that demonstrate the use of new or extended functionality. These are typically include under the examples or examples/PACKAGES directory and are further described on the *examples page*. Guidelines for input scripts include:

- commands that generate output should be commented out (except when the output is the sole purpose or the feature, e.g. for a new compute)
- commands like *log*, *echo*, *package*, *processors*, *suffix* may **not** be used in the input file (exception: "processors * * 1" or similar is acceptable when used to avoid unwanted domain decomposition of empty volumes)
- outside of the log files, no generated output should be included
- custom thermo_style settings may not include output measuring CPU or other time as it complicates comparisons between different runs
- input files should be named `in.name`, data files should be named `data.name` and log files should be named `log.version.name.<compiler>.<ncpu>`
- the total file size of all the inputs and outputs should be small
- where possible, potential files from the "potentials" folder or data file from other folders should be re-used through symbolic links

### 3.3.9 Error or warning messages and explanations (preferred)

Changed in version 4May2022.

Starting with LAMMPS version 4 May 2022, the LAMMPS developers have agreed on a new policy for error and warning messages.

Previously, all error and warning strings were supposed to be listed in the class header files with an explanation. Those would then be regularly "harvested" and transferred to alphabetically sorted lists in the manual. To avoid excessively long lists and to reduce effort, this came with a requirement to have rather generic error messages (e.g. "Illegal ... command"). To identify the specific cause, the name of the source file and the line number of the error location would be printed, so that one could look up the cause by reading the source code.

The new policy encourages more specific error messages that ideally indicate the cause directly, and requiring no further lookup. This is aided by either the {fmt} library or the C++ format library (for C++20 and later) enabling `LAMMPS_NS::Error` class methods that take a variable number of arguments and an error text that will be treated like a C++ format string. Error messages should still preferably be kept to a single line or two lines at most.

For more complex explanations or errors that have multiple possible reasons, a paragraph should be added to the *Error_details* page with an error code reference (e.g. `.. _err0001:`) then the utility function *utils::errorurl()*

can be used to generate a URL that will directly lead to that paragraph. An error for missing arguments can be easily generated using the *utils::missing_cmd_args()* convenience function. An example for this approach would be the `src/read_data.cpp` and `src/atom.cpp` files that implement the *read_data* and *atom_modify* commands and that may create *"Unknown identifier in data file"* errors that may have multiple possible reasons which complicates debugging, and thus require some additional explanation.

The transformation of existing LAMMPS code to this new scheme is ongoing. Given the size of the LAMMPS code base, it will take a significant amount of time to complete. For new code, however, following the new approach is strongly preferred. The expectation is that the new scheme will make understanding errors easier for LAMMPS users, developers, and maintainers.

### 3.3.10 Citation reminder (optional)

If there is a paper of yours describing your feature (either the algorithm/science behind the feature itself, or its initial usage, or its implementation in LAMMPS), you can add the citation to the `.cpp` source file. A description of how to add the reminder is *available elsewhere in this manual*.

If there is additional functionality (which may have been added later) described in a different publication, additional citation descriptions may be added so long as they are only registered when the corresponding keyword activating this functionality is used.

With these options, it is possible to have LAMMPS output a specific citation reminder whenever a user invokes your feature from their input script. Please note that you should *only* use this for the *most* relevant paper for a feature and a publication that you or your group authored. E.g. adding a citation in the source code for a paper by Nose and Hoover if you write a fix that implements their integrator is not the intended usage. That kind of citation should just be included in the documentation page you provide describing your contribution. If you are not sure what the best option would be, please contact the LAMMPS developers for advice.

### 3.3.11 Testing (optional)

If your contribution contains new utility functions or a supporting class (i.e. anything that does not depend on a LAMMPS object), new unit tests should be added to a suitable folder in the `unittest` tree. When adding a new LAMMPS style computing forces or selected fixes, a `.yaml` file with a test configuration and reference data should be added for the styles where a suitable tester program already exists (e.g. pair styles, bond styles, etc.). Please see *this section in the manual* for more information on how to enable, run, and expand testing.

## 3.4 LAMMPS programming style

The aim of the LAMMPS developers is to use a consistent programming style and naming conventions across the entire code base, as this helps with maintenance, debugging, and understanding the code, both for developers and users. This page provides a list of standard style choices used in LAMMPS. Some of these standards are required, while others are just preferred. Following these conventions will make it much easier to integrate your contribution. If you are uncertain, please ask.

The files *pair_lj_cut.h*, *pair_lj_cut.cpp*, *utils.h*, and *utils.cpp* may serve as representative examples.

### 3.4.1 Include files (varied)

- Header files that define a new LAMMPS style (i.e. that have a `SomeStyle(some/name,SomeName);` macro in them) should only use the include file for the base class and otherwise use forward declarations and pointers; when interfacing to a library use the PIMPL (pointer to implementation) approach where you have a pointer to a struct that contains all library specific data (and thus requires the library header) but use a forward declaration and define the struct only in the implementation file. This is a **strict** requirement since this is where type clashes between packages and hard-to-find bugs have regularly manifested in the past.

- Header files, especially those defining a "style", should only use the absolute minimum number of include files and **must not** contain any `using` statements. Typically, that would only be the header for the base class. Instead, any include statements should be put in the corresponding implementation files and forward declarations be used. For implementation files, the "include what you use" principle should be employed. However, there is the notable exception that when the `pointers.h` header is included (or the header of one of the classes derived from it), certain headers will *always* be included and thus do not need to be explicitly specified. These are: *mpi.h*, *cstddef*, *cstdio*, *cstdlib*, *string*, *utils.h*, *vector*, *fmt/format.h* (or *format* when using C++20 and later), *climits*, *cinttypes*. This also means any such file can assume that *FILE*, *NULL*, and *INT_MAX* are defined.
- Class members variables should not be initialized in the header file, but instead should be initialized either in the initializer list of the constructor or explicitly assigned in the body of the constructor. If the member variable is relevant to the functionality of a class (for example when it stores a value from a command-line argument), the member variable declaration is followed by a brief comment explaining its purpose and what its values can be. Class members that are pointers should always be initialized to `nullptr` in the initializer list of the constructor. This reduces clutter in the header and avoids accessing uninitialized pointers, which leads to hard to debug issues, class members are often implicitly initialized to `NULL` on the first use (but *not* after a *clear command*). Please see the files `reset_atoms_mol.h` and `reset_atoms_mol.cpp` as an example.
- System headers or headers from installed libraries are included with angular brackets (example: `#include <vector>`), while local include files use double quotes (example: `#include "atom.h"`)
- When including system header files from the C library use the C++-style names (`<cstdlib>` or `<cstring>`) instead of the C-style names (`<stdlib.h>` or `<string.h>`)
- The order of `#include` statements in a file `some_name.cpp` that implements a class `SomeName` defined in a header file `some_name.h` should be as follows:
  - `#include "some_name.h"` followed by an empty line
  - LAMMPS include files e.g. `#include "comm.h"` or `#include "modify.h"` in alphabetical order followed by an empty line
  - System header files from the C++ or C standard library followed by an empty line
  - `using namespace LAMMPS_NS` or other namespace imports.

### 3.4.2 Whitespace (preferred)

Source files should not contain TAB characters unless required by the syntax (e.g. in makefiles) and no trailing whitespace. Text files should have Unix-style line endings (LF-only). Git will automatically convert those in both directions when running on Windows; use dos2unix on Linux machines to convert files to Unix-style line endings. The last line of text files include a line ending.

You can check for these issues with the python scripts in the *"tools/coding_standard"* folder. When run normally with a source file or a source folder as argument, they will list all non-conforming lines. By adding the *-f* flag to the command line, they will modify the flagged files to try to remove the detected issues.

### 3.4.3 Constants (strongly preferred)

Global or per-file constants should be declared as *static constexpr* variables rather than via the pre-processor with *#define*. The name of constants should be all uppercase. This has multiple advantages:

- constants are easily identified as such by their all upper case name
- rather than a pure text substitution during pre-processing, *constexpr variables* have a type associated with them and are processed later in the parsing process where the syntax checks and type specific processing (e.g. via overloads) can be applied to them.
- compilers can emit a warning if the constant is not used and thus can be removed (we regularly check for and remove dead code like this)

- there are no unexpected substitutions and thus confusing syntax errors when compiling leading to, for instance, conflicts so that LAMMPS cannot be compiled with certain combinations of packages (this *has* happened multiple times in the past).

Pre-processor defines should be limited to macros (but consider C++ templates) and conditional compilation. If a pre-processor define must be used, it should be defined at the top of the .cpp file after the include statements and at all cost it should be avoided to put them into header files.

Some sets of commonly used constants are provided in the `MathConst` and `EwaldConst` namespaces and implemented in the files `math_const.h` and `ewald_const.h`, respectively.

There are always exceptions, special cases, and legacy code in LAMMPS, so please contact the LAMMPS developers if you are not sure.

### 3.4.4 Placement of braces (strongly preferred)

For new files added to the "src" tree, a clang-format configuration file is provided under the name *.clang-format*. This file is compatible with clang-format version 8 and later. With that file present, files can be reformatted according to the configuration with a command like: *clang-format -i new-file.cpp*. Ideally, this is done while writing the code or before a pull request is submitted. Blocks of code where the reformatting from clang-format yields hard-to-read or otherwise undesirable output may be protected with placing a pair *// clang-format off* and *// clang-format on* comments around that block.

### 3.4.5 Miscellaneous standards (varied)

- I/O is done via the C-style stdio library and **not** iostreams.
- Do not use so-called "alternative tokens" like `and`, `or`, `not` and similar, but rather use the corresponding operators `&&`, `||`, and `!`. The alternative tokens are not available by default on all compilers.
- Output to the screen and the logfile should use the corresponding FILE pointers and only be done on MPI rank 0. Use the *`utils::logmesg()`* convenience function where possible.
- Usage of C++ *virtual*, *override*, *final* keywords: Please follow the C++ Core Guideline C.128. That means, you should only use *virtual* to declare a new virtual function, *override* to indicate you are overriding an existing virtual function, and *final* to prevent any further overriding.
- Trivial destructors: Do not write destructors when they are empty and *default*.

```
// don't write destructors for A or B like this

class A : protected Pointers {
 public:
   A();
   ~A() override {}
};

class B : protected Pointers {
 public:
   B();
   ~B() override = default;
};

// instead, let the compiler create the implicit default destructor by not writing␣
 ↪it

class A : protected Pointers {
```

(continues on next page)

(continued from previous page)

```
 public:
   A();
};

class B : protected Pointers {
 public:
   B();
};
```

- Please use clang-format only to reformat files that you have contributed. For header files containing a `SomeStyle(keyword, ClassName)` macros it is required to have this macro embedded with a pair of `// clang-format off`, `// clang-format on` comments and the line must be terminated with a semicolon (;). Example:

```
#ifdef COMMAND_CLASS
// clang-format off
CommandStyle(run,Run);
// clang-format on
#else

#ifndef LMP_RUN_H
[...]
```

  You may also use `// clang-format on/off` throughout your files to protect individual sections from being reformatted.

- All files should have 0644 permissions, i.e. writable by the user only and readable by all and no executable permissions. Executable permissions (0755) should only be for shell scripts or python or similar scripts for interpreted script languages.

## 3.5 Atom styles

Classes that define an *atom style* are derived from the AtomVec class and managed by the Atom class. The atom style determines what attributes are associated with an atom and communicated when it is a ghost atom or migrates to a new processor. A new atom style can be created if one of the existing atom styles does not define all the attributes you need to store and communicate with atoms.

The file `atom_vec_atomic.cpp` is the simplest example of an atom style. Examining the code for others will make these instructions more clear.

Note that the *atom style hybrid* command can be used to define atoms or particles which have the union of properties of individual styles. Also the *fix property/atom* command can be used to add a single property (e.g. charge or a molecule ID) to a style that does not have it. It can also be used to add custom properties to an atom, with options to communicate them with ghost atoms or read them from a data file. Other LAMMPS commands can access these custom properties, as can new pair, fix, compute styles that are written to work with these properties. For example, the *set* command can be used to set the values of custom per-atom properties from an input script. All of these methods are less work than writing and testing(!) code for a new atom style.

If you follow these directions your new style will automatically work in tandem with others via the *atom_style hybrid* command.

The first step is to define a set of string lists in the constructor of the new derived class. Each list will have zero or more comma-separated strings that correspond to the variable names used in the `atom.h` header file for per-atom properties. Note that some represent per-atom vectors (q, molecule) while other are per-atom arrays (x,v). For all but the last two lists you do not need to specify any of (id,type,x,v,f). Those are included automatically as needed in the other lists.

| | |
|---|---|
| fields_grow | full list of properties which is allocated and stored |
| fields_copy | list of properties to copy atoms are rearranged on-processor |
| fields_comm | list of properties communicated to ghost atoms every step |
| fields_comm_vel | additional properties communicated if *comm_modify vel* is used |
| fields_reverse | list of properties summed from ghost atoms every step |
| fields_border | list of properties communicated with ghost atoms every reneighboring step |
| fields_border_vel | additional properties communicated if *comm_modify vel* is used |
| fields_exchange | list of properties communicated when an atom migrates to another processor |
| fields_restart | list of properties written/read to/from a restart file |
| fields_create | list of properties defined when an atom is created by *create_atoms* |
| fields_data_atom | list of properties (in order) in the Atoms section of a data file, as read by *read_data* |
| fields_data_vel | list of properties (in order) in the Velocities section of a data file, as read by *read_data* |

In these lists you can list variable names which LAMMPS already defines (in some other atom style), or you can create new variable names. You should not re-use a LAMMPS variable in your atom style that is used for something with a different meaning in another atom style. If the meaning is related, but interpreted differently by your atom style, then using the same variable name means a user must not use your style and the other style together in a *atom_style hybrid* command. Because there will only be one value of the variable and different parts of LAMMPS will then likely use it differently. LAMMPS has no way of checking for this.

If you are defining new variable names then make them descriptive and unique to your new atom style. For example choosing "e" for energy is a bad choice; it is too generic. A better choice would be "e_foo", where "foo" is specific to your style.

If any of the variable names in your new atom style do not exist in LAMMPS, you need to add them to the src/atom.h and atom.cpp files.

Search for the word "customize" or "customization" in these 2 files to see where to add your variable. Adding a flag to the 2nd customization section in `atom.h` is only necessary if your code (e.g. a pair style) needs to check that a per-atom property is defined. These flags should also be set in the constructor of the atom style child class.

In `atom.cpp`, aside from the constructor and destructor, there are 3 methods that a new variable name or flag needs to be added to.

In `Atom::peratom_create()` when using the `Atom::add_peratom()` method, a cols argument of 0 is for per-atom vectors, a length > 1 is for per-atom arrays. Note the use of the extra per-thread flag and the add_peratom_vary() method when the last dimension of the array is variable-length.

Adding the variable name to Atom::extract() enables the per-atom data to be accessed through the *LAMMPS library interface* by a calling code, including from *Python*.

The constructor of the new atom style will also typically set a few flags which are defined at the top of `atom_vec.h`. If these are unclear, see how other atom styles use them.

The grow_pointers() method is also required to make a copy of peratom data pointers, as explained in the code.

There are a number of other optional methods which your atom style can implement. These are only needed if you need to do something out-of-the-ordinary which the default operation of the AtomVec parent class does not take care of. The best way to figure out why they are sometimes useful is to look at how other atom styles use them.

- process_args = use if the atom style has arguments
- init = called before each run
- force_clear = called before force computations each timestep

A few atom styles define "bonus" data associated with some or all of their particles, such as *atom_style ellipsoid or tri*. These methods work with that data:

- copy_bonus
- clear_bonus
- pack_comm_bonus
- unpack_comm_bonus
- pack_border_bonus
- unpack_border_bonus
- pack_exchange_bonus
- unpack_exchange_bonus
- size_restart_bonus
- pack_restart_bonus
- unpack_restart_bonus
- data_atom_bonus
- memory_usage_bonus

The *atom_style body* command can define a particle geometry with an arbitrary number of values. This method reads it from a data file:

- data_body

These methods are called before or after operations handled by the parent AtomVec class. They allow an atom style to do customized operations on the per-atom values. For example *atom_style sphere* reads a diameter and density of each particle from a data file. But these need to be converted internally to a radius and mass. That operation is done in the data_atom_post() method.

- pack_restart_pre
- pack_restart_post
- unpack_restart_init
- create_atom_post
- data_atom_post
- pack_data_pre
- pack_data_post

These methods enable the *compute property/atom* command to access per-atom variables it does not already define as arguments, so that they can be written to a dump file or used by other LAMMPS commands.

- property_atom
- pack_property_atom

## 3.6 Pair styles

Classes that compute pairwise non-bonded interactions are derived from the `Pair` class. In LAMMPS, pairwise force calculations include many-body potentials such as EAM, Tersoff, or ReaxFF where particles interact without an explicit bond topology but include interactions beyond pairwise non-bonded contributions. New styles can be created to add support for additional pair potentials to LAMMPS. When the modifications are small, sometimes it is more effective to derive from an existing pair style class. This latter approach is also used by *Accelerator packages* where the accelerated style names differ from their base classes by an appended suffix.

The file `src/pair_lj_cut.cpp` is an example of a Pair class with a very simple potential function. It includes several optional methods to enable its use with *run_style respa* and *compute group/group*. *Writing new pair styles* contains a detailed discussion of writing new pair styles from scratch, and how simple and more complex pair styles can be implemented with examples from existing pair styles.

Here is a brief list of some the class methods in the Pair class that *must* be or *may* be overridden in a derived class for a new pair style.

| Required | "pure" methods that *must* be overridden in a derived class |
|---|---|
| compute | workhorse routine that computes pairwise interactions |
| settings | processes the arguments to the pair_style command |
| coeff | set coefficients for one i,j type pair, called from pair_coeff |

| Optional | methods that have a default or dummy implementation |
|---|---|
| init_one | perform initialization for one i,j type pair |
| init_style | style initialization: request neighbor list(s), error checks |
| init_list | Neighbor class callback function to pass neighbor list to pair style |
| single | force/r and energy of a single pairwise interaction between two atoms |
| compute_inner/middle/outer | versions of compute used by rRESPA |
| compute_atomic_energy | energy of one atom, equivalent to per-atom energy |
| memory_usage | return estimated amount of memory used by the pair style |
| modify_params | process arguments to pair_modify command |
| extract | provide access to internal scalar or per-type data like cutoffs |
| extract_peratom | provide access to internal per-atom data |
| setup | initialization at the beginning of a run |
| finish | called at the end of a run, e.g. to print |
| write & read_restart | write/read i,j pair coeffs to restart files |
| write & read_restart_settings | write/read global settings to restart files |
| write_data | write Pair Coeffs section to data file |
| write_data_all | write PairIJ Coeffs section to data file |
| pack & unpack_forward_comm | copy data to and from buffer if style uses forward communication |
| pack & unpack_reverse_comm | copy data to and from buffer if style uses reverse communication |
| reinit | reset all type-based parameters, called by fix adapt for example |
| reset_dt | called when the time step is changed by timestep or fix reset/dt |
| born_matrix | compute Born matrix elements for elasticity (optional) |
| single_hessian | compute the per-pair Hessian (optional) |

Here is a list of flags or settings that should be set in the constructor of the derived pair class when they differ from the default setting.

| Name of flag | Description | default |
|---|---|---|
| single_enable | 1 if single() method is implemented, 0 if missing | 1 |
| respa_enable | 1 if pair style has compute_inner/middle/outer() | 0 |
| restartinfo | 1 if pair style writes its settings to a restart | 1 |
| one_coeff | 1 if only a pair_coeff * * command is allowed | 0 |
| manybody_flag | 1 if pair style is a manybody potential | 0 |
| unit_convert_flag | value != 0 indicates support for unit conversion | 0 |
| no_virial_fdotr_compute | 1 if pair style does not call virial_fdotr_compute() | 0 |
| writedata | 1 if write_data() and write_data_all() are implemented | 0 |
| comm_forward | size of buffer (in doubles) for forward communication | 0 |
| comm_reverse | size of buffer (in doubles) for reverse communication | 0 |
| ghostneigh | 1 if cutghost is set and style uses neighbors of ghosts | 0 |
| finitecutflag | 1 if cutoff depends on diameter of atoms | 0 |
| reinitflag | 1 if style has reinit() and is compatible with fix adapt | 0 |
| ewaldflag | 1 if compatible with kspace_style ewald | 0 |
| pppmflag | 1 if compatible with kspace_style pppm | 0 |
| msmflag | 1 if compatible with kspace_style msm | 0 |
| dispersionflag | 1 if compatible with ewald/disp or pppm/disp | 0 |
| tip4pflag | 1 if compatible with kspace_style pppm/tip4p | 0 |
| dipoleflag | 1 if compatible with dipole kspace_style | 0 |
| spinflag | 1 if compatible with spin kspace_style | 0 |
| atomic_energy_enable | 1 if compute_atomic_energy() routine exists | 0 |
| born_matrix_enable | 1 if born_matrix() routine exists | 0 |
| single_hessian_enable | 1 if single_hessian() routine exists | 0 |
| centroidstressflag | CENTROID_SAME/CENTROID_AVAIL/CENTROID_NOTAVAIL | SAME |

## 3.7 Bond, angle, dihedral, improper styles

Classes that compute molecular interactions are derived from the Bond, Angle, Dihedral, and Improper classes. New styles can be created to add new potentials to LAMMPS.

*Writing new bond, angle, dihedral, and improper styles* contains a detailed discussion of writing new bond, angle, dihedral, and improper styles from scratch with examples from the existing harmonic styles.

Bond_harmonic.cpp is the simplest example of a bond style. Ditto for the harmonic forms of the angle, dihedral, and improper style commands.

Here is a brief description of common methods you define in your new derived class. See bond.h, angle.h, dihedral.h, and improper.h for details and specific additional methods.

| Required | "pure" methods that *must* be overridden in a derived class |
|---|---|
| compute | compute the molecular interactions for all listed items |
| coeff | set coefficients for one type |
| equilibrium_distance | length of bond, used by SHAKE (bond styles only) |
| equilibrium_angle | opening of angle, used by SHAKE (angle styles only) |
| write & read_restart | writes/reads coeffs to restart files |
| single | force/r (bond styles only) and energy of a single bond or angle |

| Optional | methods that have a default or dummy implementation |
|---|---|
| init | check if all coefficients are set, calls init_style() |
| init_style | check if style specific conditions are met |
| settings | apply global settings for all types |
| write & read_restart_settings | writes/reads global style settings to restart files |
| write_data | write corresponding Coeffs section(s) in data file |
| memory_usage | tally memory allocated by the style |
| extract | provide access to internal data |
| born_matrix | compute Born matrix elements for elasticity |
| reinit | reset all type-based parameters, called by fix adapt (bonds only) |
| pack & unpack_forward_comm | copy data to and from buffer in forward communication (bonds only) |
| pack & unpack_reverse_comm | copy data to and from buffer in reverse communication (bonds only) |

Here is a list of flags or settings that should be set in the constructor of the derived class when they differ from the default setting.

| Name of flag | Description | default |
|---|---|---|
| writedata | 1 if write_data() is implemented | 1 |
| single_extra | number of extra single values calculated (bond styles only) | 0 |
| partial_flag | 1 if bond type can be set to 0 and deleted (bond styles only) | 0 |
| reinitflag | 1 if style has reinit() and is compatible with fix adapt | 1 |
| comm_forward | size of buffer (in doubles) for forward communication (bond styles only) | 0 |
| comm_reverse | size of buffer (in doubles) for reverse communication (bond styles only) | 0 |
| comm_reverse_off | size of buffer for reverse communication with newton off (bond styles only) | 0 |
| born_matrix_enable | 1 if born_matrix() is implemented | 0 |
| centroidstressflag | CENTROID_AVAIL if centroid stress is different and implemented | varies |

The default value of `centroidstressflag` is `CENTROID_SAME` for bond styles and `CENTROID_AVAIL` for angle, dihedral, and improper styles. It should be set to `CENTROID_NOTAVAIL` when the centroid stress differs from the two-body virial but has not yet been implemented.

## 3.8 Compute styles

Classes that compute scalar and vector quantities like temperature and the pressure tensor, as well as classes that compute per-atom quantities like kinetic energy and the centro-symmetry parameter are derived from the Compute class. New styles can be created to add new calculations to LAMMPS.

*Writing new compute styles* contains a detailed discussion of writing new compute styles from scratch with examples from existing styles.

The `src/compute_temp.cpp` file is a simple example of computing a scalar temperature. The `src/compute_ke_atom.cpp` file is a simple example of computing per-atom kinetic energy.

Here is a brief description of methods you define in your new derived class. See `src/compute.h` for additional details.

| | |
|---|---|
| post_constructor | perform tasks that cannot be run in the constructor (optional) |
| init | perform one time setup (required) |
| init_list | neighbor list setup, if needed (optional) |
| setup | called before the first timestep of a run (optional) |

continues on next page

Table 1 – continued from previous page

| | |
|---|---|
| compute_scalar | compute a scalar quantity (optional) |
| compute_vector | compute a vector of quantities (optional) |
| compute_array | compute a 2d array of quantities (optional) |
| compute_peratom | compute one or more quantities per atom (optional) |
| compute_local | compute one or more quantities per processor (optional) |
| compute_pergrid | compute one or more quantities per grid cell (optional) |
| compute_image | compute image data for dump image (optional) |
| set_arrays | initialize per-atom arrays when a new atom is created (optional) |
| pack_forward_comm | pack a buffer with items to communicate (optional) |
| unpack_forward_comm | unpack the buffer (optional) |
| pack_reverse_comm | pack a buffer with items to reverse communicate (optional) |
| unpack_reverse_comm | unpack the buffer (optional) |
| dof_remove_pre | pre-check before removing atoms from temperature DOF (optional) |
| dof_remove | return 1 to exclude an atom from temperature DOF count (optional) |
| remove_bias | remove velocity bias from one atom (optional) |
| remove_bias_all | remove velocity bias from all atoms in group (optional) |
| reapply_bias_all | reapply velocity bias to all atoms in group (optional) |
| restore_bias | restore velocity bias for one atom after remove_bias (optional) |
| restore_bias_all | same as before, but for all atoms in group (optional) |
| lock_enable | enable locking mechanism for fix ave/time (optional) |
| lock_disable | disable locking mechanism (optional) |
| refresh | refresh internal state (e.g. for dump_modify refresh) (optional) |
| pair_setup_callback | callback invoked during pair-style setup (optional) |
| pair_tally_callback | callback function for *tally*-style computes (optional). |
| get_thermo_colname | return custom column label for thermo output (optional) |
| modify_param | called when a compute_modify request is executed (optional) |
| memory_usage | tally memory usage (optional) |

Tally-style computes are a special case, as their computation is done in two stages: the callback function is registered with the pair style and then called from the Pair::ev_tally() function, which is called for each pair after force and energy has been computed for this pair. Then the tallied values are retrieved with the standard compute_scalar or compute_vector or compute_peratom methods. The *compute styles in the TALLY package* provide *examples* for utilizing this mechanism.

## 3.9 Fix styles

In LAMMPS, a "fix" is any operation that is computed during timestepping that alters some property of the system. Essentially everything that happens during a simulation besides force computation, neighbor list construction, and output, is a "fix". This includes time integration (update of coordinates and velocities), force constraints or boundary conditions (SHAKE or walls), and diagnostics (compute a diffusion coefficient). New styles can be created to add new options to LAMMPS.

The file `src/fix_setforce.cpp` is a simple example of setting forces on atoms to prescribed values. There are dozens of fix options already in LAMMPS; choose one as a template that is similar to what you want to implement. There also is a detailed discussion of *how to write new fix styles* in LAMMPS.

Here is a brief description of methods you can define in your new derived class. See `src/fix.h` for additional details.

| Required | "pure" method that *must* be overridden in a derived class |
|---|---|
| setmask | determines when the fix is called during the timestep |

| Optional | methods that have a default or empty implementation |
|---|---|
| post_constructor | perform tasks that cannot be run in the constructor |
| init | initialization before a run |
| init_list | store pointer to neighbor list; called by neighbor list code |
| setup_pre_exchange | called before atom exchange in setup |
| setup_pre_neighbor | called before neighbor list build in setup |
| setup_post_neighbor | called after neighbor list build in setup |
| setup_pre_force | called before force computation in setup |
| setup_pre_reverse | called before reverse communication in setup |
| setup | called immediately before the first timestep and after forces are computed |
| min_setup | like setup, but for minimizations instead of MD runs |
| initial_integrate | called at very beginning of each timestep |
| post_integrate | called after the first half-step position/velocity update |
| pre_exchange | called before atom exchange on re-neighboring steps |
| pre_neighbor | called before neighbor list build |
| post_neighbor | called after neighbor list build |
| pre_force | called before pair & molecular forces are computed |
| pre_reverse | called before reverse communication of forces |
| post_force | called after pair & molecular forces are computed and communicated |
| final_integrate | called at end of each timestep |
| end_of_step | called at very end of timestep |
| post_run | called at the end of each run command |
| write_restart | dumps fix info to restart file |
| restart | uses info from restart file to re-initialize the fix |
| grow_arrays | allocate memory for atom-based arrays used by fix |
| copy_arrays | copy atom info when an atom migrates to a new processor |
| set_arrays | set per-atom values when a new atom is created |
| pack_exchange | store atom's data in a buffer |
| unpack_exchange | retrieve atom's data from a buffer |
| pack_restart | store atom's data for writing to restart file |
| unpack_restart | retrieve atom's data from a restart file buffer |
| size_restart | size of atom's data |
| maxsize_restart | max size of atom's data |
| setup_pre_force_respa | same as setup_pre_force, but for rRESPA |
| initial_integrate_respa | same as initial_integrate, but for rRESPA |
| post_integrate_respa | called after the first half integration step is done in rRESPA |
| pre_force_respa | same as pre_force, but for rRESPA |
| post_force_respa | same as post_force, but for rRESPA |
| final_integrate_respa | same as final_integrate, but for rRESPA |
| min_pre_exchange | called before atom exchange during minimization |

continues on next page

Table 2 – continued from previous page

| Optional | methods that have a default or empty implementation |
|---|---|
| min_pre_neighbor | called before neighbor list build during minimization |
| min_post_neighbor | called after neighbor list build during minimization |
| min_pre_force | called before pair & molecular forces are computed in minimizer |
| min_pre_reverse | called before reverse communication during minimization |
| min_post_force | called after pair & molecular forces are computed and communicated in minimizer |
| min_energy | return an extra energy contribution during minimization |
| min_store | store extra data for linesearch based minimization on a LIFO stack |
| min_pushstore | push the minimization LIFO stack one element down |
| min_popstore | pop the minimization LIFO stack one element up |
| min_clearstore | clear minimization LIFO stack |
| min_reset_ref | reset the reference state for the line search |
| min_step | reset or move forward on line search minimization |
| min_dof | report number of degrees of freedom *added* by this fix in minimization |
| max_alpha | report maximum allowed step size during linesearch minimization |
| pack_forward_comm | pack a buffer to communicate a per-atom quantity |
| unpack_forward_comm | unpack a buffer to communicate a per-atom quantity |
| pack_reverse_comm_size | return the dynamic size of the reverse communication buffer |
| pack_reverse_comm | pack a buffer to reverse communicate a per-atom quantity |
| unpack_reverse_comm | unpack a buffer to reverse communicate a per-atom quantity |
| dof | report number of degrees of freedom *removed* by this fix during MD |
| compute_scalar | return a global scalar property that the fix computes |
| compute_vector | return a component of a vector property that the fix computes |
| compute_array | return a component of an array property that the fix computes |
| get_thermo_colname | return custom thermo column label for a vector or array component |
| deform | called when the box size is changed |
| reset_target | called when a change of the target temperature is requested during a run |
| reset_dt | called when a change of the time step is requested during a run |
| modify_param | called when a fix_modify request is executed |
| memory_usage | report memory used by fix |
| image | pass lists of graphics objects to *dump image fix* |

Typically, only a small fraction of these methods are defined for a particular fix. The `setmask()` method is mandatory, as it determines *when* the fix will be invoked during *the evolution of a timestep*. Fixes that perform time integration (*nve*, *nvt*, *npt*) implement `initial_integrate()` and `final_integrate()` to perform velocity Verlet time stepping updates. Fixes that apply forces implement `post_force()`, i.e. after the forces on atoms have been computed and

collected to the local atoms.

Fixes that perform diagnostics typically implement `end_of_step()`. For such a fix, one of your fix arguments must set the variable "nevery" which is used to determine when to call the fix and you **must** set this variable in the constructor of your fix. By convention, this is the first argument the fix defines (after the fix-ID, group-ID, and fix style).

If the fix needs to store information for each atom that persists from timestep to timestep, it can manage that memory and migrate the info with the atoms as they move from processors to processor by implementing the `grow_arrays()`, `copy_arrays()`, `pack_exchange()`, and `unpack_exchange()` methods. Similarly, the `pack_restart()` and `unpack_restart()` methods can be implemented to store information about the fix in binary restart files. If you wish an integrator or force constraint fix to work with rRESPA (see the *run_style* command), the `initial_integrate_respa()`, `post_force_respa()`, and `final_integrate_respa()` methods can be implemented. The `compute_scalar()` and `compute_vector()` methods enable a fix to contribute values to thermodynamic output, as printed quantities and/or to be summed to the potential energy of the system. The corresponding flags (`scalar_flag`, `vector_flag`, etc.) must be set in the constructor to tell LAMMPS which of these methods are implemented.

## 3.10 Input script command style

New commands can be added to LAMMPS input scripts by adding new classes that are derived from the Command class and thus must have a "command" method. For example, the *create_atoms*, *read_data*, *velocity*, and *run* commands are all implemented in this fashion. When such a command is encountered in the LAMMPS input script, LAMMPS simply creates a class instance with the corresponding name, invokes the "command" method of the class, and passes it the arguments from the input script. The command method can perform whatever operations it wishes on LAMMPS data structures. After the command method returns the class instance is deleted.

There is a detailed discussion of *how to write new command styles* in LAMMPS.

The method that your new Command class *must* define is as follows:

| command | operations performed by the new command |
|---|---|

Of course, the new class can define other methods and variables as needed.

## 3.11 Dump styles

Classes that dump per-atom info to files are derived from the Dump class. To dump new quantities or in a new format, a new derived dump class can be added, but it is typically simpler to modify the DumpCustom class contained in the `dump_custom.cpp` file.

`src/dump_atom.cpp` is a simple example of a derived dump class.

Here is a brief description of methods you define in your new derived class. See `src/dump.h` for details.

| Required | "pure" methods that *must* be overridden |
|---|---|
| init_style | initialize style-specific state before a run |
| write_header | write the header section of a snapshot of atoms |
| pack | pack a proc's output data into a buffer |
| write_data | write a proc's data to a file |

| Optional | methods that have a default or empty implementation |
|---|---|
| count | count the number of lines a processor will output |
| openfile | open the output file (override for custom I/O) |
| convert_string | convert atom data to a string buffer |
| write_footer | write a footer after each snapshot |
| pack_forward_comm | pack a buffer for forward ghost communication |
| unpack_forward_comm | unpack a forward communication buffer |
| pack_reverse_comm | pack a buffer for reverse ghost communication |
| unpack_reverse_comm | unpack a reverse communication buffer |
| modify_param | called when dump_modify is executed (optional) |
| extract | provide access to internal data (optional) |

See the *dump* command and its *custom* style for a list of keywords for atom information that can already be dumped by DumpCustom. It includes options to dump per-atom info from Compute classes, so adding a new derived Compute class is one way to calculate new quantities to dump.

Note that new keywords for atom properties are not typically added to the *dump custom* command. Instead they are added to the *compute property/atom* command.

## 3.12 Kspace styles

Classes that compute long-range Coulombic interactions via K-space representations (Ewald, PPPM) are derived from the KSpace class. New styles can be created to add new K-space options to LAMMPS.

`src/ewald.cpp` is an example of computing K-space interactions.

Here is a brief description of methods you define in your new derived class. See `src/kspace.h` for details.

| Required | "pure" methods that *must* be overridden in a derived class |
|---|---|
| init | initialize the calculation before a run |
| setup | computation before the first timestep of a run |
| compute | every-timestep computation |

| Optional | methods that have a default or empty implementation |
|---|---|
| settings | process the arguments to the kspace_style command |
| compute_group_group | compute energy/force between two groups of atoms |
| qsum_qsq | compute total charge and charge-squared sum |
| reset_grid | called when the FFT grid size changes |
| pack_forward_grid | pack FFT grid data into a buffer for forward communication |
| unpack_forward_grid | unpack FFT grid data from a forward communication buffer |
| pack_reverse_grid | pack FFT grid data into a buffer for reverse communication |
| unpack_reverse_grid | unpack FFT grid data from a reverse communication buffer |
| timing | report timing information (used by kspace_modify) |
| timing_1d | report 1d FFT timing |
| timing_3d | report 3d FFT timing |
| modify_param | handle arguments from the kspace_modify command |
| memory_usage | tally of memory usage |

Here is a list of flags or settings that should be set in the constructor of the derived class when they differ from the default setting.

| Name of flag | Description | default |
|---|---|---|
| ewaldflag | 1 if this is an Ewald solver | 0 |
| pppmflag | 1 if this is a PPPM solver | 0 |
| msmflag | 1 if this is an MSM solver | 0 |
| dispersionflag | 1 if this is a LJ/dispersion solver | 0 |
| tip4pflag | 1 if this is a TIP4P solver | 0 |
| dipoleflag | 1 if this is a dipole solver | 0 |
| spinflag | 1 if this is a spin solver | 0 |
| triclinic_support | 1 if the style supports triclinic boxes | 1 |
| group_group_enable | 1 if compute_group_group() is implemented | 0 |
| centroidstressflag | CENTROID_SAME/CENTROID_AVAIL/CENTROID_NOTAVAIL | NOTAV. |

The pair-style flags `ewaldflag`, `pppmflag`, `msmflag`, `dispersionflag`, `tip4pflag`, `dipoleflag`, and `spinflag` must match the corresponding flags set in the pair style that is used together with the kspace style. The default value of `centroidstressflag` is `CENTROID_NOTAVAIL` for kspace styles, as the centroid stress for long-range interactions is generally not equal to the two-body virial.

## 3.13 Minimization styles

Classes that perform energy minimization derived from the Min class. New styles can be created to add new minimization algorithms to LAMMPS.

`src/min_cg.cpp` is an example of conjugate gradient minimization.

Here is a brief description of methods you define in your new derived class. See `src/min.h` for details.

| Required | "pure" methods that *must* be overridden |
|---|---|
| setup_style | initialize minimizer-specific data structures |
| reset_vectors | connect internal xvec/fvec pointers to atom coordinates |
| iterate | perform the minimization iterations; returns stop-code |

| Optional | methods that have a default or base-class implementation |
|---|---|
| init | initialize the minimization before a run |
| setup | called before minimization starts |
| setup_minimal | minimal setup variant used for restarts |
| init_style | check style-specific conditions at init time |
| run | called by LAMMPS to drive the full minimization |
| force_clear | clear forces; override for GPU/Kokkos variants |
| energy_force | evaluate energy and forces; override for Kokkos |
| fnorm_sqr | return the squared force norm; override for spin styles |
| fnorm_inf | return the inf-norm of forces; override for spin styles |
| fnorm_max | return the max force norm; override for spin styles |
| modify_param | handle arguments from the min_modify command |
| memory_usage | tally of memory usage |

The `iterate()` method must return one of the following integer stop-condition codes defined as an enum in `src/min.h`: `MAXITER`, `MAXEVAL`, `ETOL`, `FTOL`, `DOWNHILL`, `ZEROALPHA`, `ZEROFORCE`, `ZEROQUAD`, `TRSMALL`, `INTERROR`, `TIMEOUT`, or `MAXVDOTF`.

The line-search style used by the minimizer is controlled by the `linestyle` member, which takes values from the enum `{ BACKTRACK, QUADRATIC, FORCEZERO, SPIN_CUBIC, SPIN_NONE }`.

## 3.14 Region styles

Classes that define geometric regions are derived from the Region class. Regions are used elsewhere in LAMMPS to group atoms, delete atoms to create a void, insert atoms in a specified region, etc. New styles can be created to add new region shapes to LAMMPS.

The files `region_sphere.cpp` and `region_block.cpp` and their corresponding headers are representative examples of region style implementations.

Here is a brief description of methods you define in your new derived class. See `src/region.h` for details.

| Required | "pure" methods that *must* be overridden in a derived class |
|---|---|
| inside | determine whether a point is in the region |
| surface_interior | determine if a point is within a cutoff distance inside of surface |
| surface_exterior | determine if a point is within a cutoff distance outside of surface |

| Optional | methods that have a default or empty implementation |
|---|---|
| shape_update | change region shape if set by time-dependent variable |
| bbox_update | update the bounding box when the shape is variable |
| init | check style-specific conditions before a run |
| set_velocity | compute the velocity of a moving region surface |
| set_velocity_shape | shape-specific part of set_velocity |
| velocity_contact_shape | return velocity at a contact point for the specific shape |
| write_restart | write region state to a restart file |
| restart | read region state from a restart file |
| length_restart_string | return the size of the restart data |
| reset_vel | reset velocity when the region is reset |

## 3.15 Body styles

Classes that define body particles are derived from the Body class. Body particles can represent complex entities, such as surface meshes of discrete points, collections of sub-particles, deformable objects, etc.

See the *Howto body* page for an overview of using body particles and the various body styles LAMMPS supports. New styles can be created to add new kinds of body particles to LAMMPS.

Body_nparticle.cpp is an example of a body particle that is treated as a rigid body containing N sub-particles.

Here is a brief description of methods you define in your new derived class. See `src/body.h` for details.

| Required | "pure" methods that *must* be overridden |
|---|---|
| data_body | process a line from the Bodies section of a data file |
| pack_data_body | pack body data into a buffer for writing to a data file |
| write_data_body | write body data to a data file |
| noutrow | number of sub-particles output is generated for |
| noutcol | number of values per-sub-particle output is generated for |
| output | output values for the Mth sub-particle |
| image | provide geometry for dump image rendering |

| Optional | methods that have a default or empty implementation |
|---|---|
| pack_comm_body | body attributes to communicate every timestep |
| unpack_comm_body | unpacking of those attributes |
| pack_border_body | body attributes to communicate when reneighboring is done |
| unpack_border_body | unpacking of those attributes |
| radius_body | return effective radius for use by granular pair styles |

## 3.16 Granular Sub-Model styles

In granular models, particles are spheres with a finite radius and rotational degrees of freedom as further described in the *Howto granular page*. Interactions between pair of particles or particles and walls may therefore depend on many different modes of motion as described in *pair granular* and *fix wall/gran*. In both cases, the exchange of forces, torques, and heat flow between two types of bodies is defined using a GranularModel class. The GranularModel class organizes the details of an interaction using a series of granular sub-models each of which describe a particular interaction mode (e.g. normal forces or rolling friction). From a parent GranSubMod class, several types of sub-model classes are derived:

- GranSubModNormal: normal force sub-model
- GranSubModDamping: normal damping sub-model
- GranSubModTangential: tangential forces and sliding friction sub-model
- GranSubModRolling: rolling friction sub-model
- GranSubModTwisting: twisting friction sub-model
- GranSubModHeat: heat conduction sub-model

For each type of sub-model, more classes are further derived, each describing a specific implementation. For instance, from the GranSubModNormal class the GranSubModNormalHooke, GranSubModNormalHertz, and GranSubModNormalJKR classes are derived which calculate Hookean, Hertzian, or JKR normal forces, respectively. This modular structure simplifies the addition of new granular contact models as one only needs to create a new GranSubMod class without having to modify the more complex PairGranular, FixGranWall, and GranularModel classes. Most GranSubMod methods are also already defined by the parent classes so new contact models typically only require edits to a few relevant methods (e.g. methods that define coefficients and calculate forces).

Each GranSubMod class has a pointer to both the LAMMPS class and the GranularModel class which owns it, `lmp` and `gm`, respectively. The GranularModel class includes several public variables that describe the geometry/dynamics of the contact such as

| | |
|---|---|
| `xi` and `xj` | Positions of the two contacting bodies |
| `vi` and `vj` | Velocities of the two contacting bodies |
| `omegai` and `omegaj` | Angular velocities of the two contacting bodies |
| `dx` and `nx` | The displacement and normalized displacement vectors |
| `r`, `rsq`, and `rinv` | The distance, distance squared, and inverse distance |
| `radsum` | The sum of particle radii |
| `vr`, `vn`, and `vt` | The relative velocity vector and its normal and tangential components |
| `wr` | The relative rotational velocity |

These quantities, among others, are calculated in the `GranularModel->check_contact()` and `GranularModel->calculate_forces()` methods which can be referred to for more details.

To create a new GranSubMod class, it is recommended that one first looks at similar GranSubMod classes. All GranSubMod classes share several general methods which may need to be defined

| | |
|---|---|
| `mix_coeff()` | Optional method to define how coefficients are mixed for different atom types. By default, coefficients are mixed using a geometric mean. |
| `coeffs_to_local()` | Parses coefficients to define local variables. Run once at model construction. |
| `init()` | Optional method to define local variables after other GranSubMod types were created. For instance, this method may be used by a tangential model that derives parameters from the normal model. |

The Normal, Damping, Tangential, Twisting, and Rolling sub-models also have a `calculate_forces()` method which calculate the respective forces/torques. Correspondingly, the Heat sub-model has a `calculate_heat()` method. Lastly, the Normal sub-model has a few extra optional methods:

| | |
|---|---|
| `touch()` | Tests whether particles are in contact. By default, when particles overlap. |
| `pulloff_distance()` | Returns the distance at which particles stop interacting. By default, when particles no longer overlap. |
| `calculate_radius()` | Returns the radius of the contact. By default, the radius of the geometric cross section. |
| `set_fncrit()` | Defines the critical force to break the contact used by some tangential, rolling, and twisting sub-models. By default, the current total normal force including damping. |

As an example, say one wanted to create a new normal force option that consisted of a Hookean force with a piecewise stiffness. This could be done by adding a new set of files `gran_sub_mod_custom.h`:

```
#ifdef GRAN_SUB_MOD_CLASS
// clang-format off
GranSubModStyle(hooke/piecewise,GranSubModNormalHookePiecewise,NORMAL);
// clang-format on
#else

#ifndef GRAN_SUB_MOD_CUSTOM_H
#define GRAN_SUB_MOD_CUSTOM_H

#include "gran_sub_mod.h"
#include "gran_sub_mod_normal.h"

namespace LAMMPS_NS {
namespace Granular_NS {
  class GranSubModNormalHookePiecewise : public GranSubModNormal {
   public:
    GranSubModNormalHookePiecewise(class GranularModel *, class LAMMPS *);
    void coeffs_to_local() override;
    double calculate_forces() override;
   protected:
    double k1, k2, delta_switch;
  };
}    // namespace Granular_NS
}    // namespace LAMMPS_NS

#endif /*GRAN_SUB_MOD_CUSTOM_H */
#endif /*GRAN_SUB_MOD_CLASS_H */
```

and `gran_sub_mod_custom.cpp`

```
#include "gran_sub_mod_custom.h"
#include "gran_sub_mod_normal.h"
#include "granular_model.h"

using namespace LAMMPS_NS;
using namespace Granular_NS;

GranSubModNormalHookePiecewise::GranSubModNormalHookePiecewise(GranularModel *gm, LAMMPS␣
↪*lmp) :
    GranSubModNormal(gm, lmp)
{
  num_coeffs = 4;
}

/* ---------------------------------------------------------------------- */

void GranSubModNormalHookePiecewise::coeffs_to_local()
{
  k1 = coeffs[0];
  k2 = coeffs[1];
  damp = coeffs[2];
  delta_switch = coeffs[3];
}

/* ---------------------------------------------------------------------- */

double GranSubModNormalHookePiecewise::calculate_forces()
{
  double Fne;
  if (gm->delta >= delta_switch) {
    Fne = k1 * delta_switch + k2 * (gm->delta - delta_switch);
  } else {
    Fne = k1 * gm->delta;
  }
  return Fne;
}
```

## 3.17 Thermodynamic output options

The `Thermo` class computes and prints thermodynamic information to the screen and log file; see the files `thermo.cpp` and `thermo.h`.

There are four styles defined in `thermo.cpp`: "one", "multi", "yaml", and "custom". The "custom" style allows the user to explicitly list keywords for individual quantities to print when thermodynamic output is generated. The others have a fixed list of keywords. See the *thermo_style* command for a list of available quantities. The formatting of the "custom" style defaults to the "one" style, but can be adapted using *thermo_modify line*.

The thermo styles (one, multi, etc) are defined by lists of keywords with associated formats for integer and floating point numbers and identified by an enumerator constant. Adding a new style thus mostly requires defining a new list of keywords and the associated formats and then inserting the required output processing where the enumerators are identified. Search for the word "CUSTOMIZATION" with references to "thermo style" in the `thermo.cpp` file to see the locations where code will need to be added. The member function `Thermo::header()` prints output at the very beginning of a thermodynamic output block and can be used to print column headers or other front matter. The

member function `Thermo::footer()` prints output at the end of a thermodynamic output block. The formatting of the output is done by assembling a "line" (which may span multiple lines if the style inserts newline characters ("n" as in the "multi" style).

New thermodynamic keywords can also be added to `thermo.cpp` to compute new quantities for output. Search for the word "CUSTOMIZATION" with references to "keyword" in `thermo.cpp` to see the several locations where code will need to be added. Effectively, you need to define a member function that computes the property, add an if statement in `Thermo::parse_fields()` where the corresponding header string for the keyword and the function pointer is registered by calling the `Thermo::addfield()` method, and add an if statement in `Thermo::evaluate_keyword()` which is called from the `Variable` class when a thermo keyword is encountered.

**Note**

The third argument to `Thermo::addfield()` is a flag indicating whether the function for the keyword computes a floating point (FLOAT), regular integer (INT), or big integer (BIGINT) value. This information is used for formatting the thermodynamic output. Inside the function the result must then be stored either in the `dvalue`, `ivalue` or `bivalue` member variable, respectively.

Since the *thermo_style custom* command allows to use output of quantities calculated by *fixes*, *computes*, and *variables*, it may often be simpler to compute what you wish via one of those constructs, rather than by adding a new keyword to the thermo_style command.

## 3.18 Variable options

The `Variable` class computes and stores *variable* information in LAMMPS; see the file `variable.cpp`. The value associated with a variable can be periodically printed to the screen via the *print*, *fix print*, or *thermo_style custom* commands. Variables of style "equal" can compute complex equations that involve the following types of arguments:

```
thermo keywords = ke, vol, atoms, ...
other variables = v_a, v_myvar, ...
math functions = div(x,y), mult(x,y), add(x,y), ...
group functions = mass(group), xcm(group,x), ...
atom values = x[123], y[3], vx[34], ...
compute values = c_mytemp[0], c_thermo_press[3], ...
```

Adding keywords for the *thermo_style custom* command (which can then be accessed by variables) is discussed in the *Modify thermo* documentation.

Adding a new math function of one or two arguments can be done by editing one section of the `Variable::evaluate()` method. Search for the word "customize" to find the appropriate location.

Adding a new group function can be done by editing one section of the `Variable::evaluate()` method. Search for the word "customize" to find the appropriate location. You may need to add a new method to the Group class as well (see the `group.cpp` file).

Accessing a new atom-based vector can be done by editing one section of the Variable::evaluate() method. Search for the word "customize" to find the appropriate location.

Adding new *compute styles* (whose calculated values can then be accessed by variables) is discussed in the *Modify compute* documentation.

CHAPTER

FOUR

# INFORMATION FOR DEVELOPERS

This section describes the internal structure and basic algorithms of the LAMMPS code. This is a work in progress and additional information will be added incrementally depending on availability of time and requests from the LAMMPS user community.

A discussion of software engineering methods applied to LAMMPS over time and a general outline of the design and maintenance approach by the LAMMPS developers can be found in the paper LAMMPS: A Case Study For Applying Modern Software Engineering to an Established Research Software Package, in the USRSE'25 conference proceedings.

**On using AI coding agents**

Recently AI coding agents have seen massive improvements and thus have become quite popular and viable in assisting with the development of new code or with adding features to existing code. The LAMMPS developers have made some experiments and tried to assess the how effective and reliable the results are. Below are some of our observations.

A significant part of LAMMPS was written by taking some existing classes that perform similar calculations or operations and then replacing the functionality. That kind of task is well suited for a coding agent, but there are often little details that require some manual adjustments and thus careful checking of the resulting code is recommended. Similarly, porting styles to the KOKKOS or OPENMP package can be delegated in many cases, provided suitable template examples exist and the coding agent is asked to look at those. Github Copilot can also run LAMMPS unit tests and even add unit tests for new styles if a suitable template already exists. Asking a coding agent to create new functionality for which there is no similar code in either the LAMMPS sources or some other repository has a much lower chance of success.

In general, it is best to be explicit and verbose when assigning tasks to coding agents. It can also helpful to ask the agent to break up complex tasks and create a master plan. Avoid asking Github Copilot to do too much at once. For example, we have found that porting 5 or so styles to the KOKKOS or OPENMP package is reasonably efficient, without running into session timeouts. When a session is finished, asking the agent to summarize what it learned and including that context in the next session can also improve efficiency.

Another task that can often be passed successfully to a coding agent is the creation of suitable documentation. However, the results have a tendency to be overly verbose and sometimes redundant, so some manual editing to tighten it up is recommended.

Finally, coding agents are quite good at reviewing existing code. The LAMMPS developers use GitHub Copilot regularly to review submitted pull requests and over time the rate of incorrect advice has become rather low. When using an AI coding agent, it is generally recommended to use a different model for the code review than what was used to write the code.

To assist coding agents in their tasks, and particularly to provide instructions that GitHub Copilot will automatically read, we have created a file `.github/copilot-instructions.md` that can be used to inform coding agents about important steps when creating or reviewing code for LAMMPS.

When contributing code to LAMMPS that was at least partially created by a coding agent, please provide information in the pull request description to the extent of AI usage and which model or models were used.

## 4.1 Source files

The source files of the LAMMPS code are found in two directories of the distribution: `src` and `lib`. Most of the code is written in C++ but there are small a number of files in several other languages like C, Fortran, Shell script, or Python.

The core of the code is located in the `src` folder and its subdirectories. A sizable number of these files are in the `src` directory itself, but there are plenty of *packages*, which can be included or excluded when LAMMPS is built. See the *Include packages in build* section of the manual for more information about that part of the build process. LAMMPS currently supports building with *conventional makefiles* and through *CMake*. Those procedures differ in how packages are enabled or disabled for inclusion into a LAMMPS binary, so they cannot be mixed. The source files for each package are in all-uppercase subdirectories of the `src` folder, for example `src/MOLECULE` or `src/EXTRA-MOLECULE`. The `src/STUBS` subdirectory is not a package but contains a dummy MPI library, that is used when building a serial version of the code. The `src/MAKE` directory and its subdirectories contain makefiles with settings and flags for a variety of configuration and machines for the build process with traditional makefiles.

The `lib` directory contains the source code for several supporting libraries or files with configuration settings to use globally installed libraries, that are required by some optional packages. They may include python scripts that can transparently download additional source code on request. Each subdirectory, like `lib/colvars` or `lib/gpu`, contains the source files, some of which are in different languages such as Fortran or CUDA. These libraries included in the LAMMPS build, if the corresponding package is installed.

LAMMPS C++ source files almost always come in pairs, such as `src/run.cpp` (implementation file) and `src/run.h` (header file). Each pair of files defines a C++ class, for example the `LAMMPS_NS::Run` class, which contains the code invoked by the *run* command in a LAMMPS input script. As this example illustrates, source file and class names often have a one-to-one correspondence with a command used in a LAMMPS input script. Some source files and classes do not have a corresponding input script command, for example `src/force.cpp` and the `LAMMPS_NS::Force` class. They are discussed in the next section.

The names of all source files are in lower case and may use the underscore character '_' to separate words. Apart from bundled, externally maintained libraries, which may have different conventions, all C and C++ header files have a `.h` extension, all C++ files have a `.cpp` extension, and C files a `.c` extension. A few C++ classes and utility functions are implemented with only a `.h` file. Examples are the Pointers and Commands classes or the MathVec functions.

## 4.2 Class topology

Though LAMMPS has a lot of source files and classes, its class topology is not very deep, which can be seen from the *LAMMPS class topology* figure. In that figure, each name refers to a class and has a pair of associated source files in the `src` folder, for example the class `LAMMPS_NS::Memory` corresponds to the files `memory.cpp` and `memory.h`, or the class `LAMMPS_NS::AtomVec` corresponds to the files `atom_vec.cpp` and `atom_vec.h`. Full lines in the figure represent compositing: that is, the class at the base of the arrow holds a pointer to an instance of the class at the tip. Dashed lines instead represent inheritance: the class at the tip of the arrow is derived from the class at the base. Classes with a red boundary are not instantiated directly, but they represent the base classes for "styles". Those "styles" make up the bulk of the LAMMPS code and only a few representative examples are included in the figure, so it remains readable.

The *LAMMPS_NS::LAMMPS* class is the topmost class and represents what is generally referred to as an "instance of LAMMPS". It is a composite holding pointers to instances of other core classes providing the core functionality of the MD engine in LAMMPS and through them abstractions of the required operations. The constructor of the LAMMPS class will instantiate those instances, process the command-line flags, initialize MPI (if not already done) and set up file pointers for input and output. The destructor will shut everything down and free all associated memory. Thus code for the standalone LAMMPS executable in `main.cpp` simply initializes MPI, instantiates a single instance of LAMMPS

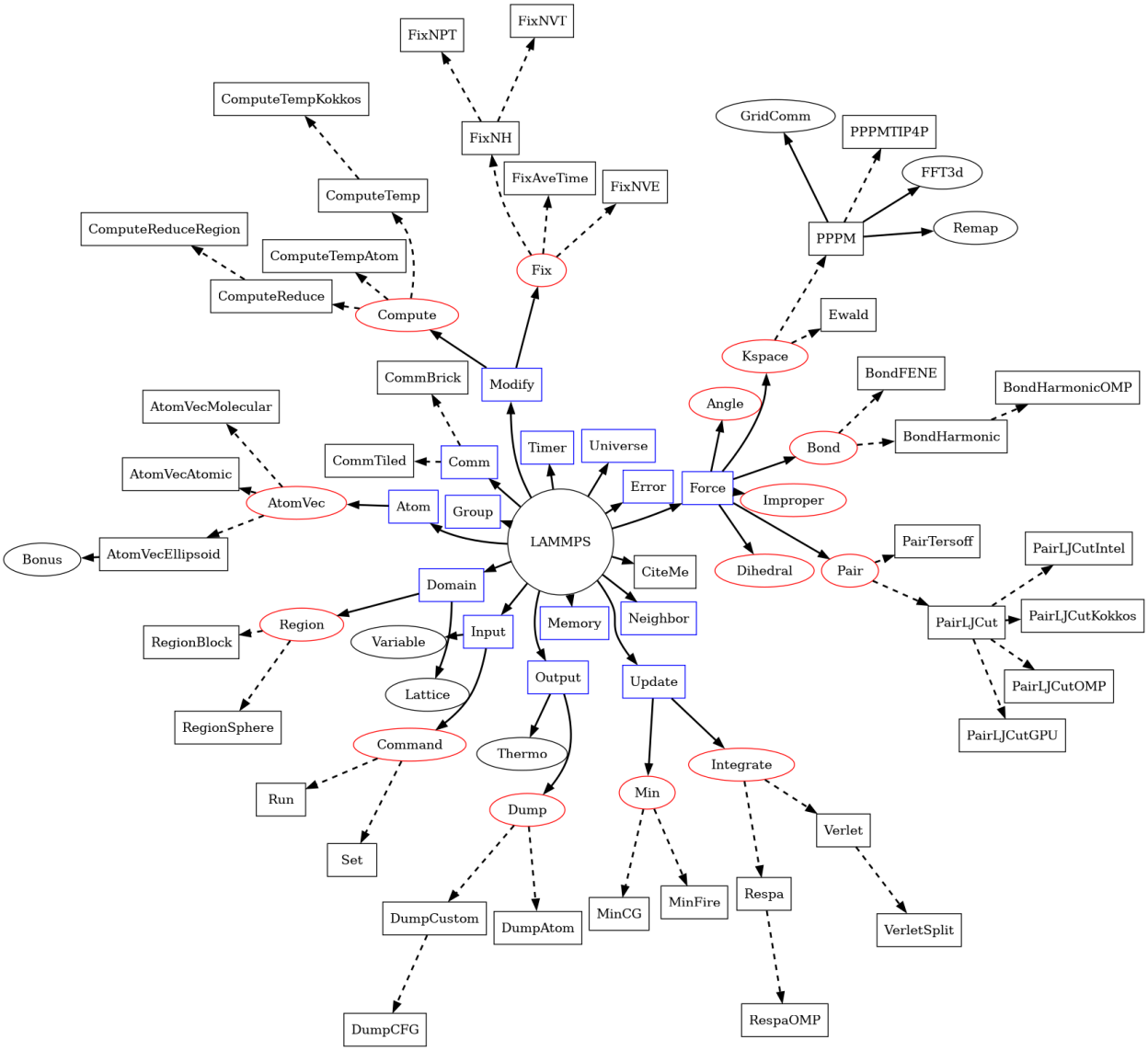



Fig. 1: LAMMPS class topology

This figure shows relations of base classes of the LAMMPS simulation package. Full lines indicate that a class holds an instance of the class it is pointing to; dashed lines point to derived classes that are given as examples of what classes may be instantiated during a LAMMPS run based on the input commands and accessed through the API define by their respective base classes. At the core is the *LAMMPS* class, which holds pointers to class instances with specific purposes. Those may hold instances of other classes, sometimes directly, or only temporarily, sometimes as derived classes or derived classes of derived classes, which may also hold instances of other classes.

while passing it the command-line flags and input script. It deletes the LAMMPS instance after the method reading the input returns and shuts down the MPI environment before it exits the executable.

The *LAMMPS_NS::Pointers* class is not shown in the *LAMMPS class topology* figure for clarity. It holds references to many of the members of the *LAMMPS_NS::LAMMPS*, so that all classes derived from *LAMMPS_NS::Pointers* have direct access to those references. From the class topology all classes with blue boundary are referenced in the Pointers class and all classes in the second and third columns, that are not listed as derived classes, are instead derived from *LAMMPS_NS::Pointers*. To initialize the pointer references in Pointers, a pointer to the LAMMPS class instance needs to be passed to the constructor. All constructors for classes derived from it, must do so and thus pass that pointer to the constructor for *LAMMPS_NS::Pointers*. The default constructor for *LAMMPS_NS::Pointers* is disabled to enforce this.

Since all storage is supposed to be encapsulated (there are a few exceptions), the LAMMPS class can also be instantiated multiple times by a calling code. Outside the aforementioned exceptions, those LAMMPS instances can be used alternately. As of the time of this writing (early 2023) LAMMPS is not yet sufficiently thread-safe for concurrent execution. When running in parallel with MPI, care has to be taken, that suitable copies of communicators are used to not create conflicts between different instances.

The LAMMPS class currently holds instances of 19 classes representing the core functionality. There are a handful of virtual parent classes in LAMMPS that define what LAMMPS calls `styles`. These are shaded red in the *LAMMPS class topology* figure. Each of these are parents of a number of child classes that implement the interface defined by the parent class. There are two main categories of these `styles`: some may only have one instance active at a time (e.g. atom, pair, bond, angle, dihedral, improper, kspace, comm) and there is a dedicated pointer variable for each of them in the corresponding composite class. Setups that require a mix of different such styles have to use a *hybrid* class instance that acts as a proxy, and manages and forwards calls to the corresponding sub-style class instances for the designated subset of atoms or data. The composite class may also have lists of class instances, e.g. `Modify` handles lists of compute and fix styles, while `Output` handles a list of dump class instances.

The exception to this scheme are the `command` style classes. These implement specific commands that can be invoked before, after, or in between runs. For these an instance of the class is created, its command() method called and then, after completion, the class instance deleted. Examples for this are the create_box, create_atoms, minimize, run, set, or velocity command styles.

For all those `styles`, certain naming conventions are employed: for the fix nve command the class is called FixNVE and the source files are `fix_nve.h` and `fix_nve.cpp`. Similarly, for fix ave/time we have FixAveTime and `fix_ave_time.h` and `fix_ave_time.cpp`. Style names are lower case and without spaces or special characters. A suffix or words are appended with a forward slash '/' which denotes a variant of the corresponding class without the suffix. To connect the style name and the class name, LAMMPS uses macros like: `AtomStyle()`, `PairStyle()`, `BondStyle()`, `RegionStyle()`, and so on in the corresponding header file. During configuration or compilation, files with the pattern `style_<name>.h` are created that consist of a list of include statements including all headers of all styles of a given type that are currently enabled (or "installed").

More details on individual classes in the *LAMMPS class topology* are as follows:

- The Memory class handles allocation of all large vectors and arrays.
- The Error class prints all (terminal) error and warning messages.
- The Universe class sets up one or more partitions of processors so that one or multiple simulations can be run, on the processors allocated for a run, e.g. by the mpirun command.
- The Input class reads and processes input (strings and files), stores variables, and invokes *commands*.
- Command style classes are derived from the Command class. They provide input script commands that perform one-time operations before/after/between simulations or which invoke a simulation. They are usually instantiated from within the Input class, its `command` method invoked, and then immediately destructed.
- The Finish class is instantiated to print statistics to the screen after a simulation is performed, by commands like run and minimize.

- The Special class walks the bond topology of a molecular system to find first, second, third neighbors of each atom. It is invoked by several commands, like *read_data*, *read_restart*, or *replicate*.
- The Atom class stores per-atom properties associated with atom styles. More precisely, they are allocated and managed by a class derived from the AtomVec class, and the Atom class simply stores pointers to them. The classes derived from AtomVec represent the different atom styles, and they are instantiated through the *atom_style* command.
- The Update class holds instances of an integrator and a minimizer class. The Integrate class is a parent style for the Verlet and r-RESPA time integrators, as defined by the *run_style* command. The Min class is a parent style for various energy minimizers.
- The Neighbor class builds and stores neighbor lists. The NeighList class stores a single list (for all atoms). A NeighRequest class instance is created by pair, fix, or compute styles when they need a particular kind of neighbor list and use the NeighRequest properties to select the neighbor list settings for the given request. There can be multiple instances of the NeighRequest class. The Neighbor class will try to optimize how the requests are processed. Depending on the NeighRequest properties, neighbor lists are constructed from scratch, aliased, or constructed by post-processing an existing list into sub-lists.
- The Comm class performs inter-processor communication, typically of ghost atom information. This usually involves MPI message exchanges with 6 neighboring processors in the 3d logical grid of processors mapped to the simulation box. There are two *communication styles*, enabling different ways to perform the domain decomposition.
- The Irregular class is used, when atoms may migrate to arbitrary processors.
- The Domain class stores the simulation box geometry, as well as geometric Regions and any user definition of a Lattice. The latter are defined by the *region* and *lattice* commands in an input script.
- The Force class computes various forces between atoms. The Pair parent class is for non-bonded or pairwise forces, which in LAMMPS also includes many-body forces such as the Tersoff 3-body potential if those are computed by walking pairwise neighbor lists. The Bond, Angle, Dihedral, Improper parent classes are styles for bonded interactions within a static molecular topology. The KSpace parent class is for computing long-range Coulombic interactions. One of its child classes, PPPM, uses the FFT3D and Remap classes to redistribute and communicate grid-based information across the parallel processors.
- The Modify class stores lists of class instances derived from the *Fix* and *Compute* base classes.
- The Group class manipulates groups that atoms are assigned to via the *group* command. It also has functions to compute various attributes of groups of atoms.
- The Output class is used to generate 3 kinds of output from a LAMMPS simulation: thermodynamic information printed to the screen and log file, dump file snapshots, and restart files. These correspond to the *Thermo*, *Dump*, and *WriteRestart* classes respectively. The Dump class is a base class, with several derived classes implementing various dump style variants.
- The Timer class logs timing information, output at the end of a run.

## 4.3 Code design

This section explains some code design choices in LAMMPS with the goal of helping developers write new code similar to the existing code. Please see the section on *Requirements for contributed code* for more specific recommendations and guidelines. While that section is organized more in the form of a checklist for code contributors, the focus here is on overall code design strategy, choices made between possible alternatives, and discussing some relevant C++ programming language constructs.

Historically, the basic design philosophy of the LAMMPS C++ code was a "C with classes" style. The motivation was to make it easy to modify LAMMPS for people without significant training in C++ programming. Data structures and code constructs were used that resemble the previous implementation(s) in Fortran. A contributing factor to this

choice was that at the time, C++ compilers were often not mature and some advanced features contained bugs or did not function as the standard required. There were also disagreements between compiler vendors as to how to interpret the C++ standard documents.

However, C++ compilers and the C++ programming language have advanced significantly. In 2020, the LAMMPS developers decided to require the C++11 standard as the minimum C++ language standard for LAMMPS. Since then, we have begun to replace C-style constructs with equivalent C++ functionality. This was taken either from the C++ standard library or implemented as custom classes or functions. The goal is to improve readability of the code and to increase code reuse through abstraction of commonly used functionality. In summer 2025, after the 22 July 2025 stable release, the minimum required C++ language standard was raised to C++17.

**Note**

Please note that as of summer 2025 there is still a sizable chunk of legacy code in LAMMPS that has not yet been refactored to reflect these style conventions in full. LAMMPS has a large code base and many contributors. There is also a hierarchy of precedence in which the code is adapted. Highest priority has been the code in the `src` folder, followed by code in packages in order of their popularity and complexity (simpler code gets adapted sooner), followed by code in the `lib` folder. Source code that is downloaded from external packages or libraries during compilation is not subject to the conventions discussed here.

### 4.3.1 Object-oriented code

LAMMPS is designed to be an object-oriented code. Each simulation is represented by an instance of the LAMMPS class. When running in parallel, each MPI process creates such an instance. This can be seen in the `main.cpp` file where the core steps of running a LAMMPS simulation are the following 3 lines of code:

```
LAMMPS *lammps = new LAMMPS(argc, argv, lammps_comm);
lammps->input->file();
delete lammps;
```

The first line creates a LAMMPS class instance and passes the command line arguments and the global communicator to its constructor. The second line triggers the LAMMPS instance to process the input (either from standard input or a provided input file) until the simulation ends. The third line deletes the LAMMPS instance. The remainder of the main.cpp file has code for error handling, MPI configuration, and other special features.

The basic LAMMPS class hierarchy which is created by the LAMMPS class constructor is shown in *LAMMPS class topology*. When input commands are processed, additional class instances are created, or deleted, or replaced. Likewise, specific member functions of specific classes are called to trigger actions such as creating atoms, computing forces, computing properties, time-propagating the system, or writing output.

#### Compositing and Inheritance

LAMMPS makes extensive use of the object-oriented programming (OOP) principles of *compositing* and *inheritance*. Classes like the `LAMMPS` class are a **composite** containing pointers to instances of other classes like `Atom`, `Comm`, `Force`, `Neighbor`, `Modify`, and so on. Each of these classes implements certain functionality by storing and manipulating data related to the simulation and providing member functions that trigger certain actions. Some of those classes like `Force` are themselves composites, containing instances of classes describing different force interactions. Similarly, the `Modify` class contains a list of `Fix` and `Compute` classes. If the input commands that correspond to these classes include the word *style*, then LAMMPS stores only a single instance of that class. E.g. *atom_style*, *comm_style*, *pair_style*, *bond_style*. If the input command does **not** include the word *style*, then there may be many instances of that class defined, for example *region*, *fix*, *compute*, *dump*.

**Inheritance** enables creation of *derived* classes that can share common functionality in their base class while providing a consistent interface. The derived classes replace (dummy or pure) functions in the base class. The higher level classes can then call those methods of the instantiated classes without having to know which specific derived class variant was

instantiated. In LAMMPS these derived classes are often referred to as "styles", e.g. pair styles, fix styles, atom styles and so on.

This is the origin of the flexibility of LAMMPS. For example, pair styles implement a variety of different non-bonded interatomic potentials functions. All details for the implementation of a potential are stored and executed in a single class.

As mentioned above, there can be multiple instances of classes derived from the `Fix` or `Compute` base classes. They represent a different facet of LAMMPS' flexibility, as they provide methods which can be called at different points within a timestep, as explained in the *How a timestep works* doc page. This allows the input script to tailor how a specific simulation is run, what diagnostic computations are performed, and how the output of those computations is further processed or output.

Additional code sharing is possible by creating derived classes from the derived classes (e.g., to implement an accelerated version of a pair style) where only a subset of the derived class methods are replaced with accelerated versions.

### Polymorphism

Polymorphism and dynamic dispatch are another OOP feature that play an important role in how LAMMPS selects what code to execute. In a nutshell, this is a mechanism where the decision of which member function to call from a class is determined at runtime and not when the code is compiled. To enable it, the function has to be declared as `virtual` and all corresponding functions in derived classes should use the `override` property. Below is a brief example.

```
class Base {
public:
 virtual ~Base() = default;
 void call();
 void normal();
 virtual void poly();
};

void Base::call() {
 normal();
 poly();
}

class Derived : public Base {
public:
 ~Derived() override = default;
 void normal();
 void poly() override;
};

// [....]

Base *base1 = new Base();
Base *base2 = new Derived();

base1->call();
base2->call();
```

The difference in behavior of the `normal()` and the `poly()` member functions is which of the two member functions is called when executing *base1->call()* versus *base2->call()*. Without polymorphism, a function within the base class can only call member functions within the same scope: that is, `Base::call()` will always call `Base::normal()`. But for the *base2->call()* case, the call of the virtual member function will be dispatched to `Derived::poly()` instead.

This mechanism results in calling functions that are within the scope of the class that was used to *create* the instance, even if they are assigned to a pointer for their base class. This is the desired behavior, and this way LAMMPS can even use styles that are loaded at runtime from a shared object file with the *plugin command*.

A special case of virtual functions are so-called pure functions. These are virtual functions that are initialized to 0 in the class declaration (see example below).

```
class Base {
public:
 virtual void pure() = 0;
};
```

This has the effect that an instance of the base class cannot be created and that derived classes **must** implement these functions. Many of the functions listed with the various class styles in the section *Modifying & extending LAMMPS* are pure functions. The motivation for this is to define the interface or API of the functions, but defer their implementation to the derived classes.

However, there are downsides to this. For example, calls to virtual functions from within a constructor, will *not* be in the scope of the derived class, and thus it is good practice to either avoid calling them or to provide an explicit scope such as `Base::poly()` or `Derived::poly()`. Furthermore, any destructors in classes containing virtual functions should be declared virtual too, so they will be processed in the expected order before types are removed from dynamic dispatch.

**Important Notes**

In order to be able to detect incompatibilities at compile time and to avoid unexpected behavior, it is crucial that all member functions that are intended to replace a virtual or pure function use the `override` property keyword. For the same reason, the use of overloads or default arguments for virtual functions should be avoided, as they lead to confusion over which function is supposed to override which, and which arguments need to be declared.

### Style Factories

In order to create class instances for different styles, LAMMPS often uses a programming pattern called *Factory*. Those are functions that create an instance of a specific derived class, say `PairLJCut` and return a pointer to the type of the common base class of that style, `Pair` in this case. To associate the factory function with the style keyword, a `std::map` class is used with function pointers indexed by their keyword (for example "lj/cut" for `PairLJCut` and "morse" for `PairMorse`). A couple of typedefs help keep the code readable, and a template function is used to implement the actual factory functions for the individual classes. Below is an example of such a factory function from the `Force` class as declared in `force.h` and implemented in `force.cpp`. The file `style_pair.h` is generated during compilation and includes all main header files (i.e. those starting with `pair_`) of pair styles and then the macro `PairStyle()` will associate the style name "lj/cut" with a factory function creating an instance of the `PairLJCut` class.

```
// from force.h
typedef Pair *(*PairCreator)(LAMMPS *);
typedef std::map<std::string, PairCreator> PairCreatorMap;
PairCreatorMap *pair_map;

// from force.cpp
template <typename S, typename T> static S *style_creator(LAMMPS *lmp)
{
  return new T(lmp);
}

// [...]
```

(continues on next page)

(continued from previous page)

```
pair_map = new PairCreatorMap();

#define PAIR_CLASS
#define PairStyle(key, Class) (*pair_map)[#key] = &style_creator<Pair, Class>;
#include "style_pair.h"
#undef PairStyle
#undef PAIR_CLASS

// from pair_lj_cut.h

#ifdef PAIR_CLASS
PairStyle(lj/cut,PairLJCut);
#else
// [...]
```

Similar code constructs are present in other files like `modify.cpp` and `modify.h` or `neighbor.cpp` and `neighbor.h`. Those contain similar macros and include `style_*.h` files for creating class instances of styles they manage.

### 4.3.2 I/O and output formatting

#### C-style stdio versus C++ style iostreams

LAMMPS uses the *stdio <https://cppreference.com/w/cpp/io/c.html>* library of the standard C library for reading from and writing to files and console instead of C++ iostreams. This is mainly motivated by better performance, better control over formatting, and less effort to achieve specific formatting.

Since mixing "stdio" and "iostreams" can lead to unexpected behavior, use of the latter is strongly discouraged. Output to the screen should *not* use the predefined `stdout` FILE pointer, but rather the `screen` and `logfile` FILE pointers managed by the LAMMPS class. Furthermore, output should generally only be done by MPI rank 0 (`comm->me == 0`). Output that is sent to both `screen` and `logfile` should use the *`utils::logmesg() convenience function`*.

We strongly discourage the use of stringstreams because the bundled {fmt} library or the C++ format library (for C++20 and later) and the customized tokenizer classes provide the same functionality in a cleaner way with better performance. This also helps maintain a consistent programming syntax with code from many different contributors.

#### Formatting with the {fmt} library and std::format

The LAMMPS source code currently includes a slightly modified copy of the {fmt} library, which is preferred over formatting with the "printf()" family of functions. When compiling for C++20 and later we switch to using the C++ format library. The namespace prefix currently remains `fmt::` through a small wrapper. In the future, this will be switched to `std::` when LAMMPS requires the C++20 standard as the minimum C++ standard. Thus only functionality compatible with the C++ format library for C++20 is accepted. Using `std::format` requires a fully C++20 compatible compiler (e.g. GCC 13 and later, Clang 14 and later, or MSVC 16.10 and later) and we use the __cpp_lib_format feature test macro to confirm the availability of `std::format`.

The primary reason for this choice is that it allows a typesafe default format for any type of supported data. This is particularly useful for formatting integers of a given size (32-bit or 64-bit) which may require different format strings depending on compile time settings or compilers/operating systems. Furthermore, {fmt} gives better performance, has more functionality, a familiar formatting syntax that has similarities to `format()` in Python, and provides a facility that can be used to integrate format strings and a variable number of arguments into custom functions in a much simpler way than the varargs mechanism of the C library.

Formatted strings are frequently created by calling the `fmt::format()` function, which will return a string as a `std::string` class instance. In contrast to the `%` placeholder in `printf()`, the {fmt} library uses `{}` to embed format

descriptors. In the simplest case, no additional characters are needed, as {fmt} will choose the default format based on the data type of the argument. Otherwise, the *utils::print()* function may be used instead of `printf()` or `fprintf()`. The equivalent std::print() function will become available in C++ 23. In addition, several LAMMPS output functions, that originally accepted a single string as argument have been overloaded to accept a format string with optional arguments as well (e.g., `Error::all()`, `Error::one()`, *utils::logmesg()*).

### Summary of the {fmt} format and std::format syntax

The syntax of the format string is "{[<argument id>][:<format spec>]}", where either the argument id or the format spec (separated by a colon ':') is optional. The argument id is usually a number starting from 0 that is the index to the arguments following the format string. By default, these are assigned in order (i.e. 0, 1, 2, 3, 4 etc.). The most common case for using argument id would be to use the same argument in multiple places in the format string without having to provide it as an argument multiple times. The argument id is rarely used in the LAMMPS source code.

More common is the use of a format specifier, which starts with a colon. This may optionally be followed by a fill character (default is ' '). If provided, the fill character **must** be followed by an alignment character ('<', '^', '>' for left, centered, or right alignment (default)). The alignment character may be used without a fill character. The next important format parameter would be the minimum width, which may be followed by a dot '.' and a precision for floating point numbers. The final character in the format string would be an indicator for the "presentation", i.e. 'd' for decimal presentation of integers, 'x' for hexadecimal, 'o' for octal, 'c' for character etc. This mostly follows the "printf()" scheme, but without requiring an additional length parameter to distinguish between different integer widths. The {fmt} library will detect those and adapt the formatting accordingly. For floating point numbers there are correspondingly, 'g' for generic presentation, 'e' for exponential presentation, and 'f' for fixed point presentation.

The format string "{:8}" would thus represent *any* type argument and be replaced by at least 8 characters; "{:<8}" would do this as left aligned, "{:^8}" as centered, "{:>8}" as right aligned. If a specific presentation is selected, the argument type must be compatible or else the {fmt} formatting code will throw an exception. Some format string examples are given below:

```
auto mesg = fmt::format("  CPU time: {:4d}:{:02d}:{:02d}\n", cpuh, cpum, cpus);
mesg = fmt::format("{:<8s}| {:<10.5g} | {:<10.5g} | {:<10.5g} |{:6.1f} |{:6.2f}\n",
                   label, time_min, time, time_max, time_sq, tmp);
utils::logmesg(lmp,"{:>6} = max # of 1-2 neighbors\n",maxall);
utils::logmesg(lmp,"Lattice spacing in x,y,z = {:.8} {:.8} {:.8}\n",
               xlattice,ylattice,zlattice);
```

which will create the following output lines:

```
CPU time:    0:02:16
Pair    | 2.0133     | 2.0133     | 2.0133     |   0.0 | 84.21
     4 = max # of 1-2 neighbors
Lattice spacing in x,y,z = 1.6795962 1.6795962 1.6795962
```

Finally, a special feature of the {fmt} library is that format parameters like the width or the precision may be also provided as arguments. In that case a nested format is used where a pair of curly braces (with an optional argument id) "{}" are used instead of the value, for example "{:{}d}" will consume two integer arguments, the first will be the value shown and the second the minimum width.

For more details and examples, please consult the {fmt} syntax documentation website or the corresponding C++ syntax reference. Since we plan to eventually transition from {fmt} to using `std::format()` of the C++ standard library, it is advisable to avoid using any extensions beyond what the C++20 standard offers.

### 4.3.3 JSON format input and output

Since LAMMPS version 12 June 2025, the LAMMPS source code includes a copy of the header-only JSON C++ library from https://json.nlohmann.me/. Same as with the {fmt} library described above some modification to the namespace has been made to avoid collisions with other uses of the same library, which may use a different, incompatible version. To have a uniform interface with other parts of LAMMPS, you should be using `#include "json.h"` or `#include "json_fwd.h"` (in header files). See the implementation of the *molecule command* for an example of using this library.

### 4.3.4 Memory management

Dynamical allocation of small data and objects can be done with the C++ commands "new" and "delete/delete[]". Large data should use the member functions of the `Memory` class, most commonly, `Memory::create()`, `Memory::grow()`, and `Memory::destroy()`, which provide variants for vectors, 2d arrays, 3d arrays, etc. These can also be used for small data.

The use of `malloc()`, `calloc()`, `realloc()` and `free()` directly is strongly discouraged. To simplify adapting legacy code into the LAMMPS code base the member functions `Memory::smalloc()`, `Memory::srealloc()`, and `Memory::sfree()` are available, which perform additional error checks for safety.

Use of these custom memory allocation functions is motivated by the following considerations:

- Memory allocation failures on *any* MPI rank during a parallel run will trigger an immediate abort of the entire parallel calculation.
- A failing "new" will trigger an exception, which is also captured by LAMMPS and triggers a global abort.
- Allocation of multidimensional arrays will be done in a C compatible fashion, but such that the storage of the actual data is stored in one large contiguous block. Thus, when MPI communication is needed, the data can be communicated directly (similar to Fortran arrays).
- The "destroy()" and "sfree()" functions may safely be called on NULL pointers.
- The "destroy()" functions will nullify the pointer variables, thus making "use after free" errors easy to detect.
- It is possible to use a larger than default memory alignment (not on all operating systems, since the allocated storage pointers must be compatible with `free()` for technical reasons).

In the practical implementation of code this means, that any pointer variables, that are class members should be initialized to a `nullptr` value in their respective constructors. That way, it is safe to call `Memory::destroy()` or `delete[]` on them before *any* allocation outside the constructor. This helps prevent memory leaks.

## 4.4 Parallel algorithms

LAMMPS is designed to enable running simulations in parallel using the MPI parallel communication standard with distributed data via domain decomposition. The parallelization aims to be efficient, and resulting in good strong scaling (= good speedup for the same system) and good weak scaling (= the computational cost of enlarging the system is proportional to the system size). Additional parallelization using GPUs or OpenMP can also be applied within the subdomain assigned to an MPI process. For clarity, most of the following illustrations show the 2d simulation case. The underlying algorithms in those cases, however, apply to both 2d and 3d cases equally well.

**Note**

The text and most of the figures in this chapter were adapted for the manual from the section on parallel algorithms in the *new LAMMPS paper*.

### 4.4.1 Partitioning

The underlying spatial decomposition strategy used by LAMMPS for distributed-memory parallelism is set with the *comm_style command* and can be either "brick" (a regular grid) or "tiled".

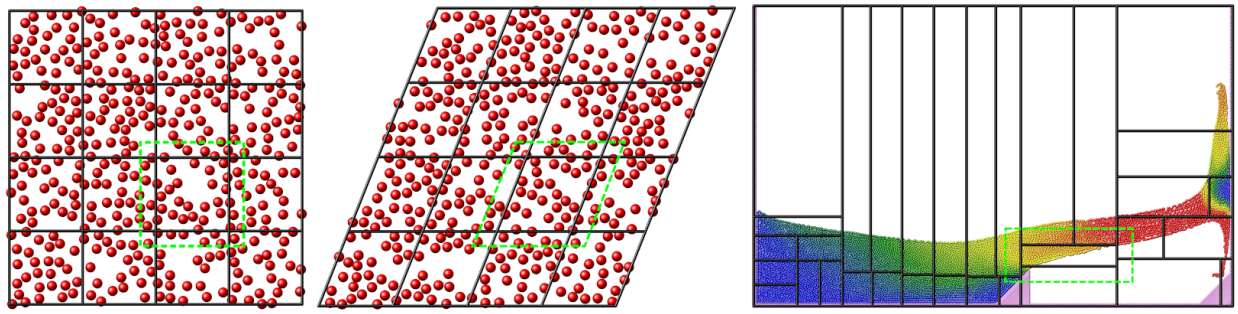


Fig. 2: Domain decomposition schemes

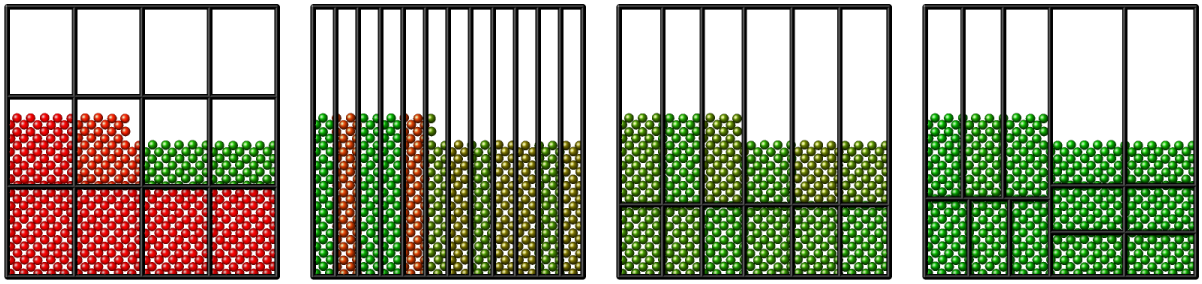
This figure shows the different kinds of domain decomposition used for MPI parallelization: "brick" on the left with an orthogonal (left) and a triclinic (middle) simulation domain, and a "tiled" decomposition (right). The black lines show the division into subdomains, and the contained atoms are "owned" by the corresponding MPI process. The green dashed lines indicate how subdomains are extended with "ghost" atoms up to the communication cutoff distance.

The LAMMPS simulation box is a 3d or 2d volume, which can be of orthogonal or triclinic shape, as illustrated in the *Domain decomposition schemes* figure for the 2d case. Orthogonal means the box edges are aligned with the $x$, $y$, $z$ Cartesian axes, and the box faces are thus all rectangular. Triclinic allows for a more general parallelepiped shape in which edges are aligned with three arbitrary vectors and the box faces are parallelograms. In each dimension, box faces can be periodic, or non-periodic with fixed or shrink-wrapped boundaries. In the fixed case, atoms which move outside the face are deleted; shrink-wrapped means the position of the box face adjusts continuously to enclose all the atoms.

For distributed-memory MPI parallelism, the simulation box is spatially decomposed (partitioned) into non-overlapping subdomains which fill the box. The default partitioning, "brick", is most suitable when atom density is roughly uniform, as shown in the left-side images of the *Domain decomposition schemes* figure. The subdomains comprise a regular grid, and all subdomains are identical in size and shape. Both the orthogonal and triclinic boxes can deform continuously during a simulation, e.g. to compress a solid or shear a liquid, in which case the processor subdomains likewise deform.

For models with non-uniform density, the number of particles per processor can be load-imbalanced with the default partitioning. This reduces parallel efficiency, as the overall simulation rate is limited by the slowest processor, i.e. the one with the largest computational load. For such models, LAMMPS supports multiple strategies to reduce the load imbalance:

- The processor grid decomposition is by default based on the simulation cell volume and tries to optimize the volume to surface ratio for the subdomains. This can be changed with the *processors command*.
- The parallel planes defining the size of the subdomains can be shifted with the *balance command*. Which can be done in addition to choosing a more optimal processor grid.
- The recursive bisectioning algorithm in combination with the "tiled" communication style can produce a partitioning with equal numbers of particles in each subdomain.



The pictures above demonstrate different decompositions for a 2d system with 12 MPI ranks. The atom colors indicate the load imbalance of each subdomain, with green being optimal and red the least optimal.

Due to the vacuum in the system, the default decomposition is unbalanced, with several MPI ranks without atoms (left). By forcing a 1x12x1 processor grid, every MPI rank does computations now, but the number of atoms per subdomain is still uneven, and the thin slice shape increases the amount of communication between subdomains (center left). With a 2x6x1 processor grid and shifting the subdomain divisions, the load imbalance is further reduced and the amount of communication required between subdomains is less (center right). And using the recursive bisectioning leads to further improved decomposition (right).

### 4.4.2 Communication

Following the selected partitioning scheme, all per-atom data is distributed across the MPI processes, which allows LAMMPS to handle very large systems provided it uses a correspondingly large number of MPI processes. To be able to compute the short-range interactions, MPI processes need not only access to the data of atoms they "own" but also information about atoms from neighboring subdomains, in LAMMPS referred to as "ghost" atoms. These are copies of atoms storing required per-atom data for up to the communication cutoff distance. The green dashed-line boxes in the *Domain decomposition schemes* figure illustrate the extended ghost-atom subdomain for one processor.

This approach is also used to implement periodic boundary conditions: atoms that lie within the cutoff distance across a periodic boundary are also stored as ghost atoms and taken from the periodic replication of the subdomain, which may be the same subdomain, e.g. if running in serial. As a consequence of this, force computation in LAMMPS is not subject to minimum image conventions and thus cutoffs may be larger than half the simulation domain.

Efficient communication patterns are needed to update the "ghost" atom data, since that needs to be done at every MD time step or minimization step. The diagrams of the *ghost atom communication* figure illustrate how ghost atom communication is performed in two stages for a 2d simulation (three in 3d) for both a regular and irregular partitioning of the simulation box. For the regular case (left) atoms are exchanged first in the $x$-direction, then in $y$, with four neighbors in the grid of processor subdomains.

In the $x$ stage, processor ranks 1 and 2 send owned atoms in their red-shaded regions to rank 0 (and vice versa). Then in the $y$ stage, ranks 3 and 4 send atoms in their blue-shaded regions to rank 0, which includes ghost atoms they received in the $x$ stage. Rank 0 thus acquires all its ghost atoms; atoms in the solid blue corner regions are communicated twice before rank 0 receives them.

For the irregular case (right) the two stages are similar, but a processor can have more than one neighbor in each direction. In the $x$ stage, MPI ranks 1,2,3 send owned atoms in their red-shaded regions to rank 0 (and vice versa). These include only atoms between the lower and upper $y$-boundary of rank 0's subdomain. In the $y$ stage, ranks 4,5,6 send atoms in their blue-shaded regions to rank 0. This may include ghost atoms they received in the $x$ stage, but only if they are needed by rank 0 to fill its extended ghost atom regions in the +/-$y$ directions (blue rectangles). Thus, in this case, ranks 5 and 6 do not include ghost atoms they received from each other (in the $x$ stage) in the atoms they send to rank 0. The key point is that while the pattern of communication is more complex in the irregular partitioning case, it can still proceed in two stages (three in 3d) via atom exchanges with only neighboring processors.

When attributes of owned atoms are sent to neighboring processors to become attributes of their ghost atoms, LAMMPS calls this a "forward" communication. On timesteps when atoms migrate to new owning processors and neighbor lists are rebuilt, each processor creates a list of its owned atoms which are ghost atoms in each of its neighbor processors.

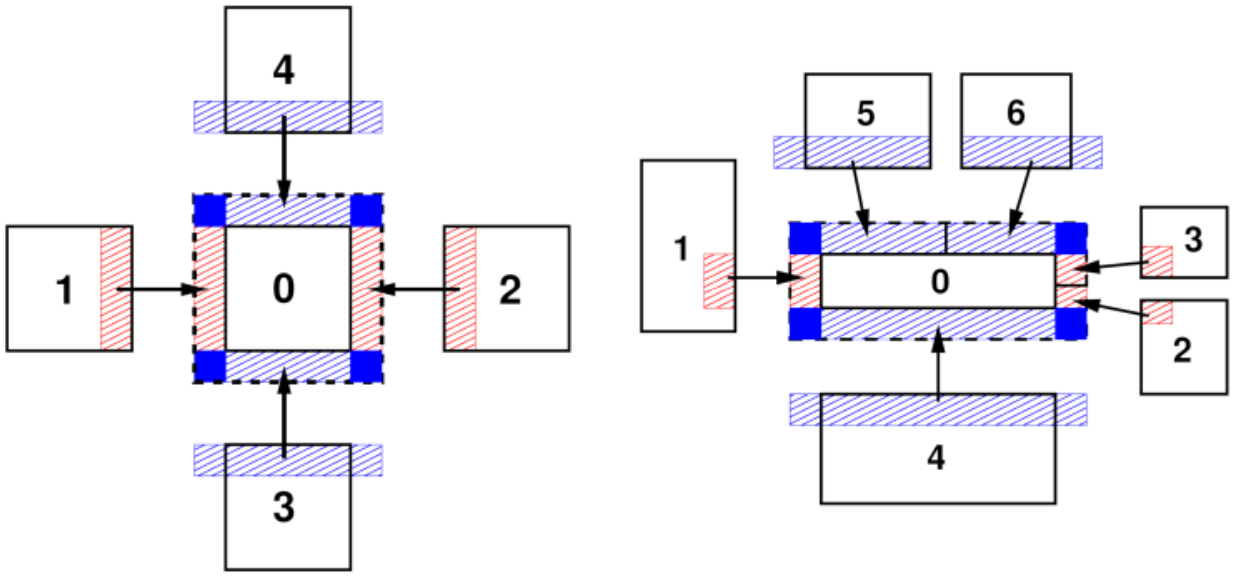



Fig. 3: ghost atom communication

This figure shows the ghost atom communication patterns between subdomains for "brick" (left) and "tiled" communication styles for 2d simulations. The numbers indicate MPI process ranks. Here the subdomains are drawn spatially separated for clarity. The dashed-line box is the extended subdomain of processor 0 which includes its ghost atoms. The red- and blue-shaded boxes are the regions of communicated ghost atoms.

These lists are used to pack per-atom coordinates (for example) into message buffers in subsequent steps until the next reneighboring.

A "reverse" communication is when computed ghost atom attributes are sent back to the processor who owns the atom. This is used (for example) to sum partial forces on ghost atoms to the complete force on owned atoms. The order of the two stages described in the *ghost atom communication* figure is inverted, and the same lists of atoms are used to pack and unpack message buffers with per-atom forces. When a received buffer is unpacked, the ghost forces are summed to owned atom forces. As in forward communication, forces on atoms in the four blue corners of the diagrams are sent, received, and summed twice (once at each stage) before owning processors have the full force.

These two operations are used in many places within LAMMPS aside from exchange of coordinates and forces, for example by manybody potentials to share intermediate per-atom values, or by rigid-body integrators to enable each atom in a body to access body properties. Here are additional details about how these communication operations are performed in LAMMPS:

- When exchanging data with different processors, forward and reverse communication is done using `MPI_Send()` and `MPI_IRecv()` calls. If a processor is "exchanging" atoms with itself, only the pack and unpack operations are performed, e.g. to create ghost atoms across periodic boundaries when running on a single processor.
- For forward communication of owned atom coordinates, periodic box lengths are added and subtracted when the receiving processor is across a periodic boundary from the sender. There is then no need to apply a minimum image convention when calculating distances between atom pairs when building neighbor lists or computing forces.
- The cutoff distance for exchanging ghost atoms is typically equal to the neighbor cutoff. But it can also set to a larger value if needed, e.g. half the diameter of a rigid body composed of multiple atoms or over 3x the length of a stretched bond for dihedral interactions. It can also exceed the periodic box size. For the regular communication pattern (left), if the cutoff distance extends beyond a neighbor processor's subdomain, then multiple exchanges are performed in the same direction. Each exchange is with the same neighbor processor, but buffers are packed/unpacked using a different list of atoms. For forward communication, in the first exchange, a processor sends only owned atoms. In subsequent exchanges, it sends ghost atoms received in previous exchanges. For

the irregular pattern (right) overlaps of a processor's extended ghost-atom subdomain with all other processors in each dimension are detected.

### 4.4.3 Neighbor lists

To compute forces efficiently, each processor creates a Verlet-style neighbor list which enumerates all pairs of atoms *i,j* (*i* = owned, *j* = owned or ghost) with separation less than the applicable neighbor list cutoff distance. In LAMMPS, the neighbor lists are stored in a multiple-page data structure; each page is a contiguous chunk of memory which stores vectors of neighbor atoms *j* for many *i* atoms. This allows pages to be incrementally allocated or deallocated in blocks as needed. Neighbor lists typically consume the most memory of any data structure in LAMMPS. The neighbor list is rebuilt (from scratch) once every few timesteps, then used repeatedly each step for force or other computations. The neighbor cutoff distance is $R_n = R_f + \Delta_s$, where $R_f$ is the (largest) force cutoff defined by the interatomic potential for computing short-range pairwise or manybody forces and $\Delta_s$ is a "skin" distance that allows the list to be used for multiple steps assuming that atoms do not move very far between consecutive time steps. Typically, the code triggers reneighboring when any atom has moved half the skin distance since the last reneighboring; this and other options of the neighbor list rebuild can be adjusted with the *neigh_modify* command.

On steps when reneighboring is performed, atoms which have moved outside their owning processor's subdomain are first migrated to new processors via communication. Periodic boundary conditions are also (only) enforced on these steps to ensure each atom is re-assigned to the correct processor. After migration, the atoms owned by each processor are stored in a contiguous vector. Periodically, each processor spatially sorts owned atoms within its vector to reorder it for improved cache efficiency in force computations and neighbor list building. For that, atoms are spatially binned and then reordered so that atoms in the same bin are adjacent in the vector. Atom sorting can be disabled or its settings modified with the *atom_modify* command.

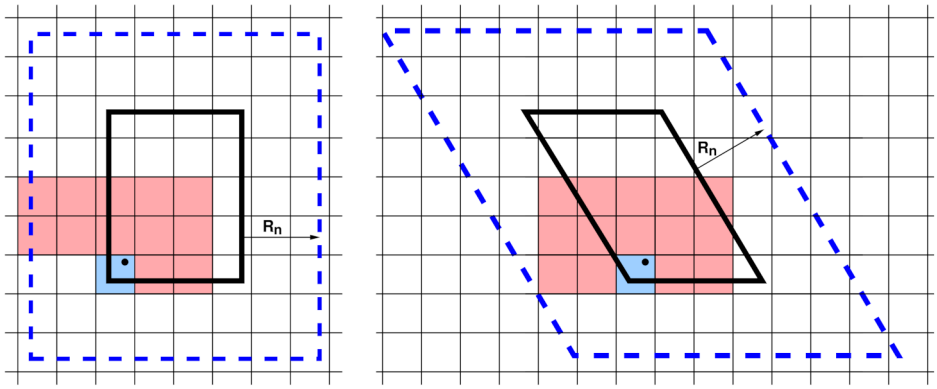



Fig. 4: neighbor list stencils

A 2d simulation subdomain (thick black line) and the corresponding ghost atom cutoff region (dashed blue line) for both orthogonal (left) and triclinic (right) domains. A regular grid of neighbor bins (thin lines) overlays the entire simulation domain and need not align with subdomain boundaries; only the portion overlapping the augmented subdomain is shown. In the triclinic case, it overlaps the bounding box of the tilted rectangle. The blue- and red-shaded bins represent a stencil of bins searched to find neighbors of a particular atom (black dot).

To build a local neighbor list in linear time, the simulation domain is overlaid (conceptually) with a regular 3d (or 2d) grid of neighbor bins, as shown in the *neighbor list stencils* figure for 2d models and a single MPI processor's subdomain. Each processor stores a set of neighbor bins which overlap its subdomain, extended by the neighbor cutoff distance $R_n$. As illustrated, the bins need not align with processor boundaries; an integer number in each dimension is fit to the size of the entire simulation box.

Most often, LAMMPS builds what is called a "half" neighbor list where each *i,j* neighbor pair is stored only once, with either atom *i* or *j* as the central atom. The build can be done efficiently by using a pre-computed "stencil" of bins around a central origin bin which contains the atom whose neighbors are being searched for. A stencil is simply a list of integer offsets in *x,y,z* of nearby bins surrounding the origin bin which are close enough to contain any neighbor atom *j*

within a distance $R_n$ from any atom $i$ in the origin bin. Note that for a half neighbor list, the stencil can be asymmetric, since each atom only need store half its nearby neighbors.

These stencils are illustrated in the figure for a half list and a bin size of $\frac{1}{2}R_n$. There are 13 red+blue stencil bins in 2d (for the orthogonal case, 15 for triclinic). In 3d there would be 63, 13 in the plane of bins that contain the origin bin and 25 in each of the two planes above it in the $z$ direction (75 for triclinic). The triclinic stencil has extra bins because the bins tile the bounding box of the entire triclinic domain, and thus are not periodic with respect to the simulation box itself. The stencil and logic for determining which *i,j* pairs to include in the neighbor list are altered slightly to account for this.

To build a neighbor list, a processor first loops over its "owned" plus "ghost" atoms and assigns each to a neighbor bin. This uses an integer vector to create a linked list of atom indices within each bin. It then performs a triply-nested loop over its owned atoms $i$, the stencil of bins surrounding atom $i$'s bin, and the $j$ atoms in each stencil bin (including ghost atoms). If the distance $r_{ij} < R_n$, then atom $j$ is added to the vector of atom $i$'s neighbors.

Here are additional details about neighbor list build options LAMMPS supports:

- The choice of bin size is an option; a size half of $R_n$ has been found to be optimal for many typical cases. Smaller bins incur additional overhead to loop over; larger bins require more distance calculations. Note that for smaller bin sizes, the 2d stencil in the figure would be of a more semicircular shape (hemispherical in 3d), with bins near the corners of the square eliminated due to their distance from the origin bin.
- Depending on the interatomic potential(s) and other commands used in an input script, multiple neighbor lists and stencils with different attributes may be needed. This includes lists with different cutoff distances, e.g. for force computation versus occasional diagnostic computations such as a radial distribution function, or for the r-RESPA time integrator which can partition pairwise forces by distance into subsets computed at different time intervals. It includes "full" lists (as opposed to half lists) where each *i,j* pair appears twice, stored once with *i* and *j*, and which use a larger symmetric stencil. It also includes lists with partial enumeration of ghost atom neighbors. The full and ghost-atom lists are used by various manybody interatomic potentials. Lists may also use different criteria for inclusion of a pairwise interaction. Typically, this simply depends only on the distance between two atoms and the cutoff distance. But for finite-size coarse-grained particles with individual diameters (e.g. polydisperse granular particles), it can also depend on the diameters of the two particles.
- When using *pair style hybrid* multiple sub-lists of the master neighbor list for the full system need to be generated, one for each sub-style, which contains only the *i,j* pairs needed to compute interactions between subsets of atoms for the corresponding potential. This means, not all *i* or *j* atoms owned by a processor are included in a particular sub-list.
- Some models use different cutoff lengths for pairwise interactions between different kinds of particles, which are stored in a single neighbor list. One example is a solvated colloidal system with large colloidal particles where colloid/colloid, colloid/solvent, and solvent/solvent interaction cutoffs can be dramatically different. Another is a model of polydisperse finite-size granular particles; pairs of particles interact only when they are in contact with each other. Mixtures with particle size ratios as high as 10-100x may be used to model realistic systems. Efficient neighbor list building algorithms for these kinds of systems are available in LAMMPS. They include a method which uses different stencils for different cutoff lengths and trims the stencil to only include bins that straddle the cutoff sphere surface. More recently a method which uses both multiple stencils and multiple bin sizes was developed; it builds neighbor lists efficiently for systems with particles of any size ratio, though other considerations (timestep size, force computations) may limit the ability to model systems with huge polydispersity.
- For small and sparse systems and as a fallback method, LAMMPS also supports neighbor list construction without binning by using a full $O(N^2)$ loop over all *i,j* atom pairs in a subdomain when using the *neighbor nsq* command.
- Dependent on the "pair" setting of the *newton* command, the "half" neighbor lists may contain **all** pairs of atoms where atom $j$ is a ghost atom (i.e. when the newton pair setting is *off*). For the newton pair *on* setting the atom $j$ is only added to the list if its $z$ coordinate is larger, or if equal the $y$ coordinate is larger, and that is equal, too, the $x$ coordinate is larger. For homogeneously dense systems, that will result in picking neighbors from a same size sector in always the same direction relative to the "owned" atom, and thus it should lead to similar length neighbor lists and reduce the chance of a load imbalance.

### 4.4.4 Long-range interactions

For charged systems, LAMMPS can compute long-range Coulombic interactions via the FFT-based particle-particle/particle-mesh (PPPM) method implemented in *kspace style pppm and its variants*. For that Coulombic interactions are partitioned into short- and long-range components. The short-ranged portion is computed in real space as a loop over pairs of charges within a cutoff distance, using neighbor lists. The long-range portion is computed in reciprocal space using a kspace style. For the PPPM implementation the simulation cell is overlaid with a regular FFT grid in 3d. It proceeds in several stages:

a) each atom's point charge is interpolated to nearby FFT grid points,

b) a forward 3d FFT is performed,

c) a convolution operation is performed in reciprocal space,

d) one or more inverse 3d FFTs are performed, and

e) electric field values from grid points near each atom are interpolated to compute its forces.

For any of the spatial-decomposition partitioning schemes each processor owns the brick-shaped portion of FFT grid points contained within its subdomain. The two interpolation operations use a stencil of grid points surrounding each atom. To accommodate the stencil size, each processor also stores a few layers of ghost grid points surrounding its brick. Forward and reverse communication of grid point values is performed similar to the corresponding *atom data communication*. In this case, electric field values on owned grid points are sent to neighboring processors to become ghost point values. Likewise charge values on ghost points are sent and summed to values on owned points.

For triclinic simulation boxes, the FFT grid planes are parallel to the box faces, but the mapping of charge and electric field values to/from grid points is done in reduced coordinates where the tilted box is conceptually a unit cube, so that the stencil and FFT operations are unchanged. However the FFT grid size required for a given accuracy is larger for triclinic domains than it is for orthogonal boxes.

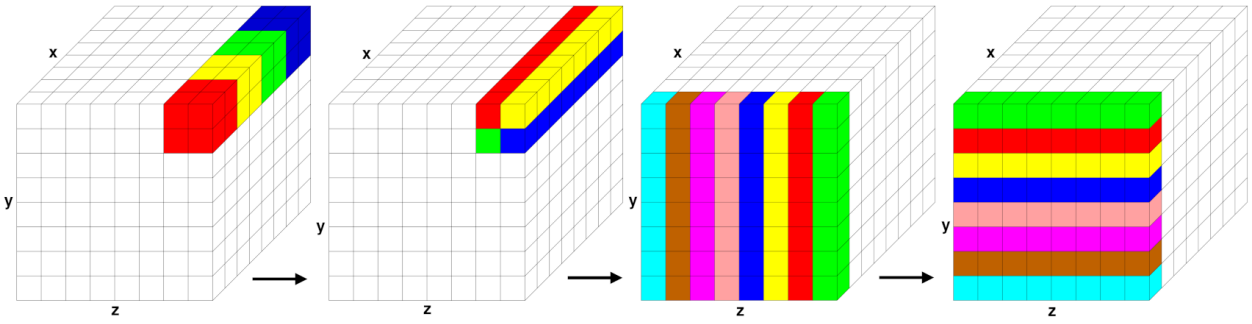



Fig. 5: Parallel FFT in PPPM

Stages of a parallel FFT for a simulation domain overlaid with an 8x8x8 3d FFT grid, partitioned across 64 processors. Within each of the 4 diagrams, grid cells of the same color are owned by a single processor; for simplicity, only cells owned by 4 or 8 of the 64 processors are colored. The two images on the left illustrate brick-to-pencil communication. The two images on the right illustrate pencil-to-pencil communication, which in this case transposes the $y$ and $z$ dimensions of the grid.

Parallel 3d FFTs require substantial communication relative to their computational cost. A 3d FFT is implemented by a series of 1d FFTs along the $x$-, $y$-, and $z$-direction of the FFT grid. Thus, the FFT grid cannot be decomposed like atoms into 3 dimensions for parallel processing of the FFTs but only in 1 (as planes) or 2 (as pencils) dimensions and in between the steps the grid needs to be transposed to have the FFT grid portion "owned" by each MPI process complete in the direction of the 1d FFTs it has to perform. LAMMPS uses the pencil-decomposition algorithm as shown in the *Parallel FFT in PPPM* figure.

Initially (far left), each processor owns a brick of same-color grid cells (actually grid points) contained within in its subdomain. A brick-to-pencil communication operation converts this layout to 1d pencils in the $x$-dimension (center

left). Again, cells of the same color are owned by the same processor. Each processor can then compute a 1d FFT on each pencil of data it wholly owns using a call to the configured FFT library. A pencil-to-pencil communication then converts this layout to pencils in the $y$ dimension (center right) which effectively transposes the $x$ and $y$ dimensions of the grid, followed by 1d FFTs in $y$. A final transpose of pencils from $y$ to $z$ (far right) followed by 1d FFTs in $z$ completes the forward FFT. The data is left in a $z$-pencil layout for the convolution operation. One or more inverse FFTs then perform the sequence of 1d FFTs and communication steps in reverse order; the final layout of resulting grid values is the same as the initial brick layout.

Each communication operation within the FFT (brick-to-pencil or pencil-to-pencil or pencil-to-brick) converts one tiling of the 3d grid to another, where a tiling in this context means an assignment of a small brick-shaped subset of grid points to each processor, the union of which comprise the entire grid. The parallel fftMPI library written for LAMMPS allows arbitrary definitions of the tiling so that an irregular partitioning of the simulation domain can use it directly. Transforming data from one tiling to another is implemented in *fftMPI* using point-to-point communication, where each processor sends data to a few other processors, since each tile in the initial tiling overlaps with a handful of tiles in the final tiling.

The transformations could also be done using collective communication across all $P$ processors with a single call to `MPI_Alltoall()`, but this is typically much slower. However, for the specialized brick and pencil tiling illustrated in *Parallel FFT in PPPM* figure, collective communication across the entire MPI communicator is not required. In the example, an $8^3$ grid with 512 grid cells is partitioned across 64 processors; each processor owns a 2x2x2 3d brick of grid cells. The initial brick-to-pencil communication (upper left to upper right) only requires collective communication within subgroups of 4 processors, as illustrated by the 4 colors. More generally, a brick-to-pencil communication can be performed by partitioning $P$ processors into $P^{\frac{2}{3}}$ subgroups of $P^{\frac{1}{3}}$ processors each. Each subgroup performs collective communication only within its subgroup. Similarly, pencil-to-pencil communication can be performed by partitioning $P$ processors into $P^{\frac{1}{2}}$ subgroups of $P^{\frac{1}{2}}$ processors each. This is illustrated in the figure for the $y \Rightarrow z$ communication (center). An eight-processor subgroup owns the front $yz$ plane of data and performs collective communication within the subgroup to transpose from a $y$-pencil to $z$-pencil layout.

LAMMPS invokes point-to-point communication by default, but also provides the option of partitioned collective communication when using a *kspace_modify collective yes* command to switch to that mode. In the latter case, the code detects the size of the disjoint subgroups and partitions the single $P$-size communicator into multiple smaller communicators, each of which invokes collective communication. Testing on a large IBM Blue Gene/Q machine at Argonne National Labs showed a significant improvement in FFT performance for large processor counts; partitioned collective communication was faster than point-to-point communication or global collective communication involving all $P$ processors.

Here are some additional details about FFTs for long-range and related grid/particle operations that LAMMPS supports:

- The fftMPI library allows each grid dimension to be a multiple of small prime factors (2,3,5), and allows any number of processors to perform the FFT. The resulting brick and pencil decompositions are thus not always as well-aligned, but the size of subgroups of processors for the two modes of communication (brick/pencil and pencil/pencil) still scale as $O(P^{\frac{1}{3}})$ and $O(P^{\frac{1}{2}})$.
- For efficiency in performing 1d FFTs, the grid transpose operations illustrated in Figure *Parallel FFT in PPPM* also involve reordering the 3d data so that a different dimension is contiguous in memory. This reordering can be done during the packing or unpacking of buffers for MPI communication.
- For large systems and particularly many MPI processes, the dominant cost for parallel FFTs is often the communication, not the computation of 1d FFTs, even though the latter scales as $N \log(N)$ in the number of grid points $N$ per grid direction. This is due to the fact that only a 2d decomposition into pencils is possible while atom data (and their corresponding short-range force and energy computations) can be decomposed efficiently in 3d.

  Reducing the number of MPI processes involved in the MPI communication will reduce this kind of overhead. By using a *hybrid MPI + OpenMP parallelization* it is still possible to use all processes for parallel computation. It will use OpenMP parallelization inside the MPI domains. While that may have a lower parallel efficiency for some part of the computation, that can be less than the communication overhead in the 3d FFTs.

  As an alternative, it is also possible to start a *multi-partition* calculation and then use the *verlet/split integrator*

to perform the PPPM computation on a dedicated, separate partition of MPI processes. This uses an integer "1:$p$" mapping of $p$ subdomains of the atom decomposition to one subdomain of the FFT grid decomposition and where pairwise non-bonded and bonded forces and energies are computed on the larger partition and the PPPM kspace computation concurrently on the smaller partition.

- LAMMPS also implements PPPM-based solvers for other long-range interactions, dipole and dispersion (Lennard-Jones), which can be used in conjunction with long-range Coulombics for point charges.
- LAMMPS implements a `GridComm` class which overlays the simulation domain with a regular grid, partitions it across processors in a manner consistent with processor subdomains, and provides methods for forward and reverse communication of owned and ghost grid point values. It is used for PPPM as an FFT grid (as outlined above) and also for the MSM algorithm, which uses a cascade of grid sizes from fine to coarse to compute long-range Coulombic forces. The GridComm class is also useful for models where continuum fields interact with particles. For example, the two-temperature model (TTM) defines heat transfer between atoms (particles) and electrons (continuum gas) where spatial variations in the electron temperature are computed by finite differences of a discretized heat equation on a regular grid. The *fix ttm/grid* command uses the `GridComm` class internally to perform its grid operations on a distributed grid instead of the original *fix ttm* which uses a replicated grid.

### 4.4.5 OpenMP Parallelism

The styles in the INTEL, KOKKOS, and OPENMP packages offer to use OpenMP thread parallelism to predominantly distribute loops over local data and thus follow an orthogonal parallelization strategy to the decomposition into spatial domains used by the *MPI partitioning*. For clarity, this section discusses only the implementation in the OPENMP package, as it is the simplest. The INTEL and KOKKOS packages offer additional options and are significantly more complex since they support more features like vectorization, accelerator support and a choice of using double, single, and mixed precision floating-point math.

A practical discussion of the actual implementation with source code examples can be found in the manual section on *writing styles for the OPENMP package*.

#### Rationale for OpenMP Integration

While LAMMPS is already well-parallelized using MPI-based domain decomposition, OpenMP provides additional benefits, particularly on modern multi-core hardware and at high node counts:

- **Efficient Resource Utilization**: On multi-core machines, MPI tasks can be communication-bandwidth limited because all MPI communication has to share the same network link for communication between nodes. Running fewer MPI tasks and using OpenMP threads can improve performance by reducing communication overhead and memory contention. However, the in-node memory bandwidth demands for multi-threading are higher, since domain decomposition promotes CPU cache locality, while the multi-thread implementation uses replicated per-thread storage to avoid data races. This has the additional overhead or requiring a reduction, which is most efficient for small numbers of threads.
- **PPPM Scaling**: Long-range electrostatics solvers like PPPM have scaling limitations with MPI communication at high node counts because the 3d-FFTs require all-to-all communications and with smaller domains the ratio of computation versus communication become less favorable. Using OpenMP threads can allow running these solvers on a subset of MPI tasks, improving overall scaling.
- **Parallelization Granularity**: OpenMP allows parallelization over particles/neighbors within a node, providing a finer grain of parallelism than domain decomposition alone.
- **Load Balancing Considerations**: The domain decomposition based MPI parallelization implicitly assumes that the simulated system is homogeneous and thus roughly the same amount of work needs to be done by each MPI rank. However, for inhomogeneous systems, this is not always the case and the available load-balancing options have limitations, too. By shifting the parallelization to OpenMP, the subdomains per MPI rank become larger and load balancing is usually more effective then.

- **Capability Computing**: Hybrid MPI+OpenMP is often essential for achieving optimal performance on large HPC clusters or supercomputers.

### Design of the OPENMP Package

One of the key decisions when implementing the OPENMP package was to keep the changes to the source code small, so that it would be easier to maintain the code and keep it in sync with the non-threaded standard implementation. This is achieved by:

- **Inheritance**: Making the OPENMP version a derived class from the regular version (e.g. `PairLJCutOMP` from `PairLJCut`) and only overriding methods that are multi-threaded or need to be modified (similar to what was done before in the OPT package).
- **Minimal Modification**: Keeping the structure in the modified code very similar so that side-by-side comparisons are still useful.
- **Helper Classes**: Offloading multi-thread support functions into three separate classes:
  - `ThrOMP`: Provides multi-thread aware functionality like "_thr" variants of tally functions via multiple inheritance.
  - `ThrData`: Manages per-thread data structures to avoid "false sharing" slowdowns.
  - `FixOMP`: Manages the global multi-thread state and settings activated by the *package omp* command.

### Avoiding data races

A key problem when implementing thread parallelism in an MD code is to avoid data races when updating accumulated per-atom properties like forces, energies, stresses, and torques. When interactions are computed, they always involve multiple atoms and thus there are race conditions when multiple threads want to update per-atom data of the same atoms.

Without a significant rewrite of the code, there are three main approaches to avoid data races.

- Use locks or **atomic operations** to guarantee that only one thread at a time updates per-thread data. This is most commonly used when the access conflicts are rare.
- Compute per-atom properties **multiple times** for all threads that "own" one or more atoms of a tuple and store only the data for those "owned" atom. This is most effective for a large number of threads and thus commonly used with GPU acceleration.
- Have **per-thread copies** of the per-atom data and update them independently and use a reduction to combine the per-thread data into the regular per-atom storage. This requires additional steps to set up, manage, clear, and combine the collected data.

The OPENMP package uses **replicated per-thread data structures** because this approach:

- retains the performance for the single-thread case unlike the other options,
- keeps the code maintainable and similar to the non-threaded version,
- and is most efficient for a small number of threads (2-8), which is a common use case, since OpenMP is typically the secondary parallelization option after MPI.

### Loop scheduling

Multi-thread parallelization is applied by distributing (outer) loops statically across threads. Typically, this is the loop over local atoms *i* when processing *i,j* pairs of atoms from a neighbor list or the list of bonds, angles, dihedrals, or impropers. Since neighbor lists typically result in a similar number of neighbors per atom (for homogeneous systems), significant load imbalances across threads are uncommon.

#### Neighbor list parallelization

In addition to force computations, neighbor list generation is parallelized. Each thread operates on a different chunk of "owned" atoms and manages its own set of neighbor "pages" using an instance of the `MyPage` page allocator.

## 4.5 Accessing per-atom data

This page discusses how per-atom data is managed in LAMMPS, how it can be accessed, what communication patterns apply, and some of the utility functions that exist for a variety of purposes.

### 4.5.1 Owned and ghost atoms

As described on the *parallel partitioning algorithms* page, LAMMPS uses a domain decomposition of the simulation domain, either in a *brick* or *tiled* manner. Each MPI process *owns* exactly one subdomain and the atoms within it. To compute forces for tuples of atoms that are spread across sub-domain boundaries, also a "halo" of *ghost* atoms are maintained within the communication cutoff distance of its subdomain.

The total number of atoms is stored in *Atom::natoms* (within any typical class this can be referred to at *atom->natoms*). The number of *owned* (or "local" atoms) are stored in *Atom::nlocal*; the number of *ghost* atoms is stored in *Atom::nghost*. The sum of *Atom::nlocal* over all MPI processes should be *Atom::natoms*. This is by default regularly checked by the Thermo class, and if the sum does not match, LAMMPS stops with a "lost atoms" error. For convenience also the property *Atom::nmax* is available, this is the maximum of *Atom::nlocal* + *Atom::nghost* across all MPI processes.

Per-atom properties are either managed by the atom style, individual classes, or as custom arrays by the individual classes. If only access to *owned* atoms is needed, they are usually allocated to be of size *Atom::nlocal*, otherwise of size *Atom::nmax*. Please note that not all per-atom properties are available or updated on *ghost* atoms. For example, per-atom velocities are only updated with *comm_modify vel yes*.

### 4.5.2 Atom indexing

When referring to individual atoms, they may be indexed by their local *index*, their index in their *Atom::x* array. This is densely populated containing first all *owned* atoms (index < *Atom::nlocal*) and then all *ghost* atoms. The order of atoms in these arrays can change due to atoms migrating between between subdomains, atoms being added or deleted, or atoms being sorted for better cache efficiency. Atoms are globally uniquely identified by their *atom ID*. There may be multiple atoms with the same atom ID present, but only one of them may be an *owned* atom.

To find the local *index* of an atom, when the *atom ID* is known, the *Atom::map()* function may be used. It will return the local atom index or -1. If the returned value is between 0 (inclusive) and *Atom::nlocal* (exclusive) it is an *owned* or "local" atom; for larger values the atom is present as a ghost atom; for a value of -1, the atom is not present on the current subdomain at all.

If multiple atoms with the same tag exist in the same subdomain, they can be found via the *Atom::sametag* array. It points to the next atom index with the same tag or -1 if there are no more atoms with the same tag. The list will be exhaustive when starting with an index of an *owned* atom, since the atom IDs are unique, so there can only be one such atom. Example code to count atoms with same atom ID in a subdomain:

```
for (int i = 0; i < atom->nlocal; ++i) {
  int count = 0;
  while (sametag[i] >= 0) {
    i = sametag[i];
    ++count;
  }
  printf("Atom ID: %ld is present %d times\n", atom->tag[i], count);
}
```

### 4.5.3 Atom class versus AtomVec classes

The *Atom* class contains all kinds of flags and counters about atoms in the system and that includes pointers to **all** per-atom properties available for atoms. However, only a subset of these pointers are non-NULL and which those are depends on the atom style. For each atom style there is a corresponding *AtomVecXXX* class derived from the *AtomVec* base class, where the XXX indicates the atom style. This *AtomVecXXX* class will update the counters and per-atom pointers if atoms are added or removed to the system or migrate between subdomains.

## 4.6 Communication patterns

This page describes various inter-processor communication operations provided by LAMMPS, mostly in the core *Comm* class. These are operations for common tasks implemented using MPI library calls. They are used by other classes to perform communication of different kinds. These operations are useful to know about when writing new code for LAMMPS that needs to communicate data between processors.

### 4.6.1 Owned and ghost atoms

As described on the *parallel partitioning algorithms* page, LAMMPS spatially decomposes the simulation domain, either in a *brick* or *tiled* manner. Each processor (MPI task) owns atoms within its subdomain and additionally stores ghost atoms within a cutoff distance of its subdomain.

#### Forward and reverse communication

As described on the *parallel communication algorithms* page, the most common communication operations are first, *forward communication* which sends owned atom information from each processor to nearby processors to store with their ghost atoms. The need to do this communication arises when data from the owned atoms is updated (e.g. their positions) and this updated information needs to be **copied** to the corresponding ghost atoms.

And second, *reverse communication*, which sends ghost atom information from each processor to the owning processor to **accumulate** (sum) the values with the corresponding owned atoms. The need for this arises when data is computed and also stored with ghost atoms (e.g. forces when using a "half" neighbor list) and thus those terms need to be added to their corresponding atoms on the process where they are "owned" atoms. Please note, that with the *newton off* setting this does not happen and the neighbor lists are constructed so that these interactions are computed on both MPI processes containing one of the atoms and only the data pertaining to the local atom is stored.

The time-integration classes in LAMMPS invoke these operations each timestep via the *forward_comm()* and *reverse_comm()* methods in the *Comm* class. Which per-atom data is communicated depends on the currently used *atom style* and whether *comm_modify vel* setting is "no" (default) or "yes".

Similarly, *Pair* style classes can invoke the *forward_comm(this)* and *reverse_comm(this)* methods in the *Comm* class to perform the same operations on per-atom data that is generated and stored within the pair style class. Note that this function requires passing the `this` pointer as the first argument to enable the *Comm* class to call the "pack" and "unpack" functions discussed below. An example of the use of these functions are many-body pair styles like the embedded-atom method (EAM) which compute intermediate values in the first part of the compute() function that need to be stored by both owned and ghost atoms for the second part of the force computation. The *Comm* class methods perform the MPI communication for buffers of per-atom data. They "call back" to the *Pair* class, so it can *pack* or *unpack* the buffer with data the *Pair* class owns. There are 4 such methods that the *Pair* class must define, assuming it uses both forward and reverse communication:

- pack_forward_comm()
- unpack_forward_comm()
- pack_reverse_comm()
- unpack_reverse_comm()

The arguments to these methods include the buffer and a list of atoms to pack or unpack. The *Pair* class also must set the *comm_forward* and *comm_reverse* variables, which store the number of values stored in the communication buffers for each operation. This means, if desired, it can choose to store multiple per-atom values in the buffer, and they will be communicated together to minimize communication overhead. The communication buffers are defined vectors containing `double` values. To correctly store integers that may be 64-bit (bigint, tagint, imageint) in the buffer, you need to use the *ubuf union* construct.

The *Fix*, *Bond*, *Compute*, and *Dump* classes can also invoke the same kind of forward and reverse communication operations using the same *Comm* class methods. Likewise, the same pack/unpack methods and comm_forward/comm_reverse variables must be defined by the calling *Fix*, *Bond*, *Compute*, or *Dump* class.

For all of these classes, there is an optional second argument to the *forward_comm()* and *reverse_comm()* call which can be used when the class performs multiple modes of communication, with different numbers of values per atom. The class should set the *comm_forward* and *comm_reverse* variables to the maximum value, but can invoke the communication for a particular mode with a smaller value. For this to work, the *pack_forward_comm()*, etc. methods typically use a class member variable to choose which values to pack/unpack into/from the buffer.

Finally, for reverse communications in *Fix* classes there is also the *reverse_comm_variable()* method that allows the communication to have a different amount of data per-atom. It invokes these corresponding callback methods:

- pack_reverse_comm_size()
- unpack_reverse_comm_size()

which have extra arguments to specify the amount of data stored in the buffer for each atom.

### 4.6.2 Higher level communication

There are also several higher-level communication operations provided in LAMMPS which work for either *brick* or *tiled* decompositions. They may be useful for a new class to invoke if it requires more sophisticated communication than the *forward* and *reverse* methods provide. The 3 communication operations described here are

- ring
- irregular
- rendezvous

You can invoke these *grep* command in the LAMMPS src directory, to see a list of classes that invoke the 3 operations.

- `grep "\->ring" *.cpp */*.cpp`
- `grep "irregular\->" *.cpp`
- `grep "\->rendezvous" *.cpp */*.cpp`

#### Ring operation

The *ring* operation is invoked via the *ring()* method in the *Comm* class.

Each processor first creates a buffer with a list of values, typically associated with a subset of the atoms it owns. Now think of the *P* processors as connected to each other in a *ring*. Each processor *M* sends data to the next *M+1* processor. It receives data from the preceding *M-1* processor. The ring is periodic so that the last processor sends to the first processor, and the first processor receives from the last processor.

Invoking the *ring()* method passes each processor's buffer in *P* steps around the ring. At each step a *callback* method, provided as an argument to ring(), in the caller is invoked. This allows each processor to examine the data buffer provided by every other processor. It may extract values needed by its atoms from the buffers, or it may alter placeholder values in the buffer. In the latter case, when the *ring* operation is complete, each processor can examine its original buffer to extract modified values.

Note that the *ring* operation is similar to an MPI_Alltoall() operation, where every processor effectively sends and receives data to every other processor. The difference is that the *ring* operation does it one step at a time, so the total volume of data does not need to be stored by every processor. However, the *ring* operation is also less efficient than MPI_Alltoall() because of the *P* stages required. So it is typically only suitable for small data buffers and occasional operations that are not time-critical.

### Irregular operation

The *irregular* operation is provided by the *Irregular* class. What LAMMPS terms irregular communication is when each processor knows what data it needs to send to what processor, but does not know what processors are sending it data. An example is when load-balancing is performed and each processor needs to send some of its atoms to new processors.

The *Irregular* class provides 5 high-level methods useful in this context:

- create_data()
- exchange_data()
- create_atom()
- exchange_atom()
- migrate_atoms()

For the *create_data()* method, each processor specifies a list of *N* datums to send, each to a specified processor. Internally, the method creates efficient data structures for performing the communication. The *exchange_data()* method triggers the communication to be performed. Each processor provides the vector of *N* datums to send, and the size of each datum. All datums must be the same size.

The *create_atom()* and *exchange_atom()* methods are similar, except that the size of each datum can be different. Typically, this is used to communicate atoms, each with a variable amount of per-atom data, to other processors.

The *migrate_atoms()* method is a convenience wrapper on the *create_atom()* and *exchange_atom()* methods to simplify communication of all the per-atom data associated with an atom so that the atom can effectively migrate to a new owning processor. It is similar to the *exchange()* method in the *Comm* class invoked when atoms move to neighboring processors (in the regular or tiled decomposition) during timestepping, except that it allows atoms to have moved arbitrarily long distances and still be properly communicated to a new owning processor.

### Rendezvous operation

Finally, the *rendezvous* operation is invoked via the *rendezvous()* method in the *Comm* class. Depending on how much communication is needed and how many processors a LAMMPS simulation is running on, it can be a much more efficient choice than the *ring()* method. It uses the *irregular* operation internally once or twice to do its communication. The rendezvous algorithm is described in detail in *(Plimpton)*, including some LAMMPS use cases.

For the *rendezvous()* method, each processor specifies a list of *N* datums to send and which processor to send each of them to. Internally, this communication is performed as an irregular operation. The received datums are returned to the caller via invocation of *callback* function, provided as an argument to *rendezvous()*. The caller can then process the received datums and (optionally) assemble a new list of datums to communicate to a new list of specific processors. When the callback function exits, the *rendezvous()* method performs a second irregular communication on the new list of datums.

Examples in LAMMPS of use of the *rendezvous* operation are the *fix rigid/small* and *fix shake* commands (for one-time identification of the rigid body atom clusters) and the identification of special_bond 1-2, 1-3 and 1-4 neighbors within molecules. See the *special_bonds* command for context.

---

**(Plimpton)** Plimpton and Knight, JPDC, 147, 184-195 (2021).

## 4.7 How a timestep works

The first and most fundamental operation within LAMMPS to understand is how a timestep is structured. Timestepping is performed by calling methods of the Integrate class instance within the Update class. Since Integrate is a base class, it will point to an instance of a derived class corresponding to what is selected by the *run_style* input script command.

In this section, the timestep implemented by the Verlet class is described. A similar timestep protocol is implemented by the Respa class, for the r-RESPA hierarchical timestepping method.

The Min base class performs energy minimization, so does not perform a literal timestep. But it has logic similar to what is described here, to compute forces and invoke fixes at each iteration of a minimization. Differences between time integration and minimization are highlighted at the end of this section.

The Verlet class is encoded in the `src/verlet.cpp` and `verlet.h` files. It implements the velocity-Verlet timestepping algorithm. The workhorse method is `Verlet::run()`, but first we highlight several other methods in the class.

- The `init()` method is called at the beginning of each dynamics run. It simply sets some internal flags, based on user settings in other parts of the code.
- The `setup()` or `setup_minimal()` methods are also called before each run. The velocity-Verlet method requires current forces be calculated before the first timestep, so these routines compute forces due to all atomic interactions, using the same logic that appears in the timestepping described next. A few fixes are also invoked, using the mechanism described in the next section. Various counters are also initialized before the run begins. The `setup_minimal()` method is a variant that has a flag for performing less setup. This is used when runs are continued and information from the previous run is still valid. For example, if repeated short LAMMPS runs are being invoked, interleaved by other commands, via the *pre no* and *every* options of the run command, the `setup_minimal()` method is used.
- The `force_clear()` method initializes force and other arrays to zero before each timestep, so that forces (torques, etc) can be accumulated.

Now for the `Verlet::run()` method. Its basic structure in hi-level pseudocode is shown below. In the actual code in `src/verlet.cpp` some of these operations are conditionally invoked.

```
loop over N timesteps:
  if timeout condition: break
  ev_set()

  fix->initial_integrate()
  fix->post_integrate()

  nflag = neighbor->decide()
  if nflag:
    fix->pre_exchange()
    domain->pbc()
    domain->reset_box()
    comm->setup()
    neighbor->setup_bins()
    comm->exchange()
    comm->borders()
    fix->pre_neighbor()
    neighbor->build()
    fix->post_neighbor()
  else:
    comm->forward_comm()

  force_clear()
```

(continues on next page)

(continued from previous page)

```
  fix->pre_force()

  pair->compute()
  bond->compute()
  angle->compute()
  dihedral->compute()
  improper->compute()
  kspace->compute()

  fix->pre_reverse()
  comm->reverse_comm()

  fix->post_force()
  fix->final_integrate()
  fix->end_of_step()

  if any output on this step:
    output->write()

# after loop
fix->post_run()
```

The `ev_set()` method (in the parent Integrate class), sets two flags (*eflag* and *vflag*) for energy and virial computation. Each flag encodes whether global and/or per-atom energy and virial should be calculated on this timestep, because some fix or variable or output will need it. These flags are passed to the various methods that compute particle interactions, so that they either compute and tally the corresponding data or can skip the extra calculations if the energy and virial are not needed. See the comments for the `Integrate::ev_set()` method, which document the flag values.

At various points of the timestep, fixes are invoked, e.g. `fix->initial_integrate()`. In the code, this is actually done via the Modify class, which stores all the Fix objects and lists of which should be invoked at what point in the timestep. Fixes are the LAMMPS mechanism for tailoring the operations of a timestep for a particular simulation. As described elsewhere, each fix has one or more methods, each of which is invoked at a specific stage of the timestep, as show in the timestep pseudocode. All the active fixes defined in an input script, that are flagged to have an `initial_integrate()` method, are invoked at the beginning of each timestep. Examples are *fix nve* or *fix nvt or fix npt* which perform the start-of-timestep velocity-Verlet integration operations to update velocities by a half-step, and coordinates by a full step. The `post_integrate()` method is next for operations that need to happen immediately after those updates. Only a few fixes use this, e.g. to reflect particles off box boundaries in the *FixWallReflect class*.

The `decide()` method in the Neighbor class determines whether neighbor lists need to be rebuilt on the current timestep (conditions can be changed using the *neigh_modify every/delay/check* command). If not, coordinates of ghost atoms are acquired by each processor via the `forward_comm()` method of the Comm class. If neighbor lists need to be built, several operations within the inner if clause of the pseudocode are first invoked. The `pre_exchange()` method of any defined fixes is invoked first. Typically, this inserts or deletes particles from the system.

Periodic boundary conditions are then applied by the Domain class via its `pbc()` method to remap particles that have moved outside the simulation box back into the box. Note that this is not done every timestep, but only when neighbor lists are rebuilt. This is so that each processor's subdomain will have consistent (nearby) atom coordinates for its owned and ghost atoms. It is also why dumped atom coordinates may be slightly outside the simulation box if not dumped on a step where the neighbor lists are rebuilt.

The box boundaries are then reset (if needed) via the `reset_box()` method of the Domain class, e.g. if box boundaries are shrink-wrapped to current particle coordinates. A change in the box size or shape requires internal information for communicating ghost atoms (Comm class) and neighbor list bins (Neighbor class) to be updated. The `setup()` method of the Comm class and `setup_bins()` method of the Neighbor class perform the update.

The code is now ready to migrate atoms that have left a processor's geometric subdomain to new processors. The `exchange()` method of the Comm class performs this operation. The `borders()` method of the Comm class then identifies ghost atoms surrounding each processor's subdomain and communicates ghost atom information to neighboring processors. It does this by looping over all the atoms owned by a processor to make lists of those to send to each neighbor processor. On subsequent timesteps, the lists are used by the `Comm::forward_comm()` method.

Fixes with a `pre_neighbor()` method are then called. These typically re-build some data structure stored by the fix that depends on the current atoms owned by each processor.

Now that each processor has a current list of its owned and ghost atoms, LAMMPS is ready to rebuild neighbor lists via the `build()` method of the Neighbor class. This is typically done by binning all owned and ghost atoms, and scanning a stencil of bins around each owned atom's bin to make a Verlet list of neighboring atoms within the force cutoff plus neighbor skin distance.

In the next portion of the timestep, all interaction forces between particles are computed, after zeroing the per-atom force vector via the `force_clear()` method. If the newton flag is set to *on* by the newton command, forces are added to both owned and ghost atoms, otherwise only to owned (aka local) atoms.

Pairwise forces are calculated first, which enables the global virial (if requested) to be calculated cheaply (at O(N) cost instead of O(N**2) at the end of the `Pair::compute()` method), by a dot product of atom coordinates and forces. By including owned and ghost atoms in the dot product, the effect of periodic boundary conditions is correctly accounted for. Molecular topology interactions (bonds, angles, dihedrals, impropers) are calculated next (if supported by the current atom style). The final contribution is from long-range Coulombic interactions, invoked by the KSpace class.

The `pre_reverse()` method in fixes is used for operations that have to be done *before* the upcoming reverse communication (e.g. to perform additional data transfers or reductions for data computed during the force computation and stored with ghost atoms).

If the newton flag is on, forces on ghost atoms are communicated and summed back to their corresponding owned atoms. The `reverse_comm()` method of the Comm class performs this operation, which is essentially the inverse operation of sending copies of owned atom coordinates to other processor's ghost atoms.

At this point in the timestep, the total force on each (local) atom is known. Additional force constraints (external forces, SHAKE, etc) are applied by Fixes that have a `post_force()` method. The second half of the velocity-Verlet integration, `final_integrate()` is then performed (another half-step update of the velocities) via fixes like nve, nvt, npt.

At the end of the timestep, fixes that contain an `end_of_step()` method are invoked. These typically perform a diagnostic calculation, e.g. the ave/time and ave/chunk fixes. The final operation of the timestep is to perform any requested output, via the `write()` method of the Output class. There are 3 kinds of LAMMPS output: thermodynamic output to the screen and log file, snapshots of atom data to a dump file, and restart files. See the *thermo_style*, *dump*, and *restart* commands for more details.

The flow of control during energy minimization iterations is similar to that of a molecular dynamics timestep. Forces are computed, neighbor lists are built as needed, atoms migrate to new processors, and atom coordinates and forces are communicated to neighboring processors. The only difference is what Fix class operations are invoked when. Only a subset of LAMMPS fixes are useful during energy minimization, as explained in their individual doc pages. The relevant Fix class methods are `min_pre_exchange()`, `min_pre_force()`, and `min_post_force()`. Each fix is invoked at the appropriate place within the minimization iteration. For example, the `min_post_force()` method is analogous to the `post_force()` method for dynamics; it is used to alter or constrain forces on each atom, which affects the minimization procedure.

After all iterations are completed, there is a `cleanup` step which calls the `post_run()` method of fixes to perform operations only required at the end of a calculation (like freeing temporary storage or creating final outputs).

## 4.8 Writing new styles

The *Modifying & extending LAMMPS* section of the manual gives an overview of how LAMMPS can be extended by writing new classes that derive from existing parent classes in LAMMPS. Here, some specific coding details are provided for writing code for LAMMPS.

### 4.8.1 Writing new pair styles

Pair styles are at the core of most simulations with LAMMPS, since they are used to compute the forces (plus energy and virial contributions, if needed) on atoms for pairs or small clusters of atoms within a given cutoff. This is often the dominant computation in LAMMPS, and sometimes even the only one. Pair styles can be grouped into multiple categories:

1. simple pairwise additive interactions of point particles (e.g. *Lennard-Jones*, *Morse*, *Buckingham*)
2. pairwise additive interactions of point particles with added *Coulomb* interactions or *only* the Coulomb interactions
3. manybody interactions of point particles (e.g. *EAM*, *Tersoff*)
4. complex interactions that include additional per-atom properties (e.g. Discrete Element Models (DEM), Peridynamics, Ellipsoids)
5. special purpose pair styles that may not even compute forces like *pair_style zero* and *pair_style tracker*, or are a wrapper for multiple kinds of interactions like *pair_style hybrid*, *pair_style list*, and *pair_style kim*

In the text below, we will discuss aspects of implementing pair styles in LAMMPS by looking at representative case studies. The design of LAMMPS allows developers to focus on the essentials, which is to compute the forces (and energies or virial contributions), enter and manage the global settings as well as the potential parameters, and the pair style specific parts of reading and writing restart and data files. Most of the complex tasks like management of the atom positions, domain decomposition and boundaries, or neighbor list creation are handled transparently by other parts of the LAMMPS code.

As shown on the page for *writing or extending pair styles*, in order to implement a new pair style, a new class must be written that is either directly or indirectly derived from the `Pair` class. If that class is directly derived from `Pair`, there are three methods that *must* be re-implemented, since they are "pure" in the base class: `Pair::compute()`, `Pair::settings()`, `Pair::coeff()`. In addition, a custom constructor is needed. All other methods are optional and have default implementations in the base class (most of which do nothing), but they may need to be overridden depending on the requirements of the model.

We are looking at the following cases:

- *Case 1: a pairwise additive model*
- *Case 2: a many-body potential*
- *Case 3: a potential requiring communication*
- *Case 4: potentials without a compute() function*

### 4.8.2 Package and build system considerations

In general, new pair styles should be added to the *EXTRA-PAIR package* unless they are an accelerated pair style and then they should be added to the corresponding accelerator package (*GPU*, *INTEL*, *KOKKOS*, *OPENMP*, *OPT*). If you feel that your contribution should be added to a different package, please consult with the LAMMPS developers first.

The contributed code needs to support the *traditional GNU make build process* **and** the *CMake build process*. For the GNU make process and if the package has an `Install.sh` file, most likely that file needs to be updated to correctly copy the sources when installing the package and properly delete them when uninstalling. This is particularly important when added a new pair style that is a derived class from an existing pair style in a package, so that its installation depends

on the the installation status of the package of the derived class. For the CMake process, it is sometimes necessary to make changes to the package specific CMake scripting in `cmake/Modules/Packages`.

---

### 4.8.3 Case 1: a pairwise additive model

In this section, we will describe the procedure of adding a simple pair style to LAMMPS: an empirical model that can be used to model liquid mercury. The pair style shall be called *born/gauss* and the complete implementation can be found in the files `src/EXTRA-PAIR/pair_born_gauss.cpp` and `src/EXTRA-PAIR/pair_born_gauss.h` of the LAMMPS source code.

#### Model and general considerations

The functional form of the model according to *(Bomont)* consists of a repulsive Born-Mayer exponential term and a temperature dependent, attractive Gaussian term.

$$E = A_0 \exp(-\alpha r) - A_1 \exp\left[-\beta (r - r_0)^2\right]$$

For the application to mercury, the following parameters are listed:

- $A_0 = 8.2464 \times 10^{13}$ eV
- $\alpha = 12.48$ Å$^{-1}$
- $\beta = 0.44$ Å$^{-2}$
- $r_0 = 3.56$ Å
- $A_1$ is temperature dependent and can be determined from $A_1 = a_0 + a_1 T + a_2 T^2$ with:
  - $a_0 = 1.97475 \times 10^{-2}$ eV
  - $a_1 = 8.40841 \times 10^{-5}$ eV/K
  - $a_2 = -2.58717 \times 10^{-8}$ eV/K$^{-2}$

With the optional cutoff, this means we have a total of 5 or 6 parameters for each pair of atom types. Additionally, we need to input a default cutoff value as a global setting.

Because of the combination of Born-Mayer with a Gaussian, the pair style shall be named "born/gauss" and thus the class name would be `PairBornGauss` and the source files `pair_born_gauss.h` and `pair_born_gauss.cpp`. Since this is a rather uncommon potential, it shall be added to the *EXTRA-PAIR* package.

#### Header file

The first segment of any LAMMPS source should be the copyright and license statement. Note the marker in the first line to indicate to editors like emacs that this file is a C++ source, even though the .h extension suggests a C source (this is a convention inherited from the very beginning of the C++ version of LAMMPS).

```
/* -*- c++ -*- ----------------------------------------------------------
   LAMMPS - Large-scale Atomic/Molecular Massively Parallel Simulator
   https://www.lammps.org/, Sandia National Laboratories
   LAMMPS development team: developers@lammps.org

   Copyright (2003) Sandia Corporation.  Under the terms of Contract
   DE-AC04-94AL85000 with Sandia Corporation, the U.S. Government retains
   certain rights in this software.  This software is distributed under
   the GNU General Public License.
```

(continues on next page)

(continued from previous page)

```
   See the README file in the top-level LAMMPS directory.
------------------------------------------------------------------------- */
```

Every pair style must be registered in LAMMPS by including the following lines of code in the second part of the header after the copyright message and before the include guards for the class definition:

```
#ifdef PAIR_CLASS
// clang-format off
PairStyle(born/gauss,PairBornGauss);
// clang-format on
#else

/* the definition of the PairBornGauss class (see below) is inserted here */

#endif
```

This block between `#ifdef PAIR_CLASS` and `#else` will be included by the `Force` class in `force.cpp` to build a map of "factory functions" that will create an instance of these classes and return a pointer to it. The map connects the name of the pair style, "born/gauss", to the name of the class, `PairBornGauss`. During compilation, LAMMPS generates a file `style_pair.h` that contains `#include` statements for all "installed" pair styles. Before including `style_pair.h` into `force.cpp`, the `PAIR_CLASS` define is set and the `PairStyle(name,class)` macro defined. The code of the macro adds the installed pair styles to the "factory map" which enables the *pair_style command* to create the pair style instance.

The list of header files to include is automatically updated by the build system if there are new files, so the presence of the new header file in the `src/EXTRA-PAIR` folder and the enabling of the EXTRA-PAIR package will trigger LAMMPS to include the new pair style when it is (re-)compiled. The "// clang-format" format comments are needed so that running *clang-format* on the file will not insert unwanted blanks between "born", "/", and "gauss" which would break the `PairStyle` macro.

The third part of the header file is the actual class definition of the `PairBornGauss` class. This has the prototypes for all member functions that will be implemented by this pair style. This includes *a few required and a number of optional functions*. All functions that were labeled in the base class as "virtual" must be given the "override" property, as it is done in the code shown below.

The "override" property helps to detect unexpected mismatches because compilation will stop with an error in case the signature of a function is changed in the base class without also changing it in all derived classes. For example, if this change added an optional argument with a default value, then all existing source code *calling* the function would not need changes and still compile, but the function in the derived class would no longer override the one in the base class due to the different number of arguments and the behavior of the pair style is thus changed in an unintended way. Using the "override" keyword prevents such issues.

```
#ifndef LMP_PAIR_BORN_GAUSS_H
#define LMP_PAIR_BORN_GAUSS_H

#include "pair.h"

namespace LAMMPS_NS {

class PairBornGauss : public Pair {
 public:
  PairBornGauss(class LAMMPS *);
  ~PairBornGauss() override;
```

(continues on next page)

(continued from previous page)

```
  void compute(int, int) override;
  void settings(int, char **) override;
  void coeff(int, char **) override;
  double init_one(int, int) override;

  void write_restart(FILE *) override;
  void read_restart(FILE *) override;
  void write_restart_settings(FILE *) override;
  void read_restart_settings(FILE *) override;
  void write_data(FILE *) override;
  void write_data_all(FILE *) override;

  double single(int, int, int, int, double, double, double, double &) override;
  void *extract(const char *, int &) override;
```

Also, variables and arrays for storing global settings and potential parameters are defined. Since these are internal to the class, they are placed after a "protected:" label.

```
 protected:
  double cut_global;
  double **cut;
  double **biga0, **alpha, **biga1, **beta, **r0;
  double **a0, **a1, **a2;
  double **offset;

  virtual void allocate();
};
}    // namespace LAMMPS_NS
#endif
```

### Implementation file

We move on to the implementation of the `PairBornGauss` class in the `pair_born_gauss.cpp` file. This file also starts with a LAMMPS copyright and license header. Below that notice is typically the space where comments may be added with additional information about this specific file, the author(s), affiliation(s), and email address(es). This way the contributing author(s) can be easily contacted, when there are questions about the implementation later. Since the file(s) may be around for a long time, it is beneficial to use some kind of "permanent" email address, if possible.

```
/* ----------------------------------------------------------------------
   LAMMPS - Large-scale Atomic/Molecular Massively Parallel Simulator
   https://www.lammps.org/, Sandia National Laboratories
   LAMMPS development team: developers@lammps.org

   Copyright (2003) Sandia Corporation.  Under the terms of Contract
   DE-AC04-94AL85000 with Sandia Corporation, the U.S. Government retains
   certain rights in this software.  This software is distributed under
   the GNU General Public License.

   See the README file in the top-level LAMMPS directory.
------------------------------------------------------------------------- */
```

(continues on next page)

(continued from previous page)

```
// Contributing author: Axel Kohlmeyer, Temple University, akohlmey@gmail.com

#include "pair_born_gauss.h"

#include "atom.h"
#include "comm.h"
#include "error.h"
#include "fix.h"
#include "force.h"
#include "memory.h"
#include "neigh_list.h"

#include <cmath>
#include <cstring>

using namespace LAMMPS_NS;
```

The second section of the implementation file has various include statements. The include file for the class header has to come first, then a block of LAMMPS classes (sorted alphabetically) followed by a block of system headers and others, if needed. Note the standardized C++ notation for headers of C-library functions (`cmath` instead of `math.h`). The final statement of this segment imports the `LAMMPS_NS::` namespace globally for this file. This way, all LAMMPS specific functions and classes do not have to be prefixed with `LAMMPS_NS::`.

### Constructor and destructor (required)

The first two functions in the implementation source file are typically the constructor and the destructor.

Pair styles are different from most classes in LAMMPS that define a "style", as their constructor only uses the LAMMPS class instance pointer as an argument, but **not** the arguments of the *pair_style command*. Instead, those arguments are processed in the `Pair::settings()` function (or rather the version in the derived class). The constructor is the place where global defaults are set and specifically flags are set indicating which optional features of a pair style are available.

```
/* ---------------------------------------------------------------------- */

PairBornGauss::PairBornGauss(LAMMPS *lmp) : Pair(lmp)
{
  writedata = 1;
}
```

The *writedata = 1;* statement indicates that the pair style is capable of writing the current pair coefficient parameters to data files. That is, the class implements specific versions for `Pair::data()` and `Pair::data_all()`. Other statements that could be added here would be *single_enable = 1;* or *respa_enable = 0;* to indicate that the `Pair::single()` function is present and the `Pair::compute_(inner|middle|outer)` functions are not, but those are also the default settings and already set in the base class.

In the destructor, we need to delete all memory that was allocated by the pair style, usually to hold force field parameters that were entered with the *pair_coeff command*. Most of those array pointers will need to be declared in the derived class header, but some (e.g. setflag, cutsq) are already declared in the base class.

```
PairBornGauss::~PairBornGauss()
{
  if (allocated) {
    memory->destroy(setflag);
    memory->destroy(cutsq);
```

(continues on next page)

(continued from previous page)

```
    memory->destroy(cut);
    memory->destroy(biga0);
    memory->destroy(alpha);
    memory->destroy(biga1);
    memory->destroy(beta);
    memory->destroy(r0);
    memory->destroy(offset);
  }
}
```

### Settings and coefficients (required)

To enter the global pair style settings and the pair style parameters, the functions `Pair::settings()` and `Pair::coeff()` need to be re-implemented. The arguments to the `settings()` function are the arguments given to the *pair_style command*. Normally, those would already be processed as part of the constructor, but moving this to a separate function allows users to change global settings like the default cutoff without having to reissue all pair_coeff commands or re-read the `Pair Coeffs` sections from the data file. In the `settings()` function, also the arrays for storing parameters, to define cutoffs, track which pairs of parameters have been explicitly set and allocated and, if needed, initialized. In this case, the memory allocation and initialization are moved to a function `allocate()`.

```
/* ----------------------------------------------------------------------
   allocate all arrays
------------------------------------------------------------------------- */

void PairBornGauss::allocate()
{
  allocated = 1;
  int np1 = atom->ntypes + 1;

  memory->create(setflag, np1, np1, "pair:setflag");
  for (int i = 1; i < np1; i++)
    for (int j = i; j < np1; j++) setflag[i][j] = 0;

  memory->create(cutsq, np1, np1, "pair:cutsq");
  memory->create(cut, np1, np1, "pair:cut");
  memory->create(biga0, np1, np1, "pair:biga0");
  memory->create(alpha, np1, np1, "pair:alpha");
  memory->create(biga1, np1, np1, "pair:biga1");
  memory->create(beta, np1, np1, "pair:beta");
  memory->create(r0, np1, np1, "pair:r0");
  memory->create(offset, np1, np1, "pair:offset");
}

/* ----------------------------------------------------------------------
   global settings
------------------------------------------------------------------------- */

void PairBornGauss::settings(int narg, char **arg)
{
  if (narg != 1) error->all(FLERR, "Pair style born/gauss must have exactly one argument
↪");
  cut_global = utils::numeric(FLERR, arg[0], false, lmp);
```

(continues on next page)

(continued from previous page)

```
  // reset per-type pair cutoffs that have been explicitly set previously

  if (allocated) {
    for (int i = 1; i <= atom->ntypes; i++)
      for (int j = i; j <= atom->ntypes; j++)
        if (setflag[i][j]) cut[i][j] = cut_global;
  }
}
```

The arguments to the `coeff()` function are the arguments to the *pair_coeff command*. The function is also called when processing the `Pair Coeffs` or `PairIJ Coeffs` sections of data files. In the case of the `Pair Coeffs` section, there is only one atom type per line and thus the first argument is duplicated. Since the atom type arguments of the *pair_coeff command* may be a range (e.g. *3 for atom types 1, 2, and 3), the corresponding arguments are passed to the *utils::bounds()* function which will then return the low and high end of the range. Note that the `setflag` array is set to 1 for all pairs of atom types processed by this call. This information is later used in the `init_one()` function to determine if any coefficients are missing and, if supported by the potential, generate those missing coefficients from the selected mixing rule.

```
/* ----------------------------------------------------------------------
   set coeffs for one or more type pairs
------------------------------------------------------------------------- */

void PairBornGauss::coeff(int narg, char **arg)
{
  if (narg < 7 || narg > 8) error->all(FLERR, "Incorrect args for pair coefficients");
  if (!allocated) allocate();

  int ilo, ihi, jlo, jhi;
  utils::bounds(FLERR, arg[0], 1, atom->ntypes, ilo, ihi, error);
  utils::bounds(FLERR, arg[1], 1, atom->ntypes, jlo, jhi, error);

  double biga0_one = utils::numeric(FLERR, arg[2], false, lmp);
  double alpha_one = utils::numeric(FLERR, arg[3], false, lmp);
  double biga1_one = utils::numeric(FLERR, arg[4], false, lmp);
  double beta_one = utils::numeric(FLERR, arg[5], false, lmp);
  double r0_one = utils::numeric(FLERR, arg[6], false, lmp);
  double cut_one = cut_global;
  if (narg == 10) cut_one = utils::numeric(FLERR, arg[7], false, lmp);

  int count = 0;
  for (int i = ilo; i <= ihi; i++) {
    for (int j = MAX(jlo, i); j <= jhi; j++) {
      biga0[i][j] = biga0_one;
      alpha[i][j] = alpha_one;
      biga1[i][j] = biga1_one;
      beta[i][j] = beta_one;
      r0[i][j] = r0_one;
      cut[i][j] = cut_one;
      setflag[i][j] = 1;
      count++;
    }
```

(continues on next page)

(continued from previous page)

```
  }

  if (count == 0) error->all(FLERR, "Incorrect args for pair coefficients");
}
```

### Initialization

The `init_one()` function is called during the *"init" phase* of a simulation. This is where potential parameters are checked for completeness, derived parameters computed (e.g. the "offset" of the potential energy at the cutoff distance for use with the *pair_modify shift yes* command). If a pair style supports generating "mixed" parameters (i.e. where both atoms of a pair have a different atom type) using a "mixing rule" from the parameters of the type with itself, this is the place to compute and store those mixed values. The *born/gauss* pair style does not support mixing, so we only check for completeness. Another purpose of the `init_one()` function is to symmetrize the potential parameter arrays. The return value of the function is the cutoff for the given pair of atom types. This information is used by the neighbor list code to determine the largest cutoff and then build the neighbor lists accordingly.

```
/* ----------------------------------------------------------------------
   init for one type pair i,j and corresponding j,i
------------------------------------------------------------------------- */

double PairBornGauss::init_one(int i, int j)
{
  if (setflag[i][j] == 0) error->all(FLERR, "All pair coeffs are not set");

  if (offset_flag) {
    double dr = cut[i][j] - r0[i][j];
    offset[i][j] =
        biga0[i][j] * exp(-alpha[i][j] * cut[i][j]) - biga1[i][j] * exp(-beta[i][j] * dr␣
↪* dr);
  } else
    offset[i][j] = 0.0;

  biga0[j][i] = biga0[i][j];
  alpha[j][i] = alpha[i][j];
  biga1[j][i] = biga1[i][j];
  beta[j][i] = beta[i][j];
  r0[j][i] = r0[i][j];
  offset[j][i] = offset[i][j];

  return cut[i][j];
}
```

### Computing forces from the neighbor list (required)

The `compute()` function is the "workhorse" of a pair style. This is where we have the nested loops over all pairs of particles from the neighbor list to compute forces and - if needed - energies and virials.

The first part is to define some variables for later use and store cached copies of data or pointers that we need to access frequently. Also, this is a good place to call `Pair::ev_init()`, which initializes several flags derived from the *eflag* and *vflag* parameters signaling whether the energy and virial need to be tallied and whether only globally or also per-atom.

```
/* ---------------------------------------------------------------------- */

void PairBornGauss::compute(int eflag, int vflag)
{
  int i, j, ii, jj, inum, jnum, itype, jtype;
  double xtmp, ytmp, ztmp, delx, dely, delz, evdwl, fpair;
  double rsq, r, dr, aexp, bexp, factor_lj;
  int *ilist, *jlist, *numneigh, **firstneigh;

  evdwl = 0.0;
  ev_init(eflag, vflag);

  double **x = atom->x;
  double **f = atom->f;
  int *type = atom->type;
  int nlocal = atom->nlocal;
  double *special_lj = force->special_lj;
  int newton_pair = force->newton_pair;

  inum = list->inum;
  ilist = list->ilist;
  numneigh = list->numneigh;
  firstneigh = list->firstneigh;
```

The outer loop (index *i*) is over local atoms of our sub-domain. Typically, the value of *inum* (the number of neighbor lists) is the same as the number of local atoms (= atoms *owned* by this sub-domain). But when the pair style is used as a sub-style of a *hybrid pair style* or neighbor list entries are removed with *neigh_modify exclude*, this number may be smaller. The array `list->ilist` has the (local) indices of the atoms for which neighbor lists have been created. Then `list->numneigh` is an *inum* sized array with the number of entries of each list of neighbors, and `list->firstneigh` is a list of pointers to those lists.

For efficiency reasons, cached copies of some properties of the outer loop atoms are also initialized.

```
// loop over neighbors of my atoms

for (ii = 0; ii < inum; ii++) {
  i = ilist[ii];
  xtmp = x[i][0];
  ytmp = x[i][1];
  ztmp = x[i][2];
  itype = type[i];
  jlist = firstneigh[i];
  jnum = numneigh[i];
```

The inner loop (index *j*) processes the neighbor lists. The neighbor list code encodes extra information using the upper 3 bits. The 2 highest bits encode whether a pair is a regular pair of neighbor (= 0) or a pair of 1-2 (= 1), 1-3 (= 2), or 1-4 (= 3) *"special" neighbor*. The next highest bit encodes whether the pair stores data in a `fix neigh/history` instance (an undocumented internal fix style). The `sbmask()` inline function extracts those bits and converts them into a number. This number is used to look up the corresponding scaling factor for the non-bonded interaction from the `force->special_lj` array and stores it in the *factor_lj* variable. Due to the additional bits, the value of *j* would be out of range when accessing data from per-atom arrays, so we apply the NEIGHMASK constant with a bit-wise and operation to mask them out. This step *must* be done, even if a pair style does not use special bond scaling of forces and energies to avoid segmentation faults.

With the corrected *j* index, it is now possible to compute the distance of the pair. For efficiency reasons, the square

root is only taken *after* the check for the cutoff (which has been stored as squared cutoff by the `Pair` base class). For some pair styles, like the 12-6 Lennard-Jones potential, computing the square root can be avoided entirely.

```
for (jj = 0; jj < jnum; jj++) {
  j = jlist[jj];
  factor_lj = special_lj[sbmask(j)];
  j &= NEIGHMASK;

  delx = xtmp - x[j][0];
  dely = ytmp - x[j][1];
  delz = ztmp - x[j][2];
  rsq = delx * delx + dely * dely + delz * delz;
  jtype = type[j];
```

The following block of code is the actual application of the model potential to compute the force. Note, that *fpair* is the pair-wise force divided by the distance, as this simplifies the projection of the x-, y-, and z-components of the force vector by simply multiplying with the respective distances in those directions.

```
if (rsq < cutsq[itype][jtype]) {
  r = sqrt(rsq);
  dr = r - r0[itype][jtype];
  aexp = biga0[itype][jtype] * exp(-alpha[itype][jtype] * r);
  bexp = biga1[itype][jtype] * exp(-beta[itype][jtype] * dr * dr);
  fpair = alpha[itype][jtype] * aexp;
  fpair -= 2.0 * beta[itype][jtype] * dr * bexp;
  fpair *= factor_lj / r;
```

In the next block, the force is added to the per-atom force arrays. This pair style uses a "half" neighbor list (each pair is listed only once) so we take advantage of the fact that $\vec{F}_{ij} = -\vec{F}_{ji}$, i.e. apply Newton's third law. The force is *always* stored when the atom is a "local" atom. Index *i* atoms are always "local" (i.e. *i* < nlocal); index *j* atoms may be "ghost" atoms (*j* >= nlocal).

Depending on the settings used with the *newton command*, those pairs are only listed once globally (newton_pair == 1), then forces must be stored even with ghost atoms and after all forces are computed a "reverse communication" is performed to add those ghost atom forces to their corresponding local atoms. If the setting is disabled, then the extra communication is skipped, since for pairs straddling sub-domain boundaries, the forces are computed twice and only stored with the local atoms in the domain that *owns* it.

```
f[i][0] += delx * fpair;
f[i][1] += dely * fpair;
f[i][2] += delz * fpair;
if (newton_pair || j < nlocal) {
  f[j][0] -= delx * fpair;
  f[j][1] -= dely * fpair;
  f[j][2] -= delz * fpair;
}
```

The `ev_tally()` function tallies global or per-atom energy and virial. For typical MD simulations, the potential energy is merely a diagnostic and only needed on output. Similarly, the pressure may only be computed for (infrequent) thermodynamic output. For all timesteps where this information is not needed either, *eflag* or *evflag* are zero and the computation and call to the tally function skipped. Note that evdwl is initialized to zero at the beginning of the function, so that it still is valid to access it, even if the energy is not computed (e.g. when only the virial is needed).

```
      if (eflag) evdwl = factor_lj * (aexp - bexp - offset[itype][jtype]);
      if (evflag) ev_tally(i, j, nlocal, newton_pair, evdwl, 0.0, fpair, delx, dely,␣
```

(continues on next page)

(continued from previous page)

```
↪delz);
    }
  }
}
```

If only the global virial is needed and no energy, then calls to `ev_tally()` can be avoided altogether, and the global virial can be computed more efficiently from the dot product of the total per-atom force vector and the position vector of the corresponding atom, $\vec{F} \cdot \vec{r}$. This has to be done *after* all pair-wise forces are computed and *before* the reverse communication to collect data from ghost atoms, since the position has to be the position that was used to compute the force, i.e. *not* the "local" position if that ghost atom is a periodic copy.

```
  if (vflag_fdotr) virial_fdotr_compute();
}
```

### Computing force and energy for a single pair

Certain features in LAMMPS only require computing interactions between individual pairs of atoms and the (optional) `single()` function is needed to support those features (e.g. for tabulation of force and energy with *pair_write*). This is a repetition of the force kernel in the `compute()` function, but only for a single pair of atoms, where the (squared) distance is provided as a parameter (so it may not even be an existing distance between two specific atoms). The energy is returned as the return value of the function and the force as the *fforce* reference. Note, that this is, similar to how *fpair* is used in the `compute()` function, the magnitude of the force along the vector between the two atoms *divided* by the distance.

The `single()` function is optional, but it is expected to be implemented for any true pair-wise additive potential. Many-body potentials and special case potentials do not implement it. In a few special cases (EAM, long-range Coulomb), the `single()` function implements the pairwise additive part of the complete force interaction and depends on either pre-computed properties (derivative of embedding term for EAM) or post-computed non-pair-wise force contributions (KSpace style in case of long-range Coulomb).

The member variable *single_enable* should be set to 0 in the constructor, if it is not implemented (its default value is 1).

```
/* ---------------------------------------------------------------------- */

double PairBornGauss::single(int /*i*/, int /*j*/, int itype, int jtype, double rsq,
                             double /*factor_coul*/, double factor_lj, double &fforce)
{
  double r, dr, aexp, bexp;

  r = sqrt(rsq);
  dr = r - r0[itype][jtype];
  aexp = biga0[itype][jtype] * exp(-alpha[itype][jtype] * r);
  bexp = biga1[itype][jtype] * exp(-beta[itype][jtype] * dr * dr);

  fforce = factor_lj * (alpha[itype][jtype] * aexp - 2.0 * dr * beta[itype][jtype] *␣
↪bexp) / r;
  return factor_lj * (aexp - bexp - offset[itype][jtype]);
}
```

### Reading and writing of restart files

Support for writing and reading binary restart files is provided by the following four functions. Writing is only done by MPI processor rank 0. The output of global (not related to atom types) settings is usually delegated to the `write_restart_settings()` function. This restart facility is commonly only used, if there are small number of per-type parameters. For potentials that use per-element parameters or tabulated data and read these from files, those parameters and the name of the potential file are not written to restart files and the *pair_coeff command* has to re-issued when restarting. For pair styles like "born/gauss" that do support writing to restart files, this is not required.

Implementing the functions to read and write binary restart files is optional. The member variable *restartinfo* should be set to 0 in the constructor, if they are not implemented (its default value is 1).

```
/* ----------------------------------------------------------------------
   proc 0 writes to restart file
------------------------------------------------------------------------- */

void PairBornGauss::write_restart(FILE *fp)
{
  write_restart_settings(fp);

  int i, j;
  for (i = 1; i <= atom->ntypes; i++) {
    for (j = i; j <= atom->ntypes; j++) {
      fwrite(&setflag[i][j], sizeof(int), 1, fp);
      if (setflag[i][j]) {
        fwrite(&biga0[i][j], sizeof(double), 1, fp);
        fwrite(&alpha[i][j], sizeof(double), 1, fp);
        fwrite(&biga1[i][j], sizeof(double), 1, fp);
        fwrite(&beta[i][j], sizeof(double), 1, fp);
        fwrite(&r0[i][j], sizeof(double), 1, fp);
        fwrite(&cut[i][j], sizeof(double), 1, fp);
      }
    }
  }
}

/* ----------------------------------------------------------------------
   proc 0 writes to restart file
------------------------------------------------------------------------- */

void PairBornGauss::write_restart_settings(FILE *fp)
{
  fwrite(&cut_global, sizeof(double), 1, fp);
  fwrite(&offset_flag, sizeof(int), 1, fp);
  fwrite(&mix_flag, sizeof(int), 1, fp);
}
```

Similarly, on reading, only MPI processor rank 0 has opened the restart file and will read the data. The data is then distributed across all parallel processes using calls to `MPI_Bcast()`. Before reading atom type specific data, the corresponding storage needs to be allocated. Order and number or storage size of items read must be exactly the same as when writing, or else the data will be read incorrectly.

Reading uses the *utils::sfread* utility function to detect read errors and short reads, so that LAMMPS can abort if that happens, e.g. when the restart file is corrupted.

```
/* ----------------------------------------------------------------------
   proc 0 reads from restart file, bcasts
------------------------------------------------------------------------- */

void PairBornGauss::read_restart(FILE *fp)
{
  read_restart_settings(fp);

  allocate();

  int i, j;
  int me = comm->me;
  for (i = 1; i <= atom->ntypes; i++) {
    for (j = i; j <= atom->ntypes; j++) {
      if (me == 0) utils::sfread(FLERR, &setflag[i][j], sizeof(int), 1, fp, nullptr,␣
↪error);
      MPI_Bcast(&setflag[i][j], 1, MPI_INT, 0, world);
      if (setflag[i][j]) {
        if (me == 0) {
          utils::sfread(FLERR, &biga0[i][j], sizeof(double), 1, fp, nullptr, error);
          utils::sfread(FLERR, &alpha[i][j], sizeof(double), 1, fp, nullptr, error);
          utils::sfread(FLERR, &biga1[i][j], sizeof(double), 1, fp, nullptr, error);
          utils::sfread(FLERR, &beta[i][j], sizeof(double), 1, fp, nullptr, error);
          utils::sfread(FLERR, &r0[i][j], sizeof(double), 1, fp, nullptr, error);
          utils::sfread(FLERR, &cut[i][j], sizeof(double), 1, fp, nullptr, error);
        }
        MPI_Bcast(&biga0[i][j], 1, MPI_DOUBLE, 0, world);
        MPI_Bcast(&alpha[i][j], 1, MPI_DOUBLE, 0, world);
        MPI_Bcast(&biga1[i][j], 1, MPI_DOUBLE, 0, world);
        MPI_Bcast(&beta[i][j], 1, MPI_DOUBLE, 0, world);
        MPI_Bcast(&r0[i][j], 1, MPI_DOUBLE, 0, world);
        MPI_Bcast(&cut[i][j], 1, MPI_DOUBLE, 0, world);
      }
    }
  }
}

/* ----------------------------------------------------------------------
   proc 0 reads from restart file, bcasts
------------------------------------------------------------------------- */

void PairBornGauss::read_restart_settings(FILE *fp)
{
  if (comm->me == 0) {
    utils::sfread(FLERR, &cut_global, sizeof(double), 1, fp, nullptr, error);
    utils::sfread(FLERR, &offset_flag, sizeof(int), 1, fp, nullptr, error);
    utils::sfread(FLERR, &mix_flag, sizeof(int), 1, fp, nullptr, error);
  }
  MPI_Bcast(&cut_global, 1, MPI_DOUBLE, 0, world);
  MPI_Bcast(&offset_flag, 1, MPI_INT, 0, world);
  MPI_Bcast(&mix_flag, 1, MPI_INT, 0, world);
}
```

### Writing coefficients to data files

The `write_data()` and `write_data_all()` functions are optional and write out the current state of the *pair_coeff settings* as "Pair Coeffs" or "PairIJ Coeffs" sections to a data file when using the *write_data command*. The `write_data()` only writes out the diagonal elements of the pair coefficient matrix, as that is required for the format of the "Pair Coeffs" section. It is called when the "pair" option of the *write_data command* is "ii" (the default). This is suitable for force fields where *all* off-diagonal terms of the pair coefficient matrix are generated from mixing. If explicit settings for off-diagonal elements were made, LAMMPS will print a warning, as those would be lost. To avoid this, the "pair ij" option of *write_data* can be used which will trigger calling the `write_data_all()` function instead, which will write out all settings of the pair coefficient matrix (regardless of whether they were originally created from mixing or not).

These data file output functions are only useful for true pair-wise additive potentials, where the potential parameters can be entered through *multiple pair_coeff commands*. Pair styles that require a single "pair_coeff * *" command are not compatible with reading their parameters from data files. For pair styles like *born/gauss* that do support writing to data files, the potential parameters will be read from the data file, if present, and *pair_coeff commands* may not be needed.

The member variable `writedata` should be set to 1 in the constructor, if these functions are implemented (the default value is 0).

```
/* ----------------------------------------------------------------------
   proc 0 writes to data file
------------------------------------------------------------------------- */

void PairBornGauss::write_data(FILE *fp)
{
  for (int i = 1; i <= atom->ntypes; i++)
    fprintf(fp, "%d %g %g %g %g %g\n", i, biga0[i][i], alpha[i][i], biga1[i][i],␣
↪beta[i][i],
            r0[i][i]);
}

/* ----------------------------------------------------------------------
   proc 0 writes all pairs to data file
------------------------------------------------------------------------- */

void PairBornGauss::write_data_all(FILE *fp)
{
  for (int i = 1; i <= atom->ntypes; i++)
    for (int j = i; j <= atom->ntypes; j++)
      fprintf(fp, "%d %d %g %g %g %g %g %g\n", i, j, biga0[i][j], alpha[i][j],␣
↪biga1[i][j],
              beta[i][j], r0[i][j], cut[i][j]);
}
```

### Give access to internal data

The purpose of the `extract()` function is to facilitate access to internal data of the pair style by other parts of LAMMPS. One possible application is to use *fix adapt* to gradually change potential parameters during a run. Here, we implement access to the pair coefficient matrix parameters.

```
/* ---------------------------------------------------------------------- */

void *PairBornGauss::extract(const char *str, int &dim)
{
```

(continues on next page)

(continued from previous page)

```
  dim = 2;
  if (strcmp(str, "biga0") == 0) return (void *) biga0;
  if (strcmp(str, "biga1") == 0) return (void *) biga1;
  if (strcmp(str, "r0") == 0) return (void *) r0;
  return nullptr;
}
```

Since the mercury potential, for which we have implemented the born/gauss pair style, has a temperature dependent parameter "biga1", we can automatically adapt the potential based on the Taylor-MacLaurin expansion for "biga1" when performing a simulation with a temperature ramp. LAMMPS commands for that application are given below:

```
variable tlo  index 300.0
variable thi  index 600.0
variable temp equal ramp(v_tlo,v_thi)
variable biga1 equal (-2.58717e-8*v_temp+8.40841e-5)*v_temp+1.97475e-2

fix             1 all nvt temp ${tlo} ${thi} 0.1
fix             2 all adapt 1 pair born/gauss biga1 * * v_biga1
```

### 4.8.4 Case 2: a many-body potential

Since there is a detailed description of the purpose and general layout of a pair style in the previous case, we will focus on where the implementation of a typical many-body potential *differs* from a pair-wise additive potential. We will use the implementation of the Tersoff potential as *pair_style tersoff* as an example. The complete implementation can be found in the files `src/MANYBODY/pair_tersoff.cpp` and `src/MANYBODY/pair_tersoff.h` of the LAMMPS source code.

#### Constructor

In the constructor, several *pair style flags* must be set differently for many-body potentials:

- the potential is not pair-wise additive, so the `single()` function cannot be used. This is indicated by setting the *single_enable* member variable to 0 (default value is 1)
- many-body potentials are usually not written to *binary restart files*. This is indicated by setting the member variable *restartinfo* to 0 (default is 1)
- many-body potentials typically read *all* parameters from a file which stores parameters indexed with a string (e.g. the element). For this, only a single *pair_coeff* * * command is allowed. This requirement is set and checked for, when the member variable *one_coeff* is set to 1 (default value is 0)
- many-body potentials can produce incorrect results if pairs of atoms are excluded from the neighbor list, e.g. explicitly by *neigh_modify exclude* or implicitly through defining bonds, angles, etc. and having a *special_bonds setting* that is not "special_bonds lj/coul 1.0 1.0 1.0". LAMMPS will check for this and print a suitable warning, when the member variable *manybody_flag* is set to 1 (default value is 0).

```
PairTersoff::PairTersoff(LAMMPS *lmp) : Pair(lmp)
{
  single_enable = 0;
  restartinfo = 0;
  one_coeff = 1;
  manybody_flag = 1;
```

### Neighbor list request

For computing the three-body interactions of the Tersoff potential a "full" neighbor list (both atoms of a pair are listed in each other's neighbor list) is required. By default a "half" neighbor list is requested (each pair is listed only once). The request is made in the `init_style()` function. A more in-depth discussion of neighbor lists in LAMMPS and how to request them is in *this section of the documentation*

Also, additional conditions must be met for some global settings which are checked in the `init_style()` function.

```
/* ----------------------------------------------------------------------
   init specific to this pair style
------------------------------------------------------------------------- */

void PairTersoff::init_style()
{
  if (atom->tag_enable == 0)
    error->all(FLERR,"Pair style Tersoff requires atom IDs");
  if (force->newton_pair == 0)
    error->all(FLERR,"Pair style Tersoff requires newton pair on");

  // need a full neighbor list

  neighbor->add_request(this,NeighConst::REQ_FULL);
}
```

### Computing forces from the neighbor list

Computing forces for a many-body potential is usually more complex than for a pair-wise additive potential and there are multiple components. For Tersoff, there is a pair-wise additive two-body term (two nested loops over indices $i$ and $j$) and a three-body term (three nested loops over indices $i$, $j$, and $k$). Since the neighbor list has all neighbors up to the maximum cutoff (for the two-body term), but the three-body interactions have a significantly shorter cutoff, a "short neighbor list" is also constructed at the same time while computing the two-body term and looping over the neighbor list for the first time.

```
if (rsq < cutshortsq) {
  neighshort[numshort++] = j;
  if (numshort >= maxshort) {
    maxshort += maxshort/2;
    memory->grow(neighshort,maxshort,"pair:neighshort");
  }
}
```

For the two-body term, only a half neighbor list would be needed, even though we have requested a full list (for the three-body loops). Rather than computing all interactions twice, we skip over half of the entries. This is done in a slightly complex way to make certain the same choice is made across all subdomains and so that there is no load imbalance introduced.

```
jtag = tag[j];
if (itag > jtag) {
  if ((itag+jtag) % 2 == 0) continue;
} else if (itag < jtag) {
  if ((itag+jtag) % 2 == 1) continue;
} else {
  if (x[j][2] < x[i][2]) continue;
  if (x[j][2] == ztmp && x[j][1] < ytmp) continue;
```

(continues on next page)

(continued from previous page)

```
  if (x[j][2] == ztmp && x[j][1] == ytmp && x[j][0] < xtmp) continue;
}
```

For the three-body term, there is one additional nested loop and it uses the "short" neighbor list, accumulated previously.

```
// three-body interactions
// skip immediately if I-J is not within cutoff
double fjxtmp,fjytmp,fjztmp;

for (jj = 0; jj < numshort; jj++) {
  j = neighshort[jj];
  jtype = map[type[j]];

  [...]

  for (kk = 0; kk < numshort; kk++) {
    if (jj == kk) continue;
    k = neighshort[kk];
    ktype = map[type[k]];

    [...]
  }
[...]
```

### Reading potential parameters

For the Tersoff potential, the parameters are listed in a file and associated with triples of elements. Because we have set the `one_coeff` flag to 1 in the constructor, there may only be a single *pair_coeff* * * line in the input for this pair style, and as a consequence the `coeff()` function will only be called once. Thus, the `coeff()` function has to do three tasks, each of which is delegated to a function in the `PairTersoff` class:

1. map elements to atom types. Those follow the potential file name in the command arguments and are processed by the `map_element2type()` function.
2. read and parse the potential parameter file in the `read_file()` function.
3. Build data structures where the original and derived parameters are indexed by all possible triples of atom types and thus can be looked up quickly in the loops for the force computation

```
void PairTersoff::coeff(int narg, char **arg)
{
  if (!allocated) allocate();

  map_element2type(narg-3,arg+3);

  // read potential file and initialize potential parameters

  read_file(arg[2]);
  setup_params();
}
```

### 4.8.5 Case 3: a potential requiring communication

For some models, the interactions between atoms depends on properties of their environment which have to be computed *before* the the forces can be computed. Since LAMMPS is designed to run in parallel using a *domain decomposition strategy*, not all information of the atoms may be directly available and thus communication steps may be need to collect data from ghost atoms of neighboring subdomains or send data to ghost atoms for application during the pairwise computation.

Specifically, two communication patterns are needed: a "reverse communication" and a "forward communication". The reverse communication collects data added to "ghost" atoms from neighboring sub-domains and sums it to their corresponding "local" atoms. This communication is only required and thus executed when the `Force::newton_pair` setting is 1 (i.e. *newton on*, the default). The forward communication is used to copy computed per-atom data from "local" atoms to their corresponding "ghost" atoms in neighboring sub-domains.

For this we will look at how the embedding term of the *embedded atom potential EAM* is implemented in LAMMPS. The complete implementation of this pair style can be found in the files `src/MANYBODY/pair_eam.cpp` and `src/MANYBODY/pair_eam.h` of the LAMMPS source code.

#### Allocating additional per-atom storage

First suitable (local) per-atom arrays (*rho*, *fp*, *numforce*) are allocated. These have to be large enough to include ghost atoms, are not used outside the `compute()` function and are re-initialized to zero once per timestep.

```
if (atom->nmax > nmax) {
  memory->destroy(rho);
  memory->destroy(fp);
  memory->destroy(numforce);
  nmax = atom->nmax;
  memory->create(rho,nmax,"pair:rho");
  memory->create(fp,nmax,"pair:fp");
  memory->create(numforce,nmax,"pair:numforce");
}
```

#### Reverse communication

Then a first loop over all pairs (*i* and *j*) is performed, where data is stored in the *rho* array representing the electron density at the site of *i* contributed from all neighbors *j*. Since the EAM pair style uses a half neighbor list (for efficiency reasons), a reverse communication is needed to collect the contributions to *rho* from ghost atoms (only if *newton on* is set for pair styles).

```
if (newton_pair) comm->reverse_comm(this);
```

To support the reverse communication, two functions must be defined: `pack_reverse_comm()` that copies relevant data into a buffer for ghost atoms and `unpack_reverse_comm()` that takes the collected data and adds it to the *rho* array for the corresponding local atoms that match the ghost atoms. In order to allocate sufficiently sized buffers, a flag must be set in the pair style constructor. Since in this case a single double precision number is communicated per atom, the *comm_reverse* member variable is set to 1 (default is 0 = no reverse communication).

```
int PairEAM::pack_reverse_comm(int n, int first, double *buf)
{
  int i,m,last;

  m = 0;
  last = first + n;
  for (i = first; i < last; i++) buf[m++] = rho[i];
```

(continues on next page)

(continued from previous page)

```
  return m;
}

void PairEAM::unpack_reverse_comm(int n, int *list, double *buf)
{
  int i,j,m;

  m = 0;
  for (i = 0; i < n; i++) {
    j = list[i];
    rho[j] += buf[m++];
  }
}
```

### Forward communication

From the density array *rho*, the derivative of the embedding energy *fp* is computed. The computation is only done for "local" atoms, but for the force computation, that property also is needed on ghost atoms. For that a forward communication is needed.

```
comm->forward_comm(this);
```

Similar to the reverse communication, this requires implementing a `pack_forward_comm()` and an `unpack_forward_comm()` function. Since there is one double precision number per atom that needs to be communicated, we must set the *comm_forward* member variable to 1 (default is 0 = no forward communication).

```
int PairEAM::pack_forward_comm(int n, int *list, double *buf, int pbc_flag, int *pbc)
{
  int i,j,m;

  m = 0;
  for (i = 0; i < n; i++) {
    j = list[i];
    buf[m++] = fp[j];
  }
  return m;
}

void PairEAM::unpack_forward_comm(int n, int first, double *buf)
{
  int i,m,last;

  m = 0;
  last = first + n;
  for (i = first; i < last; i++) fp[i] = buf[m++];
}
```

## 4.8.6 Case 4: potentials without a compute() function

A small number of pair style classes do not implement a `compute()` function, but instead use that of a different pair style.

### Embedded atom variants "eam/fs" and "eam/alloy"

The pair styles *eam/fs and eam/alloy* share the same model and potential function as the *eam pair style*. They differ in the format of the potential files. Pair style *eam* supports only potential files for single elements. For multi-element systems, the mixed terms are computed from mixed parameters. The *eam/fs* and *eam/alloy* pair styles, however, **require** the use of a single potential file for all elements where the mixed element potential is included in the tabulation. That enables more accurate models for alloys, since the mixed terms can be adjusted for a better representation of material properties compared to terms created from mixing of per-element terms in the `PairEAM` class.

We take a closer at the *eam/alloy* pair style. The complete implementation is in the files `src/MANYBODY/pair_eam_alloy.cpp` and `src/MANYBODY/pair_eam_alloy.h`.

The `PairEAMAlloy` class is derived from `PairEAM` and not `Pair` and overrides only a small number of functions:

```
class PairEAMAlloy : virtual public PairEAM {
 public:
  PairEAMAlloy(class LAMMPS *);
  void coeff(int, char **) override;

 protected:
  void read_file(char *) override;
  void file2array() override;
};
```

All other functionality is inherited from the base classes. In the constructor we set the `one_coeff` flag and the `many_body` flag to 1 to indicate the different behavior.

```
PairEAMAlloy::PairEAMAlloy(LAMMPS *lmp) : PairEAM(lmp)
{
  one_coeff = 1;
  manybody_flag = 1;
}
```

The `coeff()` function (not shown here) implements the different behavior when processing the *pair_coeff command*. The `read_file()` and `file2array()` replace the corresponding `PairEAM` class functions to accommodate the different data and file format.

### AIREBO and AIREBO-M potentials

The AIREBO-M potential differs from the better known AIREBO potential in that it use a Morse potential instead of a Lennard-Jones potential for non-bonded interactions. Since this difference is very minimal compared to the entire potential, both potentials are implemented in the `PairAIREBO` class and which non-bonded potential is used is determined by the value of the `morseflag` flag, which would be set to either 0 or 1.

```
class PairAIREBOMorse : public PairAIREBO {
 public:
  PairAIREBOMorse(class LAMMPS *);
  void settings(int, char **) override;
};
```

The `morseflag` variable defaults to 0 and is set to 1 in the `PairAIREBOMorse::settings()` function which is called by the *pair_style* command. This function delegates all command argument processing and setting of other parameters to the `PairAIREBO::settings()` function of the base class.

```
void PairAIREBOMorse::settings(int narg, char **arg)
{
```

(continues on next page)

(continued from previous page)

```
  PairAIREBO::settings(narg, arg);

  morseflag = 1;
}
```

The complete implementation is in the files `src/MANYBODY/pair_airebo.cpp`, `src/MANYBODY/pair_airebo.h`, `src/MANYBODY/pair_airebo_morse.cpp`, `src/MANYBODY/pair_airebo_morse.h`.

---

**(Bomont)** Bomont, Bretonnet, J. Chem. Phys. 124, 054504 (2006)

### 4.8.7 Writing new bond, angle, dihedral, and improper styles

Bond, angle, dihedral, and improper styles are used to compute bonded interactions between groups of two, three, or four atoms whose topology is described in a data file or set by other commands. They are derived from the `Bond`, `Angle`, `Dihedral`, and `Improper` base classes, respectively. As shown on the corresponding pages *Bond, angle, dihedral, improper styles*, the classes that compute these molecular interactions share a common design.

In general, new styles for bonded interactions should be added to the *EXTRA-MOLECULE package*. If you feel that your contribution should be added to a different package (e.g. the *MOLECULE package*), please consult with the *LAMMPS developers* first.

Before implementing an accelerated style the corresponding plain CPU version should be implemented and properly tested. The accelerated version should then be derived from the plain CPU version so that only the code relevant for acceleration is re-implemented. Those should then be added to the corresponding accelerator package (*KOKKOS*, *OPENMP*, *INTEL*).

The contributed code needs to support the *traditional GNU make build process* **and** the *CMake build process*. This is usually automatic unless the new style uses some external library, which is uncommon for bonded interactions.

It is highly recommended also to include one or more example input decks and *force-style unit tests*. The tests will be particularly useful to test consistency between the `compute()` and `single()` methods, plain and accelerated versions, and to test correct restarting and writing and reading of data files.

---

### 4.8.8 Case 1: Implementing a bond style

In this section we describe the procedure of implementing a new bond style. As a concrete example we use the harmonic bond style *bond_style harmonic*, the simplest non-trivial bond style. The implementation can be found in the files `src/MOLECULE/bond_harmonic.cpp` and `src/MOLECULE/bond_harmonic.h` of the LAMMPS source code. It implements the potential:

$$E = K(r - r_0)^2$$

with the spring constant $K$ and the equilibrium distance $r_0$ as per-type coefficients.

#### Header file

The header file `bond_harmonic.h` starts with the standard LAMMPS copyright and license block (see, for example, the discussion in *Writing a new pair style* for details).

After the copyright block, every bond style must be registered in LAMMPS by including the following lines before the include guards:

```
#ifdef BOND_CLASS
// clang-format off
BondStyle(harmonic,BondHarmonic);
// clang-format on
#else
```

This block between `#ifdef BOND_CLASS` and `#else` will be included by the `Force` class in `force.cpp` to build a map of factory functions for bond styles. The map connects the name of the bond style, "harmonic", with the name of the class, `BondHarmonic`. During compilation, LAMMPS generates a file `style_bond.h` that contains `#include` statements for all installed bond styles. Before including `style_bond.h` into `force.cpp`, the `BOND_CLASS` define is set and the `BondStyle(name,class)` macro defined. The `// clang-format` comments prevent `clang-format` from reformatting the macro argument in a way that would break it.

Analogously, an angle style would use `#ifdef ANGLE_CLASS` and `AngleStyle(name,class)`, a dihedral style would use `#ifdef DIHEDRAL_CLASS` and `DihedralStyle(name,class)`, and an improper style would use `#ifdef IMPROPER_CLASS` and `ImproperStyle(name,class)`.

The class definition itself follows after the include guard:

```
#ifndef LMP_BOND_HARMONIC_H
#define LMP_BOND_HARMONIC_H

#include "bond.h"

namespace LAMMPS_NS {

class BondHarmonic : public Bond {
 public:
  BondHarmonic(class LAMMPS *);
  ~BondHarmonic() override;
  void compute(int, int) override;
  void coeff(int, char **) override;
  double equilibrium_distance(int) override;
  void write_restart(FILE *) override;
  void read_restart(FILE *) override;
  void write_data(FILE *) override;
  double single(int, double, int, int, double &) override;
  void born_matrix(int, double, int, int, double &, double &) override;
  void *extract(const char *, int &) override;

 protected:
  double *k, *r0;

  virtual void allocate();
};

}    // namespace LAMMPS_NS
#endif
#endif
```

The class is derived from `Bond` and overrides all *pure* virtual methods as require by the C++ standard (`compute()`, `coeff()`, `equilibrium_distance()`, `write_restart()`, `read_restart()`, and `single()`) and several optional ones. All overriding methods use the `override` keyword to allow the compiler to detect mismatches. The per-type coefficient arrays `k` and `r0` are declared as protected member variables. The `allocate()` method is declared as `virtual` so that derived classes can override it if needed.

> **Note**
>
> Ideally, there should be no additional `#include` statements outside of `#include "bond.h"`. Where possible, forward declarations should be used and the proper include statements only added to the implementation file (discussed below). Exceptions are headers for C++ container classes. Please see *Requirements for contributions to LAMMPS* and *LAMMPS programming style* for more information and this and related issues.

### Implementation file

The implementation file `bond_harmonic.cpp` starts with the same copyright block and includes, followed by `using namespace LAMMPS_NS;`.

### Constructor and destructor

The constructor is minimal; it only calls the base class constructor and sets the `born_matrix_enable` flag to enable the optional `born_matrix()` method:

```
BondHarmonic::BondHarmonic(LAMMPS *_lmp) : Bond(_lmp), k(nullptr), r0(nullptr)
{
  born_matrix_enable = 1;
}
```

It is recommended to always initialize all pointers to `nullptr` in the initializer list so that they can be safely deleted in the destructor.

The destructor frees the per-type coefficient arrays, but only if `allocated` is true and `copymode` is false. The `copymode` flag is set by Kokkos-accelerated styles that copy the base class rather than owning the storage:

```
BondHarmonic::~BondHarmonic()
{
  if (allocated && !copymode) {
    memory->destroy(setflag);
    memory->destroy(k);
    memory->destroy(r0);
  }
}
```

### The allocate() method

The `allocate()` helper allocates all per-type arrays. Types are one-indexed, so arrays of size `nbondtypes + 1` are created. The `setflag` array, which tracks whether coefficients have been set for each type, is also allocated here and initialized to zero:

```
void BondHarmonic::allocate()
{
  allocated = 1;
  const int np1 = atom->nbondtypes + 1;

  memory->create(k, np1, "bond:k");
  memory->create(r0, np1, "bond:r0");

  memory->create(setflag, np1, "bond:setflag");
  for (int i = 1; i < np1; i++) setflag[i] = 0;
}
```

The `setflag` array is declared in the `Bond` base class. It is used by `Bond::init()` to check that coefficients have been set for all bond types.

### The coeff() method (required)

The `coeff()` method processes the arguments of the *bond_coeff* command. The first argument is always the type range (e.g. `1*3`, handled by `utils::bounds()`); subsequent arguments are the potential coefficients.

```
void BondHarmonic::coeff(int narg, char **arg)
{
  if (narg != 3) error->all(FLERR, "Incorrect args for bond coefficients" +␣
↪utils::errorurl(21));
  if (!allocated) allocate();

  int ilo, ihi;
  utils::bounds(FLERR, arg[0], 1, atom->nbondtypes, ilo, ihi, error);

  double k_one = utils::numeric(FLERR, arg[1], false, lmp);
  double r0_one = utils::numeric(FLERR, arg[2], false, lmp);

  int count = 0;
  for (int i = ilo; i <= ihi; i++) {
    k[i] = k_one;
    r0[i] = r0_one;
    setflag[i] = 1;
    count++;
  }

  if (count == 0) error->all(FLERR, "Incorrect args for bond coefficients" +␣
↪utils::errorurl(21));
}
```

The `utils::bounds()` function converts the type string into integer lower and upper bounds `ilo` and `ihi`. After setting the coefficients for each type in the range, `setflag[i]` is set to 1. The final check guards against a range that matches no types.

### The compute() method (required)

The `compute()` method iterates over all bonds in the local bond list (`neighbor->bondlist`) and accumulates forces and energies. The two arguments `eflag` and `vflag` control whether energies and virial contributions are computed. The actual energy and virial accumulation is handled by calling `ev_tally()` at the end of each iteration step.

```
void BondHarmonic::compute(int eflag, int vflag)
{
  int i1, i2, n, type;
  double delx, dely, delz, ebond, fbond;
  double rsq, r, dr, rk;

  ebond = 0.0;
  ev_init(eflag, vflag);

  double **x = atom->x;
  double **f = atom->f;
  int **bondlist = neighbor->bondlist;
```

(continues on next page)

(continued from previous page)

```
  int nbondlist = neighbor->nbondlist;
  int nlocal = atom->nlocal;
  int newton_bond = force->newton_bond;

  for (n = 0; n < nbondlist; n++) {
    i1 = bondlist[n][0];
    i2 = bondlist[n][1];
    type = bondlist[n][2];

    delx = x[i1][0] - x[i2][0];
    dely = x[i1][1] - x[i2][1];
    delz = x[i1][2] - x[i2][2];

    rsq = delx * delx + dely * dely + delz * delz;
    r = sqrt(rsq);
    dr = r - r0[type];
    rk = k[type] * dr;

    // force & energy

    if (r > 0.0)
      fbond = -2.0 * rk / r;
    else
      fbond = 0.0;

    if (eflag) ebond = rk * dr;

    // apply force to each of 2 atoms

    if (newton_bond || i1 < nlocal) {
      f[i1][0] += delx * fbond;
      f[i1][1] += dely * fbond;
      f[i1][2] += delz * fbond;
    }

    if (newton_bond || i2 < nlocal) {
      f[i2][0] -= delx * fbond;
      f[i2][1] -= dely * fbond;
      f[i2][2] -= delz * fbond;
    }

    if (evflag) ev_tally(i1, i2, nlocal, newton_bond, ebond, fbond, delx, dely, delz);
  }
}
```

Before the loop, `ev_init(eflag, vflag)` initializes the energy and virial accumulators and sets several flags (including `evflag`) that control which quantities are computed. The force scalar `fbond` is defined such that the force on atom `i1` is `fbond * (x[i1] - x[i2])` and the force on atom `i2` is `-fbond * (x[i1] - x[i2])`. This sign convention matches the force as the negative gradient of the energy.

The Newton's third law check (`newton_bond || i < nlocal`) is important for correctness in parallel runs: when `newton_bond` is on, forces are computed for both atoms even when they are on different MPI ranks, relying on a subsequent reverse communication to add up the contributions; when `newton_bond` is off, only forces on local atoms

(`i < nlocal`) are accumulated.

### The equilibrium_distance() method (required for bond styles)

The `equilibrium_distance()` method returns the equilibrium bond length for the given type. This is used, for instance, by the *fix shake* and *fix rattle* commands:

```
double BondHarmonic::equilibrium_distance(int i)
{
  return r0[i];
}
```

### The single() method (required for bond styles)

The `single()` method computes the force divided by $r$ and the potential energy for a single pair of atoms. It is called by *compute bond/local* and related commands:

```
double BondHarmonic::single(int type, double rsq, int /*i*/, int /*j*/, double &fforce)
{
  double r = sqrt(rsq);
  double dr = r - r0[type];
  double rk = k[type] * dr;
  fforce = 0;
  if (r > 0.0) fforce = -2.0 * rk / r;
  return rk * dr;
}
```

The return value is the potential energy. `fforce` is set to `-dE/dr / r` (i.e. the force divided by the interatomic distance) so that it can be easily converted into the x-, y-, and z-direction force components by multiplying with $\Delta x$, $\Delta y$, and $\Delta z$, respectively.

### Restart and data file methods (required)

The `write_restart()` method is called by proc 0 to write the per-type coefficients to the binary restart file. The matching `read_restart()` method is called when reading a restart file; it allocates storage, reads coefficients on proc 0, and broadcasts them to all ranks.

```
void BondHarmonic::write_restart(FILE *fp)
{
  fwrite(&k[1], sizeof(double), atom->nbondtypes, fp);
  fwrite(&r0[1], sizeof(double), atom->nbondtypes, fp);
}

void BondHarmonic::read_restart(FILE *fp)
{
  allocate();

  if (comm->me == 0) {
    utils::sfread(FLERR, &k[1], sizeof(double), atom->nbondtypes, fp, nullptr, error);
    utils::sfread(FLERR, &r0[1], sizeof(double), atom->nbondtypes, fp, nullptr, error);
  }
  MPI_Bcast(&k[1], atom->nbondtypes, MPI_DOUBLE, 0, world);
  MPI_Bcast(&r0[1], atom->nbondtypes, MPI_DOUBLE, 0, world);
```

(continues on next page)

(continued from previous page)

```
  for (int i = 1; i <= atom->nbondtypes; i++) setflag[i] = 1;
}

void BondHarmonic::write_data(FILE *fp)
{
  for (int i = 1; i <= atom->nbondtypes; i++) fprintf(fp, "%d %g %g\n", i, k[i], r0[i]);
}
```

Note that index 0 of each array is unused (types are one-indexed), so `&k[1]` is the start of the actual data. Using `utils::sfread()` (instead of `fread()`) provides error checking.

---

### 4.8.9 Case 2: Implementing an angle style

Angle styles are similar to bond styles but involve three atoms. The implementation follows the same pattern as for bond styles, so for details not mentioned here, you can refer to the corresponding section documenting the implementation of bond styles. As an example we use *angle_style harmonic* which implements the potential:

$$E = K(\theta - \theta_0)^2$$

with spring constant $K$ and equilibrium angle $\theta_0$.

The header file uses `#ifdef ANGLE_CLASS` and `AngleStyle(name,class)` for the registration block, and the class is derived from `Angle`:

```
#ifdef ANGLE_CLASS
// clang-format off
AngleStyle(harmonic,AngleHarmonic);
// clang-format on
#else

#ifndef LMP_ANGLE_HARMONIC_H
#define LMP_ANGLE_HARMONIC_H

#include "angle.h"

namespace LAMMPS_NS {

class AngleHarmonic : public Angle {
 public:
  AngleHarmonic(class LAMMPS *);
  ~AngleHarmonic() override;
  void compute(int, int) override;
  void coeff(int, char **) override;
  double equilibrium_angle(int) override;
  void write_restart(FILE *) override;
  void read_restart(FILE *) override;
  void write_data(FILE *) override;
  double single(int, int, int, int) override;
  void *extract(const char *, int &) override;
```

(continues on next page)

(continued from previous page)

```
 protected:
  double *k, *theta0;

  virtual void allocate();
};

}    // namespace LAMMPS_NS
#endif
#endif
```

Key differences from a bond style:

- The registration macro is `AngleStyle` and the include guard `#ifdef ANGLE_CLASS`.
- The base class is `Angle` (`#include "angle.h"`).
- The `equilibrium_distance()` method is replaced by `equilibrium_angle()` (used by *fix shake*).
- The `single()` method signature takes four atom indices instead of a distance squared plus two atom indices.
- The `compute()` loop iterates over `neighbor->anglelist` and `neighbor->nanglelist` involving three-atom groups.
- The types array is `atom->nangletypes`.

The `compute()` method for angle styles uses three atom indices (`i1`, `i2`, `i3`) from `neighbor->anglelist[n][0..3]`, where the type is at index 3. The equilibrium angle `theta0` is stored in radians (converted from degrees in `coeff()`).

---

### 4.8.10 Case 3: Implementing a dihedral style

Dihedral styles compute interactions among four atoms forming a torsion angle. Harmonic dihedrals implement the potential:

$$E = K\left[1 + d\cos(n\phi)\right]$$

where $K$ is the force constant, $d$ is the sign (+1 or -1), and $n$ is the multiplicity (periodicity). The implementation can be found in `src/MOLECULE/dihedral_harmonic.cpp` and `src/MOLECULE/dihedral_harmonic.h`.

The header file uses `#ifdef DIHEDRAL_CLASS` and `DihedralStyle(name,class)`, and the class is derived from `Dihedral`:

```
#ifdef DIHEDRAL_CLASS
// clang-format off
DihedralStyle(harmonic,DihedralHarmonic);
// clang-format on
#else

#ifndef LMP_DIHEDRAL_HARMONIC_H
#define LMP_DIHEDRAL_HARMONIC_H

#include "dihedral.h"

namespace LAMMPS_NS {
```

(continues on next page)

(continued from previous page)

```
class DihedralHarmonic : public Dihedral {
 public:
  DihedralHarmonic(class LAMMPS *);
  ~DihedralHarmonic() override;
  void compute(int, int) override;
  void coeff(int, char **) override;
  void write_restart(FILE *) override;
  void read_restart(FILE *) override;
  void write_data(FILE *) override;

 protected:
  double *k, *cos_shift, *sin_shift;
  int *sign, *multiplicity;

  virtual void allocate();
};

}    // namespace LAMMPS_NS
#endif
#endif
```

Key differences from bond and angle styles:

- The base class is `Dihedral` (`#include "dihedral.h"`).
- There is no `equilibrium_distance()` or `equilibrium_angle()` method; dihedral styles have no analogue required by other LAMMPS subsystems.
- There is no `single()` method in the base class for dihedrals.
- The `compute()` loop uses four atom indices from `neighbor->dihedrallist[n][0..4]` (type at index 4).
- The types array is `atom->ndihedraltypes`.
- The `writedata` flag must be set to 1 in the constructor if `write_data()` is implemented (the default is 0 for dihedral styles).

In the constructor, the `writedata` flag is set:

```
DihedralHarmonic::DihedralHarmonic(LAMMPS *_lmp) :
    Dihedral(_lmp), k(nullptr), cos_shift(nullptr), sin_shift(nullptr), sign(nullptr),
    multiplicity(nullptr)
{
  writedata = 1;
}
```

### 4.8.11 Case 4: Implementing an improper style

Improper styles compute interactions involving four atoms where the geometry represents an out-of-plane deformation rather than a conventional torsion. Harmonic impropers implement:

$$E = K(\chi - \chi_0)^2$$

where $K$ is the force constant and $\chi_0$ is the equilibrium improper angle.

The header file uses `#ifdef IMPROPER_CLASS` and `ImproperStyle(name,class)`, and the class is derived from `Improper`:

```
#ifdef IMPROPER_CLASS
// clang-format off
ImproperStyle(harmonic,ImproperHarmonic);
// clang-format on
#else

#ifndef LMP_IMPROPER_HARMONIC_H
#define LMP_IMPROPER_HARMONIC_H

#include "improper.h"

namespace LAMMPS_NS {

class ImproperHarmonic : public Improper {
 public:
  ImproperHarmonic(class LAMMPS *);
  ~ImproperHarmonic() override;
  void compute(int, int) override;
  void coeff(int, char **) override;
  void write_restart(FILE *) override;
  void read_restart(FILE *) override;
  void write_data(FILE *) override;
  void *extract(const char *, int &) override;

 protected:
  double *k, *chi;

  virtual void allocate();
};

}    // namespace LAMMPS_NS
#endif
#endif
```

Key differences from dihedral styles:

- The base class is `Improper` (`#include "improper.h"`).
- The `compute()` loop uses four atom indices from `neighbor->improperlist[n][0..4]` (type at index 4).
- The types array is `atom->nimpropertypes`.
- A `symmatoms` array in the base class can be used to record which atom in the quadruplet is the central atom of symmetry. In `ImproperHarmonic`, `symmatoms[0] = 1` indicates that the first atom in the quadruplet is the atom of symmetry.

Constructor setup:

```
ImproperHarmonic::ImproperHarmonic(LAMMPS *_lmp) : Improper(_lmp), k(nullptr),␣
↪chi(nullptr)
{
  writedata = 1;
  // the first atom in the quadruplet is the atom of symmetry
```

(continues on next page)

(continued from previous page)

```
  symmatoms[0] = 1;
}
```

### 4.8.12 Writing a new fix style

Writing fix styles is a flexible way of extending LAMMPS. Users can implement many things using fixes. Some fix styles are only used internally to support compute styles or pair styles:

- change particles attributes (positions, velocities, forces, etc.). Examples: `FixNVE`, `FixFreeze`.
- read or write data. Example: `FixRestart`.
- adding or modifying properties due to geometry. Example: `FixWall`.
- interacting with other subsystems or external code: Examples: `FixTTM`, `FixExternal`, `FixMDI`
- saving information for analysis or future use (previous positions, for instance). Examples: `FixAveTime`, `FixStoreState`.

All fixes are derived from the `Fix` base class and must have a constructor with the signature: `FixPrintVel(class LAMMPS *, int, char **)`.

Every fix must be registered in LAMMPS by writing the following lines of code in the header before include guards:

```
#ifdef FIX_CLASS
// clang-format off
FixStyle(print/vel,FixPrintVel);
// clang-format on
#else
/* the definition of the FixPrintVel class comes here */
...
#endif
```

Where `print/vel` is the style name of your fix in the input script and `FixPrintVel` is the name of the class. The header file would be called `fix_print_vel.h` and the implementation file `fix_print_vel.cpp`. These conventions allow LAMMPS to automatically integrate it into the executable when compiling and associate your new fix class with the designated keyword when it parses the input script.

Let's write a simple fix which will print the average velocity at the end of each timestep. First of all, implement a constructor:

```
FixPrintVel::FixPrintVel(LAMMPS *lmp, int narg, char **arg)
: Fix(lmp, narg, arg)
{
  if (narg < 4) utils::missing_cmd_args(FLERR, "fix print/vel", error);

  nevery = utils::inumeric(FLERR,arg[3],false,lmp);
  if (nevery <= 0)
    error->all(FLERR,"Illegal fix print/vel nevery value: {}", nevery);
}
```

In the constructor you should parse the fix arguments which are specified in the script. All fixes have pretty much the same syntax: `fix <fix-ID> <fix group> <fix name> <fix arguments ...>`. The first 3 parameters are parsed by Fix base class constructor, while `<fix arguments>` should be parsed by you. In our case, we need to specify how often we want to print an average velocity. For instance, once in 50 timesteps: `fix 1 print/vel 50`. There is a special variable in the Fix class called `nevery` which specifies how often the method `end_of_step()` is called. Thus all we need to do is just set it up.

The next method we need to implement is `setmask()`:

```
int FixPrintVel::setmask()
{
  int mask = 0;
  mask |= FixConst::END_OF_STEP;
  return mask;
}
```

Here the we specify which methods of the fix should be called during *execution of a timestep*. The constant `END_OF_STEP` corresponds to the `end_of_step()` method.

```
void FixPrintVel::end_of_step()
{
  // for add3, scale3
  using namespace MathExtra;

  double** v = atom->v;
  int nlocal = atom->nlocal;
  double localAvgVel[4]; // 4th element for particles count
  memset(localAvgVel, 0, 4 * sizeof(double));
  for (int particleInd = 0; particleInd < nlocal; ++particleInd) {
    add3(localAvgVel, v[particleInd], localAvgVel);
  }
  localAvgVel[3] = nlocal;
  double globalAvgVel[4];
  memset(globalAvgVel, 0, 4 * sizeof(double));
  MPI_Allreduce(localAvgVel, globalAvgVel, 4, MPI_DOUBLE, MPI_SUM, world);
  scale3(1.0 / globalAvgVel[3], globalAvgVel);
  if ((comm->me == 0) && screen) {
    utils::print(screen, "{}, {}, {}\n",
                 globalAvgVel[0], globalAvgVel[1], globalAvgVel[2]);
  }
}
```

In the code above, we use MathExtra routines defined in `math_extra.h`. There are bunch of math functions to work with arrays of doubles as with math vectors. It is also important to note that LAMMPS code should always assume to be run in parallel and that atom data is thus distributed across the MPI ranks. Thus you can only process data from local atoms directly and need to use MPI library calls to combine or exchange data. For serial execution, LAMMPS comes bundled with the MPI STUBS library that contains the MPI library function calls in dummy versions that only work for a single MPI rank.

In this code we use an instance of Atom class. This object is stored in the Pointers class (see `pointers.h`) which is the base class of the Fix base class. This object contains references to various class instances (the original instances are created and held by the LAMMPS class) with all global information about the simulation system. Data from the Pointers class is available to all classes inherited from it using protected inheritance. Hence when you write you own class, which is going to use LAMMPS data, don't forget to inherit from Pointers or pass a Pointer to it to all functions that need access. When writing fixes we inherit from class Fix which is inherited from Pointers so there is no need to inherit from it directly.

The code above computes average velocity for all particles in the simulation. Yet you have one unused parameter in fix call from the script: `group_name`. This parameter specifies the group of atoms used in the fix. So we should compute the average for all particles in the simulation only if `group_name == "all"`, but it can be any group. The group membership information of an atom is contained in the *mask* property of an atom and the bit corresponding to a given group is stored in the groupbit variable which is defined in Fix base class:

```
for (int i = 0; i < nlocal; ++i) {
  if (atom->mask[i] & groupbit) {
  // Do all job here
  }
}
```

Class Atom encapsulates atoms positions, velocities, forces, etc. Users can access them using the particle index. Note, that particle indexes are usually changed every few timesteps because of neighbor list rebuilds and spatial sorting (to improve cache efficiency).

Let us consider another Fix example: We want to have a fix which stores atoms position from the previous time step in your fix. The local atoms indexes may not be valid on the next iteration. In order to handle this situation there are several methods which should be implemented:

- `double memory_usage()`: return how much memory the fix uses (optional)
- `void grow_arrays(int)`: do reallocation of the per-particle arrays in your fix
- `void copy_arrays(int i, int j, int delflag)`: copy i-th per-particle information to j-th particle position. Used when atom sorting is performed. if delflag is set and atom j owns a body, move the body information to atom i.
- `void set_arrays(int i)`: sets i-th particle related information to zero

Note, that if your class implements these methods, it must add calls of add_callback and delete_callback to the constructor and destructor. Since we want to store positions of atoms from the previous timestep, we need to add `double** xold` to the header file. Than add allocation code to the constructor:

```
FixSavePos::FixSavePos(LAMMPS *lmp, int narg, char **arg), xold(nullptr)
{
//...
  memory->create(xold, atom->nmax, 3, "FixSavePos:x");
  atom->add_callback(0);
}

FixSavePos::~FixSavePos() {
  atom->delete_callback(id, 0);
  memory->destroy(xold);
}
```

Implement the aforementioned methods:

```
double FixSavePos::memory_usage()
{
  int nmax = atom->nmax;
  double bytes = 0.0;
  bytes += nmax * 3 * sizeof(double);
  return bytes;
}

void FixSavePos::grow_arrays(int nmax)
{
  memory->grow(xold, nmax, 3, "FixSavePos:xold");
}

void FixSavePos::copy_arrays(int i, int j, int delflag)
```

(continues on next page)

(continued from previous page)

```
{
  memcpy(xold[j], xold[i], sizeof(double) * 3);
}

void FixSavePos::set_arrays(int i)
{
  memset(xold[i], 0, sizeof(double) * 3);
}

int FixSavePos::pack_exchange(int i, double *buf)
{
  int m = 0;
  buf[m++] = xold[i][0];
  buf[m++] = xold[i][1];
  buf[m++] = xold[i][2];

  return m;
}

int FixSavePos::unpack_exchange(int nlocal, double *buf)
{
  int m = 0;
  xold[nlocal][0] = buf[m++];
  xold[nlocal][1] = buf[m++];
  xold[nlocal][2] = buf[m++];

  return m;
}
```

Now, a little bit about memory allocation. We use the Memory class which is just a bunch of template functions for allocating 1D and 2D arrays. So you need to add include `memory.h` to have access to them.

Finally, if you need to write/read some global information used in your fix to the restart file, you might do it by setting the flag `restart_global = 1` in the constructor and implementing methods `void write_restart(FILE *fp)` and `void restart(char *buf)`. If, in addition, you want to write the per-atom property to restart files then these additional settings and functions are needed:

- a fix flag indicating this needs to be set `restart_peratom = 1;`
- `atom->add_callback()` and `atom->delete_callback()` must be called a second time with the final argument set to 1 instead of 0 (indicating restart processing instead of per-atom data memory management).
- the functions `void pack_restart(int i, double *buf)` and `void unpack_restart(int nlocal, int nth)` need to be implemented

### 4.8.13 Writing new compute styles

Compute styles are used to calculate global or per-atom scalar, vector, or array quantities or local or per-chunk or grid based properties from the current state of the simulation. Examples include temperatures, pressures, kinetic energies, and user-defined quantities. All compute styles are derived from the `Compute` base class, which is defined in `src/compute.h`. An overview of the available methods and flags is given on the page for *modifying or extending compute styles*.

Computes are executed *on demand* and thus require a "consumer" like *fix ave/time*, *compute reduce*, or a *custom thermo style* or *custom dump style*. This is different from *Writing a new fix style* which are executed at *pre-determined steps*

*during a run* and either at every steps or with a set frequency. Some compute instances are also created and used internally, e.g. for *thermo output*. It should be noted that some computes do not actually perform computations (e.g. computes for potential energy or virial) but rather suitable flags are set and then the corresponding data is collected by the force computation as needed and then the compute merely makes that pre-collected data available. Such computes may not be executed between runs but only during a run (or minimization).

In general, new compute styles should be added to the *EXTRA-COMPUTE package*. It is usually not necessary to create accelerated compute styles except for the *KOKKOS* package where there is a significant benefit to avoid time consuming data transfers between GPU and host. If you feel that your contribution should be added to a different package, please consult with the *LAMMPS developers* first. The contributed code needs to support the *traditional GNU make build process* **and** the *CMake build process*.

---

### 4.8.14 Case 1: A global scalar compute

In this section we describe how to implement a compute that produces a single global scalar value. As a concrete example we use *compute temp*, which computes the kinetic temperature of a group of atoms. The full implementation can be found in `src/compute_temp.cpp` and `src/compute_temp.h`.

#### Header file

Every compute style must be registered in LAMMPS by including the following lines after the copyright block and before the include guards for the class definition:

```
#ifdef COMPUTE_CLASS
// clang-format off
ComputeStyle(temp,ComputeTemp);
// clang-format on
#else
```

The block between `#ifdef COMPUTE_CLASS` and `#else` will be included by the `Modify` class in `modify.cpp` to build a map of factory functions. The map connects the style name `temp` with the class name `ComputeTemp`. During compilation, LAMMPS constructs a file `style_compute.h` that `#include`s all installed compute style headers. The `// clang-format` comments prevent the `clang-format` tool from incorrectly inserting spaces inside the macro arguments.

The class definition:

```
#ifndef LMP_COMPUTE_TEMP_H
#define LMP_COMPUTE_TEMP_H

#include "compute.h"

namespace LAMMPS_NS {

class ComputeTemp : public Compute {
 public:
  ComputeTemp(class LAMMPS *, int, char **);
  ~ComputeTemp() override;
  void init() override {}
  void setup() override;
  double compute_scalar() override;
  void compute_vector() override;
```

(continues on next page)

(continued from previous page)

```
 protected:
  double tfactor;

  virtual void dof_compute();
};

}    // namespace LAMMPS_NS
#endif
#endif
```

The class derives from `Compute` and overrides the pure virtual method `init()` (here with an empty inline body since there is no initialization needed for this example) as well as several optional methods. All overriding methods carry the `override` keyword.

### Implementation file

### Constructor

The constructor calls the base class constructor and sets the flags that tell LAMMPS what kind of output this compute produces:

```
ComputeTemp::ComputeTemp(LAMMPS *lmp, int narg, char **arg) : Compute(lmp, narg, arg)
{
  if (narg != 3) error->all(FLERR, "Illegal compute temp command");

  scalar_flag = vector_flag = 1;
  size_vector = 6;
  extscalar = 0;
  extvector = 1;
  tempflag = 1;

  vector = new double[size_vector];
}
```

The flags have the following meanings:

- `scalar_flag = 1`: this compute provides a global scalar via `compute_scalar()`.
- `vector_flag = 1`: this compute also provides a global vector via `compute_vector()`.
- `size_vector = 6`: the vector has 6 components (the six components of the kinetic energy tensor).
- `extscalar = 0`: the scalar is *intensive* (does not scale with the number of atoms since temperature is an intensive property).
- `extvector = 1`: each vector component is *extensive* (since kinetic energy scales with the number of atoms).
- `tempflag = 1`: marks this compute as providing a temperature, which is used by thermostat fixes to find a compatible temperature compute.

The `vector` array is allocated in the constructor; the base class `Compute` provides a pointer `vector` that derived classes should point to their allocated storage.

The destructor frees the `vector` array if `copymode` is false. The `copymode` flag is set to `0` by default and set to `1` by the `ComputeTempKokkos` class. This class is derived from `ComputeTemp` and replaces the storage assigned to `vector` with a Kokkos specific storage object to facilitate convenient data transfer between host and accelerator devices:

```
ComputeTemp::~ComputeTemp()
{
  if (!copymode) delete[] vector;
}
```

### The init() method (required)

The `init()` method must be overridden in every compute style; it is pure virtual in the base class. It is called once before each run or minimization to perform one-time initialization (e.g., checking that all required fixes or computes exist, requesting neighbor lists, etc.). Here, `init()` has nothing to do and is given an empty body inline in the header.

### The setup() method (optional)

The optional `setup()` method is called at the start of each run, after `init()`. For `ComputeTemp`, it computes the number of degrees of freedom:

```
void ComputeTemp::setup()
{
  dynamic = 0;
  if (dynamic_user || group->dynamic[igroup]) dynamic = 1;
  dof_compute();
}
```

The `dynamic` flag controls whether the number of degrees of freedom is re-computed every timestep (needed when atoms enter or leave the group or the simulation).

### The compute_scalar() method (optional)

The `compute_scalar()` method computes the temperature and returns it as a double:

```
double ComputeTemp::compute_scalar()
{
  invoked_scalar = update->ntimestep;

  double **v = atom->v;
  double *mass = atom->mass;
  double *rmass = atom->rmass;
  int *type = atom->type;
  int *mask = atom->mask;
  int nlocal = atom->nlocal;

  double t = 0.0;

  if (rmass) {
    for (int i = 0; i < nlocal; i++)
      if (mask[i] & groupbit)
        t += (v[i][0] * v[i][0] + v[i][1] * v[i][1] + v[i][2] * v[i][2]) * rmass[i];
  } else {
    for (int i = 0; i < nlocal; i++)
      if (mask[i] & groupbit)
        t += (v[i][0] * v[i][0] + v[i][1] * v[i][1] + v[i][2] * v[i][2]) * mass[type[i]];
  }
```

(continues on next page)

(continued from previous page)

```
  MPI_Allreduce(&t, &scalar, 1, MPI_DOUBLE, MPI_SUM, world);
  if (dynamic) dof_compute();
  if (dof < 0.0 && natoms_temp > 0.0)
    error->all(FLERR, "Temperature compute degrees of freedom < 0");
  scalar *= tfactor;
  return scalar;
}
```

There are several important points:

- `invoked_scalar` is set to the current timestep number to allow other computes and fixes to check whether the result is fresh and avoid redundant re-computation.
- The group membership is checked via `mask[i] & groupbit`. The `groupbit` variable is defined in the base class `Compute` and corresponds to the group specified in the compute command.
- Atom data may be distributed across MPI ranks. Only *local* atoms (`i < atom->nlocal`) are iterated. `MPI_Allreduce` is used to sum the result across all ranks.
- The result is stored in `scalar` (declared in the base class) and also returned.
- The branch on `rmass` handles the case where atoms have individual masses (e.g. for granular simulations) versus per-type masses.

### The compute_vector() method (optional)

The `compute_vector()` method computes the kinetic energy tensor and stores it in the pre-allocated `vector` array:

```
void ComputeTemp::compute_vector()
{
  invoked_vector = update->ntimestep;

  double **v = atom->v;
  double *mass = atom->mass;
  double *rmass = atom->rmass;
  int *type = atom->type;
  int *mask = atom->mask;
  int nlocal = atom->nlocal;

  double massone, t[6];
  for (int i = 0; i < 6; i++) t[i] = 0.0;

  for (int i = 0; i < nlocal; i++)
    if (mask[i] & groupbit) {
      if (rmass)
        massone = rmass[i];
      else
        massone = mass[type[i]];
      t[0] += massone * v[i][0] * v[i][0];
      t[1] += massone * v[i][1] * v[i][1];
      t[2] += massone * v[i][2] * v[i][2];
      t[3] += massone * v[i][0] * v[i][1];
      t[4] += massone * v[i][0] * v[i][2];
      t[5] += massone * v[i][1] * v[i][2];
    }
```

(continues on next page)

(continued from previous page)

```
  MPI_Allreduce(t, vector, 6, MPI_DOUBLE, MPI_SUM, world);
  for (int i = 0; i < 6; i++) vector[i] *= force->mvv2e;
}
```

The result must be stored in the `vector` array (declared in the base class) and `invoked_vector` must be set to mark the result as fresh.

---

### 4.8.15 Case 2: A per-atom compute

In this section we describe a compute that produces a per-atom quantity. As an example we use *compute ke/atom*, which computes the kinetic energy of each atom. The full implementation can be found in `src/compute_ke_atom.cpp` and `src/compute_ke_atom.h`.

#### Header file

The registration block uses `ComputeStyle(ke/atom,ComputeKEAtom)`. The class definition:

```
class ComputeKEAtom : public Compute {
 public:
  ComputeKEAtom(class LAMMPS *, int, char **);
  ~ComputeKEAtom() override;
  void init() override;
  void compute_peratom() override;
  double memory_usage() override;

 private:
  int nmax;
  double *ke;
};
```

The compute overrides `init()`, `compute_peratom()`, and `memory_usage()`. The per-atom data array `ke` and its current allocated size `nmax` are private members.

#### Constructor

```
ComputeKEAtom::ComputeKEAtom(LAMMPS *lmp, int narg, char **arg) :
    Compute(lmp, narg, arg), ke(nullptr)
{
  if (narg != 3) error->all(FLERR, "Illegal compute ke/atom command");

  peratom_flag = 1;
  size_peratom_cols = 0;

  nmax = 0;
}
```

The flags have the following meanings:

- `peratom_flag = 1`: marks this compute as providing per-atom data via `compute_peratom()`.
- `size_peratom_cols = 0`: the per-atom data is a vector (one value per atom); set to a non-zero value for arrays with multiple values per atom.

The destructor must free ke:

```
ComputeKEAtom::~ComputeKEAtom()
{
  memory->destroy(ke);
}
```

### The init() method (required)

```
void ComputeKEAtom::init()
{
  if (modify->get_compute_by_style(style).size() > 1)
    if (comm->me == 0) error->warning(FLERR, "More than one compute {}", style);
}
```

This `init()` implementation emits a warning if more than one compute of this style exists, since having duplicates wastes computation. Most compute styles implement at least a minimal `init()` that checks for required fixes or computes and requests neighbor lists when needed.

### The compute_peratom() method (optional)

```
void ComputeKEAtom::compute_peratom()
{
  invoked_peratom = update->ntimestep;

  // grow ke array if necessary

  if (atom->nmax > nmax) {
    memory->destroy(ke);
    nmax = atom->nmax;
    memory->create(ke, nmax, "ke/atom:ke");
    vector_atom = ke;
  }

  // compute kinetic energy for each atom in group

  double mvv2e = force->mvv2e;
  double **v = atom->v;
  double *mass = atom->mass;
  double *rmass = atom->rmass;
  int *mask = atom->mask;
  int *type = atom->type;
  int nlocal = atom->nlocal;

  if (rmass)
    for (int i = 0; i < nlocal; i++) {
      if (mask[i] & groupbit) {
        ke[i] = 0.5 * mvv2e * rmass[i] *
            (v[i][0] * v[i][0] + v[i][1] * v[i][1] + v[i][2] * v[i][2]);
      } else
        ke[i] = 0.0;
    }
```

(continues on next page)

(continued from previous page)

```
  else
    for (int i = 0; i < nlocal; i++) {
      if (mask[i] & groupbit) {
        ke[i] = 0.5 * mvv2e * mass[type[i]] *
            (v[i][0] * v[i][0] + v[i][1] * v[i][1] + v[i][2] * v[i][2]);
      } else
        ke[i] = 0.0;
    }
}
```

There are several important points:

- `invoked_peratom` is set to the current timestep to mark the result as fresh.
- The `ke` array is grown if the number of owned+ghost atoms (`atom->nmax`) has increased since the last allocation. The base class pointer `vector_atom` must be updated whenever `ke` is reallocated. Setting `vector_atom = ke` after each reallocation ensures that callers always see the current storage.
- Atoms not in the group have their per-atom value set to 0.0.
- Only *local* atoms (`i < nlocal`) are processed; ghost atoms are not included.

#### The memory_usage() method (optional)

The `memory_usage()` method returns an estimate of the memory usage in bytes. This is used by the *info command* and for diagnostic output:

```
double ComputeKEAtom::memory_usage()
{
  double bytes = (double) nmax * sizeof(double);
  return bytes;
}
```

---

### 4.8.16 Notes on flags and output types

The `Compute` base class provides several flags and member variables that control how the compute's output is used elsewhere in LAMMPS. The most important ones for new compute styles are summarized below.

| Flag | Meaning |
|---|---|
| scalar_flag | 1 if compute_scalar() is implemented |
| vector_flag | 1 if compute_vector() is implemented |
| array_flag | 1 if compute_array() is implemented |
| peratom_flag | 1 if compute_peratom() is implemented |
| local_flag | 1 if compute_local() is implemented |
| size_vector | length of the global vector |
| size_array_rows | number of rows of the global array |
| size_array_cols | number of columns of the global array |
| size_peratom_cols | number of columns in the per-atom array (0=vector) |
| extscalar | 0 if scalar is intensive, 1 if extensive |
| extvector | 0 if each vector component is intensive, 1 extensive |
| extarray | 0 if each array column is intensive, 1 extensive |
| tempflag | 1 if compute produces a temperature |
| pressflag | 1 if compute produces a pressure |
| vector | pointer to the global vector output array |
| array | pointer to the global 2D array output |
| vector_atom | pointer to the per-atom vector output array |
| array_atom | pointer to the per-atom 2D array output |

For the output arrays (vector, array, vector_atom, array_atom), the derived class is responsible for allocating storage and pointing the base class pointer to it. The scalar variable is declared in the base class and need not be separately allocated.

The invoked_scalar, invoked_vector, invoked_array, and invoked_peratom variables should be set to update->ntimestep at the start of the corresponding compute method to allow other routines to detect whether the output is current.

### 4.8.17 Writing a new command style

Command styles allow to do system manipulations or interfaces to the operating system.

In the text below, we will discuss the implementation of one example. As shown on the page for *writing or extending command styles*, in order to implement a new command style, a new class must be written that is either directly or indirectly derived from the Command class. There is just one method that must be implemented: Command::command(). In addition, a custom constructor is needed to get access to the members of the LAMMPS class like the Error class to print out error messages. The Command::command() method processes the arguments passed to the command in the input and executes it. Any other methods would be for the convenience of implementation of the new command.

In general, new command styles should be added to the *EXTRA-COMMAND package*. If you feel that your contribution should be added to a different package, please consult with the *LAMMPS developers* first. The contributed code needs to support the *traditional GNU make build process* **and** the *CMake build process*.

---

### 4.8.18 Case 1: Implementing the geturl command

In this section, we will describe the procedure of adding a simple command style to LAMMPS: the *geturl command* that allows to download files directly without having to rely on an external program like "wget" or "curl". The complete implementation can be found in the files src/EXTRA-COMMAND/geturl.cpp and src/EXTRA-COMMAND/geturl.h of the LAMMPS source code.

### Interfacing the *libcurl* library

Rather than implementing the various protocols for downloading files, we rely on an external library: libcurl library. This requires that the library and its headers are installed. For the traditional GNU make build system, this simply requires edits to the machine makefile to add compilation flags like for other libraries. For the CMake based build system, we need to add some lines to the file `cmake/Modules/Packages/EXTRA-COMMAND.cmake`:

```
find_package(CURL QUIET COMPONENTS HTTP HTTPS)
option(WITH_CURL "Enable libcurl support" ${CURL_FOUND})
if(WITH_CURL)
  find_package(CURL REQUIRED COMPONENTS HTTP HTTPS)
  target_compile_definitions(lammps PRIVATE -DLAMMPS_CURL)
  target_link_libraries(lammps PRIVATE CURL::libcurl)
endif()
```

The first `find_package()` command uses a built-in CMake module to find an existing *libcurl* installation with development headers and support for using the HTTP and HTTPS protocols. The "QUIET" flag ensures that there is no screen output and no error if the search fails. The status of the search is recorded in the "${CURL_FOUND}" variable. That variable sets the default of the WITH_CURL option, which toggles whether support for *libcurl* is included or not.

The second `find_package()` uses the "REQUIRED" flag to produce an error if the WITH_CURL option was set to `True`, but no suitable *libcurl* implementation with development support was found. This construct is used so that the CMake script code inside the `if(WITH_CURL)` and `endif()` block can be expanded later to download and compile *libcurl* as part of the LAMMPS build process, if it is not found locally. The `target_compile_definitions()` function added the define `-DLAMMPS_CURL` to the compilation flags when compiling objects for the LAMMPS library. This allows to always compile the *geturl command*, but use pre-processing to compile in the interface to *libcurl* only when it is present and usable and otherwise stop with an error message about the unavailability of *libcurl* to execute the functionality of the command.

### Header file

The first segment of any LAMMPS source should be the copyright and license statement. Note the marker in the first line to indicate to editors like emacs that this file is a C++ source, even though the .h extension suggests a C source (this is a convention inherited from the very beginning of the C++ version of LAMMPS).

```
/* -*- c++ -*- ----------------------------------------------------------
   LAMMPS - Large-scale Atomic/Molecular Massively Parallel Simulator
   https://www.lammps.org/, Sandia National Laboratories
   LAMMPS development team: developers@lammps.org

   Copyright (2003) Sandia Corporation.  Under the terms of Contract
   DE-AC04-94AL85000 with Sandia Corporation, the U.S. Government retains
   certain rights in this software.  This software is distributed under
   the GNU General Public License.

   See the README file in the top-level LAMMPS directory.
------------------------------------------------------------------------- */
```

Every command style must be registered in LAMMPS by including the following lines of code in the second part of the header after the copyright message and before the include guards for the class definition:

```
#ifdef COMMAND_CLASS
// clang-format off
CommandStyle(geturl,GetURL);
```

(continues on next page)

(continued from previous page)

```
// clang-format on
#else
```

This block between `#ifdef COMMAND_CLASS` and `#else` will be included by the `Input` class in `input.cpp` to build a map of "factory functions" that will create an instance of a Command class and call its `command()` method. The map connects the name of the command `geturl` with the name of the class `GetURL`. During compilation, LAMMPS constructs a file `style_command.h` that contains `#include` statements for all "installed" command styles. Before including `style_command.h` into `input.cpp`, the `COMMAND_CLASS` define is set and the `CommandStyle(name,class)` macro defined. The code of the macro adds the installed command styles to the "factory map" which enables the `Input` to execute the command.

The list of header files to include in `style_command.h` is automatically updated by the build system if there are new files, so the presence of the new header file in the `src/EXTRA-COMMAND` folder and the enabling of the EXTRA-COMMAND package will trigger LAMMPS to include the new command style when it is (re-)compiled. The "// clang-format" format comments are needed so that running *clang-format* on the file will not insert unwanted blanks which would break the `CommandStyle` macro.

The third part of the header file is the actual class definition of the `GetURL` class. This has the custom constructor and the `command()` method implemented by this command style. For the constructor there is nothing to do but to pass the `lmp` pointer to the base class. Since the `command()` method is labeled "virtual" in the base class, it must be given the "override" property.

```
#ifndef LMP_GETURL_H
#define LMP_GETURL_H

#include "command.h"

namespace LAMMPS_NS {

class GetURL : public Command {
 public:
  GetURL(class LAMMPS *lmp) : Command(lmp) {};
  void command(int, char **) override;
};
}    // namespace LAMMPS_NS
#endif
#endif
```

The "override" property helps to detect unexpected mismatches because compilation will stop with an error in case the signature of a function is changed in the base class without also changing it in all derived classes.

### Implementation file

We move on to the implementation of the `GetURL` class in the `geturl.cpp` file. This file also starts with a LAMMPS copyright and license header. Below that notice is typically the space where comments may be added with additional information about this specific file, the author(s), affiliation(s), and email address(es). This way the contributing author(s) can be easily contacted, when there are questions about the implementation later. Since the file(s) may be around for a long time, it is beneficial to use some kind of "permanent" email address, if possible.

```
/* ----------------------------------------------------------------------
   LAMMPS - Large-scale Atomic/Molecular Massively Parallel Simulator
   https://www.lammps.org/, Sandia National Laboratories
   LAMMPS development team: developers@lammps.org
```

(continues on next page)

(continued from previous page)

```
   Copyright (2003) Sandia Corporation.  Under the terms of Contract
   DE-AC04-94AL85000 with Sandia Corporation, the U.S. Government retains
   certain rights in this software.  This software is distributed under
   the GNU General Public License.

   See the README file in the top-level LAMMPS directory.
------------------------------------------------------------------------- */

/* ----------------------------------------------------------------------
   Contributing authors:  Axel Kohlmeyer (Temple U),
------------------------------------------------------------------------- */

#include "geturl.h"

#include "comm.h"
#include "error.h"

#if defined(LAMMPS_CURL)
#include <curl/curl.h>
#endif

using namespace LAMMPS_NS;
```

The second section of the implementation file has various include statements. The include file for the class header has to come first, then a couple of LAMMPS classes (sorted alphabetically) followed by the header for the *libcurl* interface. This is wrapped into an `#ifdef` block so that LAMMPS will compile this file without error when the *libcurl* header is not available and thus the define not set. The final statement of this segment imports the `LAMMPS_NS::` namespace globally for this file. This way, all LAMMPS specific functions and classes do not have to be prefixed with `LAMMPS_NS::`.

### The command() function (required)

Since the required custom constructor is trivial and implemented in the header, there is only one function that must be implemented for a command style and that is the `command()` function.

```
void GetURL::command(int narg, char **arg)
{
#if !defined(LAMMPS_CURL)
  error->all(FLERR, "LAMMPS has not been compiled with libcurl support");
#else
  if (narg < 1) utils::missing_cmd_args(FLERR, "geturl", error);
  int verify = 1;
  int overwrite = 1;
  int verbose = 0;
```

This first part also has the `#ifdef` block depending on the LAMMPS_CURL define. This way the command will simply print an error, if *libcurl* is not available but will not fail to compile. Furthermore, it sets the defaults for the following optional arguments.

```
// process arguments

std::string url = arg[0];
```

(continues on next page)

(continued from previous page)

```
// sanity check

if ((url.find(':') == std::string::npos) || (url.find('/') == std::string::npos))
  error->all(FLERR, "URL '{}' is not a supported URL", url);

std::string output = url.substr(url.find_last_of('/') + 1);
if (output.empty()) error->all(FLERR, "URL '{}' must end in a file string", url);
```

This block stores the positional, i.e. non-optional argument of the URL to be downloaded and adds a couple of sanity checks on the string to make sure it is a valid URL. Also it derives the default name of the output file from the URL.

```
int iarg = 1;
while (iarg < narg) {
  if (strcmp(arg[iarg], "output") == 0) {
    if (iarg + 2 > narg) utils::missing_cmd_args(FLERR, "geturl output", error);
    output = arg[iarg + 1];
    ++iarg;
  } else if (strcmp(arg[iarg], "overwrite") == 0) {
    if (iarg + 2 > narg) utils::missing_cmd_args(FLERR, "geturl overwrite", error);
    overwrite = utils::logical(FLERR, arg[iarg + 1], false, lmp);
    ++iarg;
  } else if (strcmp(arg[iarg], "verify") == 0) {
    if (iarg + 2 > narg) utils::missing_cmd_args(FLERR, "geturl verify", error);
    verify = utils::logical(FLERR, arg[iarg + 1], false, lmp);
    ++iarg;
  } else if (strcmp(arg[iarg], "verbose") == 0) {
    if (iarg + 2 > narg) utils::missing_cmd_args(FLERR, "geturl verbose", error);
    verbose = utils::logical(FLERR, arg[iarg + 1], false, lmp);
    ++iarg;
  } else {
    error->all(FLERR, "Unknown geturl keyword: {}", arg[iarg]);
  }
  ++iarg;
}
```

This block parses the optional arguments following the URL and stops with an error if there are arguments missing or an unknown argument is encountered.

```
// only download files from rank 0

if (comm->me != 0) return;

if (!overwrite && platform::file_is_readable(output)) return;

// open output file for writing

FILE *out = fopen(output.c_str(), "wb");
if (!out)
  error->all(FLERR, "Cannot open output file {} for writing: {}", output,␣
 ↪utils::getsyserror());
```

Here all MPI ranks other than 0 will return, so that the URL download will only happen from a single MPI rank. For that rank the output file is opened for writing using the C library function `fopen()`.

```
// initialize curl and perform download

CURL *curl;
curl_global_init(CURL_GLOBAL_DEFAULT);
curl = curl_easy_init();
if (curl) {
  (void) curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
  (void) curl_easy_setopt(curl, CURLOPT_WRITEDATA, (void *) out);
  (void) curl_easy_setopt(curl, CURLOPT_FILETIME, 1L);
  (void) curl_easy_setopt(curl, CURLOPT_FAILONERROR, 1L);
  if (verbose && screen) {
    (void) curl_easy_setopt(curl, CURLOPT_VERBOSE, 1L);
    (void) curl_easy_setopt(curl, CURLOPT_STDERR, (void *) screen);
  }
  if (!verify) {
    (void) curl_easy_setopt(curl, CURLOPT_SSL_VERIFYPEER, 0L);
    (void) curl_easy_setopt(curl, CURLOPT_SSL_VERIFYHOST, 0L);
  }
  auto res = curl_easy_perform(curl);
  if (res != CURLE_OK) {
    long response = 0L;
    curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &response);
    error->one(FLERR, "Download of {} failed with: {} {}", output, curl_easy_
↪strerror(res),
               response);
  }
  curl_easy_cleanup(curl);
```

This block now implements the actual URL download with the selected options via the "easy" interface of *libcurl*. For the details of what these function calls do, please have a look at the *libcurl documentation.

```
  }
  curl_global_cleanup();
  fclose(out);
#endif
}
```

Finally, the previously opened file is closed and the command is complete.

### 4.8.19 Creating styles for the OPENMP package

The OPENMP package in LAMMPS provides an accelerated execution of many LAMMPS styles with multi-threading using OpenMP pragmas. Adding support for a new style to the OPENMP package involves creating a derived class that overrides the standard serial method.

For a high-level discussion on the rationale and parallelization strategy of the OpenMP package, see *OpenMP Parallelism*.

In the text below, we will describe important steps of the procedure of adding multi-thread style to the OPENMP package by looking at a few selected example sections of source files in the OPENMP package.

### Package and build system considerations

The OPENMP package is one of the accelerator packages. New styles should be added to the *src/OPENMP* directory. The contributed code needs to support both the CMake build process and (if applicable) the traditional GNU make process.

### Case study: a pairwise additive model

For many pair styles, adding OpenMP support involves creating a derived class that inherits from the serial pair style class. Let's look at the *pair_lj_cut* style and its OpenMP-accelerated variant.

### Header file

The header file for an OPENMP-enabled pair style (e.g., *pair_lj_cut_omp.h*) typically derives from the original style class (e.g., *PairLJCut*).

```
#ifndef LMP_PAIR_LJ_CUT_OMP_H
#define LMP_PAIR_LJ_CUT_OMP_H

#include "pair_lj_cut.h"
#include "thr_omp.h"

namespace LAMMPS_NS {

class PairLJCutOMP : public PairLJCut, public ThrOMP {
 public:
  PairLJCutOMP(class LAMMPS *);
  virtual ~PairLJCutOMP() {}

  void compute(int, int) override;
  double memory_usage() override;

 private:
  template <int EVFLAG, int EFLAG, int NEWTON_PAIR>
  void eval(int ifrom, int ito, ThrData *const thr);
};
}
```

The class inherits from *PairLJCut* (the base serial style) and *ThrOMP*, a utility class that manages the OpenMP-specific data, such as thread-local force and energy accumulation arrays.

### Constructor and methods

The constructor for an OMP style typically just calls the base class constructor and initializes the *ThrOMP* base:

```
PairLJCutOMP::PairLJCutOMP(LAMMPS *lmp) :
  PairLJCut(lmp), ThrOMP(lmp, THR_PAIR)
{
  respa_enable = 0;
}
```

- The *THR_PAIR* argument tells the *ThrOMP* base class that this is a pair-style acceleration, which influences how data is allocated and managed.
- Setting *respa_enable = 0* is common for OMP accelerated styles as RESPA acceleration often requires separate implementation logic.

### Why only few methods?

You may notice that only a few methods are overridden. Most methods (e.g., *coeff()*, *init_one()*, *write_restart()*) are inherited directly from *PairLJCut*. This is possible because:

- The serial class already handles the data structures and input parsing.
- The *compute()* method is the part that consumes most of the CPU time and is called in every step during a run.
- *single()* is not overridden because it cannot be efficiently parallelized with OpenMP. It represents just one iteration of the inner loop of the main loop constructs of the *compute()* function.

### Utility Functions for Threading

The *ThrOMP* class and related LAMMPS infrastructure provide several key methods to manage parallel execution:

- *loop_setup_thr(int nlocal)*: This method calculates the range of atoms (or neighbors) each thread should process based on the number of threads and the total count. It ensures an even workload distribution across threads.
- *FixOMP::get_thr(int tid)*: This method retrieves the *ThrData* object for a specific thread ID (*tid*). *ThrData* maintains thread-private buffers for forces and energies, avoiding race conditions during the force computation.
- *reduce_thr(ThrData* thr, int eflag, int vflag)*: After the parallel region, this method is used to perform the reduction of per-thread force, energy, and virial contributions into the global arrays (e.g., *atom->f*).

### *ev_setup_thr()* and *ev_tally_thr()*

These methods replace their non-OpenMP counterparts (*ev_setup()* and *ev_tally()*) to support thread-local data:

- *ev_setup_thr(eflag, vflag, thr)*: Unlike *ev_setup()*, which initializes global energy/virial flags, *ev_setup_thr()* initializes the thread-specific *ThrData* structures to track energy and virial contributions independently per thread.
- *ev_tally_thr(i, j, nlocal, newton, evdwl, ecoul, fpair, delx, dely, delz, thr)*: Instead of updating global force/energy arrays directly, *ev_tally_thr()* updates the per-thread tally arrays stored within the provided *ThrData* object (*thr*). This allows multiple threads to record force/energy contributions simultaneously without locking or memory contention.

### Performance Optimization with Templates and *_noalias*

The template-based approach in *eval<EVFLAG, EFLAG, NEWTON_PAIR>()* is a powerful optimization technique:

- **Branch Elimination**: By passing state as template parameters, the compiler can generate specialized machine code for each combination of flags (e.g., *NEWTON_PAIR=1*, *EFLAG=0*). This removes conditional branches (*if* statements) from the inner loop, which is critical for performance because CPU branch predictors often struggle with these toggles in tight loops.
- **`_noalias`**: The *_noalias* keyword (or compiler hint) tells the compiler that pointers (e.g., force arrays) do not point to the same memory addresses as other pointers used in the same context. This enables the compiler to aggressively optimize load/store operations and loop unrolling, as it no longer needs to ensure memory consistency across potential pointer aliasing.

### Porting complex many-body styles (e.g., PairSWOMP, PairTersoffOMP)

Many-body pair styles like Stillinger-Weber (*PairSW*) or Tersoff (*PairTersoff*) differ significantly from pairwise additive potentials. In pairwise additive potentials (like LJ), the force on atom *i* depends only on its neighbors *j*. Many-body styles, however, compute forces based on three-body (or higher-order) terms, meaning the force on atom *i* depends on the environment of atom *j* as well.

### Differences from Pairwise Additive Potentials

1. **Neighbor List Requirements**: Pairwise additive potentials typically require a "half" neighbor list. Many-body potentials, however, usually require a "full" neighbor list to iterate over all neighbors of both atoms *i* and *j* when calculating the three-body interaction *i*-*j*-*k*.
2. **Computational Structure**: Instead of a simple nested loop, many-body potentials involve complex kernels that iterate over neighbors of neighbors. This significantly increases the complexity of the parallel execution and data locality requirements.
3. **Communication**: Due to the three-body nature, the potential needs access to the positions of all atoms within the cutoff of *j*, even if *j* is a ghost atom. This increases the amount of ghost-atom data that must be kept synchronized.

### Case Study: PairEAMOMP

Many-body pair styles like EAM are more complex because they require multiple passes (e.g., one pass to accumulate density, one to compute forces). In *PairEAMOMP*, this is handled by having the *compute()* function orchestrate multiple parallel loops, each calling its own *eval()* variant:

```
// First loop: accumulate density
#pragma omp parallel
{
  eval_rho<...>(...);
}

// Communication step:
comm->forward_comm(this);

// Second loop: compute forces
#pragma omp parallel
{
  eval_force<...>(...);
}
```

### Case Study: PairSWOMP and PairTersoffOMP

In *PairSWOMP* and *PairTersoffOMP*, the acceleration is handled by parallelizing the nested loops while maintaining thread-safety.

- **Data Locality**: Because these styles require a full neighbor list, the *PairTersoffOMP::compute()* function must carefully partition the neighbor list among threads, ensuring that every thread processes a unique set of interactions without redundant calculations.
- **Tallying**: Just as in pairwise styles, *ev_tally_thr()* is used to accumulate energy/virial. However, since the many-body calculation is more involved, the tallying is often called multiple times or with different arguments to properly decompose the contributions of the 2-body and 3-body terms.
- **Kernel Specialization**: Similar to the pairwise case, the *eval()* kernels for many-body styles are heavily templated to handle various scenarios (e.g., different Newton's third law configurations).

When porting these styles, the primary challenge is to ensure that the many-body logic remains correct within a multi-threaded environment. Developers must carefully synchronize data access, especially for shared neighbor list information and thread-local atom data, by utilizing the *ThrData* structure.

### Accelerating bonded force styles

Accelerating bonded force styles follows the same philosophy as pair styles. A *BondLJCutOMP* class would inherit from *BondLJCut* and *ThrOMP*. The same principle applies to angle, dihedral, and improper styles. The *compute()* function again loops over the bonds using OpenMP pragmas, using the same thread-local tallying mechanisms as pair styles.

```
void BondLJCutOMP::compute(int eflag, int vflag)
{
  // ...
  #pragma omp parallel
  {
     // loop over bond list in chunks
     // use ev_tally_thr for tallies
  }
  // ...
}
```

The reduction logic and templated kernels remain nearly identical to those used in pair styles, making it relatively straightforward to port any bonded interaction style to OpenMP.

### Implementation file

The core of the OPENMP implementation resides in the *.cpp* file (e.g., *pair_lj_cut_omp.cpp*).

The *compute()* function is overridden to trigger the parallel execution and handle thread synchronization:

```
void PairLJCutOMP::compute(int eflag, int vflag)
{
  // ... (initialization of evflag, vflag, etc.)

  const int nthreads = comm->nthreads;
  const int inum = list->inum;

  #if defined(_OPENMP)
  #pragma omp parallel default(none) shared(eflag,vflag)
  #endif
  {
    // ... (setup thread-local data)

    if (newton_pair) eval<1,1,1>(...);
    else eval<1,1,0>(...);
  }
  // ... (reduce per-thread forces/energies/virials)
}
```

1. The *compute()* function initializes the OpenMP parallel region using *#pragma omp parallel*.
2. The *eval()* method (a template function) encapsulates the actual force-calculation loop. It uses template parameters (*EVFLAG*, *EFLAG*, *NEWTON_PAIR*) to optimize the loop based on whether energy/virial calculations are enabled and if Newton's third law is used, avoiding runtime conditionals inside the hot loop.
3. The *eval()* method uses *ThrData* to accumulate force and energy contributions safely within each thread.
4. After the parallel loop, the results from each thread's local buffers are reduced into the main force arrays, handling thread-safety during the reduction process.

By templating the *eval()* method, we can compile multiple variants of the kernel at build time, significantly improving performance by removing branches inside the main force loop.

Refer to existing files in *src/OPENMP/* for comprehensive implementation details and complex scenarios involving many-body interactions or communication.

### 4.8.20 Writing or updating documentation

Every contribution that adds a new style or command, or that changes the behavior or the arguments of an existing one, **must** come with matching documentation. This is a *strict requirement* for inclusion into the LAMMPS distribution. The documentation sources are written in reStructuredText (files with a `.rst` suffix in the `doc/src/` folder) and translated to HTML and PDF with Sphinx. All text must be written in American English and the `.rst` files may only contain 7-bit ASCII characters, so they can be cleanly converted to PDF via PDFLaTeX. Special characters and formulas can be included as embedded math typeset in a LaTeX subset.

This page complements the previous sections on *writing new styles*: where those describe the *C++ code* of a style, this one describes the *documentation* that has to accompany it. We use the *pair_style born/gauss* page (file `doc/src/pair_born_gauss.rst`) and the *fix nve* page (file `doc/src/fix_nve.rst`) as running examples, since both follow the established conventions closely and are short enough to quote in full.

A documentation page is not complete by itself: a new style also has to be registered in the corresponding command tables and overview lists, and the build and consistency checks described at the *bottom of this page* must pass without errors or warnings.

### 4.8.21 Anatomy of a style documentation page

A typical per-style documentation page is built from a fixed sequence of elements. The order and the exact wording of the section headers matter, because several of them are verified by automated checks and because common include files (like `accel_styles.rst`) are shared across all pages. The most reliable way to create a new page is to copy the page of a similar, existing style and adapt it.

#### Index entries

The very first lines of a documentation page are the Sphinx index directives. There must be one `.. index::` entry for **every** style that is documented on the page, *including every accelerated variant* (`/gpu`, `/intel`, `/kk`, `/omp`, `/opt`).

```
.. index:: pair_style born/gauss
.. index:: pair_style born/gauss/kk
.. index:: pair_style born/gauss/omp
```

A single documentation file may describe several closely related styles (for example `doc/src/pair_born.rst` documents `born`, `born/coul/long`, `born/coul/msm`, and `born/coul/wolf`). In that case each of those styles, and each of their accelerated variants, needs its own `.. index::` line. The completeness of these index entries is verified by `make style_check` (see *below*), which parses the `...Style(name,class)` macros from the C++ headers and reports any style or accelerated variant whose index entry is missing.

#### Title and command heading

The page title is the name of the command followed by the word "command", underlined with a row of = characters that must be at least as long as the title text.

```
pair_style born/gauss command
=============================
```

When a page documents multiple styles, it is customary to list the primary style in the title (or to use a more generic title), and to make sure each documented style has an index entry and is reachable through the command tables.

In reStructuredText a heading is indicated by underlining it with characters and the the level of a heading is set by kind of character used. The underline must be **at least as long** as the heading text; a shorter underline will trigger an error message, and the convention is to match the length exactly. LAMMPS uses a fixed set of characters for the heading levels so that pages nest consistently. This matters in particular for the combined PDF/LaTeX manual, where all pages are concatenated into a single document and therefore have to share one global hierarchy. From the outermost to the innermost level the characters are:

- `======` – the title of an individual page
- `------` – a section
- `^^^^^^` – a subsection
- `"""""""` – a subsubsection

A per-style reference page needs only two of these levels: `======` for the page title (the line ending in "command") and `"""""""` for each of the standard section headers (Syntax, Examples, Description, ...). Pages with a deeper structure – such as Howto or Programmer Guide pages, and this developer page – additionally use `------` and `^^^^^^` for the intermediate levels.

The page title and the section headers show up as collapsible, expandable entries in the navigation sidebar of the HTML manual (the theme is configured with a navigation depth of three levels) and in the table of contents of the PDF manual.

### Accelerator Variants line

If, and only if, one or more accelerated variants of the style exist, an "Accelerator Variants:" line follows the title. It lists the full names of the accelerated styles in italics.

```
Accelerator Variants: *born/gauss/kk*, *born/gauss/omp*
```

This line and the `.. index::` entries for the accelerated variants must be kept consistent with each other and with the actual accelerated styles present in the source tree. Further down the page, the shared file `accel_styles.rst` is included (see *below*); `make style_check` enforces that a page which lists accelerated index entries also includes `accel_styles.rst` and, conversely, that a page without accelerated styles does not.

### Syntax, Examples, and Description

The body of the page begins with a "Syntax" section. It shows the command in a `LAMMPS` code block and then explains each argument with a bullet list.

```
Syntax
"""""""

.. code-block:: LAMMPS

   pair_style born/gauss cutoff

* born/gauss = name of the pair style
* cutoff = global cutoff (distance units)
```

The "Examples" section gives one or more short, realistic command sequences, again in a `LAMMPS` code block.

```
Examples
""""""""

.. code-block:: LAMMPS

```

(continues on next page)

(continued from previous page)

```
   pair_style born/gauss 10.0
   pair_coeff 1 1 8.2464e13 12.48 0.042644277 0.44 3.56
```

The "Description" section is the main explanatory text. It describes the model or behavior, typically including the functional form as a `.. math::` block and a list of the coefficients with their units.

```
Description
"""""""""""

.. versionadded:: 4Jul2026

Pair style *born/gauss* computes pairwise interactions ...

.. math::

   E = A_0 \exp \left( -\alpha r \right) - A_1 \exp\left[ -\beta \left(r - r_0 \right)^2␣
↪\right]
```

### Version markers

New publicly visible commands or newly added keywords must be marked with `.. versionadded:: TBD`; behavior that changed for an existing command uses `.. versionchanged:: TBD`. The literal text `TBD` is replaced automatically with the upcoming version date at release time. These directives are standalone markers: any explanatory text follows as an ordinary, *unindented* paragraph below the directive, **not** as the indented body of the directive. Version markers do not apply to internal (uppercase) style names.

### Accelerator styles include

For any page that has accelerated variants, a standard block describing how to use accelerated styles is pulled in from the shared file `accel_styles.rst`, set off by horizontal rules:

```
----------

.. include:: accel_styles.rst

----------
```

Using the shared include keeps the description of the accelerator suffixes uniform across all pages, so this text should never be written out by hand.

### Category-specific information sections

After the description, most styles have one or more information sections whose header and content are largely standardized per style category. Copy the wording from a closely related style and adjust it for the behavior of your style. Common examples are:

- For **pair styles**: a section titled "Mixing, shift, table, tail correction, restart, rRESPA info" describing whether the style supports *pair_modify* mixing, the shift, table, and tail options, whether it writes to *binary restart files*, and which *run_style respa* keywords it supports.
- For **fix styles**: a section titled "Restart, fix_modify, output, run start/stop, minimize info" describing what (if anything) the fix writes to restart files, which *fix_modify* options it honors, what global or per-atom quantities it makes available to *output commands*, and whether it is invoked during *energy minimization*.

- For **compute styles**: an "Output info" section describing the shape and meaning of the computed scalar, vector, or array and its units.

These sections should be present even when the answer is "none" or "not supported"; the *fix nve* page is a good template for the minimal case.

### Restrictions, related commands, and defaults

Three short sections close out the page.

The "Restrictions" section documents any conditions that must be met for the style to be usable. The most common one is a **package requirement**: the style is only available when LAMMPS was built with a particular package, with a pointer to the *Build package* page.

```
Restrictions
""""""""""""

This pair style is only enabled if LAMMPS was built with the EXTRA-PAIR
package.  See the :doc:`Build package <Build_package>` page for more
info.
```

Restrictions are *not* limited to package membership, however. Many styles impose additional conditions; copy the exact wording from the page of a similar style. The most common categories are:

- **Package requirement:** only enabled when built with a given package (almost all styles outside the core); always link to *Build package*.
- **Newton setting:** the style requires the *newton* setting to be "on" (common for many-body and communication-heavy pair styles) or, more rarely, "off".
- **Command ordering:** the command must appear before or after another command – for example before the simulation box is defined by *create_box* or *read_data*, after a prerequisite fix, or before any *pair_coeff* commands.
- **Atom style requirement:** the style needs an *atom style* that stores particular per-atom data (e.g. charge, finite size/radius, torque and angular momentum, density or temperature, or support for bonds and angles).
- **Dependency on another command:** the style only works in conjunction with a specific fix, pair style, bond style, or *kspace_style*.
- **Uniqueness or mutual exclusivity:** only a single instance of the style may be defined, or certain keyword combinations cannot be used together.
- **Geometry and dimensionality:** the style only works for 2d or only for 3d systems, or only for orthogonal (not triclinic) simulation boxes.
- **Integrator and parallel constraints:** the style is incompatible with *run_style respa*, is not invoked during *energy minimization*, or has limitations under MPI domain decomposition.

If there are no restrictions at all, the section contains only the single word `none`.

The "Related commands" section is a comma-separated list of links (`:doc:` roles) to closely related commands. The "Default" section describes the default settings of any optional keywords, or the single word `none` if there are none.

### Typical section order

Putting it together, the canonical order of a per-style page is:

1. `.. index::` entries (one per documented style and accelerated variant)
2. title with the `command` suffix
3. "Accelerator Variants:" line (only if accelerated styles exist)

4. Syntax
5. Examples
6. Description (with a `versionadded`/`versionchanged` marker if applicable)
7. `.. include:: accel_styles.rst` block (only if accelerated styles exist)
8. category-specific info section(s) (Mixing... / Restart... / Output info / ...)
9. Restrictions
10. Related commands
11. Default
12. literature citations (if any)

#### Literature citations

If the page cites publications, the citations are placed at the very bottom of the file behind a horizontal rule, each with a unique anchor label and a bold key.

```
----------

.. _Bomont:

**(Bomont)** Bomont, Bretonnet, J. Chem. Phys. 124, 054504 (2006)
```

The in-text reference then uses `:ref:`(Bomont) <Bomont>``. Citation anchor labels must be **unique across all `.rst` files**, so a distinctive label (often including a number or the style name) is needed; `make anchor_check` reports duplicate labels. See the bottom of `doc/fix_nh.rst` for more elaborate examples of citation formatting.

### 4.8.22 Integrating a new style into the command lists

Creating the per-style page is not enough: a new style must also be added to the compact command table and to the alphabetical overview list for its category, or the documentation will build with warnings and `make style_check` will fail. For a pair style, two files need an entry:

- `Commands_pair.rst` contains a multi-column table. Add the style in alphabetical order; the accelerator suffix letters are appended in parentheses (`g` = GPU, `i` = INTEL, `k` = KOKKOS, `o` = OPENMP, `t` = OPT) and the link target is the documentation file:

  ```
  * :doc:`born/gauss (ko) <pair_born_gauss>`
  ```

- `pair_style.rst` contains the alphabetical overview list with a short, one-line description after the link:

  ```
  * :doc:`born/gauss <pair_born_gauss>` - Born-Mayer / Gaussian potential
  ```

The other style categories follow the same pattern with different files. `make style_check` (via `utils/check-styles.py`) checks exactly these files for completeness:

| Category | Compact table file | Overview list file |
|---|---|---|
| pair | Commands_pair.rst | pair_style.rst |
| bond | Commands_bond.rst | bond_style.rst |
| angle | Commands_bond.rst | angle_style.rst |
| dihedral | Commands_bond.rst | dihedral_style.rst |
| improper | Commands_bond.rst | improper_style.rst |
| compute | Commands_compute.rst | compute.rst |
| fix | Commands_fix.rst | fix.rst |
| kspace | Commands_kspace.rst | (none) |
| dump | Commands_dump.rst | (none) |
| command | Commands_all.rst | (none) |

The suffix letters in the compact table must be listed in the canonical order `g i k o t`, with no separators (for example `ko`, never `ok` or `o,k`).

### 4.8.23 Documenting an accelerated variant of an existing style

When a new accelerated variant (e.g. a `/kk` or `/omp` version) is added for a style that already exists and is already documented, several small edits are required rather than a new page:

1. Add a `.. index::` line for the new accelerated style near the top of the existing page.
2. Add or extend the "Accelerator Variants:" line below the title so it lists the new variant.
3. Make sure the page includes the shared `accel_styles.rst` block; if the style previously had no accelerated variants, add that include (with surrounding horizontal rules).
4. Add the corresponding suffix **letter** to the style's entry in the compact command table (`Commands_pair.rst`, `Commands_fix.rst`, ...), keeping the canonical `g i k o t` letter order.

The overview list file (`pair_style.rst`, `fix.rst`, ...) does not encode the suffix letters and therefore does not need to change for a new accelerated variant.

### 4.8.24 Adding and referencing example inputs

For most new features it is preferred (see *the examples requirement*) to add a small, fast example input under the `examples` or `examples/PACKAGES` directory. The documentation page should point the reader to it. Two conventions are in common use; both are acceptable:

- mention the directory inside the "Description" or "Examples" section, for instance a line like `Example input scripts available: examples/PACKAGES/foo`;
- reference specific input files by their path in the relevant discussion.

### 4.8.25 Other documentation that may need updating

Depending on the nature of the contribution, additional documentation may have to be created or updated:

- **New package:** add a description to `Packages_details.rst` and an entry to the package list, and, if the package needs special build steps, add the corresponding build instructions.
- **Complex features:** if a feature needs more background or a multi-command workflow to be used correctly, contribute a *Howto document* and add it to the toctree in `Howto.rst`.
- **Library or API changes:** additions or changes to the C++ library interface or the Fortran module require doxygen comments in the source and matching updates to the "Programmer Guide" section of the manual.

- **Error explanations:** errors with multiple possible causes can be given a detailed explanation on the `Error_details` page with an error code anchor, referenced from the code via *`utils::errorurl()`* (see the *error message requirements*).
- **Figures and PDFs:** inline figures live in `doc/JPG` and supplementary PDF files in `doc/PDF`; see existing pages for how to embed them.

### 4.8.26 Building and checking the documentation

Before submitting, build and proofread the documentation from within the `doc` directory:

```
make html          # build the HTML manual; there should be no warnings
make spelling      # run the spell checker
make anchor_check  # check for duplicate anchors and labels
make style_check   # check completeness of style tables and index entries
make check         # run all consistency checks at once
```

The `make check` target bundles `anchor_check`, `style_check`, `package_check`, `char_check`, and `role_check`. The `char_check` target rejects any non-ASCII character in the `.rst` files, and the `role_check` target catches malformed roles and directives (for example a `:doc:` or `:ref:` role not immediately followed by a backtick, a `text <target>` hyperlink missing its role, or a directive written with a single colon instead of `::`). Carefully read and proofread the generated HTML page before opening the pull request.

If the spell checker reports a false positive (such as an author surname or an established acronym), add it to `doc/utils/sphinx-config/false_positives.txt`.

## 4.9 Notes for developers and code maintainers

This section documents how some of the code functionality within LAMMPS works at a conceptual level. Comments on code in source files typically document what a variable stores, what a small section of code does, or what a function does and its input/outputs. The topics on this page are intended to document code functionality at a higher level.

**Available notes**

- *Reading and parsing of text and text files*
- *Requesting and accessing neighbor lists*
- *Errors, warnings, and informational messages*
- *Choosing between a custom atom style, fix property/atom, and fix STORE/ATOM*
- *Fix contributions to instantaneous energy, virial, and cumulative energy*
- *KSpace PPPM FFT grids*

### 4.9.1 Reading and parsing of text and text files

Classes in LAMMPS frequently need to read in additional data from a file, e.g. potential parameters from a potential file for manybody potentials. LAMMPS provides several custom classes and convenience functions to simplify the process. They offer the following benefits:

- better code reuse and fewer lines of code needed to implement reading and parsing data from a file
- better detection of format errors, incompatible data, and better error messages

- exit with an error message instead of silently converting only part of the text to a number or returning a 0 on unrecognized text and thus reading incorrect values
- re-entrant code through avoiding global static variables (as used by `strtok()`)
- transparent support for translating unsupported UTF-8 characters to their ASCII equivalents (the text-to-value conversion functions **only** accept ASCII characters)

In most cases (e.g. potential files) the same data is needed on all MPI ranks. Then it is best to do the reading and parsing only on MPI rank 0, and communicate the data later with one or more `MPI_Bcast()` calls. For reading generic text and potential parameter files the custom classes *TextFileReader* and *PotentialFileReader* are available. They allow reading the file as individual lines for which they can return a tokenizer class (see below) for parsing the line. Or they can return blocks of numbers as a vector directly. The documentation on *File reader classes* contains an example for a typical case.

When reading per-atom data, the data on each line of the file usually needs to include an atom ID so it can be associated with a particular atom. In that case the data can be read in multi-line chunks and broadcast to all MPI ranks with *utils::read_lines_from_file()*. Those chunks are then split into lines, parsed, and applied only to atoms the MPI rank "owns".

For splitting a string (incrementally) into words and optionally converting those to numbers, the *Tokenizer* and *ValueTokenizer* can be used. Those provide a superset of the functionality of `strtok()` from the C-library and the latter also includes conversion to different types. Any errors while processing the string in those classes will result in an exception, which can be caught and the error processed as needed. Unlike the C-library functions `atoi()`, `atof()`, `strtol()`, or `strtod()` the conversion will check if the converted text is a valid integer or floating point number and will not silently return an unexpected or incorrect value. For example, `atoi()` will return 12 when converting "12.5", while the ValueTokenizer class will throw an *InvalidIntegerException* if *ValueTokenizer::next_int()* is called on the same string.

### 4.9.2 Requesting and accessing neighbor lists

LAMMPS uses Verlet-style neighbor lists to avoid having to loop over *all* pairs of *all* atoms when computing pairwise properties with a cutoff (e.g. pairwise forces or radial distribution functions). There are three main algorithms that can be selected by the *neighbor command*: *bin* (the default, uses binning to achieve linear scaling with system size), *nsq* (without binning, quadratic scaling), *multi* (with binning, optimized for varying cutoffs or polydisperse granular particles). In addition to how the neighbor lists are constructed a number of different variants of neighbor lists need to be created (e.g. "full" or "half") for different purposes and styles and those may be required in every time step ("perpetual") or on some steps ("occasional").

The neighbor list creation is managed by the `Neighbor` class. Individual classes can obtain a neighbor list by creating an instance of a `NeighRequest` class which is stored in a list inside the `Neighbor` class. The `Neighbor` class will then analyze the various requests and apply optimizations where neighbor lists that have the same settings will be created only once and then copied, or a list may be constructed by processing a neighbor list from a different request that is a superset of the requested list. The neighbor list build is then *processed in parallel*.

The most commonly required neighbor list is a so-called "half" neighbor list, where each pair of atoms is listed only once (except when the *newton command setting* for pair is off; in that case pairs straddling subdomains or periodic boundaries will be listed twice). Thus these are the default settings when a neighbor list request is created in:

```
void Pair::init_style()
{
  neighbor->add_request(this);
}

void Pair::init_list(int /*id*/, NeighList *ptr)
{
```

(continues on next page)

(continued from previous page)

```
  list = ptr;
}
```

The `this` pointer argument is required so the neighbor list code can access the requesting class instance to store the assembled neighbor list with that instance by calling its `init_list()` member function. The optional second argument (omitted here) contains a bitmask of flags that determines the kind of neighbor list requested. The default value used here asks for a perpetual "half" neighbor list.

Non-default values of the second argument need to be used to adjust a neighbor list request to the specific needs of a style. The *tersoff* pair style, for example, needs a "full" neighbor list:

```
void PairTersoff::init_style()
{
  // [...]
  neighbor->add_request(this, NeighConst::REQ_FULL);
}
```

When a pair style supports r-RESPA time integration with different cutoff regions, the request flag may depend on the corresponding r-RESPA settings. Here is an example from pair style lj/cut:

```
void PairLJCut::init_style()
{
  int list_style = NeighConst::REQ_DEFAULT;

  if (update->whichflag == 1 && utils::strmatch(update->integrate_style, "^respa")) {
    auto respa = (Respa *) update->integrate;
    if (respa->level_inner >= 0) list_style = NeighConst::REQ_RESPA_INOUT;
    if (respa->level_middle >= 0) list_style = NeighConst::REQ_RESPA_ALL;
  }
  neighbor->add_request(this, list_style);
  // [...]
}
```

Granular pair styles need neighbor lists based on particle sizes and not cutoff and also may need to store data across timesteps ("history"). For example with:

```
if (use_history) neighbor->add_request(this, NeighConst::REQ_SIZE | NeighConst::REQ_
↪HISTORY);
else neighbor->add_request(this, NeighConst::REQ_SIZE);
```

In case a class would need to make multiple neighbor list requests with different settings, each request can set an id which is then used in the corresponding `init_list()` function to assign it to the suitable pointer variable. This is done for example by the *pair style meam*:

```
void PairMEAM::init_style()
{
// [...]
  neighbor->add_request(this, NeighConst::REQ_FULL)->set_id(1);
  neighbor->add_request(this)->set_id(2);
}
void PairMEAM::init_list(int id, NeighList *ptr)
{
  if (id == 1) listfull = ptr;
```

(continues on next page)

(continued from previous page)

```
  else if (id == 2) listhalf = ptr;
}
```

Fixes may require a neighbor list that is only build occasionally (or just once) and this can also be indicated by a flag. As an example here is the request from the `FixPeriNeigh` class which is created internally by *Peridynamics pair styles*:

```
neighbor->add_request(this, NeighConst::REQ_FULL | NeighConst::REQ_OCCASIONAL);
```

It is also possible to request a neighbor list that uses a different cutoff than what is usually inferred from the pair style settings (largest cutoff of all pair styles plus neighbor list skin). The following is used in the *compute rdf* command implementation:

```
if (cutflag)
  neighbor->add_request(this, NeighConst::REQ_OCCASIONAL)->set_cutoff(mycutneigh);
else
  neighbor->add_request(this, NeighConst::REQ_OCCASIONAL);
```

The neighbor list request function has a slightly different set of arguments when created by a command style. In this case the neighbor list is *always* an occasional neighbor list, so that flag is not needed. However for printing the neighbor list summary the name of the requesting command should be set. Below is the request from the *delete atoms* command:

```
neighbor->add_request(this, "delete_atoms", NeighConst::REQ_FULL);
```

### 4.9.3 Errors, warnings, and informational messages

LAMMPS has specialized functionality to handle errors (which should terminate LAMMPS), warning messages (which should indicate possible problems *without* terminating LAMMPS), and informational text for messages about the progress and chosen settings. We *strongly* encourage using these facilities and to *stay away* from using `printf()` or `fprintf()` or `std::cout` or `std::cerr` and calling `MPI_Abort()` or `exit()` directly. Warnings and informational messages should be printed only on MPI rank 0 to avoid flooding the output when running in parallel with many MPI processes.

**Errors**

When LAMMPS encounters an error, for example a syntax error in the input, then a suitable error message should be printed giving a brief, one line remark about the reason and then call either `Error::all()` or `Error::one()`. `Error::all()` must be called when the failing code path is executed by *all* MPI processes and the error condition will appear for *all* MPI processes the same. If desired, each MPI process may set a flag to either 0 or 1 and then MPI_Allreduce() searching for the maximum can be used to determine if there was an error on *any* of the MPI processes and make this information available to *all*. `Error::one()` in contrast needs to be called when only one or a few MPI processes execute the code path or can have the error condition. `Error::all()` is generally the preferred option.

Calling these functions does not abort LAMMPS directly, but rather throws either a `LAMMPSException` (from `Error::all()`) or a `LAMMPSAbortException` (from `Error::one()`). These exceptions are caught by the LAMMPS `main()` program and then handled accordingly. The reason for this approach is to support applications, especially graphical applications like *LAMMPS-GUI*, that are linked to the LAMMPS library and have a mechanism to avoid that an error in LAMMPS terminates the application. By catching the exceptions, the application can delete the failing LAMMPS class instance and create a new one to try again. In a similar fashion, the *LAMMPS Python module* checks for this and then re-throws a corresponding Python exception, which in turn can be caught by the calling Python code.

There are multiple "signatures" that can be called:

- `Error::all(FLERR, "Error message")`: this will abort LAMMPS with the error message "Error message", followed by the last line of input that was read and processed before the error condition happened.

- `Error::all(FLERR, Error::NOLASTLINE, "Error message")`: this is the same as before but without the last line of input. This is preferred for errors that would happen *during* a *run* or *minimization*, since the last line would be showing the "run" or "minimize" command yet those are unrelated to the command *causing* the error.
- `Error::all(FLERR, idx, "Error message")`: this is for argument parsing where "idx" is the index (starting at 0) of the argument for a LAMMPS command that is causing the failure (use `Error::COMMAND` for the command itself). For index 0, i.e. the first command argument, you need to use the constant `Error::ARGZERO` to work around the inability of some compilers to disambiguate between a NULL pointer and an integer constant 0, even with an added type cast. The output may also include the last input line *before* and *after* substituting variables, if they differ. A textual indicator is pointing to the specific word that failed. Using the constant `Error::NOPOINTER` in place of the *idx* argument will suppress the marker and then the behavior is like the *idx* argument is not provided.

FLERR is a macro containing the filename and line where the Error class is called and that information is appended to the error message. This allows to quickly find the relevant source code causing the error. For all three signatures, the single string "Error message" may be replaced with a format string using '{}' placeholders and followed by a variable number of arguments, one for each placeholder. This format string and the arguments are then handed for formatting to the {fmt} library (which is bundled with LAMMPS) or the C++ format library (for C++20 and later) and thus allow processing similar to the "format()" functionality in Python.

**Note**

Commands that accept wildcard arguments, for example *fix ave/time*, use `utils::expand_args()` to convert the wildcards into a list of explicit arguments. This function accepts a pointer address as an optional argument, which will be set to a map to the original arguments from the expanded argument indices. Please see the corresponding source code for details on how to apply this map in error messages.

For complex errors, that can have multiple causes and which cannot be explained in a single line, you can append to the error message, the string created by `utils::errorurl()`, which then provides a URL pointing to a paragraph of the *Errors and warnings details* that corresponds to the number provided. Example:

```
error->all(FLERR, "Unknown identifier in data file: {}{}", keyword, utils::errorurl(1));
```

This will output something like this:

```
ERROR: Unknown identifier in data file: Massess
For more information see https://docs.lammps.org/err0001 (src/read_data.cpp:1482)
Last input line: read_data       data.peptide
```

Where the URL points to the first paragraph with explanations on the *Errors and warnings details* page in the manual.

**Warnings**

To print warnings, the `Errors::warning()` function should be used. It also requires the FLERR macros as first argument to easily identify the location of the warning in the source code. Same as with the error functions above, the function has two variants: one just taking a single string as final argument and a second that uses the {fmt} library to make it similar to, say, `fprintf()`. One motivation to use this function is that it will output warnings with always the same capitalization of the leading "WARNING" string. A second is that it has a built in rate limiter. After a given number (by default 100), that can be set via the *thermo_modify command* no more warnings are printed. Also, warnings are written consistently to both screen and logfile or not, depending on the settings for *screen* or *logfile* output.

**Note**

Unlike `Error::all()`, the warning function will produce output on *every* MPI process, so it typically would be

prefixed with an if statement testing for `comm->me == 0`, i.e. limiting output to MPI rank 0.

**Informational messages**

Finally, for informational message LAMMPS has the *utils::logmesg() convenience function*. It also uses the {fmt} library to support using a format string followed by a matching number of arguments. It will output the resulting formatted text to both, the screen and the logfile and will honor the corresponding settings about whether this output is active and to which file it should be send. Same as for `Error::warning()`, it would produce output for every MPI process and thus should usually be called only on MPI rank 0 to avoid flooding the output when running with many parallel processes.

### 4.9.4 Choosing between a custom atom style, fix property/atom, and fix STORE/ATOM

There are multiple ways to manage per-atom data within LAMMPS. Often the per-atom storage is only used locally and managed by the class that uses it. If the data has to persist between multiple time steps and migrate with atoms when they move from sub-domain to sub-domain or across periodic boundaries, then using a custom atom style, or *fix property/atom*, or the internal fix STORE/ATOM are possible options.

- Using the atom style is usually the most programming effort and mostly needed when the per-atom data is an integral part of the model like a per-atom charge or diameter and thus should be part of the Atoms section of a *data file*.
- Fix property/atom is useful if the data is optional or should be entered by the user, or accessed as a (named) custom property. In this case the fix should be entered as part of the input (and not internally) which allows to enter and store its content with data files.
- Fix STORE/ATOM should be used when the data should be accessed internally only and thus the fix can be created internally.

### 4.9.5 Fix contributions to instantaneous energy, virial, and cumulative energy

Fixes can calculate contributions to the instantaneous energy and/or virial of the system, both in a global and peratom sense. Fixes that perform thermostatting or barostatting can calculate the cumulative energy they add to or subtract from the system, which is accessed by the *ecouple* and *econserve* thermodynamic keywords. This subsection explains how both work and what flags to set in a new fix to enable this functionality.

Let's start with thermostatting and barostatting fixes. Examples are the *fix langevin* and *fix npt* commands. Here is what the fix needs to do:

- Set the variable *ecouple_flag* = 1 in the constructor. Also set *scalar_flag* = 1, *extscalar* = 1, and *global_freq* to a timestep increment which matches how often the fix is invoked.
- Implement a compute_scalar() method that returns the cumulative energy added or subtracted by the fix, e.g. by rescaling the velocity of atoms. The sign convention is that subtracted energy is positive, added energy is negative. This must be the total energy added to the entire system, i.e. an "extensive" quantity, not a per-atom energy. Cumulative means the summed energy since the fix was instantiated, even across multiple runs. This is because the energy is used by the *econserve* thermodynamic keyword to check that the fix is conserving the total energy of the system, i.e. potential energy + kinetic energy + coupling energy = a constant.

And here is how the code operates:

- The Modify class makes a list of all fixes that set *ecouple_flag* = 1.
- The *thermo_style custom* command defines *ecouple* and *econserve* keywords.
- These keywords sum the energy contributions from all the *ecouple_flag* = 1 fixes by invoking the *energy_couple()* method in the Modify class, which calls the *compute_scalar()* method of each fix in the list.

Next, here is how a fix contributes to the instantaneous energy and virial of the system. First, it sets any or all of these flags to a value of 1 in their constructor:

- *energy_global_flag* to contribute to global energy, example: *fix indent*
- *energy_peratom_flag* to contribute to peratom energy, *fix cmap*
- *virial_global_flag* to contribute to global virial, example: *fix wall*
- *virial_peratom_flag* to contribute to peratom virial, example: *fix wall*

The fix must also do the following:

- For global energy, implement a compute_scalar() method that returns the energy added or subtracted on this timestep. Here the sign convention is that added energy is positive, subtracted energy is negative.
- For peratom energy, invoke the ev_init(eflag,vflag) function each time the fix is invoked, which initializes peratom energy storage. The value of eflag may need to be stored from an earlier call to the fix during the same timestep. See how the *fix cmap* command does this in src/MOLECULE/fix_cmap.cpp. When an energy for one or more atoms is calculated, invoke the ev_tally() function to tally the contribution to each atom. Both the ev_init() and ev_tally() methods are in the parent Fix class.
- For global and/or peratom virial, invoke the v_init(vflag) function each time the fix is invoked, which initializes virial storage. When forces on one or more atoms are calculated, invoke the v_tally() function to tally the contribution. Both the v_init() and v_tally() methods are in the parent Fix class. Note that there are several variants of v_tally(); choose the one appropriate to your fix.

**Note**

The ev_init() and ev_tally() methods also account for global and peratom virial contributions. Thus you do not need to invoke the v_init() and v_tally() methods if the fix also calculates peratom energies.

The fix must also specify whether (by default) to include or exclude these contributions to the global/peratom energy/virial of the system. For the fix to include the contributions, set either or both of these variables in the constructor:

- *thermo_energy* = 1, for global and peratom energy
- *thermo_virial* = 1, for global and peratom virial

Note that these variables are zeroed in fix.cpp. Thus if you don't set the variables, the contributions will be excluded (by default).

However, the user has ultimate control over whether to include or exclude the contributions of the fix via the *fix modify* command:

- fix modify *energy yes* to include global and peratom energy contributions
- fix modify *virial yes* to include global and peratom virial contributions

If the fix contributes to any of the global/peratom energy/virial values for the system, it should be explained on the fix doc page, along with the default values for the *energy yes/no* and *virial yes/no* settings of the *fix modify* command.

Finally, these 4 contributions are included in the output of 4 computes:

- global energy in *compute pe*
- peratom energy in *compute pe/atom*
- global virial in *compute pressure*
- peratom virial in *compute stress/atom*

These computes invoke a method of the Modify class to include contributions from fixes that have the corresponding flags set, e.g. *energy_peratom_flag* and *thermo_energy* for *compute pe/atom*.

Note that each compute has an optional keyword to either include or exclude all contributions from fixes. Also note that *compute pe* and *compute pressure* are what is used (by default) by *thermodynamic output* to calculate values for its *pe* and *press* keywords.

### 4.9.6 KSpace PPPM FFT grids

The various *KSpace PPPM* styles in LAMMPS use FFTs to solve Poisson's equation. This subsection describes:

- how FFT grids are defined
- how they are decomposed across processors
- how they are indexed by each processor
- how particle charge and electric field values are mapped to/from the grid

An FFT grid cell is a 3d volume; grid points are corners of a grid cell and the code stores values assigned to grid points in vectors or 3d arrays. A global 3d FFT grid has points indexed 0 to N-1 inclusive in each dimension.

Each processor owns two subsets of the grid, each subset is brick-shaped. Depending on how it is used, these subsets are allocated as a 1d vector or 3d array. Either way, the ordering of values within contiguous memory x fastest, then y, z slowest.

For the `3d decomposition` of the grid, the global grid is partitioned into bricks that correspond to the subdomains of the simulation box that each processor owns. Often, this is a regular 3d array (Px by Py by Pz) of bricks, where P = number of processors = Px * Py * Pz. More generally it can be a tiled decomposition, where each processor owns a brick and the union of all the bricks is the global grid. Tiled decompositions are produced by load balancing with the RCB algorithm; see the *balance rcb* command.

For the `FFT decompostion` of the grid, each processor owns a brick that spans the entire x dimension of the grid while the y and z dimensions are partitioned as a regular 2d array (P1 by P2), where P = P1 * P2.

The following indices store the inclusive bounds of the brick a processor owns, within the global grid:

```
nFOO_in = 3d decomposition brick
nFOO_fft = FFT decomposition brick
nFOO_out = 3d decomposition brick + ghost cells
```

where `FOO` corresponds to `xlo, xhi, ylo, yhi, zlo,` or `zhi`.

The `in` and `fft` indices are from 0 to N-1 inclusive in each dimension, where N is the grid size.

The `out` indices index an array which stores the `in` subset of the grid plus ghost cells that surround it. These indices can thus be < 0 or >= N.

The number of ghost cells a processor owns in each of the 6 directions is a function of:

```
neighbor skin distance (since atoms can move outside a proc subdomain)
qdist = offset or charge from atom due to TIP4P fictitious charge
order = mapping stencil size
shift = factor used when order is an even number (see below)
```

Here is an explanation of how the PPPM variables `order`, `nlower` / `nupper`, `shift`, and `OFFSET` work. They are the relevant variables that determine how atom charge is mapped to grid points and how field values are mapped from grid points to atoms:

```
order = # of nearby grid points in each dim that atom charge/field are mapped to/from
nlower,nupper = extent of stencil around the grid point an atom is assigned to
OFFSET = large integer added/subtracted when mapping to avoid int(-0.75) = 0 when -1 is␣
↪the desired result
```

The particle_map() method assigns each atom to a grid point.

If order is even, say 4:

```
atom is assigned to grid point to its left (in each dim)
shift = OFFSET
nlower = -1, nupper = 2, which are offsets from assigned grid point
window of mapping grid pts is thus 2 grid points to left of atom, 2 to right
```

If order is odd, say 5:

```
atom is assigned to left/right grid pt it is closest to (in each dim)
shift = OFFSET + 0.5
nlower = 2, nupper = 2
if point is in left half of cell, then window of affected grid pts is 3 grid points to␣
↪left of atom, 2 to right
if point is in right half of cell, then window of affected grid pts is 2 grid points to␣
↪left of atom, 3 to right
```

These settings apply to each dimension, so that if order = 5, an atom's charge is mapped to 125 grid points that surround the atom.

## 4.10 Notes for updating code written for older LAMMPS versions

This section documents how C++ source files that are available *outside of the LAMMPS source distribution* (e.g. in external USER packages or as source files provided as a supplement to a publication) that are written for an older version of LAMMPS and thus need to be updated to be compatible with the current version of LAMMPS. Due to the active development of LAMMPS it is likely to always be incomplete. Please contact developers@lammps.org in case you run across an issue that is not (yet) listed here. Please also review the latest information about the LAMMPS *programming style conventions*, especially if you are considering to submit the updated version for inclusion into the LAMMPS distribution.

Available topics in mostly chronological order are:

- *Setting flags in the constructor*
- *Rename of pack/unpack_comm() to pack/unpack_forward_comm()*
- *Use ev_init() to initialize variables derived from eflag and vflag*
- *Use utils::count_words() functions instead of atom->count_words()*
- *Use utils::numeric() functions instead of force->numeric()*
- *Use utils::open_potential() function to open potential files*
- *Use symbolic Atom and AtomVec constants instead of numerical values*
- *Simplify customized error messages*
- *Use of "override" instead of "virtual"*
- *Simplified and more compact neighbor list requests*
- *Split of fix STORE into fix STORE/GLOBAL and fix STORE/PERATOM*

- *Rename of fix STORE/PERATOM to fix STORE/ATOM and change of arguments*
- *Use Output::get_dump_by_id() instead of Output::find_dump()*
- *Refactored grid communication using Grid3d/Grid2d classes instead of GridComm*
- *FLERR as first argument to minimum image functions in Domain class*
- *Use utils::logmesg() instead of error->warning()*

---

### 4.10.1 Setting flags in the constructor

As LAMMPS gains additional functionality, new flags may need to be set in the constructor or a class to signal compatibility with such features. Most of the time the defaults are chosen conservatively, but sometimes the conservative choice is the uncommon choice, and then those settings need to be made when updating code.

Pair styles:

- `manybody_flag`: set to 1 if your pair style is not pair-wise additive
- `restartinfo`: set to 0 if your pair style does not store data in restart files

### 4.10.2 Rename of pack/unpack_comm() to pack/unpack_forward_comm()

Changed in version 8Aug2014.

In this change set, the functions to pack/unpack data into communication buffers for *forward communications* were renamed from `pack_comm()` and `unpack_comm()` to `pack_forward_comm()` and `unpack_forward_comm()`, respectively. Also the meaning of the return value of these functions was changed: rather than returning the number of items per atom stored in the buffer, now the total number of items added (or unpacked) needs to be returned. Here is an example from the *PairEAM* class. Of course the member function declaration in corresponding header file needs to be updated accordingly.

Old:

```
int PairEAM::pack_comm(int n, int *list, double *buf, int pbc_flag, int *pbc)
{
  int m = 0;
  for (int i = 0; i < n; i++) {
    int j = list[i];
    buf[m++] = fp[j];
  }
  return 1;
}
```

New:

```
int PairEAM::pack_forward_comm(int n, int *list, double *buf, int pbc_flag, int *pbc)
{
  int m = 0;
  for (int i = 0; i < n; i++) {
    int j = list[i];
    buf[m++] = fp[j];
  }
  return m;
}
```

**Note**

Because the various "pack" and "unpack" functions are defined in the respective base classes as dummy functions doing nothing, and because of the the name mismatch the custom versions in the derived class will no longer be called, there will be no compilation error when this change is not applied. Only calculations will suddenly produce incorrect results because the required forward communication calls will cease to function correctly.

### 4.10.3 Use ev_init() to initialize variables derived from eflag and vflag

Changed in version 29Mar2019.

There are several variables that need to be initialized based on the values of the "eflag" and "vflag" variables and since sometimes there are new bits added and new variables need to be set to 1 or 0. To make this consistent across all styles, there is now an inline function `ev_init(eflag, vflag)` that makes those settings consistently and calls either `ev_setup()` or `ev_unset()`. Example from a pair style:

Old:

```
if (eflag || vflag) ev_setup(eflag, vflag);
else evflag = vflag_fdotr = eflag_global = eflag_atom = 0;
```

New:

```
ev_init(eflag, vflag);
```

Not applying this change will not cause a compilation error, but can lead to inconsistent behavior and incorrect tallying of energy or virial.

### 4.10.4 Use utils::count_words() functions instead of atom->count_words()

Changed in version 2Jun2020.

The "count_words()" functions for parsing text have been moved from the Atom class to the *utils namespace*. The "count_words()" function in "utils" uses the Tokenizer class internally to split a line into words and count them, thus it will not modify the argument string as the function in the Atoms class did and thus had a variant using a copy buffer. Unlike the old version, the new version does not remove comments. For that you can use the *utils::trim_comment() function* as shown in the example below.

Old:

```
nwords = atom->count_words(line);
int nwords = atom->count_words(buf);
```

New:

```
nwords = utils::count_words(line);
int nwords = utils::count_words(utils::trim_comment(buf));
```

**See also**

*utils::count_words()*, *utils::trim_comment()*

### 4.10.5 Use utils::numeric() functions instead of force->numeric()

Changed in version 18Sep2020.

The "numeric()" conversion functions (including "inumeric()", "bnumeric()", and "tnumeric()") have been moved from the Force class to the *utils namespace*. Also they take an additional argument that selects whether the `Error::all()` or `Error::one()` function should be called in case of an error. The former should be used when *all* MPI processes call the conversion function and the latter *must* be used when they are called from only one or a subset of the MPI processes.

Old:

```
val = force->numeric(FLERR, arg[1]);
num = force->inumeric(FLERR, arg[2]);
```

New:

```
val = utils::numeric(FLERR, true, arg[1], lmp);
num = utils::inumeric(FLERR, false, arg[2], lmp);
```

**See also**

*utils::numeric()*, *utils::inumeric()*, *utils::bnumeric()*, *utils::tnumeric()*

### 4.10.6 Use utils::open_potential() function to open potential files

Changed in version 18Sep2020.

The *utils::open_potential()* function must be used to replace calls to `force->open_potential()` and should be used to replace `fopen()` for opening potential files for reading. The custom function does three additional steps compared to `fopen()`: 1) it will try to parse the `UNITS:` and `DATE:` metadata and will stop with an error on a units mismatch and will print the date info, if present, in the log file; 2) for pair styles that support it, it will set up possible automatic unit conversions based on the embedded unit information and LAMMPS' current units setting; 3) it will not only try to open a potential file at the given path, but will also search in the folders listed in the `LAMMPS_POTENTIALS` environment variable. This allows potential files to reside in a common location instead of having to copy them around for simulations.

Old:

```
fp = force->open_potential(filename);
fp = fopen(filename, "r");
```

New:

```
fp = utils::open_potential(filename, lmp);
```

### 4.10.7 Use symbolic Atom and AtomVec constants instead of numerical values

Changed in version 18Sep2020.

Properties in LAMMPS that were represented by integer values (0, 1, 2, 3) to indicate settings in the `Atom` and `AtomVec` classes (or classes derived from it) (and its derived classes) have been converted to use scoped enumerators instead.

| Symbolic Constant | Value | Symbolic Constant | Value | Symbolic Constant | Value |
|---|---|---|---|---|---|
| Atom::GROW | 0 | Atom::ATOMIC | 0 | Atom::MAP_NONE | 0 |
| Atom::RESTART | 1 | Atom::MOLECULAR | 1 | Atom::MAP_ARRAY | 1 |
| Atom::BORDER | 2 | Atom::TEMPLATE | 2 | Atom::MAP_HASH | 2 |
| AtomVec::PER_ATOM | 0 | AtomVec::PER_TYPE | 1 | Atom::MAP_YES | 3 |

Old:

```
molecular = 0;
mass_type = 1;
if (atom->molecular == 2)
if (atom->map_style == 2)
atom->add_callback(0);
atom->delete_callback(id,1);
```

New:

```
molecular = Atom::ATOMIC;
mass_type = AtomVec::PER_TYPE;
if (atom->molecular == Atom::TEMPLATE)
if (atom->map_style == Atom::MAP_HASH)
atom->add_callback(Atom::GROW);
atom->delete_callback(id,Atom::RESTART);
```

### 4.10.8 Simplify customized error messages

Changed in version 14May2021.

Aided by features of the bundled {fmt} library, error messages now can have a variable number of arguments and the string will be interpreted as a {fmt} style format string so that error messages can be easily customized without having to use temporary buffers and `sprintf()`. Example:

Old:

```
if (fptr == NULL) {
  char str[128];
  sprintf(str,"Cannot open AEAM potential file %s",filename);
  error->one(FLERR,str);
}
```

New:

```
if (fptr == nullptr)
  error->one(FLERR, "Cannot open AEAM potential file {}: {}", filename,␣
 ↪utils::getsyserror());
```

### 4.10.9 Use of "override" instead of "virtual"

Changed in version 17Feb2022.

Since LAMMPS requires C++17, we switched to use the "override" keyword instead of "virtual" to indicate polymorphism in derived classes. This allows the C++ compiler to better detect inconsistencies when an override is intended or not. Please note that "override" has to be added to **all** polymorph functions in derived classes and "virtual" *only* to the function in the base class (or the destructor). Here is an example from the `FixWallReflect` class:

Old:

```
FixWallReflect(class LAMMPS *, int, char **);
virtual ~FixWallReflect();
int setmask();
void init();
void post_integrate();
```

New:

```
FixWallReflect(class LAMMPS *, int, char **);
~FixWallReflect() override;
int setmask() override;
void init() override;
void post_integrate() override;
```

This change set will neither cause a compilation failure, nor will it change functionality, but if you plan to submit the updated code for inclusion into the LAMMPS distribution, it will be requested for achieve a consistent *programming style*.

### 4.10.10 Simplified function names for forward and reverse communication

Changed in version 24Mar2022.

Rather than using the function name to distinguish between the different forward and reverse communication functions for styles, LAMMPS now uses the type of the "this" pointer argument.

Old:

```
comm->forward_comm_pair(this);
comm->forward_comm_fix(this);
comm->forward_comm_compute(this);
comm->forward_comm_dump(this);
comm->reverse_comm_pair(this);
comm->reverse_comm_fix(this);
comm->reverse_comm_compute(this);
comm->reverse_comm_dump(this);
```

New:

```
comm->forward_comm(this);
comm->reverse_comm(this);
```

This change is **required** or else the code will not compile.

### 4.10.11 Simplified and more compact neighbor list requests

Changed in version 24Mar2022.

This change set reduces the amount of code required to request a neighbor list. It enforces consistency and no longer requires to change internal data of the request. More information on neighbor list requests can be *found here*. Example from the `ComputeRDF` class:

Old:

```
int irequest = neighbor->request(this,instance_me);
neighbor->requests[irequest]->pair = 0;
neighbor->requests[irequest]->compute = 1;
neighbor->requests[irequest]->occasional = 1;
if (cutflag) {
  neighbor->requests[irequest]->cut = 1;
  neighbor->requests[irequest]->cutoff = mycutneigh;
}
```

New:

```
auto req = neighbor->add_request(this, NeighConst::REQ_OCCASIONAL);
if (cutflag) req->set_cutoff(mycutneigh);
```

Public access to the `NeighRequest` class data members has been removed so this update is **required** to avoid compilation failure.

### 4.10.12 Split of fix STORE into fix STORE/GLOBAL and fix STORE/PERATOM

Changed in version 15Sep2022.

This change splits the GLOBAL and PERATOM modes of fix STORE into two separate fixes STORE/GLOBAL and STORE/PERATOM. There was very little shared code between the two fix STORE modes and the two different code paths had to be prefixed with if statements. Furthermore, some flags were used differently in the two modes leading to confusion. Splitting the code into two fix styles, makes it more easily maintainable. Since these are internal fixes, there is no user visible change.

Old:

```
#include "fix_store.h"

FixStore *fix = dynamic_cast<FixStore *>(
   modify->add_fix(fmt::format("{} {} STORE peratom 1 13",id_pole,group->names[0]));

FixStore *fix = dynamic_cast<FixStore *>(modify->get_fix_by_id(id_pole));
```

New:

```
#include "fix_store_peratom.h"

FixStorePeratom *fix = dynamic_cast<FixStorePeratom *>(
   modify->add_fix(fmt::format("{} {} STORE/PERATOM 1 13",id_pole,group->names[0]));

FixStorePeratom *fix = dynamic_cast<FixStorePeratom *>(modify->get_fix_by_id(id_pole));
```

Old:

```
#include "fix_store.h"

FixStore *fix = dynamic_cast<FixStore *>(
   modify->add_fix(fmt::format("{} {} STORE global 1 1",id_fix,group->names[igroup]));

FixStore *fix = dynamic_cast<FixStore *>(modify->get_fix_by_id(id_fix));
```

New:

```
#include "fix_store_global.h"

FixStoreGlobal *fix = dynamic_cast<FixStoreGlobal *>(
   modify->add_fix(fmt::format("{} {} STORE/GLOBAL 1 1",id_fix,group->names[igroup]));

FixStoreGlobal *fix = dynamic_cast<FixStoreGlobal *>(modify->get_fix_by_id(id_fix));
```

This change is **required** or else the code will not compile.

### 4.10.13 Rename of fix STORE/PERATOM to fix STORE/ATOM and change of arguments

Changed in version 28Mar2023.

The available functionality of the internal fix to store per-atom properties was expanded to enable storing data with ghost atoms and to support binary restart files. With those changes, the fix was renamed to fix STORE/ATOM and the number and order of (required) arguments has changed.

Old syntax: `ID group-ID STORE/PERATOM rflag n1 n2 [n3]`

- *rflag* = 0/1, *no*/*yes* store per-atom values in restart file
- $n1 = 1, n2 = 1$, no $n3 \rightarrow$ per-atom vector, single value per atom
- $n1 = 1, n2 > 1$, no $n3 \rightarrow$ per-atom array, *n2* values per atom
- $n1 = 1, n2 > 0, n3 > 0 \rightarrow$ per-atom tensor, *n2* x *n3* values per atom

New syntax: `ID group-ID STORE/ATOM n1 n2 gflag rflag`

- $n1 = 1, n2 = 0 \rightarrow$ per-atom vector, single value per atom
- $n1 > 1, n2 = 0 \rightarrow$ per-atom array, *n1* values per atom
- $n1 > 0, n2 > 0 \rightarrow$ per-atom tensor, *n1* x *n2* values per atom
- *gflag* = 0/1, *no*/*yes* communicate per-atom values with ghost atoms
- *rflag* = 0/1, *no*/*yes* store per-atom values in restart file

Since this is an internal fix, there is no user visible change.

### 4.10.14 Use Output::get_dump_by_id() instead of Output::find_dump()

Changed in version 15Sep2022.

The accessor function to individual dump style instances has been changed from `Output::find_dump()` returning the index of the dump instance in the list of dumps to `Output::get_dump_by_id()` returning a pointer to the dump directly. Example:

Old:

```
int idump = output->find_dump(arg[iarg+1]);
if (idump < 0)
  error->all(FLERR,"Dump ID in hyper command does not exist");
memory->grow(dumplist,ndump+1,"hyper:dumplist");
dumplist[ndump++] = idump;

[...]
```

(continues on next page)

(continued from previous page)

```
if (dumpflag)
  for (int idump = 0; idump < ndump; idump++)
    output->dump[dumplist[idump]]->write();
```

New:

```
auto idump = output->get_dump_by_id(arg[iarg+1]);
if (!idump) error->all(FLERR,"Dump ID {} in hyper command does not exist", arg[iarg+1]);
dumplist.emplace_back(idump);

[...]

if (dumpflag) for (auto idump : dumplist) idump->write();
```

This change is **required** or else the code will not compile.

### 4.10.15 Refactored grid communication using Grid3d/Grid2d classes instead of GridComm

Changed in version 22Dec2022.

The `GridComm` class was for creating and communicating distributed grids was replaced by the `Grid3d` class with added functionality. A `Grid2d` class was also added for additional flexibility.

The new functionality and commands using the two grid classes are discussed on the following documentation pages:

- *Using distributed grids*
- *Use of distributed grids within style classes*

If you have custom LAMMPS code, which uses the GridComm class, here are some notes on how to adapt it for using the Grid3d class.

(1) The constructor has changed to allow the `Grid3d` / `Grid2d` classes to partition the global grid across processors, both for owned and ghost grid cells. Previously any class which called `GridComm` performed the partitioning itself and that information was passed in the `GridComm::GridComm()` constructor. There are several "set" functions which can be called to alter how `Grid3d` / `Grid2d` perform the partitioning. They should be sufficient for most use cases of the grid classes.

(2) The partitioning is triggered by the `setup_grid()` method.

(3) The `setup()` method of the `GridComm` class has been replaced by the `setup_comm()` method in the new grid classes. The syntax for the `forward_comm()` and `reverse_comm()` methods is slightly altered as is the syntax of the associated pack/unpack callback methods. But the functionality of these operations is the same as before.

(4) The new `Grid3d` / `Grid2d` classes have additional functionality for dynamic load-balancing of grids and their associated data across processors. This did not exist in the `GridComm` class.

This and more is explained in detail on the *Use of distributed grids within style classes* page. The following LAMMPS source files can be used as illustrative examples for how the new grid classes are used by computes, fixes, and various KSpace solvers which use distributed FFT grids:

- `src/fix_ave_grid.cpp`
- `src/compute_property_grid.cpp`
- `src/EXTRA-FIX/fix_ttm_grid.cpp`
- `src/KSPACE/pppm.cpp`

This change is **required** or else the code will not compile.

### 4.10.16 FLERR as first argument to minimum image functions in Domain class

Changed in version 12Jun2025.

The `Domain::minimum_image()` and `Domain::minimum_image_big()` functions were changed to take the `FLERR` macros as first argument. This way the error message indicates *where* the function was called instead of pointing to the implementation of the function. Example:

Old:

```
double delx1 = x[i1][0] - x[i2][0];
double dely1 = x[i1][1] - x[i2][1];
double delz1 = x[i1][2] - x[i2][2];
domain->minimum_image(delx1, dely1, delz1);
double r1 = sqrt(delx1 * delx1 + dely1 * dely1 + delz1 * delz1);

double delx2 = x[i3][0] - x[i2][0];
double dely2 = x[i3][1] - x[i2][1];
double delz2 = x[i3][2] - x[i2][2];
domain->minimum_image_big(delx2, dely2, delz2);
double r2 = sqrt(delx2 * delx2 + dely2 * dely2 + delz2 * delz2);
```

New:

```
double delx1 = x[i1][0] - x[i2][0];
double dely1 = x[i1][1] - x[i2][1];
double delz1 = x[i1][2] - x[i2][2];
domain->minimum_image(FLERR, delx1, dely1, delz1);
double r1 = sqrt(delx1 * delx1 + dely1 * dely1 + delz1 * delz1);

double delx2 = x[i3][0] - x[i2][0];
double dely2 = x[i3][1] - x[i2][1];
double delz2 = x[i3][2] - x[i2][2];
domain->minimum_image_big(FLERR, delx2, dely2, delz2);
double r2 = sqrt(delx2 * delx2 + dely2 * dely2 + delz2 * delz2);
```

This change is **required** or else the code will not compile.

### 4.10.17 Use utils::logmesg() instead of error->warning()

Changed in version 22Jul2025.

The `Error::message()` method has been removed since its functionality has been superseded by the *utils::logmesg()* function.

Old:

```
 if (comm->me == 0) {
   error->message(FLERR, "INFO: About to read data file: {}", filename);
}
```

New:

```
if (comm->me == 0) utils::logmesg(lmp, "INFO: About to read data file: {}\n", filename);
```

This change is **required** or else the code will not compile.

## 4.11 Writing plugins

Plugins provide a mechanism to add functionality to a LAMMPS executable without recompiling LAMMPS. The functionality for this and the *plugin command* are implemented in the *PLUGIN package* which must be installed to use plugins.

Plugins use the operating system's capability to load dynamic shared object (DSO) files in a way similar shared libraries and then reference specific functions in those DSOs. Any DSO file with plugins has to include an initialization function with a specific name, "lammpsplugin_init", that has to follow specific rules described below. When loading the DSO with the "plugin" command, this function is looked up and called and will then register the contained plugin(s) with LAMMPS.

When the environment variable `LAMMPS_PLUGIN_PATH` is set, then LAMMPS will search the directory (or directories) listed in this path for files with names that end in `plugin.so` (e.g. `helloplugin.so`) and will try to load the contained plugins automatically at start-up. For plugins that are loaded this way, the behavior of LAMMPS should be identical to a binary where the corresponding code was compiled in statically as a package.

From the programmer perspective this can work because of the object oriented design of LAMMPS where all pair style commands are derived from the class Pair, all fix style commands from the class Fix and so on and usually only functions present in those base classes are called directly. When a *pair_style command* command or *fix command* command is issued a new instance of such a derived class is created. This is done by a so-called factory function which is mapped to the style name. Thus when, for example, the LAMMPS processes the command `pair_style lj/cut 2.5`, LAMMPS will look up the factory function for creating the `PairLJCut` class and then execute it. The return value of that function is a `Pair *` pointer and the pointer will be assigned to the location for the currently active pair style.

A DSO file with a plugin thus has to implement such a factory function and register it with LAMMPS so that it gets added to the map of available styles of the given category. To register a plugin with LAMMPS an initialization function has to be present in the DSO file called `lammpsplugin_init` which is called with three `void *` arguments: a pointer to the current LAMMPS instance, a pointer to the opened DSO handle, and a pointer to the registration function. The registration function takes two arguments: a pointer to a `lammpsplugin_t` struct with information about the plugin and a pointer to the current LAMMPS instance. Please see below for an example of how the registration is done.

### 4.11.1 Members of `lammpsplugin_t`

| Member | Description |
|---|---|
| version | LAMMPS Version string the plugin was compiled for |
| style | Style of the plugin (pair, bond, fix, command, etc.) |
| name | Name of the plugin style |
| info | String with information about the plugin |
| author | String with the name and email of the author |
| creator.v1 | Pointer to factory function for pair, bond, angle, dihedral, improper, kspace, command, or minimize styles |
| creator.v2 | Pointer to factory function for compute, fix, region, or run styles |
| handle | Pointer to the open DSO file handle |

Only one of the two alternate creator entries can be used at a time and which of those is determined by the style of plugin. The "creator.v1" element is for factory functions of supported styles computing forces (i.e. pair, bond, angle, dihedral, or improper styles), command styles, or minimize styles and the function takes as single argument the pointer to the LAMMPS instance. The factory function is cast to the `lammpsplugin_factory1` type before assignment. The "creator.v2" element is for factory functions creating an instance of a fix, compute, region, or run style and takes three arguments: a pointer to the LAMMPS instance, an integer with the length of the argument list and a `char **` pointer

to the list of arguments. The factory function pointer needs to be cast to the `lammpsplugin_factory2` type before assignment.

### 4.11.2 Pair style example

As an example, a hypothetical pair style plugin "morse2" implemented in a class `PairMorse2` in the files `pair_morse2.h` and `pair_morse2.cpp` with the factory function and initialization function would look like this:

```
#include "lammpsplugin.h"
#include "version.h"
#include "pair_morse2.h"

using namespace LAMMPS_NS;

static Pair *morse2creator(LAMMPS *lmp)
{
  return new PairMorse2(lmp);
}

extern "C" void lammpsplugin_init(void *lmp, void *handle, void *regfunc)
{
  lammpsplugin_regfunc register_plugin = (lammpsplugin_regfunc) regfunc;
  lammpsplugin_t plugin;

  plugin.version = LAMMPS_VERSION;
  plugin.style   = "pair";
  plugin.name    = "morse2";
  plugin.info    = "Morse2 variant pair style v1.0";
  plugin.author  = "Axel Kohlmeyer (akohlmey@gmail.com)";
  plugin.creator.v1 = (lammpsplugin_factory1 *) &morse2creator;
  plugin.handle  = handle;
  (*register_plugin)(&plugin,lmp);
}
```

The factory function in this example is called `morse2creator()`. It receives a pointer to the LAMMPS class as only argument and thus has to be assigned to the *creator.v1* member of the plugin struct and cast to the `lammpsplugin_factory1` function pointer type. It returns a pointer to the allocated class instance derived from the `Pair` class. This function may be declared static to avoid clashes with other plugins. The name of the derived class, `PairMorse2`, however must be unique inside the entire LAMMPS executable.

### 4.11.3 Fix style example

If the factory function is for a fix or compute, which take three arguments (a pointer to the LAMMPS class, the number of arguments and the list of argument strings), then the pointer type is `lammpsplugin_factory2` and it must be assigned to the *creator.v2* member of the plugin struct. Below is an example for that:

```
#include "lammpsplugin.h"
#include "version.h"
#include "fix_nve2.h"

using namespace LAMMPS_NS;

static Fix *nve2creator(LAMMPS *lmp, int argc, char **argv)
{
```

(continues on next page)

(continued from previous page)

```
  return new FixNVE2(lmp,argc,argv);
}

extern "C" void lammpsplugin_init(void *lmp, void *handle, void *regfunc)
{
  lammpsplugin_regfunc register_plugin = (lammpsplugin_regfunc) regfunc;
  lammpsplugin_t plugin;

  plugin.version = LAMMPS_VERSION;
  plugin.style   = "fix";
  plugin.name    = "nve2";
  plugin.info    = "NVE2 variant fix style v1.0";
  plugin.author  = "Axel Kohlmeyer (akohlmey@gmail.com)";
  plugin.creator.v2 = (lammpsplugin_factory2 *) &nve2creator;
  plugin.handle  = handle;
  (*register_plugin)(&plugin,lmp);
}
```

### 4.11.4 Command style example

Command styles also use the first variant of factory function as demonstrated in the following example, which also shows that the implementation of the plugin class may be within the same source file as the plugin interface code:

```
#include "lammpsplugin.h"

#include "comm.h"
#include "error.h"
#include "command.h"
#include "version.h"

#include <cstring>

namespace LAMMPS_NS {
  class Hello : public Command {
   public:
    Hello(class LAMMPS *lmp) : Command(lmp) {};
    void command(int, char **);
  };
}

using namespace LAMMPS_NS;

void Hello::command(int argc, char **argv)
{
   if (argc != 1) error->all(FLERR,"Illegal hello command");
   if (comm->me == 0)
     utils::logmesg(lmp,fmt::format("Hello, {}!\n",argv[0]));
}

static void hellocreator(LAMMPS *lmp)
{
  return new Hello(lmp);
```

(continues on next page)

(continued from previous page)

```
}

extern "C" void lammpsplugin_init(void *lmp, void *handle, void *regfunc)
{
  lammpsplugin_t plugin;
  lammpsplugin_regfunc register_plugin = (lammpsplugin_regfunc) regfunc;

  plugin.version = LAMMPS_VERSION;
  plugin.style   = "command";
  plugin.name    = "hello";
  plugin.info    = "Hello world command v1.1";
  plugin.author  = "Axel Kohlmeyer (akohlmey@gmail.com)";
  plugin.creator.v1 = (lammpsplugin_factory1 *) &hellocreator;
  plugin.handle  = handle;
  (*register_plugin)(&plugin,lmp);
}
```

### 4.11.5 Additional Details

The initialization function **must** be called `lammpsplugin_init`, it **must** have C bindings and it takes three void pointers as arguments. The first is a pointer to the LAMMPS class that calls it and it needs to be passed to the registration function. The second argument is a pointer to the internal handle of the DSO file, this needs to be added to the plugin info struct, so that the DSO can be closed and unloaded when all its contained plugins are unloaded. The third argument is a function pointer to the registration function and needs to be stored in a variable of `lammpsplugin_regfunc` type and then called with a pointer to the `lammpsplugin_t` struct and the pointer to the LAMMPS instance as arguments to register a single plugin. There may be multiple calls to multiple plugins in the same initialization function.

To register a plugin a struct of the `lammpsplugin_t` needs to be filled with relevant info: current LAMMPS version string, kind of style, name of style, info string, author string, pointer to factory function, and the DSO handle. The registration function is called with a pointer to the address of this struct and the pointer of the LAMMPS class. The registration function will then add the factory function of the plugin style to the respective style map under the provided name. It will also make a copy of the struct in a global list of all loaded plugins and update the reference counter for loaded plugins from this specific DSO file.

The pair style itself (i.e. the PairMorse2 class in this example) can be written just like any other pair style that is included in LAMMPS. For a plugin, the use of the `PairStyle` macro in the section encapsulated by `#ifdef PAIR_CLASS` is not needed, since the mapping of the class name to the style name is done by the plugin registration function with the information from the `lammpsplugin_t` struct. It may be included in case the new code is intended to be later included in LAMMPS directly.

A plugin may be registered under an existing style name. In that case the plugin will override the existing code. This can be used to modify the behavior of existing styles or to debug new versions of them without having to re-compile or re-install all of LAMMPS.

Changed in version 12Jun2025.

When using the *clear* command, plugins are not unloaded but restored to their respective style maps. This also applies when multiple LAMMPS instances are created and deleted through the library interface. The *plugin load* load command may be issued again, but for existing plugins they will be skipped. To replace plugins they must be explicitly unloaded with *plugin unload*. When multiple LAMMPS instances are created concurrently, any loaded plugins will be added to the global list of plugins, but are not immediately available to any LAMMPS instance that was created before loading the plugin. To "import" such plugins, the *plugin restore* may be used. Plugins are only removed when they are explicitly unloaded or the LAMMPS interface is "finalized".

### 4.11.6 Compiling plugins

Plugins need to be compiled with the same compilers and libraries (e.g. MPI) and compilation settings (MPI on/off, OpenMP, integer sizes) as the LAMMPS executable and library. Otherwise the plugin will likely not load due to mismatches in the function signatures (LAMMPS is C++ so scope, type, and number of arguments are encoded into the symbol names and thus differences in them will lead to failed plugin load commands). Compilation of the plugin can be managed via both, CMake or traditional GNU makefiles. Some examples that can be used as a template are in the `examples/plugins` folder. The CMake script code has some small adjustments to allow building the plugins for running unit tests with them.

Another example that converts the KIM package into a plugin can be found in the `examples/kim/plugin` folder. No changes to the sources of the KIM package themselves are needed; only the plugin interface and loader code needs to be added. This example only supports building with CMake, but is probably a more typical example. To compile you need to run CMake with `-DLAMMPS_SOURCE_DIR=<path/to/lammps/src/folder>`. Other configuration setting are identical to those for compiling LAMMPS.

A second example for a plugin from a package is in the `examples/PACKAGES/pace/plugin` folder that will create a plugin from the ML-PACE package. In this case the bulk of the code is in a static external library that is being downloaded and compiled first and then combined with the pair style wrapper and the plugin loader. This example also contains a NSIS script that can be used to create an Installer package for Windows (the mutual licensing terms of the external library and LAMMPS conflict when distributing binaries, so the ML-PACE package cannot be linked statically, but the LAMMPS headers required to build the plugin are also available under a less restrictive license). This will automatically set the required environment variable and launching a (compatible) LAMMPS binary will load and register the plugin and the ML-PACE package can then be used as it was linked into LAMMPS.

---

You can find additional LAMMPS plugins in the LAMMPS plugins source code repository on GitHub

## 4.12 Adding tests for unit testing


- *Tests for utility functions*
  - *FFT Testing Infrastructure*
- *Tests for individual LAMMPS commands*
- *Tests for the C-style library interface*
- *Tests for the Python module and package*
- *Tests for the Fortran interface*
- *Tests for the C++-style library interface*
- *Tests for reading and writing file formats*
- *Tests for styles computing or modifying forces*
- *Tests for granular (DEM) models*
  - *Analytic-only (chaotic bulk)*
  - *Adding a new reference (YAML) file*
  - *Adding a new test program*
  - *References*
- *Tests for programs in the tools folder*

- *Troubleshooting failed unit tests*


This section discusses adding or expanding tests for the unit test infrastructure included into the LAMMPS source code distribution. Unlike example inputs, unit tests focus on testing the "local" behavior of individual features, tend to run fast, and should be set up to cover as much of the added code as possible. When contributing code to the distribution, the LAMMPS developers will appreciate if additions to the integrated unit test facility are included.

Given the complex nature of MD simulations where many operations can only be performed when suitable "real" simulation environment has been set up, not all tests will be unit tests in the strict definition of the term. They are rather executed on a more abstract level by issuing LAMMPS script commands and then inspecting the changes to the internal data. For some classes of tests, generic test programs have been written that can be applied to parts of LAMMPS that use the same interface (via polymorphism) and those are driven by input files, so tests can be added by simply adding more of those input files. Those tests should be seen more as a hybrid between unit and regression tests.

When adding tests it is recommended to also *enable support for code coverage reporting*, and study the coverage reports so that it is possible to monitor which parts of the code of a given file are executed during the tests and which tests would need to be added to increase the coverage.

The tests are grouped into categories and corresponding folders. The following sections describe how the tests are implemented and executed in those categories with increasing complexity of tests and implementation.

### 4.12.1 Tests for utility functions

These tests are driven by programs in the `unittest/utils` folder and most closely resemble conventional unit tests. There is one test program for each namespace or group of classes or file. The naming convention for the sources and executables is that they start with with `test_`. The following sources and groups of tests are currently available:

| File name: | Test name: | Description: |
|---|---|---|
| `test_argutils.cpp` | ArgInfo | Tests for `ArgInfo` class used by LAMMPS |
| `test_fmtlib.cpp` | FmtLib | Tests for `{fmt}` or `std::format` functions used by LAMMPS |
| `test_math_eigen_impl.cpp` | MathEigen | Tests for `MathEigen::` classes and functions |
| `test_mempool.cpp` | MemPool | Tests for *`MyPage`* and *`MyPoolChunk`* |
| `test_tokenizer.cpp` | Tokenizer | Tests for *`Tokenizer`* and *`ValueTokenizer`* |
| `test_utils.cpp` | Utils | Tests for `utils::` *functions* |
| `test_fft3d.cpp` | FFT3D | Tests for standard FFT3d wrapper (KISS, FFTW3, MKL, NVPL) |
| `test_fft3d_kokkos.cpp` | FFT3DKokkos | Tests for KOKKOS FFT3d wrapper (CPU and GPU back ends) |

To add tests either an existing source file needs to be modified or a new source file needs to be added to the distribution and enabled for testing. To add a new file suitable CMake script code needs to be added to the `CMakeLists.txt` file in the `unittest/utils` folder. Example:

```
add_executable(test_tokenizer test_tokenizer.cpp)
target_link_libraries(test_tokenizer PRIVATE lammps GTest::GMockMain GTest::GMock␣
↪GTest::GTest)
add_test(Tokenizer test_tokenizer)
```

This adds instructions to build the `test_tokenizer` executable from `test_tokenizer.cpp` and links it with the GoogleTest libraries and the LAMMPS library as well as it uses the `main()` function from the GoogleMock library of GoogleTest. The third line registers the executable as a test program to be run from `ctest` under the name `Tokenizer`.

The test executable itself will execute multiple individual tests through the GoogleTest framework. In this case each test consists of creating a tokenizer class instance with a given string and explicit or default separator choice, and then executing member functions of the class and comparing their results with expected values. A few examples:

```
TEST(Tokenizer, empty_string)
{
    Tokenizer t("", " ");
    ASSERT_EQ(t.count(), 0);
}

TEST(Tokenizer, two_words)
{
    Tokenizer t("test word", " ");
    ASSERT_EQ(t.count(), 2);
}

TEST(Tokenizer, default_separators)
{
    Tokenizer t(" \r\n test \t word \f");
    ASSERT_THAT(t.next(), Eq("test"));
    ASSERT_THAT(t.next(), Eq("word"));
    ASSERT_EQ(t.count(), 2);
}
```

Each of these TEST functions will become an individual test run by the test program. When using the `ctest` command as a front end to run the tests, their output will be suppressed and only a summary printed, but adding the '-V' option will then produce output from the tests above like the following:

```
[...]
1: [ RUN      ] Tokenizer.empty_string
1: [       OK ] Tokenizer.empty_string (0 ms)
1: [ RUN      ] Tokenizer.two_words
1: [       OK ] Tokenizer.two_words (0 ms)
1: [ RUN      ] Tokenizer.default_separators
1: [       OK ] Tokenizer.default_separators (0 ms)
[...]
```

The MathEigen test collection has been adapted from a standalone test and does not use the GoogleTest framework and thus not representative. The other test sources, however, can serve as guiding examples for additional tests.

### FFT Testing Infrastructure

Added in version 10Dec2025.

The FFT tests (`test_fft3d.cpp` and `test_fft3d_kokkos.cpp`) validate the LAMMPS FFT wrapper implementations for both standard (CPU) and KOKKOS (CPU/GPU) back ends. These tests require the KSPACE package and use specialized helper utilities to ensure FFT correctness across different library back ends (KISS FFT, FFTW3, MKL, NVPL, cuFFT, hipFFT, etc.).

**Building and Running FFT Tests:**

The FFT tests are automatically enabled when `ENABLE_TESTING=ON` and `PKG_KSPACE=ON` are set during CMake configuration. For KOKKOS FFT tests, `PKG_KOKKOS=ON` is also required.

Run only FFT tests using the `ctest` command of the CMake software:

```
ctest -R FFT3D          # Run all tests with FFT3D in their name
ctest -R FFT3D -V       # Same as above but with verbose output
ctest -L fft            # Run all tests labeled with 'fft'
```

Tests automatically skip configurations requiring libraries or back ends not available in the current build (e.g., FFTW3, MPI, CUDA).

**FFT Test Helper Header:**

The testing infrastructure uses `fft_test_helpers.h` which contains test data generators, validators, and utilities.

For runtime configuration detection, tests use the existing `Info` class API (`Info::has_package()`, `Info::has_accelerator_feature()`, etc.).

The `fft_test_helpers.h` header provides three main namespaces:

1. **FFTTestHelpers** - utility functions: `FFTBuffer` (RAII wrapper), `idx3d()` (index conversion), `scaled_tolerance()` (grid-size-dependent precision)
2. **FFTTestData** - test data generators: `DeltaFunctionGenerator`, `ConstantGenerator`, `SineWaveGenerator`, `GaussianGenerator`, `RandomComplexGenerator`, `MixedModesGenerator`
3. **FFTValidation** - validators: `RoundTripValidator`, `KnownAnswerValidator`, `ParsevalValidator`, `HermitianSymmetryValidator`, `LinearityValidator`

**Example Test:**

```
TEST_F(FFT3DTest, RoundTrip_32x32x32) {
    FFTBuffer original(32, 32, 32), fft_result(32, 32, 32), recovered(32, 32, 32);

    GaussianGenerator generator(2.0);
    generator.generate(original.data(), 32, 32, 32);

    fft->compute(original.data(), fft_result.data(), FFT3d::FORWARD);
    fft->compute(fft_result.data(), recovered.data(), FFT3d::BACKWARD);

    RoundTripValidator validator(original.data(), recovered.data(), 32, 32, 32,
                                 scaled_tolerance(ROUNDTRIP_TOLERANCE, 32, 32, 32));
    EXPECT_TRUE(validator.validate());
}
```

**Precision and Tolerances:**

FFT tests use precision-aware tolerances that automatically adjust based on floating-point precision (`-D FFT_SINGLE=ON` vs `-D FFT_SINGLE=off`), grid size, and accelerator back end. Base tolerances (`ROUNDTRIP_TOLERANCE`, `PARSEVAL_TOLERANCE`, etc.) are defined in `fft_test_helpers.h`. Use `scaled_tolerance()` to adjust for grid size effects.

### 4.12.2 Tests for individual LAMMPS commands

The tests `unittest/commands` are a bit more complex as they require to first create a *LAMMPS* class instance and then use the *C++ API* to pass individual commands to that LAMMPS instance. For that reason these tests use a GoogleTest "test fixture", i.e. a class derived from `testing::Test` that will create (and delete) the required LAMMPS class instance for each set of tests in a `TEST_F()` function. Please see the individual source files for different examples of setting up suitable test fixtures. Here is an example for implementing a test using a fixture by first checking the default value and then issuing LAMMPS commands and checking whether they have the desired effect:

```
TEST_F(SimpleCommandsTest, ResetTimestep)
{
    ASSERT_EQ(lmp->update->ntimestep, 0);

    BEGIN_HIDE_OUTPUT();
    command("reset_timestep 10");
    END_HIDE_OUTPUT();
    ASSERT_EQ(lmp->update->ntimestep, 10);

    BEGIN_HIDE_OUTPUT();
    command("reset_timestep 0");
    END_HIDE_OUTPUT();
    ASSERT_EQ(lmp->update->ntimestep, 0);

    TEST_FAILURE(".*ERROR: Timestep must be >= 0.*", command("reset_timestep -10"););
    TEST_FAILURE(".*ERROR: Illegal reset_timestep .*", command("reset_timestep"););
    TEST_FAILURE(".*ERROR: Illegal reset_timestep .*", command("reset_timestep 10 10"););
    TEST_FAILURE(".*ERROR: Expected integer .*", command("reset_timestep xxx"););
}
```

Please note the use of the `BEGIN_HIDE_OUTPUT` and `END_HIDE_OUTPUT` functions that will capture output from running LAMMPS. This is normally discarded but by setting the verbose flag (via setting the `TEST_ARGS` environment variable, `TEST_ARGS=-v`) it can be printed and used to understand why tests fail unexpectedly.

The specifics of so-called "death tests", i.e. conditions where LAMMPS should fail and throw an exception, are implemented in the `TEST_FAILURE()` macro. These tests operate by capturing the screen output when executing the failing command and then comparing that with a provided regular expression string pattern. Example:

```
TEST_F(SimpleCommandsTest, UnknownCommand)
{
    TEST_FAILURE(".*ERROR: Unknown command.*", lmp->input->one("XXX one two"););
}
```

The following test programs are currently available:

| File name: | Test name: | Description: |
|---|---|---|
| `test_simple_commands.cpp` | SimpleCommands | Tests for LAMMPS commands that do not require a box |
| `test_lattice_region.cpp` | LatticeRegion | Tests to validate the *lattice* and *region* commands |
| `test_groups.cpp` | GroupTest | Tests to validate the *group* command |
| `test_variables.cpp` | VariableTest | Tests to validate the *variable* command |
| `test_kim_commands.cpp` | KimCommands | Tests for several commands from the *KIM package* |
| `test_reset_atoms.cpp` | ResetAtoms | Tests to validate the *reset_atoms* sub-commands |

### 4.12.3 Tests for the C-style library interface

Tests for validating the LAMMPS C-style library interface are in the `unittest/c-library` folder. They text either utility functions or LAMMPS commands, but use the functions implemented in `src/library.cpp` as much as possible. There may be some overlap with other tests as far as the LAMMPS functionality is concerned, but the focus is on testing the C-style library API. The tests are distributed over multiple test programs which try to match the grouping of the functions in the source code and *in the manual*.

This group of tests also includes tests invoking LAMMPS in parallel through the library interface, provided that LAMMPS was compiled with MPI support. These include tests where LAMMPS is run in multi-partition mode or

only on a subset of the MPI world communicator. The CMake script code for adding this kind of test looks like this:

```
if (BUILD_MPI)
  add_executable(test_library_mpi test_library_mpi.cpp)
  target_link_libraries(test_library_mpi PRIVATE lammps GTest::GTest GTest::GMock)
  target_compile_definitions(test_library_mpi PRIVATE ${TEST_CONFIG_DEFS})
  add_mpi_test(NAME LibraryMPI NUM_PROCS 4 COMMAND $<TARGET_FILE:test_library_mpi>)
endif()
```

Note the custom function `add_mpi_test()` which adapts how `ctest` will execute the test so it is launched in parallel (with 4 MPI ranks).

### 4.12.4 Tests for the Python module and package

The `unittest/python` folder contains primarily tests for classes and functions in the LAMMPS python module but also for commands in the PYTHON package. These tests are only enabled, if the necessary prerequisites are detected or enabled during configuration and compilation of LAMMPS (shared library build enabled, Python interpreter found, Python development files found).

The Python tests are implemented using the `unittest` standard Python module and split into multiple files with similar categories as the tests for the C-style library interface.

### 4.12.5 Tests for the Fortran interface

Tests for using the Fortran module are in the `unittest/fortran` folder. Since they are also using the GoogleTest library, they require test wrappers written in C++ that will call fortran functions with a C function interface through ISO_C_BINDINGS which will in turn call the functions in the LAMMPS Fortran module.

### 4.12.6 Tests for the C++-style library interface

The tests in the `unittest/cplusplus` folder are somewhat similar to the tests for the C-style library interface, but do not need to test the convenience and utility functions that are only available through the C-style library interface. Instead they focus on the more generic features that are used in LAMMPS internally. This part of the unit tests is currently still mostly in the planning stage.

### 4.12.7 Tests for reading and writing file formats

The `unittest/formats` folder contains test programs for reading and writing files like data files, restart files, potential files or dump files. This covers simple things like the file i/o convenience functions in the `utils::` namespace to complex tests of atom styles where creating and deleting of atoms with different properties is tested in different ways and through script commands or reading and writing of data or restart files.

### 4.12.8 Tests for styles computing or modifying forces

These are tests common configurations for pair styles, bond styles, angle styles, kspace styles and certain fix styles. Those are tests driven by some test executables build from sources in the `unittest/force-styles` folder and use LAMMPS input template and data files as well as input files in YAML format from the `unittest/force-styles/tests` folder. The YAML file names have to follow some naming conventions so they get associated with the test programs and categorized and listed with canonical names in the list of tests as displayed by `ctest -N`. If you add a new YAML file, you need to re-run CMake to update the corresponding list of tests.

A minimal YAML file for a (molecular) pair style test will looks something like the following (see `mol-pair-zero.yaml`):

```
---
lammps_version: 24 Aug 2020
date_generated: Tue Sep 15 09:44:21 202
epsilon: 1e-14
prerequisites: ! |
  atom full
  pair zero
pre_commands: ! ""
post_commands: ! ""
input_file: in.fourmol
pair_style: zero 8.0
pair_coeff: ! |
  * *
extract: ! ""
natoms: 29
init_vdwl: 0
init_coul: 0

[...]
```

The following table describes the available keys and their purpose. The `Tester` column lists which test program(s) use each key. `all` means every force-style tester (`test_pair_style`, `test_bond_style`, `test_angle_style`, `test_dihedral_style`, `test_improper_style`, and `test_fix_timestep`), and `bonded interaction tests` means the `test_bond_style`, `test_angle_style`, `test_dihedral_style`, and `test_improper_style` testers. The `Required` column indicates whether a valid YAML file (for the listed tester) must contain the key (`yes`) or whether it may be omitted (`no`). The reference generator always writes the required keys; the optional keys are either metadata added by hand (`skip_tests`, `tags`) or reference data that is only emitted when the tested style actually provides it (e.g. `global_scalar` only for a fix with scalar output):

| Key: | Tester: | Required: | Description: |
|---|---|---|---|
| lammps_version | all | yes | LAMMPS version used to last update the reference data |
| date_generated | all | yes | date when the file was last updated |
| epsilon | all | yes | base value for the relative precision required for tests to pass |
| skip_tests | all | no | request to skip the indicated test fixtures (see table below) |
| tags | all | no | used to classify tests and to adjust behavior of test fixtures (see table below) |
| prerequisites | all | yes | list of style kind / style name pairs required to run the test |
| pre_commands | all | yes | LAMMPS commands to be executed before the input template file is read |

continues on next page

Table 1 – continued from previous page

| Key: | Tester: | Required: | Description: |
|---|---|---|---|
| post_commands | all | yes | LAMMPS commands to be executed right before the actual tests |
| input_file | all | yes | LAMMPS input file template |
| input_coeffs | test_fix_timestep | yes | file with the force-field and group setup commands applied after the input template |
| natoms | all | yes | number of atoms in the input file template |
| pair_style | test_pair_style | yes | arguments to the pair_style command to be tested |
| pair_coeff | test_pair_style | yes | list of pair_coeff arguments to set parameters for the input template |
| init_vdwl | test_pair_style | yes | non-Coulomb pair energy after "run 0" |
| init_coul | test_pair_style | yes | Coulomb pair energy after "run 0" |
| run_vdwl | test_pair_style | yes | non-Coulomb pair energy after "run 4" |
| run_coul | test_pair_style | yes | Coulomb pair energy after "run 4" |
| bond_style | test_bond_style | yes | arguments to the bond_style command to be tested |
| bond_coeff | test_bond_style | yes | list of bond_coeff arguments to set parameters |
| angle_style | test_angle_style | yes | arguments to the angle_style command to be tested |
| angle_coeff | test_angle_style | yes | list of angle_coeff arguments to set parameters |
| dihedral_style | test_dihedral_style | yes | arguments to the dihedral_style command to be tested |
| dihedral_coeff | test_dihedral_style | yes | list of dihedral_coeff arguments to set parameters |
| improper_style | test_improper_style | yes | arguments to the improper_style command to be tested |
| improper_coeff | test_improper_style | yes | list of improper_coeff arguments to set parameters |
| init_energy | bonded interaction tests | yes | bonded interaction energy after "run 0" |
| run_energy | bonded interaction tests | yes | bonded interaction energy after "run 4" |

continues on next page

Table 1 – continued from previous page

| Key: | Tester: | Required: | Description: |
|---|---|---|---|
| equilibrium | test_bond_style | yes | equilibrium distance for each type |
| equilibrium | test_angle_style | yes | equilibrium angle for each type |
| extract | all but test_fix_timestep | yes | list of keywords supported by the style's `extract()` method and their dimension |
| init_stress | all but test_fix_timestep | yes | stress tensor after "run 0" |
| init_forces | all but test_fix_timestep | yes | forces on atoms after "run 0" |
| run_stress | all | no | stress tensor after the run (omitted by `test_fix_timestep` when the fix has no virial contribution) |
| run_forces | all but test_fix_timestep | yes | forces on atoms after "run 4" |
| run_pos | test_fix_timestep | yes | per-atom positions after the run |
| run_vel | test_fix_timestep | yes | per-atom velocities after the run |
| run_torque | test_fix_timestep | no | per-atom torques after the run (only when the atom style stores torque) |
| global_scalar | test_fix_timestep | no | the global scalar output of the tested fix, if any |
| global_vector | test_fix_timestep | no | the global vector output of the tested fix, if any |

These reference files can be validated against the JSON schema file `tools/json/force-style-test-schema.json` with the `check-jsonschema` tool, which catches typos in keys, missing required keys, and values of the wrong type. For example, to validate all of them at once:

```
check-jsonschema --schemafile tools/json/force-style-test-schema.json \
    unittest/force-styles/tests/*.yaml
```

See the *JSON support files* section of the *Auxiliary tools* documentation for how to install `check-jsonschema`.

The test program will read all this data from the YAML file and then create a LAMMPS instance, apply the settings/commands from the YAML file as needed and then issue a "run 0" command, write out a restart file, a data file and a coeff file. The actual test will then compare computed energies, stresses, and forces with the reference data, issue a "run 4" command and compare to the second set of reference data. This will be run with both the newton_pair setting enabled and disabled and is expected to generate the same results (allowing for some numerical noise). Then it will restart from the previously generated restart and compare with the reference and also start from the data file. A final check will use multi-cutoff r-RESPA (if supported by the pair style) at a 1:1 split and compare to the Verlet results. These sets of tests are run with multiple test fixtures for accelerated styles: OPT, OPENMP and INTEL (the latter two with 4 OpenMP threads enabled), and three mutually exclusive KOKKOS fixtures selected by the active back end: the `kokkos_omp` fixture requires the KOKKOS package compiled with the OpenMP back end and uses 4 OpenMP threads, while the `kokkos_serial` fixture only runs when the Serial back end is the sole back end of the KOKKOS package (with any other back end enabled the host execution space would not be Serial, so this configuration must be tested with a separate build). Both of these host fixtures skip when a GPU back end (CUDA, HIP, SYCL) is enabled, since the

KOKKOS package then must run on the GPU. The third fixture, `kokkos_gpu`, is the complement: it runs only when a GPU back end is enabled (using `-k on g 1`) and is skipped on host-only builds. Because enabling the KOKKOS package with a GPU back end aborts when no usable device is present, this fixture first probes for a compatible GPU at runtime with `Info::has_kokkos_gpu_device()` (the KOKKOS package analog of `Info::has_gpu_device()` for the GPU package) and skips transparently when none is available, so the test suite can be run unchanged on machines without a GPU. For these tests the relative error (epsilon) is lowered by a common factor due to the additional numerical noise, but the tests are still comparing to the same reference data.

The KOKKOS fixtures also support the KOKKOS package compiled for reduced precision with `-D KOKKOS_PREC=mixed` (compute in single precision, accumulate in double precision) or `-D KOKKOS_PREC=single`: the test tolerance is then relaxed by a large additional factor, similar to what is done for the mixed and single precision variants of the GPU package. Individual tests can be skipped for a given fixture by listing the fixture name in the `skip_tests:` field of the YAML file (e.g. `skip_tests: kokkos_omp kokkos_serial kokkos_gpu`). A skip entry may also be qualified by the KOKKOS precision, e.g. `kokkos_serial_single` or `kokkos_omp_mixed`, which skips the test only for that combination of fixture and precision. This is used for tests whose reference quantities cannot be meaningfully compared in reduced precision, for example global force totals that are the cancellation sum of large per-atom contributions in a charge-neutral system.

The test fixture names accepted by `skip_tests` (each fixture runs the corresponding variant or check and self-skips when its package or back end is not available) are listed below. Not every fixture exists for every style kind (e.g. `gpu`, `intel`, and `opt` are used by the pair-style tester, while `numdiff` is used by the bonded-style testers).

| Fixture | Description |
|---|---|
| plain | the unmodified (base class) style, without a suffix |
| omp | the `/omp` variant from the OPENMP package (run with 4 threads) |
| intel | the `/intel` variant from the INTEL package |
| opt | the `/opt` variant from the OPT package |
| gpu | the `/gpu` variant from the GPU package |
| kokkos_serial | the `/kk` variant from the KOKKOS package with a Serial-only build |
| kokkos_omp | the `/kk` variant from the KOKKOS package with the OpenMP back end (run with 4 threads) |
| kokkos_gpu | the `/kk` variant from the KOKKOS package with a GPU back end |
| single | consistency check of the style's `single()` method against `compute()` |
| extract | check of the style's `extract()` keywords (base style) |
| extract_omp | check of the style's `extract()` keywords (`/omp` variant) |
| numdiff | check of the forces against a numerical derivative of the energy |

The `kokkos_omp`, `kokkos_serial`, and `kokkos_gpu` entries may be qualified with `_single` or `_mixed` (e.g. `kokkos_gpu_single`), as noted above.

The `tags:` field of a YAML file lists keywords that classify a test or request special handling from the test fixtures. The fixtures query them with `TestConfig::has_tag()` so that style-specific behavior is selected by a descriptive tag instead of by hard-coded style names. The recognized tags are:

| Tag | Purpose |
|---|---|
| gpu_ | The GPU package variant of the style does not support mixed precision GPU mode; the `gpu` fixture skips the test when the GPU package was compiled for mixed precision. |
| gpu_ | The GPU package variant of the style does not support single precision GPU mode (e.g. `born/coul/long/cs/gpu`); the `gpu` fixture skips the test when the GPU package was compiled for single precision. |
| single_t | The style cannot run correctly with more than one thread in the test (e.g. `dpd` uses multiple per-thread pRNGs; `snap` and `pace` due to their implementation), so the threaded fixtures (`omp`, `intel`, `kokkos_omp`) run it with a single thread. |
| no_re | The `fix_timestep` tester does not exercise this style under *run_style respa*: rigid fixes need additional work to test correctly with r-RESPA, `fix nve/limit` and `fix recenter` do not support it, stochastic integrators and barostats (`brownian`, `gjf`, `press/langevin`) draw their random numbers differently under r-RESPA, and velocity-dependent forcing fixes (`viscous`, `accelerate/cos`) and the isokinetic `nvk` integrator follow a different trajectory under r-RESPA - in all of these cases the verlet and r-RESPA runs cannot match. |
| no_re | The `fix_timestep` tester does not exercise a timestep change for this style. The fix rejects a timestep reset (its `Fix::reset_dt()` raises an error, e.g. *fix move*), which would otherwise abort the test. |
| ellipsoid | The test includes ellipsoids and thus requires *fix nve/asphere*. |
| spica | The test setup uses `pair_style lj/spica` instead of the default `pair_style zero` (required by the `spica` angle style). |
| slow | The test runs significantly longer than others and `ctest -LE slow` would skip it. |
| noWindows | Indicates that this test must be skipped on Windows; use `ctest -LE noWindows` |
| unstable | The test exhibits numerically unstable behavior on some platforms, e.g. ARM64; Until a proper correction is found, tests can be skipped with `ctest -LE unstable`. |
| generated | Indicates that a test input was regenerated. *Remove* after confirming the correctness of the updated YAML file. |

Additional tests will check whether all listed extract keywords are supported and have the correct dimensionality and the final set of tests will set up a few pairs of atoms explicitly and in such a fashion that the forces on the atoms computed from `Pair::compute()` will match individually with the results from `Pair::single()`, if the pair style does support that functionality.

With this scheme a large fraction of the code of any tested pair style will be executed and consistent results are required for different settings and between different accelerated pair style variants and the base class, as well as for computing individual pairs through the `Pair::single()` method where supported.

The `test_pair_style` tester is used with 4 categories of test inputs:

- pair styles compatible with molecular systems using bonded interactions and exclusions. For pair styles requiring a KSpace style the KSpace computations are disabled. The YAML files match the pattern "mol-pair-*.yaml" and the tests are correspondingly labeled with "MolPairStyle:*"
- pair styles not compatible with the previous input template. The YAML files match the pattern "atomic-pair-*.yaml" and the tests are correspondingly labeled with "AtomicPairStyle:*"
- manybody pair styles. The YAML files match the pattern "atomic-pair-*.yaml" and the tests are correspondingly labeled with "AtomicPairStyle:*"
- kspace styles. The YAML files match the pattern "kspace-*.yaml" and the tests are correspondingly labeled with "KSpaceStyle:*". In these cases a compatible pair style is defined, but the computation of the pair style contributions is disabled.

The `test_bond_style`, `test_angle_style`, `test_dihedral_style`, and `test_improper_style` tester programs are set up in a similar fashion and share support functions with the pair style tester. The final group of tests in this section is for fix styles that add/manipulate forces and velocities, e.g. for time integration, thermostats and more.

Adding a new test is easiest done by copying and modifying an existing YAML file for a style that is similar to one to be tested. The file name should follow the naming conventions described above and after copying the file, the first step is to replace the style names where needed. The coefficient values do not have to be meaningful, just in a reasonable range for the given system. It does not matter if some forces are large, for as long as they do not diverge.

The template input files define a large number of index variables at the top that can be modified inside the YAML file to control the behavior. For example, if a pair style requires a "newton on" setting, the following can be used in as the "pre_commands" section:

```
pre_commands: ! |
  variable newton_pair delete
  variable newton_pair index on
```

And for a pair style requiring a kspace solver the following would be used as the "post_commands" section:

```
post_commands: ! |
  pair_modify table 0
  kspace_style pppm/tip4p 1.0e-6
  kspace_modify gewald 0.3
  kspace_modify compute no
```

Note that this disables computing the kspace contribution, but still will run the setup. The "gewald" parameter should be set explicitly to speed up the run. For styles with long-range electrostatics, typically two tests are added one using the (slower) analytic approximation of the erfc() function and the other using the tabulated coulomb, to test both code paths. The reference results in the YAML files then should be compared manually, if they agree well enough within the limits of those two approximations.

The `test_pair_style` and equivalent programs have special command-line options to update the YAML files. Running a command like

```
test_pair_style mol-pair-lennard_mdf.yaml -g new.yaml
```

will read the settings from the `mol-pair-lennard_mdf.yaml` file and then compute the reference data and write a new file with to `new.yaml`. If this step fails, there are likely some (LAMMPS or YAML) syntax issues in the YAML file that need to be resolved and then one can compare the two files to see if the output is as expected.

It is also possible to do an update in place with:

```
test_pair_style mol-pair-lennard_mdf.yaml -u
```

And one can finally run the full set of tests with:

```
test_pair_style mol-pair-lennard_mdf.yaml
```

This will just print a summary of the groups of tests. When using the "-v" flag the test will also keep any LAMMPS output and when using the "-s" flag, there will be some statistics reported on the relative errors for the individual checks which can help to figure out what would be a good choice of the epsilon parameter. It should be as small as possible to catch any unintended side effects from changes elsewhere, but large enough to accommodate the numerical noise due to the implementation of the potentials and differences in compilers.

> **Note**
>
> These kinds of tests can be very sensitive to compiler optimization and thus the expectation is that they pass with compiler optimization turned off. When compiler optimization is enabled, there may be some failures, but one has to carefully check whether those are acceptable due to the enhanced numerical noise from reordering floating-point math operations or due to the compiler mis-compiling the code. That is not always obvious.

### 4.12.9 Tests for granular (DEM) models

Added in version 4Jul2026.

The `unittest/granular` folder contains a YAML-driven test suite for discrete element method (DEM) / granular models, built in the same spirit as the force-style tests above but specialized for time-resolved trajectories of small granular systems.

Currently, there are 11 test programs. This set of unit tests is still a work-in-progress and the tests have not yet been thoroughly vetted. Tests may be added, updated, or removed. The first six test programs, `test_dem_01` through `test_dem_06`, reproduce the test surface of the MFiX-DEM verification studies of *Garg et al.* (the individual cases are also described in the MFiX-DEM manual). `test_dem_07` through `test_dem_11` add coverage of benchmark cases from the granular literature: rolling resistance, cohesion, and two-particle collisions follow the software-agnostic DEM benchmark of *Mohajeri et al.*, and the bulk angle of repose and multi-sphere clump cases follow the round-robin study of *Saomoto et al.*. The particle-impact-level cases – oblique wall impact (`test_dem_05`), two-sphere and spinning-sphere collisions (`test_dem_09`) and the elastic Hertzian normal impact (`test_dem_11`) – follow the benchmark of *Chung and Ooi*. The test programs are:

| Program | Scenario |
|---|---|
| `test_dem_01` | a freely falling particle bouncing off a wall |
| `test_dem_02` | a particle bouncing repeatedly (convergence to the hard-sphere limit) |
| `test_dem_03` | two stacked particles in continuous compression between two walls |
| `test_dem_04` | a sphere sliding then rolling without slipping on a rough surface |
| `test_dem_05` | an oblique collision of a sphere (and a superellipsoid) with a wall |
| `test_dem_06` | a single particle settling to its terminal velocity under fluid drag |
| `test_dem_07` | a spinning sphere damped to rest by rolling resistance (`rolling sds`) |
| `test_dem_08` | cohesive/adhesive contact: the DMT and JKR pull-off force |
| `test_dem_09` | two-sphere head-on, oblique (shear), and spinning-sphere collisions |
| `test_dem_10` | bulk behavior: a settling pile, the angle of repose, and a rigid clump |
| `test_dem_11` | elastic Hertzian normal impact (peak contact mechanics) |

Every test program shares the same driver logic, implemented in `unittest/granular/test_dem_common.cpp` and compiled into the `granular_tests` support library; each `test_dem_0N.cpp` only contains the two GoogleTest fixtures (`newton_on` and `newton_off`). As with the force-style tests, the reference systems are defined by YAML files in the `unittest/granular/tests` folder and registered as CTest cases by their file name (`dem0N-*.yaml` becomes test `DEM0N:*`); adding or removing a YAML file requires re-running CMake.

Unlike the force-style tests, the entire system is built *from the YAML file* rather than from a fixed input template. A YAML file provides an optional `variables` block (emitted as *index variables* so they can be substituted as `${name}` anywhere in the command strings), `pre_commands` that create the geometry, `pair_style` / `pair_coeff` that select the contact model, and `post_commands` that add the integrator, gravity, walls and drag. The trajectory is then advanced in a sequence of `run_segments` and, after each segment, the per-atom positions, velocities, torques, angular velocities (spheres) and angular momenta (ellipsoids/superellipsoids) are compared against the recorded reference. A minimal example (`dem01-hooke-3d-si.yaml`) looks like:

```
---
lammps_version: 30 Mar 2026
tags: granular
epsilon: 1e-10
prerequisites: ! |
  atom sphere
  pair gran/hooke
variables: ! |
  knorm 1.0e4
  gnorm 10.0
  diam 0.2
  dens 2600.0
  grav 9.81
  z0 0.5
pre_commands: ! |
  units si
  dimension 3
  boundary f f f
  atom_style sphere
  region box block -0.5 0.5 -0.5 0.5 0.0 1.0 units box
  create_box 1 box
  create_atoms 1 single 0.0 0.0 ${z0} units box
  set group all diameter ${diam} density ${dens}
  comm_modify vel yes
  timestep 0.001
pair_style: gran/hooke ${knorm} NULL ${gnorm} NULL 0.0 0
pair_coeff: ! |
  * *
post_commands: ! |
  fix grav all gravity ${grav} vector 0.0 0.0 -1.0
  fix integr all nve/sphere
  fix zwall all wall/gran hooke ${knorm} NULL ${gnorm} NULL 0.0 0 zplane 0.0 NULL
run_segments: ! |
  250 150 300
analytic_enable: yes
analytic_model: freefall
analytic_tol: 1.0e-9
analytic_segment: 0
# run_pos / run_vel / run_torque / run_omega / run_angmom blocks follow
```

The following table describes the available keys:

| Key: | Description: |
|---|---|
| epsilon | relative precision required for the recorded (regression) reference data |
| prerequisites | list of style kind / style name pairs required to run the test |
| variables | name/value pairs exposed as `${name}` index variables for substitution |
| pre_commands | commands that build the geometry (units, box, atoms, `set`, timestep) |
| pair_style / pair_coeff | the particle-particle contact model |
| post_commands | fixes added after the geometry (integrator, gravity, walls, drag) |
| run_segments | whitespace-separated list of run lengths; state is captured after each |
| run_pos, run_vel | reference positions and velocities, as `segment tag x y z` rows |
| run_torque, run_omega, run_angmom | reference torque / angular velocity / angular momentum (when applicable) |
| analytic_enable | `yes` to also assert a closed-form (analytic) model |
| analytic_model | which analytic model to evaluate (see below) |
| analytic_tol | relative tolerance for the analytic assertion (looser than `epsilon`) |
| analytic_segment | run segment at which the analytic model is checked (`-1` means the last) |
| analytic_only | `yes` to record/check *only* the analytic model and skip the per-atom regression (for chaotic bulk tests; see below) |

The per-atom reference blocks use a `segment tag x y z` row format, so a single block holds the data for all run segments and the row order does not matter. Because granular/atomic systems do not build an atom map by default, the reference generator iterates over local atoms by tag rather than calling `Atom::map()`.

Each test runs as a pure regression check (the recorded data is reproduced to within `epsilon`) under both the `newton on` and `newton off` fixtures, which are expected to give identical results. In addition, a test may opt in to an *analytic* check that compares a derived quantity against a closed-form solution implemented in `unittest/granular/test_analytic_models.cpp`. The analytic tolerance is deliberately loose, because the soft-sphere DEM result only approaches the idealized (hard-sphere or instantaneous-contact) solution. The models currently implemented are:

| Model: | Checks: |
|---|---|
| freefall | ballistic motion before contact: $z = z_0 - gt^2/2$, $v_z = -gt$ |
| bounce_height | hard-sphere apex after the k-th bounce $h_k = r + e^{2k}(h_0 - r)$ |
| stack_energy | conservation of total mechanical energy for an elastic two-particle stack |
| slip_cessation | rolling-without-slipping limit $u = 5u_0/7$, $\omega = u/r$ |
| oblique_impact | gross-sliding rebound $v_x' = v_x - \mu(1+e)v_z$, $\omega_y = \frac{5}{2}\mu(1+e)v_z/r$ |
| terminal_velocity_linear | Stokes drag terminal velocity $v_{term} = mg/\gamma$ |
| terminal_velocity_schiller_naumann | Schiller-Naumann terminal velocity from $mg = \frac{1}{2}C_d\rho_g\pi r^2 v^2$ |
| rolling_decay | linear spin-down under rolling resistance: $\omega = \omega_0 - \frac{5\mu_r g}{2r}t$ |
| pulloff_dmt | DMT pull-off force at contact $\lvert F\rvert = 4\pi\gamma R_{\text{eff}}$ |
| collision_restitution | two-sphere momentum conservation and restitution $e = -(v_1' - v_2')/(v_1 - v_2)$ |
| angle_of_repose | measured heap slope $\arctan(z_{\max}/r_{\max})$ lies within a `[lo, hi]` band |
| hertz_normal_impact | Hertzian peak energy balance $\frac{1}{2}\mu_{red}V_{rela}^2 = \frac{2}{5}P_{max}\alpha_{max}$ |
| spin_impact | gross-sliding rebound of a spinning sphere: $v_x' = \mu(1+e)v_n$, $\omega_y' = \omega_0 - \frac{5}{2}\mu(1+e)v_n/r$ |
| spin_no_friction | counter-spinning spheres with zero contact slip keep their spin and gain no tangential velocity |

`test_dem_06` exercises both *fix viscous* (linear Stokes drag) and the *fix viscous/nonlinear* style that was added together with these tests for the Schiller-Naumann drag correlation.

### Analytic-only (chaotic bulk)

Most tests are bit-for-bit regressions that reproduce identically under `newton on` and `newton off`. A few bulk scenarios – notably the angle-of-repose pile in `test_dem_10` – are *chaotic*: a long pour-and-settle trajectory amplifies the round-off differences between summation orders, so the per-atom state is not reproducible across `newton` settings or platforms even though the bulk observable (the heap angle) is robust. Such a YAML sets `analytic_only: yes`. The generator then records no per-atom reference blocks, and the driver checks only the analytic model. `test_dem_10` pairs this with a short, deterministic `dem10-settle-*` regression (a small lattice block relaxing into contact) and a deterministic `dem10-clump-*` case (a *fix rigid/small* tetrahedral clump bouncing on a granular wall) so the bit-for-bit code path is still covered.

### Adding a new reference (YAML) file

Copy an existing `dem0N-*.yaml` for a similar scenario, adjust the `variables`, `pre_commands`, `pair_style/pair_coeff` and `post_commands` for the new model, and give it a new name matching the `dem0N-*.yaml` pattern of the test program it belongs to. Leave out the reference data blocks initially, then (re)generate them in place with:

```
TEST_ARGS=-u ctest -R DEM0N:myvariant
```

or by running the driver directly (`test_dem_0N dem0N-myvariant.yaml -u`). Do **not** write the generated file to a sibling `dem0N-*.yaml` name (for example with the `-g newfile.yaml` option pointing into the `tests` folder), because the `CONFIGURE_DEPENDS` glob would then register it as an extra, stale test. After adding the file, re-run CMake so the new test is registered, then verify it with `ctest -V -R DEM0N:myvariant` (the `-s` option of the driver reports per-quantity error statistics, which helps when choosing `epsilon` and the analytic tolerance).

### Adding a new test program

Create `test_dem_0N.cpp` as a thin copy of an existing one (only the GoogleTest suite name changes), add an `add_executable/register_dem_tests` pair to `unittest/granular/CMakeLists.txt`, and add the corresponding `dem0N-*.yaml` reference files. If the new scenario needs a closed-form check, add a named model to `test_analytic_models.cpp` that reads its parameters from the `variables` block (and reads masses, radii, etc. from the live LAMMPS instance to avoid depending on derived quantities) and assert it with `EXPECT_LE` on the relative error.

### References

**(Garg et al., 2012)** R. Garg, J. Galvin, T. Li, and S. Pannala, Open-source MFIX-DEM software for gas-solids flows: Part I – Verification studies, Powder Technology, 220, 122-137 (2012), https://doi.org/10.1016/j.powtec.2011.09.019

**(Mohajeri et al., 2024)** M. J. Mohajeri, C. Coetzee, and D. L. Schott, A software-agnostic benchmark for DEM simulation of cohesive and non-cohesive materials, Powder Technology, 447, 120136 (2024), https://doi.org/10.1016/j.powtec.2024.120136

**(Saomoto et al., 2023)** H. Saomoto, N. Kikkawa, S. Moriguchi, Y. Nakata, et al., Round robin test on angle of repose: DEM simulation results collected from 16 groups around the world, Soils and Foundations, 63, 101272 (2023), https://doi.org/10.1016/j.sandf.2023.101272

**(Chung and Ooi, 2011)** Y. C. Chung and J. Y. Ooi, Benchmark tests for verifying discrete element modelling codes at particle impact level, Granular Matter, 13, 643-656 (2011), https://doi.org/10.1007/s10035-011-0277-0

## 4.12.10 Tests for programs in the tools folder

The `unittest/tools` folder contains tests for programs in the `tools` folder. This currently only contains tests for the LAMMPS shell, which are implemented as a python scripts using the `unittest` Python module and launching the tool commands through the `subprocess` Python module.

### 4.12.11 Troubleshooting failed unit tests

There are by default no unit tests for newly added features (e.g. pair, fix, or compute styles) unless your pull request also includes tests for these added features. If you are modifying some existing LAMMPS features, you may see failures for existing tests, if your modifications have some unexpected side effects or your changes render the existing test invalid. If you are adding an accelerated version of an existing style, then only tests for INTEL, KOKKOS (with the OpenMP or Serial host back ends, depending on how the KOKKOS package was configured), OPENMP, and OPT will be run automatically. Tests for the GPU package are time consuming and thus are only run *after* a merge, or when a special label, `gpu_unit_tests` is added to the pull request. After the test has started, it is often best to remove the label since every PR activity will re-trigger the test (that is a limitation of triggering a test with a label). Support for unit tests using KOKKOS with GPU acceleration is currently not supported.

When you see a failed build on GitHub, click on `Details` to be taken to the corresponding LAMMPS Jenkins CI web page. Click on the "Exit" symbol near the `Logout` button on the top right of that page to go to the "classic view". In the classic view, there is a list of the individual runs that make up this test run (they are shown but cannot be inspected in the default view). You can click on any of those. Clicking on `Test Result` will display the list of failed tests. Click on the "Status" column to sort the tests based on their Failed or Passed status. Then click on the failed test to expand its output.

For example, the following output snippet shows the failed unit test

```
[ RUN      ] PairStyle.gpu
/home/builder/workspace/dev/pull_requests/ubuntu_gpu/unit_tests/cmake_gpu_opencl_mixed_
↪smallbig_clang_static/unittest/force-styles/test_main.cpp:63: Failure
Expected: (err) <= (epsilon)
Actual: 0.00018957912910606503 vs 0.0001
Google Test trace:
/home/builder/workspace/dev/pull_requests/ubuntu_gpu/unit_tests/cmake_gpu_opencl_mixed_
↪smallbig_clang_static/unittest/force-styles/test_main.cpp:56: EXPECT_FORCES: init_
↪forces (newton off)
/home/builder/workspace/dev/pull_requests/ubuntu_gpu/unit_tests/cmake_gpu_opencl_mixed_
↪smallbig_clang_static/unittest/force-styles/test_main.cpp:64: Failure
Expected: (err) <= (epsilon)
Actual: 0.00022892713393549854 vs 0.0001
```

The failed assertions provide line numbers in the test source (e.g. `test_main.cpp:56`), from which one can understand what specific assertion failed.

Note that the force style engine runs one of a small number of systems in a rather off-equilibrium configuration with a few atoms for a few steps, writes data and restart files, uses *the clear command* to reset LAMMPS, and then runs from those files with different settings (e.g. newton on/off) and integrators (e.g. verlet vs. respa). Beyond potential issues/bugs in the source code, the mismatch between the expected and actual values could be that force arrays are not properly cleared between multiple run commands or that class members are not correctly initialized or written to or read from a data or restart file.

While the epsilon (relative precision) for a single, IEEE 754 compliant, double precision floating point operation is at about 2.2e-16, the achievable precision for the tests is lower due to most numbers being sums over intermediate results for which the non-associativity of floating point math leads to larger errors. As a rule of thumb, the test epsilon can often be in the range 5.0e-14 to 1.0e-13. But for "noisy" force kernels, e.g. those a larger amount of arithmetic operations involving *exp()*, *log()* or *sin()* functions, and also due to the effect of compiler optimization or differences between compilers or platforms, epsilon may need to be further relaxed, sometimes epsilon can be relaxed to 1.0e-12. If interpolation or lookup tables are used, epsilon may need to be set to 1.0e-10 or even higher. For tests of accelerated styles, the per-test epsilon is multiplied by empirical factors that take into account the differences in the order of floating point operations or that some or most intermediate operations may be done using approximations or with single precision floating point math.

To rerun a failed unit test individually, change to the `build` directory and run the test with verbose output. For example,

```
env TEST_ARGS=-v ctest -R ^MolPairStyle:lj_cut_coul_long -V
```

`ctest` with the `-V` flag also shows the exact command of the test. One can then use `gdb --args` to further debug and catch exceptions with the test command, for example,

```
gdb --args /path/to/lammps/build/test_pair_style /path/to/lammps/unittest/force-styles/
↪tests/mol-pair-lj_cut_coul_long.yaml
```

It is recommended to configure the build with `-D BUILD_SHARED_LIBS=on` and use a custom linker to shorten the build time during recompilation. Installing *ccache* in your development environment helps speed up recompilation by caching previous compilations and detecting when the same compilation is being done again. Please see *Development build options* for further details.

## 4.13 C++ base classes

LAMMPS is designed to be used as a C++ class library where one can set up and drive a simulation through creating a class instance and then calling some abstract operations or commands on that class or its member class instances. These are interfaced to the *C library API*, which providing an additional level of abstraction simplification for common operations. The C API is also the basis for calling LAMMPS from Python or Fortran.

When used from a C++ program, most of the symbols and functions in LAMMPS are wrapped into the `LAMMPS_NS` namespace so they will not collide with your own classes or other libraries. This, however, does not extend to the additional libraries bundled with LAMMPS in the lib folder and some of the low-level code of some packages.

Behind the scenes this is implemented through inheritance and polymorphism where base classes define the abstract interface and derived classes provide the specialized implementation for specific models or optimizations or ports to accelerator platforms. This document will provide an outline of the fundamental class hierarchy and some selected examples for derived classes of specific models.

**Note**

Please see the *note about thread-safety* in the library Howto doc page.

---

### 4.13.1 LAMMPS Class

The LAMMPS class is encapsulating an MD simulation state and thus it is the class that needs to be created when starting a new simulation system state. The LAMMPS executable essentially creates one instance of this class and passes the command-line flags and tells it to process the provided input (a file or `stdin`). It shuts the class down when control is returned to it and then exits. When using LAMMPS as a library from another code it is required to create an instance of this class, either directly from C++ with `new LAMMPS()` or through one of the library interface functions like *`lammps_open()`* of the C-library interface, or the *`lammps.lammps`* class constructor of the Python module, or the *`lammps()`* constructor of the Fortran module.

In order to avoid clashes of function names, all of the core code in LAMMPS is placed into the `LAMMPS_NS` namespace. Functions or variables outside of that namespace must be "static", i.e. visible only to the scope of the file/object they are defined in. Code in packages or the libraries in the `lib` folder may not adhere to this as some of them are adapted from legacy code or consist of external libraries with their own requirements and policies.

---

class **LAMMPS**

*LAMMPS* simulation instance.

The *LAMMPS* class contains pointers of all constituent class instances and global variables that are used by a *LAMMPS* simulation. Its contents represent the entire state of the simulation.

The *LAMMPS* class manages the components of an MD simulation by creating, deleting, and initializing instances of the classes it is composed of, processing command-line flags, and providing access to some global properties. The specifics of setting up and running a simulation are handled by the individual component class instances.

### Public Functions

const char ***non_pair_suffix**() const

Return suffix for non-pair styles depending on pair_only_flag.

**Returns**
suffix or null pointer

const char ***match_style**(const char *style, const char *name)

Return name of package that a specific style belongs to.

This function checks the given name against all list of styles for all types of styles and if the name and the style match, it returns which package this style belongs to.

**Parameters**

- **style** – Type of style (e.g. atom, pair, fix, etc.)
- **name** – Name of style

**Returns**
Name of the package this style is part of

**LAMMPS**(argv &args, MPI_Comm)

Create a *LAMMPS* simulation instance

**Parameters**

- **args** – list of arguments
- **communicator** – MPI communicator used by this *LAMMPS* instance

**LAMMPS**(int, char**, MPI_Comm)

Create a *LAMMPS* simulation instance

The *LAMMPS* constructor starts up a simulation by allocating all fundamental classes in the necessary order, parses input switches and their arguments, initializes communicators, screen and logfile output FILE pointers.

**Parameters**

- **narg** – number of arguments
- **arg** – list of arguments
- **communicator** – MPI communicator used by this *LAMMPS* instance

**~LAMMPS**()

Shut down a *LAMMPS* simulation instance

The *LAMMPS* destructor shuts down the simulation by deleting top-level class instances, closing screen and log files for the global instance (aka "world") and files and MPI communicators in sub-partitions ("universes"). Then it deletes the fundamental class instances and copies of data inside the class.

#### Public Static Functions

static std::vector<char*> **argv_pointers**(argv &args)

Create vector of argv char pointers including terminating nullptr element

**Parameters**
**args** – list of arguments

**Returns**
vector of argument pointers

class **Pointers**

Base class for *LAMMPS* features.

The *Pointers* class contains references to many of the pointers and members of the *LAMMPS_NS::LAMMPS* class. Derived classes thus gain access to the constituent class instances in the *LAMMPS* composite class and thus to the core functionality of *LAMMPS*.

This kind of construct is needed, since the *LAMMPS* constructor should only be run once per *LAMMPS* instance and thus classes cannot be derived from *LAMMPS* itself. The *Pointers* class constructor instead only initializes C++ references to component pointer in the *LAMMPS* class.

Subclassed by *Atom*, *CiteMe*, *Input*, *LabelMap*, *PotentialFileReader*

### 4.13.2 LAMMPS Atom and AtomVec Base Classes

class **Atom** : protected *Pointers*

Class to provide access to atom data.

The Atom class provides access to atom style related global settings and per-atom data that is stored with atoms and migrates with them from sub-domain to sub-domain as atoms move around. This includes topology data, which is stored with either one specific atom or all atoms involved depending on the settings of the *newton command*.

The actual per-atom data is allocated and managed by one of the various classes derived from the AtomVec class as determined by the *atom_style command*. The pointers in the Atom class are updated by the AtomVec class as needed.

#### Public Functions

**Atom**(class *LAMMPS**)

*Atom* class constructor

This resets and initializes all kinds of settings, parameters, and pointer variables for per-atom arrays. This also initializes the factory for creating instances of classes derived from the AtomVec base class, which correspond to the selected atom style.

**Parameters**
**_lmp** – pointer to the base *LAMMPS* class

int **find_custom**(const char*, int&, int&)

Find a custom per-atom property with given name.

This function returns the list index of a custom per-atom property with the name "name", also returning by reference its data type and number of values per atom.

**Parameters**

- **name** – Name of the property (w/o a "i_" or "d_" or "i2_" or "d2_" prefix)
- **&flag** – Returns data type of property: 0 for int, 1 for double
- **&cols** – Returns number of values: 0 for a single value, 1 or more for a vector of values

**Returns**

index of property in the respective list of properties

int **find_custom_ghost**(const char*, int&, int&, int&)

Find a custom per-atom property with given name and retrieve ghost property.

This function returns the list index of a custom per-atom property with the name "name", also returning by reference its data type, number of values per atom, and if it is communicated to ghost particles. Classes rarely need to check on ghost communication and so *find_custom* is typically preferred to this function. See *pair amoeba* for an example where checking ghost communication is necessary.

**Parameters**

- **name** – Name of the property (w/o a "i_" or "d_" or "i2_" or "d2_" prefix)
- **&flag** – Returns data type of property: 0 for int, 1 for double
- **&cols** – Returns number of values: 0 for a single value, 1 or more for a vector of values
- **&ghost** – Returns whether property is communicated to ghost atoms: 0 for no, 1 for yes

**Returns**

index of property in the respective list of properties

virtual int **add_custom**(const char*, int, int, int ghost = 0)

Add a custom per-atom property with the given name and type and size.

This function will add a custom per-atom property with one or more values with the name "name" to the list of custom properties. This function is called, e.g. from *fix property/atom*.

**Parameters**

- **name** – Name of the property (w/o a "i_" or "d_" or "i2_" or "d2_" prefix)
- **flag** – Data type of property: 0 for int, 1 for double
- **cols** – Number of values: 0 for a single value, 1 or more for a vector of values
- **ghost** – Whether property is communicated to ghost atoms: 0 for no, 1 for yes

**Returns**

index of property in the respective list of properties

virtual void **remove_custom**(int, int, int)

Remove a custom per-atom property of a given type and size.

This will remove a property that was requested, e.g. by the *fix property/atom* command. It frees the allocated memory and sets the pointer to `nullptr` for the entry in the list so it can be reused. The lists of these pointers are never compacted or shrunk, so that indices to name mappings remain valid.

**Parameters**

- **index** – Index of property in the respective list of properties
- **flag** – Data type of property: 0 for int, 1 for double
- **cols** – Number of values: 0 for a single value, 1 or more for a vector of values

virtual void ***extract**(const char*)

Provide access to internal data of the *Atom* class by keyword

This function is a way to access internal per-atom data. This data is distributed across MPI ranks and thus only the data for "local" atoms are expected to be available. Whether also data for "ghost" atoms is stored and up-to-date depends on various simulation settings.

This table lists a large part of the supported names, their data types, length of the data area, and a short description.

| Name | Type | Items per atom | Description |
|---|---|---|---|
| mass | double | 1 | per-type mass. This array is **NOT** a per-atom array but of length `ntypes`+1, element 0 is ignored. |
| id | tagint | 1 | atom ID of the particles |
| type | int | 1 | atom type of the particles |
| mask | int | 1 | bitmask for mapping to groups. Individual bits are set to 0 or 1 for each group. |
| image | imageint | 1 | 3 image flags encoded into a single integer. See *lammps_encode_image_flags()*. |
| x | double | 3 | x-, y-, and z-coordinate of the particles |
| v | double | 3 | x-, y-, and z-component of the velocity of the particles |
| f | double | 3 | x-, y-, and z-component of the force on the particles |
| molecule | int | 1 | molecule ID of the particles |
| q | double | 1 | charge of the particles |
| mu | double | 3 | dipole moment of the particles |
| omega | double | 3 | x-, y-, and z-component of rotational velocity of the particles |
| angmom | double | 3 | x-, y-, and z-component of angular momentum of the particles |
| torque | double | 3 | x-, y-, and z-component of the torque on the particles |
| radius | double | 1 | radius of the (extended) particles |
| rmass | double | 1 | per-atom mass of the particles. `nullptr` if per-type masses are used. See the *rmass_flag* setting. |
| ellipsoid | int | 1 | 1 if the particle is an ellipsoidal particle, 0 if not |
| line | int | 1 | 1 if the particle is a line particle, 0 if not |
| tri | int | 1 | 1 if the particle is a triangulated particle, 0 if not |
| body | int | 1 | 1 if the particle is a body particle, 0 if not |
| quat | double | 4 | four quaternion components of the particles |
| temperature | double | 1 | temperature of the particles |
| heatflow | double | 1 | heatflow of the particles |
| i_name | int | 1 | single integer value defined by fix property/atom vector name |
| d_name | double | 1 | single double value defined by fix property/atom vector name |
| i2_name | int | N | N integer values defined by fix property/atom array name |
| d2_name | double | N | N double values defined by fix property/atom array name |

***See also***

*lammps_extract_atom()*, *lammps_extract_atom_datatype()*, *lammps_extract_atom_size()*

**See also**

*extract_datatype*, *extract_size*

**Parameters**

**name** – string with the keyword of the desired property. Typically the name of the pointer variable returned

**Returns**

pointer to the requested data cast to `void *` or `nullptr`

int **extract_datatype**(const char*)

Provide data type info about internal data of the *Atom* class

Added in version 18Sep2020.

**See also**

*extract extract_size*

**Parameters**

**name** – string with the keyword of the desired property.

**Returns**

data type constant for desired property or -1

int **extract_size**(const char*, int)

Provide vector or array size info of internal data of the *Atom* class

Added in version 19Nov2024.

**See also**

*extract extract_datatype*

**Parameters**

- **name** – string with the keyword of the desired property.
- **type** – either LMP_SIZE_ROWS or LMP_SIZE_COLS for per-atom array or ignored

**Returns**

size of the vector or size of the array for the requested dimension or -1

struct **PerAtom**

### 4.13.3 LAMMPS Input Base Class

class **Input** : protected *Pointers*

Class for processing commands and input files.

The Input class contains methods for reading, pre-processing and parsing LAMMPS commands and input files and will dispatch commands to the respective class instances or contain the code to execute the commands directly. It also contains the instance of the Variable class which performs computations and text substitutions.

#### Public Functions

**Input**(class *LAMMPS**, int, char**)

*Input* class constructor

This sets up the input processing, processes the *-var* and *-echo* command-line flags, holds the factory of commands and creates and initializes an instance of the Variable class.

To execute a command, a specific class instance, derived from `Command`, is created, then its `command()` member function executed, and finally the class instance is deleted.

**Parameters**

- **lmp** – pointer to the base *LAMMPS* class
- **argc** – number of entries in *argv*
- **argv** – argument vector

void **file**()

Process all input from the `FILE *` pointer *infile*

This will read lines from *infile*, parse and execute them until the end of the file is reached. The *infile* pointer will usually point to `stdin` or the input file given with the `-in` command-line flag.

void **file**(const char*)

Process all input from the file *filename*

This function opens the file at the path *filename*, puts the current file pointer stored in *infile* on a stack and instead assigns *infile* with the newly opened file pointer. Then it will call the *Input::file()* function to read, parse and execute the contents of that file. When the end of the file is reached, it is closed and the previous file pointer from the infile file pointer stack restored to *infile*.

**Parameters**
**filename** – name of file with *LAMMPS* commands

char ***one**(const std::string&)

Process a single command from a string in *single*

This function takes the text in *single*, makes a copy, parses that, executes the command and returns the name of the command (without the arguments). If there was no command in *single* it will return `nullptr`.

**Parameters**
**single** – string with *LAMMPS* command

**Returns**
string with name of the parsed command w/o arguments

### 4.13.4 Citation management for contributed features

LAMMPS provides a built-in mechanism to remind users to cite publications that describe the implementation of contributed features. This is implemented through the *CiteMe* class.

#### Overview

When users use specific features in LAMMPS, a citation reminder can be configured in the contributed source code to remind users which publications they should cite. Citations are output in three ways:

- To the screen (with configurable verbosity)
- To the log file (with configurable verbosity)
- To an optional BibTeX file for easy integration with bibliography managers

The system automatically deduplicates citations, so each publication is listed only once even if multiple features reference it or they are used multiple times.

#### Adding a citation reminder to contributed code

If you are contributing a new feature to LAMMPS you can add a reminder for citing a publication describing the implementation of that feature. This is a simple two-step process:

**Step 1: Define the citation string**

At the top of your implementation file (typically a `.cpp` file), define a static string containing the citation in BibTeX format. The string should start with a single line description including a DOI URL, followed by a complete BibTeX entry. Example:

```
static const char cite_my_feature[] =
  "my_feature command: https://doi.org/10.1234/example.doi\n\n"
  "@Article{AuthorYear,\n"
  " author = {First Author and Second Author},\n"
  " title = {Title of the Paper},\n"
  " journal = {Journal Name},\n"
  " year = 2024,\n"
  " volume = 100,\n"
  " pages = {1-10}\n"
  "}\n\n";
```

**Step 2: Register the citation**

In your style's constructor, add a call to register the citation. Always check that the `citeme` pointer instance in the LAMMPS is available before calling, since the pointer will be NULL when citation tracking is disabled via a command line option:

```
MyStyle::MyStyle(LAMMPS *lmp) : Parent(lmp)
{
  if (lmp->citeme) lmp->citeme->add(cite_my_feature);
  // ... rest of constructor code
}
```

**Output:**

The example from above will produce by default the following output:

```
CITE-CITE-CITE-CITE-CITE-CITE-CITE-CITE-CITE-CITE-CITE-CITE-CITE
```

(continues on next page)

(continued from previous page)

```
Your simulation uses code contributions which should be cited:

- my_feature command: https://doi.org/10.1234/example.doi

The file log.cite lists these citations in BibTeX format.

CITE-CITE-CITE-CITE-CITE-CITE-CITE-CITE-CITE-CITE-CITE-CITE-CITE
```

### Implementation details

The *CiteMe* class uses hash-based deduplication to ensure each citation is shown only once, even if multiple features in a simulation reference the same publication. Only MPI rank 0 performs citation output to avoid flooding the output in parallel runs.

Citations added with *CiteMe::add()* are:

1. Written immediately to the BibTeX file (if enabled)
2. Buffered for screen and log file output
3. Flushed to screen and log file at appropriate times (typically at the end of the run or when LAMMPS has reached the end of the input file).

For the complete API documentation see the class reference below:

---

class **CiteMe** : protected *Pointers*

The *CiteMe* class provides a mechanism for *LAMMPS* to remind users to cite relevant publications when they use specific contributed features. This ensures proper attribution for the scientific work underlying *LAMMPS* implementations.

#### Public Functions

**CiteMe**(class *LAMMPS**, int, int, const char*)

Constructor for *CiteMe* class

**Parameters**

- **lmp** – Pointer to the main *LAMMPS* object
- **_screen** – Output mode for screen (VERBOSE or TERSE)
- **_logfile** – Output mode for log file (VERBOSE or TERSE)
- **_file** – Optional filename for BibTeX citation output file (can be NULL)

**~CiteMe**() override

Destructor flushes any remaining citations

void **add**(const std::string &reference)

Register a publication for citation output

Adds a citation to the set of publications to be cited. Each citation should contain a BibTeX entry and is output only once, even if add() is called multiple times with the same citation. The citation string should start with a brief description including a DOI, followed by a complete BibTeX entry.

This method should typically be called in the constructor of a style that implements a published method or algorithm.

**Parameters**
**reference** – String containing the citation in BibTeX format with DOI header

void **flush**()

Flush accumulated citation buffers to screen and log file

Outputs all pending citations to the screen and log file with appropriate formatting. Called automatically by the destructor when LAMMPS terminates or is reset by the *clear* command and at the end of a *run* or *minimize* command.

## 4.14 Platform abstraction functions

The `platform` sub-namespace inside the `LAMMPS_NS` namespace provides a collection of wrapper and convenience functions and utilities that perform common tasks for which platform specific code would be required or for which a more high-level abstraction would be convenient and reduce duplicated code. This reduces redundant implementations and encourages consistent behavior and thus has some overlap with the *"utils" sub-namespace*.

### 4.14.1 Time functions

double `LAMMPS_NS::platform::`**cputime**()

Return the consumed CPU time for the current process in seconds

This is a wrapper around the POSIX function getrusage() and its Windows equivalent. It is to be used in a similar fashion as MPI_Wtime(). Its resolution may be rather low so it can only be trusted when observing processes consuming CPU time of at least a few seconds.

**Returns**
used CPU time in seconds

double `LAMMPS_NS::platform::`**walltime**()

Return the wall clock state for the current process in seconds

This clock is counting continuous time and is initialized during load of the executable/library. Its absolute value must be considered arbitrary and thus elapsed wall times are measured in taking differences. It is therefore to be used in a similar fashion as MPI_Wtime() but has a different offset, usually leading to better resolution.

**Returns**
wall clock time in seconds

void `LAMMPS_NS::platform::`**usleep**(int usec)

Suspend execution for a microsecond interval

This emulates the usleep(3) BSD function call also mentioned in POSIX.1-2001. This is not a precise delay; it may be longer, but not shorter.

**Parameters**
**usec** – length of delay in microseconds

### 4.14.2 Platform information functions

std::string `LAMMPS_NS::platform::`**os_info**()

Return string with the operating system version and architecture info

**Returns**
string with info about the OS and the platform is is running on

std::string LAMMPS_NS::platform::**compiler_info**()

Return string with compiler version info

This function uses predefined compiler macros to identify Compilers and their version and configuration info.

**Returns**
string with the compiler information text

std::string LAMMPS_NS::platform::**cxx_standard**()

Return string with C++ standard version used to compile *LAMMPS*.

This function uses predefined compiler macros to identify the C++ standard version used to compile *LAMMPS* with.

**Returns**
string with the C++ standard version or "unknown"

std::string LAMMPS_NS::platform::**openmp_standard**()

Return string with OpenMP standard version info

This function uses predefined compiler macros to identify OpenMP support and the supported version of the standard.

**Returns**
string with the openmp information text

std::string LAMMPS_NS::platform::**mpi_vendor**()

Return string with MPI vendor info

This function uses predefined macros to identify the vendor of the MPI library used.

**Returns**
string with the MPI vendor information text

std::string LAMMPS_NS::platform::**mpi_info**(int &major, int &minor)

Return string with MPI version info

This function uses predefined macros and MPI function calls to identify the version of the MPI library used.

**Parameters**

- **major** – major version of the MPI standard (set on exit)
- **minor** – minor version of the MPI standard (set on exit)

**Returns**
string with the MPI version information text

std::string LAMMPS_NS::platform::**compress_info**()

Return string with list of available compression types and executables

This function tests which of the supported compression executables are available for reading or writing compressed files where supported.

**Returns**
string with list of available compression tools

### 4.14.3 File and path functions and global constants

Since we are requiring C++17 to compile LAMMPS, you can also make use of the functionality of the C++ filesystem library. The following functions are in part convenience functions or emulate the behavior of similar Python functions or Unix shell commands. Please note that the you need to use the `string()` member function of the std::filesystem::path class to get access to the path as a C++ string class instance.

char LAMMPS_NS::platform::**filepathsep**[] = "/"

Platform specific file path component separator

This is a string with the character that separates directories and filename in paths on a platform. If multiple are characters are provided, the first is the preferred one.

char LAMMPS_NS::platform::**pathvarsep** = ':'

Platform specific path environment variable component separator

This is the character that separates entries in "PATH"-style environment variables.

const char *LAMMPS_NS::platform::**guesspath**(FILE *fp, char *buf, int len)

Try to detect pathname from FILE pointer

Currently only supported on Linux, MacOS, and Windows. Otherwise will report "(unknown)".

On Linux the folder /proc/self/fd holds symbolic links to the actual pathnames associated with each open file descriptor of the current process. On MacOS the same kind of information can be obtained using `fcntl(fd, F_GETPATH,buf)`. On Windows we use `GetFinalPathNameByHandleA()` which is available with Windows Vista and later. If the buffer is too small (< 16 bytes) a null pointer is returned.

This function is used to provide a filename with error messages in functions where the filename is not passed as an argument, but the FILE * pointer.

**Parameters**

- **fp** – FILE pointer struct from STDIO library for which we want to detect the name
- **buf** – storage buffer for pathname. output will be truncated if not large enough
- **len** – size of storage buffer. output will be truncated to this length - 1

**Returns**

pointer to the storage buffer with path or a NULL pointer if buf is invalid or the buffer size is too small

std::string LAMMPS_NS::platform::**path_basename**(const std::string &path)

Strip off leading part of path, return just the filename

**Parameters**

**path** – file path

**Returns**

file name

std::string LAMMPS_NS::platform::**path_dirname**(const std::string &path)

Return the directory part of a path. Return "." if empty

**Parameters**

**path** – file path

**Returns**

directory name

std::string LAMMPS_NS::platform::**path_join**(const std::string &a, const std::string &b)

Join two pathname segments

This uses the forward slash '/' character unless *LAMMPS* is compiled for Windows where it uses the backward slash '\'

**Parameters**

- **a** – first path
- **b** – second path

**Returns**
combined path

bool LAMMPS_NS::platform::**file_is_readable**(const std::string &path)

Check if file exists and is readable

**Parameters**
**path** – file path

**Returns**
true if file exists and is readable

bool LAMMPS_NS::platform::**file_is_writable**(const std::string &path)

Check if file can be opened for writing

**Parameters**
**path** – file path

**Returns**
true if file can be opened for writing

std::string LAMMPS_NS::platform::**file_redirect**(const std::string &path)

Return target path if the file is a 'redirect file'

Git uses 'redirect files' instead of symbolic links on Windows since the Windows file system has no symbolic links. The redirect file is a text file with just one line: the symbolic link target path. This function opens the path parameter and reads a line. If that line is a readable file, it returns that path, otherwise the original path. The check is only performed when compiled for Windows. Otherwise the original path is always returned.

**Parameters**
**path** – file path to check

**Returns**
the redirected path or the original path

bool LAMMPS_NS::platform::**is_console**(FILE *fp)

Check if a file pointer may be connected to a console

**Parameters**
**fp** – file pointer

**Returns**
true if the file pointer is flagged as a TTY

double LAMMPS_NS::platform::**disk_free**(const std::string &path)

Return free disk space in bytes of file system pointed to by path

Returns -1.0 if the path is invalid or free space reporting not supported.

**Parameters**
**path** – file or folder path in file system

**Returns**

std::vector<std::string> LAMMPS_NS::platform::**list_directory**(const std::string &dir)

Get list of entries in a directory

This provides a list of strings of the entries in the directory without the leading path name while also skipping over ".." and ".".

**Parameters**
**dir** – path to directory

**Returns**
vector with strings of all directory entries

int LAMMPS_NS::platform::**chdir**(const std::string &path)

Change current directory

**Parameters**
**path** – new current working directory path

**Returns**
-1 if unsuccessful, otherwise >= 0

int LAMMPS_NS::platform::**mkdir**(const std::string &path)

Create a directory or directory path

Unlike the the `mkdir()` or `_mkdir()` functions of the C library, this function will also try to create non-existing sub-directories in case they don't exist, and thus it behaves like the `mkdir -p` command rather than plain `mkdir` or `md` in a Unix or Windows shell, respectively.

**Parameters**
**path** – directory path

**Returns**
-1 if unsuccessful, otherwise >= 0

int LAMMPS_NS::platform::**rmdir**(const std::string &path)

Delete a directory

Unlike the the `rmdir()` or `_rmdir()` functions of the C library, this function will check for the contents of the folder and recurse into any sub-folders, if necessary, and delete all contained folders and their contents before deleting the folder *path*.

**Parameters**
**path** – directory path

**Returns**
-1 if unsuccessful, otherwise >= 0

int LAMMPS_NS::platform::**unlink**(const std::string &path)

Delete a file

**Parameters**
**path** – path to file to be deleted

**Returns**
0 on success, -1 on error

### 4.14.4 Standard I/O function wrappers

bigint LAMMPS_NS::platform::**END_OF_FILE** = -1

constant to seek to the end of the file

bigint LAMMPS_NS::platform::**ftell**(FILE *fp)

Get current file position

**Parameters**
**fp** – FILE pointer of the given file

**Returns**
current FILE pointer position cast to a bigint

int LAMMPS_NS::platform::**fseek**(FILE *fp, bigint pos)

Set absolute file position

If the absolute position is END_OF_FILE, then position at the end of the file.

**Parameters**

- **fp** – FILE pointer of the given file
- **pos** – new position of the FILE pointer

**Returns**
0 if successful, otherwise -1

int LAMMPS_NS::platform::**ftruncate**(FILE *fp, bigint length)

Truncate file to a given length and re-position file pointer

**Parameters**

- **fp** – FILE pointer of the given file
- **length** – length to which the file is being truncated to

**Returns**
0 if successful, otherwise -1

FILE *LAMMPS_NS::platform::**popen**(const std::string &cmd, const std::string &mode)

Open a pipe to a command for reading or writing

**Parameters**

- **cmd** – command for the pipe
- **mode** – "r" for reading from *cmd* or "w" for writing to *cmd*

**Returns**
file pointer to the pipe if successful or null

int LAMMPS_NS::platform::**pclose**(FILE *fp)

Close a previously opened pipe

**Parameters**
**fp** – FILE pointer for the pipe

**Returns**
exit status of the pipe command or -1 in case of errors

### 4.14.5 Environment variable functions

int LAMMPS_NS::platform::**putenv**(const std::string &vardef)

Add variable to the environment

**Parameters**
**vardef** – variable name or variable definition (NAME=value)

**Returns**
-1 if failure otherwise 0

int LAMMPS_NS::platform::**unsetenv**(const std::string &variable)

Delete variable from the environment

**Parameters**
**variable** – variable name

**Returns**
-1 if failure otherwise 0

std::vector<std::string> LAMMPS_NS::platform::**list_pathenv**(const std::string &var)

Get list of entries in a path environment variable

This provides a list of strings of the entries in an environment variable that is containing a "path" like "PATH" or "LD_LIBRARY_PATH".

**Parameters**
**var** – name of the environment variable

**Returns**
vector with strings of all entries in that path variable

std::string LAMMPS_NS::platform::**find_exe_path**(const std::string &cmd)

Find pathname of an executable in the standard search path

This function will traverse the list of directories in the PATH environment variable and look for the executable *cmd*. If the file exists and is executable the full path is returned as string, otherwise an empty string is returned.

On Windows the *cmd* string must not include an extension as this function will automatically append the extensions ".exe", ".com" and ".bat" and look for those paths. On Windows also the current directory is checked (and first), but otherwise is not checked unless "." exists in the PATH environment variable.

Because of the nature of the check, this will not detect shell functions built-in command or aliases.

**Parameters**
**cmd** – name of command

**Returns**
vector with strings of all directory entries

### 4.14.6 Dynamically loaded object or library functions

void *LAMMPS_NS::platform::**dlopen**(const std::string &fname)

Open a shared object file or library

**Parameters**
**fname** – name or path of the shared object

**Returns**
handle to the shared object or null

int LAMMPS_NS::platform::**dlclose**(void *handle)

Close a shared object

This releases the object corresponding to the provided handle. Resolved symbols associated with this handle may not be used after this call

**Parameters**
**handle** – handle to an opened shared object

**Returns**
0 if successful, non-zero of not

void *LAMMPS_NS::platform::**dlsym**(void *handle, const std::string &symbol)

Resolve a symbol in shared object

**Parameters**

- **handle** – handle to an opened shared object
- **symbol** – name of the symbol to extract

**Returns**
pointer to the resolved symbol or null

std::string LAMMPS_NS::platform::**dlerror**()

Obtain error diagnostic info after dynamic linking function calls

Return a human-readable string describing the most recent error that occurred when using one of the functions for dynamic loading objects the last call to this function. If there was no error, the string is empty.

**Returns**
string with error message or empty

### 4.14.7 Compressed file I/O functions

bool LAMMPS_NS::platform::**has_compress_extension**(const std::string &file)

Check if a file name ends in a known extension for a compressed file format

Currently supported file extensions are: .gz, .bz2, .zst, .xz, .lzma, .lz4, .br, and .7z

**Parameters**
**file** – name of the file to check

**Returns**
true if the file has a known extension, otherwise false

FILE *LAMMPS_NS::platform::**compressed_read**(const std::string &file)

Open pipe to compressed text file for reading

**Parameters**
**file** – name of the file to open

**Returns**
FILE pointer to pipe using for reading the compressed file.

FILE *LAMMPS_NS::platform::**compressed_write**(const std::string &file)

Open pipe to compressed text file for writing

**Parameters**
**file** – name of the file to open

**Returns**
FILE pointer to pipe using for reading the compressed file.

## 4.15 Utility functions

The utils sub-namespace inside the LAMMPS_NS namespace provides a collection of convenience functions and utilities that perform common tasks that are required repeatedly throughout the LAMMPS code like reading or writing

to files with error checking or translation of strings into specific types of numbers with checking for validity. This reduces redundant implementations and encourages consistent behavior and thus has some overlap with the *"platform" sub-namespace*.


- *I/O with status check and similar functions*
- *String to number conversions with validity check*
- *String processing*
- *Potential file functions*
- *Argument processing*
- *Convenience functions*
- *Customized standard functions*


### 4.15.1 I/O with status check and similar functions

The the first two functions are wrappers around the corresponding C library calls `fgets()` or `fread()`. They will check if there were errors on reading or an unexpected end-of-file state was reached. In that case, the functions will stop with an error message, indicating the name of the problematic file, if possible unless the *error* argument is a NULL pointer.

The *utils::fgets_trunc()* function will work similar for `fgets()` but it will read in a whole line (i.e. until the end of line or end of file), but store only as many characters as will fit into the buffer including a final newline character and the terminating NULL byte. If the line in the file is longer it will thus be truncated in the buffer. This function is used by *utils::read_lines_from_file()* to read individual lines but make certain they follow the size constraints.

The *utils::read_lines_from_file()* function will read the requested number of lines of a maximum length into a buffer and will return 0 if successful or 1 if not. It also guarantees that all lines are terminated with a newline character and the entire buffer with a NULL character.

---

void `LAMMPS_NS::utils::`**`sfgets`**(const char *srcname, int srcline, char *s, int size, FILE *fp, const char *filename, Error *error)

Safe wrapper around fgets() which aborts on errors or EOF and prints a suitable error message to help debugging.

Use nullptr as the error parameter to avoid the abort on EOF or error.

**Parameters**

- **`srcname`** – name of the calling source file (from FLERR macro)
- **`srcline`** – line in the calling source file (from FLERR macro)
- **`s`** – buffer for storing the result of fgets()
- **`size`** – size of buffer s (max number of bytes read by fgets())
- **`fp`** – file pointer used by fgets()
- **`filename`** – file name associated with fp (may be a null pointer; then *LAMMPS* will try to detect)
- **`error`** – pointer to Error class instance (for abort) or nullptr

void LAMMPS_NS::utils::**sfread**(const char *srcname, int srcline, void *s, size_t size, size_t num, FILE *fp, const char *filename, Error *error)

Safe wrapper around fread() which aborts on errors or EOF and prints a suitable error message to help debugging.

Use nullptr as the error parameter to avoid the abort on EOF or error.

**Parameters**

- **srcname** – name of the calling source file (from FLERR macro)
- **srcline** – line in the calling source file (from FLERR macro)
- **s** – buffer for storing the result of fread()
- **size** – size of data elements read by fread()
- **num** – number of data elements read by fread()
- **fp** – file pointer used by fread()
- **filename** – file name associated with fp (may be a null pointer; then *LAMMPS* will try to detect)
- **error** – pointer to Error class instance (for abort) or nullptr

char *LAMMPS_NS::utils::**fgets_trunc**(char *s, int size, FILE *fp)

Wrapper around fgets() which reads whole lines but truncates the data to the buffer size and ensures a newline char at the end.

This function is useful for reading line based text files with possible comments that should be parsed later. This applies to data files, potential files, atomfile variable files and so on. It is used instead of fgets() by utils::read_lines_from_file().

**Parameters**

- **s** – buffer for storing the result of fgets()
- **size** – size of buffer s (max number of bytes returned)
- **fp** – file pointer used by fgets()

int LAMMPS_NS::utils::**read_lines_from_file**(FILE *fp, int nlines, int nmax, char *buffer, int me, MPI_Comm comm)

Read N lines of text from file into buffer and broadcast them

This function uses repeated calls to fread() to fill a buffer with newline terminated text. If a line does not end in a newline (e.g. at the end of a file), it is added. The caller has to allocate an nlines by nmax sized buffer for storing the text data. Reading is done by MPI rank 0 of the given communicator only, and thus only MPI rank 0 needs to provide a valid file pointer.

**Parameters**

- **fp** – file pointer used by fread
- **nlines** – number of lines to be read
- **nmax** – maximum length of a single line
- **buffer** – buffer for storing the data.
- **me** – MPI rank of calling process in MPI communicator
- **comm** – MPI communicator for broadcast

**Returns**

1 if the read was short, 0 if read was successful

### 4.15.2 String to number conversions with validity check

These functions should be used to convert strings to numbers. They are are strongly preferred over C library calls like `atoi()` or `atof()` since they check if the **entire** string is a valid (floating-point or integer) number, and will error out instead of silently returning the result of a partial conversion or zero in cases where the string is not a valid number. This behavior improves detecting typos or issues when processing input files.

Similarly the *utils::logical()* function will convert a string into a boolean and will only accept certain words.

The *do_abort* flag should be set to `true` in case this function is called only on a single MPI rank, as that will then trigger the a call to `Error::one()` for errors instead of `Error::all()` and avoids a "hanging" calculation when run in parallel.

Please also see *utils::is_integer()* and *utils::is_double()* for testing strings for compliance without conversion.

---

double LAMMPS_NS::utils::**numeric**(const char *file, int line, const std::string &str, bool do_abort, *LAMMPS* *lmp)

Convert a string to a floating point number while checking if it is a valid floating point or integer number

**Parameters**

- **file** – name of source file for error message
- **line** – line number in source file for error message
- **str** – string to be converted to number
- **do_abort** – determines whether to call Error::one() or Error::all()
- **lmp** – pointer to top-level *LAMMPS* class instance

**Returns**

double precision floating point number

double LAMMPS_NS::utils::**numeric**(const char *file, int line, const char *str, bool do_abort, *LAMMPS* *lmp)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

**Parameters**

- **file** – name of source file for error message
- **line** – line number in source file for error message
- **str** – string to be converted to number
- **do_abort** – determines whether to call Error::one() or Error::all()
- **lmp** – pointer to top-level *LAMMPS* class instance

**Returns**

double precision floating point number

int LAMMPS_NS::utils::**inumeric**(const char *file, int line, const std::string &str, bool do_abort, *LAMMPS* *lmp)

Convert a string to an integer number while checking if it is a valid integer number (regular int)

**Parameters**

- **file** – name of source file for error message

- **line** – line number in source file for error message
- **str** – string to be converted to number
- **do_abort** – determines whether to call Error::one() or Error::all()
- **lmp** – pointer to top-level *LAMMPS* class instance

**Returns**
integer number (regular int)

int LAMMPS_NS::utils::**inumeric**(const char *file, int line, const char *str, bool do_abort, *LAMMPS* *lmp)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

**Parameters**

- **file** – name of source file for error message
- **line** – line number in source file for error message
- **str** – string to be converted to number
- **do_abort** – determines whether to call Error::one() or Error::all()
- **lmp** – pointer to top-level *LAMMPS* class instance

**Returns**
double precision floating point number

bigint LAMMPS_NS::utils::**bnumeric**(const char *file, int line, const std::string &str, bool do_abort, *LAMMPS* *lmp)

Convert a string to an integer number while checking if it is a valid integer number (bigint)

**Parameters**

- **file** – name of source file for error message
- **line** – line number in source file for error message
- **str** – string to be converted to number
- **do_abort** – determines whether to call Error::one() or Error::all()
- **lmp** – pointer to top-level *LAMMPS* class instance

**Returns**
integer number (bigint)

bigint LAMMPS_NS::utils::**bnumeric**(const char *file, int line, const char *str, bool do_abort, *LAMMPS* *lmp)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

**Parameters**

- **file** – name of source file for error message
- **line** – line number in source file for error message
- **str** – string to be converted to number
- **do_abort** – determines whether to call Error::one() or Error::all()
- **lmp** – pointer to top-level *LAMMPS* class instance

**Returns**
double precision floating point number

tagint LAMMPS_NS::utils::**tnumeric**(const char *file, int line, const std::string &str, bool do_abort, *LAMMPS* *lmp)

Convert a string to an integer number while checking if it is a valid integer number (tagint)

**Parameters**

- **file** – name of source file for error message
- **line** – line number in source file for error message
- **str** – string to be converted to number
- **do_abort** – determines whether to call Error::one() or Error::all()
- **lmp** – pointer to top-level *LAMMPS* class instance

**Returns**

integer number (tagint)

tagint LAMMPS_NS::utils::**tnumeric**(const char *file, int line, const char *str, bool do_abort, *LAMMPS* *lmp)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

**Parameters**

- **file** – name of source file for error message
- **line** – line number in source file for error message
- **str** – string to be converted to number
- **do_abort** – determines whether to call Error::one() or Error::all()
- **lmp** – pointer to top-level *LAMMPS* class instance

**Returns**

double precision floating point number

int LAMMPS_NS::utils::**logical**(const char *file, int line, const std::string &str, bool do_abort, *LAMMPS* *lmp)

Convert a string to a boolean while checking whether it is a valid boolean term. Valid terms are 'yes', 'no', 'true', 'false', 'on', 'off', and '1', '0'. Only lower case is accepted.

**Parameters**

- **file** – name of source file for error message
- **line** – line number in source file for error message
- **str** – string to be converted to logical
- **do_abort** – determines whether to call Error::one() or Error::all()
- **lmp** – pointer to top-level *LAMMPS* class instance

**Returns**

1 if string resolves to "true", otherwise 0

int LAMMPS_NS::utils::**logical**(const char *file, int line, const char *str, bool do_abort, *LAMMPS* *lmp)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

**Parameters**

- **file** – name of source file for error message
- **line** – line number in source file for error message

- **str** – string to be converted to logical
- **do_abort** – determines whether to call Error::one() or Error::all()
- **lmp** – pointer to top-level *LAMMPS* class instance

**Returns**
1 if string resolves to "true", otherwise 0

---

### 4.15.3 String processing

The following are functions to help with processing strings and parsing files or arguments.

---

char *LAMMPS_NS::utils::**strdup**(const std::string &text)

Make C-style copy of string in new storage

This allocates a storage buffer and copies the C-style or C++ style string into it. The buffer is allocated with "new" and thus needs to be deallocated with "delete[]".

**Parameters**
**text** – string that should be copied

**Returns**
new buffer with copy of string

std::string LAMMPS_NS::utils::**lowercase**(const std::string &line)

Convert string to lowercase

**Parameters**
**line** – string that should be converted

**Returns**
new string with all lowercase characters

std::string LAMMPS_NS::utils::**uppercase**(const std::string &line)

Convert string to uppercase

**Parameters**
**line** – string that should be converted

**Returns**
new string with all uppercase characters

std::string LAMMPS_NS::utils::**trim**(const std::string &line)

Trim leading and trailing whitespace. Like TRIM() in Fortran.

**Parameters**
**line** – string that should be trimmed

**Returns**
new string without whitespace (string)

std::string LAMMPS_NS::utils::**trim_comment**(const std::string &line)

Return string with anything from the first '#' character onward removed

**Parameters**
**line** – string that should be trimmed

**Returns**
new string without comment (string)

std::string LAMMPS_NS::utils::**strcompress**(const std::string &text)

Compress whitespace in a string

Added in version 4Feb2025.

This function compresses whitespace in a string to just a single blank.

**Parameters**
**text** – the text to be compressed

**Returns**
string with whitespace compressed to single blanks

std::string LAMMPS_NS::utils::**strip_style_suffix**(const std::string &style, *LAMMPS* *lmp)

Remove style suffix from string if suffix flag is active

This will try to undo the effect from using the *suffix command* or the *-suffix/-sf* command-line flag and return correspondingly modified string.

**Parameters**

- **style** – string of style name
- **lmp** – pointer to the *LAMMPS* class (has suffix_flag and suffix strings)

**Returns**
processed string

std::string LAMMPS_NS::utils::**star_subst**(const std::string &name, bigint step, int pad)

Replace first '*' character in a string with a number, optionally zero-padded

If there is no '' *character in the string, return the original string. If the number requires more characters than the value of the *pad* argument*, do not add zeros; otherwise add as many zeroes as needed to the left to make the the number representation *pad* characters wide.

**Parameters**

- **name** – string with file containing a '*' (or not)
- **step** – step number to replace the (first) '*'
- **pad** – zero-padding (may be zero)

**Returns**
processed string

inline bool LAMMPS_NS::utils::**has_utf8**(const std::string &line)

Check if a string will likely have UTF-8 encoded characters

UTF-8 uses the 7-bit standard ASCII table for the first 127 characters and all other characters are encoded as multiple bytes. For the multi-byte characters the first byte has either the highest two, three, or four bits set followed by a zero bit and followed by one, two, or three more bytes, respectively, where the highest bit is set and the second highest bit set to 0. The remaining bits combined are the character code, which is thus limited to 21-bits.

For the sake of efficiency this test only checks if a character in the string has the highest bit set and thus is very likely an UTF-8 character. It will not be able to tell this this is a valid UTF-8 character or whether it is a 2-byte, 3-byte, or 4-byte character.

*See also*
utils::utf8_subst()

**Parameters**
**line** – string that should be checked

**Returns**
true if string contains UTF-8 encoded characters (bool)

std::string LAMMPS_NS::utils::**utf8_subst**(const std::string &line)
Replace known UTF-8 characters with ASCII equivalents

*See also*
utils::has_utf8()

**Parameters**
**line** – string that should be converted

**Returns**
new string with ascii replacements (string)

size_t LAMMPS_NS::utils::**count_words**(const char *text)
Count words in C-string, ignore any whitespace matching " \t\r\n\f"

**Parameters**
**text** – string that should be searched

**Returns**
number of words found

size_t LAMMPS_NS::utils::**count_words**(const std::string &text)
Count words in string, ignore any whitespace matching " \t\r\n\f"

**Parameters**
**text** – string that should be searched

**Returns**
number of words found

size_t LAMMPS_NS::utils::**count_words**(const std::string &text, const std::string &separators)
Count words in string with custom choice of separating characters

**Parameters**
- **text** – string that should be searched
- **separators** – string containing characters that will be treated as whitespace

**Returns**
number of words found

size_t LAMMPS_NS::utils::**trim_and_count_words**(const std::string &text, const std::string &separators = "\t\r\n\f")

Count words in a single line, trim anything from '#' onward

**Parameters**

- **text** – string that should be trimmed and searched
- **separators** – string containing characters that will be treated as whitespace

**Returns**
number of words found

template<typename **T**>
std::string LAMMPS_NS::utils::**join**(const std::vector<*T*> &values, const std::string &sep)

Take list of values and join them with a given separator text.

This is the inverse operation of what the Tokenizer classes do. This is a generalization of the join_words() function and similar to fmt::join() but only supports the vector STL container, and to use the begin/end iterator version you have to use: `utils::join(std::vector(x.begin(), x.end()), sep);`. This approach can also be used to support other STL containers.

**Parameters**

- **values** – STL vector with values
- **sep** – separator string (may be empty)

**Returns**
string with the concatenated values and separators

std::string LAMMPS_NS::utils::**join_words**(const std::vector<std::string> &words, const std::string &sep)

Take list of words and join them with a given separator text.

This is the inverse operation of what the split_words() function and Tokenizer classes do.

**Parameters**

- **words** – STL vector with strings
- **sep** – separator string (may be empty)

**Returns**
string with the concatenated words and separators

std::vector<std::string> LAMMPS_NS::utils::**split_words**(const std::string &text)

Take text and split into non-whitespace words.

This can handle strings with single and double quotes, escaped quotes, and escaped codes within quotes, but due to using an STL container and STL strings is rather slow because of making copies. Designed for parsing command-lines and similar text and not for time critical processing. Use a tokenizer class if performance matters.

***See also***
*Tokenizer*, *ValueTokenizer*

**Parameters**
**text** – string that should be split

**Returns**
STL vector with the words

std::vector<std::string> LAMMPS_NS::utils::**split_lines**(const std::string &text)

Take multi-line text and split into lines

**Parameters**
**text** – string that should be split

**Returns**
STL vector with the lines

bool LAMMPS_NS::utils::**strsame**(const std::string &text1, const std::string &text2)

Compare two string while ignoring whitespace

Added in version 4Feb2025.

This function compares two strings while skipping over any kind of whitespace (blank, tab, newline, carriage return, etc.).

**Parameters**

- **text1** – the first text to be compared
- **text2** – the second text to be compared

**Returns**
true if the non-whitespace part of the two strings matches, false if not

bool LAMMPS_NS::utils::**strmatch**(const std::string &text, const std::string &pattern)

Match text against a simplified regex pattern

This function provides flexible and specific matching of a string against a pattern using a simplified regular expression. It is supposed to be a safe, more specific and simple to use API to find pattern matches that is compatible with both, C-style strings and C++ strings. The purpose is to replace uses of either strncmp() or strstr() in the code which are more complex to use and may have incorrect matches. sub-strings safely. With strncmp() for finding prefixes, the number of characters to match must be counted, which can lead to errors, while using "^pattern" will do the same by just adding the '^' prefix. Matching for suffixes using strstr() is not as specific as 'pattern$', and complex matches, e.g. "^rigid.*\small.*", to match all small body optimized rigid fixes require only one test.

This function uses a mini regex-module adapted and customized from https://github.com/kokke/tiny-regex-c which is in the public domain. The regular expressions support the following standard and custom regex elements:

```
'.'         Dot, matches any character
'^'         Start anchor, matches beginning of string
'$'         End anchor, matches end of string
'*'         Asterisk, match zero or more (greedy)
'+'         Plus, match one or more (greedy)
'?'         Question, match zero or one (non-greedy)
'[abc]'     Character class, match if one of {'a', 'b', 'c'}
'[a-zA-Z]' Character ranges, the character set of the ranges { a-z | A-Z }
'\s'        Whitespace, \t \f \r \n \v and spaces
'\S'        Non-whitespace
'\w'        Alphanumeric, [a-zA-Z_0-9]
'\W'        Non-alphanumeric
'\d'        Digits, [0-9]
'\D'        Non-digits
```

```
'\i'       Integer chars, [0-9], '+' and '-'
'\I'       Non-integers
'\f'       Floating point number chars, [0-9], '.', 'e', 'E', '+' and '-'
'\F'       Non-floats
```

They do **not** support:

```
'[^abc]'   Inverted class
'a|b'      Branches
'(abc)+'   Groups
```

Please note that regular expressions are different from globbing.

The use of std::string arguments allows for simple concatenation even with char * type variables. Example: utils::strmatch(text, std::string("^") + charptr)

**Parameters**

- **text** – the text to be matched against the pattern
- **pattern** – the search pattern, which may contain regexp markers

**Returns**

true if the pattern matches, false if not

std::string LAMMPS_NS::utils::**strfind**(const std::string &text, const std::string &pattern)

Find sub-string that matches a simplified regex pattern

This function is a companion function to utils::strmatch(). Arguments and logic is the same, but instead of a boolean, it returns the sub-string that matches the regex pattern. There can be only one match. This can be used as a more flexible alternative to strstr().

**Parameters**

- **text** – the text to be matched against the pattern
- **pattern** – the search pattern, which may contain regexp markers

**Returns**

the string that matches the pattern or an empty one

bool LAMMPS_NS::utils::**is_integer**(const std::string &str)

Check if string can be converted to valid integer

**Parameters**

**str** – string that should be checked

**Returns**

true, if string contains valid a integer, false otherwise

bool LAMMPS_NS::utils::**is_double**(const std::string &str)

Check if string can be converted to valid floating-point number

**Parameters**

**str** – string that should be checked

**Returns**

true, if string contains valid number, false otherwise

bool LAMMPS_NS::utils::**is_id**(const std::string &str)

Check if string is a valid ID ID strings may contain only letters, numbers, and underscores.

**Parameters**
**str** – string that should be checked

**Returns**
true, if string contains valid id, false otherwise

int LAMMPS_NS::utils::**is_type**(const std::string &str)

Check if string is a valid type label, or numeric type, or numeric type range. Numeric type or type range may only contain digits or the ':' *character. Type label strings may not contain a digit, or a* ':', or a '#' character as the first character to distinguish them from comments and numeric types or type ranges. They also may not contain any whitespace. If the string is a valid numeric type or type range the function returns 0, if it is a valid type label the function returns 1, otherwise it returns -1.

**Parameters**
**str** – string that should be checked

**Returns**
0, 1, or -1, depending on whether the string is valid numeric type, valid type label or neither, respectively

---

### 4.15.4 Potential file functions

std::string LAMMPS_NS::utils::**get_potential_file_path**(const std::string &path)

Determine full path of potential file. If file is not found in current directory, search directories listed in LAMMPS_POTENTIALS environment variable

**Parameters**
**path** – file path

**Returns**
full path to potential file

std::string LAMMPS_NS::utils::**get_potential_date**(const std::string &path, const std::string &potential_name)

Read potential file and return DATE field if it is present

**Parameters**

- **path** – file path
- **potential_name** – name of potential that is being read

**Returns**
DATE field if present

std::string LAMMPS_NS::utils::**get_potential_units**(const std::string &path, const std::string &potential_name)

Read potential file and return UNITS field if it is present

**Parameters**

- **path** – file path
- **potential_name** – name of potential that is being read

**Returns**
UNITS field if present

int LAMMPS_NS::utils::**get_supported_conversions**(const int property)

Return bitmask of available conversion factors for a given property

**Parameters**
**property** – property to be converted

**Returns**
bitmask indicating available conversions

double LAMMPS_NS::utils::**get_conversion_factor**(const int property, const int conversion)

Return unit conversion factor for given property and selected from/to units

**Parameters**

- **property** – property to be converted
- **conversion** – constant indicating the conversion

**Returns**
conversion factor

FILE *LAMMPS_NS::utils::**open_potential**(const std::string &name, *LAMMPS* *lmp, int *auto_convert)

Open a potential file as specified by *name*

If opening the file directly fails, the function will search for it in the list of folder pointed to by the environment variable LAMMPS_POTENTIALS (if it is set).

If the potential file has a UNITS tag in the first line, the tag's value is compared to the current unit style setting. The behavior of the function then depends on the value of the *auto_convert* parameter. If it is a null pointer, then the unit values must match or else the open will fail with an error. Otherwise the bitmask that *auto_convert* points to is used check for compatibility with possible automatic conversions by the calling function. If compatible, the bitmask is set to the required conversion or utils::NOCONVERT.

**Parameters**

- **name** – file- or pathname of the potential file
- **lmp** – pointer to top-level *LAMMPS* class instance
- **auto_convert** – pointer to unit conversion bitmask or nullptr

**Returns**
FILE pointer of the opened potential file or nullptr

---

### 4.15.5 Argument processing

template<typename **TYPE**>
void LAMMPS_NS::utils::**bounds**(const char *file, int line, const std::string &str, bigint nmin, bigint nmax, *TYPE* &nlo, *TYPE* &nhi, Error *error, int failed = -2)

Compute index bounds derived from a string with a possible wildcard

This functions processes the string in *str* and set the values of *nlo* and *nhi* according to the following five cases:

- a single number, i: nlo = i; nhi = i;
- a single asterisk, *: nlo = nmin; nhi = nmax;
- a single number followed by an asterisk, i*: nlo = i; nhi = nmax;
- a single asterisk followed by a number, *i: nlo = nmin; nhi = i;

- two numbers with an asterisk in between. i*j: nlo = i; nhi = j;

**Template Parameters**
**TYPE** – the type of the index that is subject to the wildcard expansion

**Parameters**

- **file** – name of source file for error message
- **line** – line number in source file for error message
- **str** – string to be processed
- **nmin** – smallest possible lower bound
- **nmax** – largest allowed upper bound
- **nlo** – **[out]** lower bound
- **nhi** – **[out]** upper bound
- **error** – pointer to Error class for out-of-bounds messages
- **failed** – argument index with failed expansion (optional)

template<typename **TYPE**>
void LAMMPS_NS::utils::**bounds_typelabel**(const char *file, int line, const std::string &str, bigint nmin, bigint nmax, *TYPE* &nlo, *TYPE* &nhi, *LAMMPS* *lmp, int mode)

Same as utils::bounds(), but string may be a typelabel

Added in version 27June2024.

This functions adds the following case to *utils::bounds()*:

- a single type label, typestr: nlo = nhi = label2type(typestr)

**Template Parameters**
**TYPE** – the type of the index that is subject to the boundary expansion

**Parameters**

- **file** – name of source file for error message
- **line** – line number in source file for error message
- **str** – string to be processed
- **nmin** – smallest possible lower bound
- **nmax** – largest allowed upper bound
- **nlo** – **[out]** lower bound
- **nhi** – **[out]** upper bound
- **lmp** – pointer to top-level *LAMMPS* class instance
- **mode** – select labelmap using constants from *Atom* class

int LAMMPS_NS::utils::**expand_args**(const char *file, int line, int narg, char **arg, int mode, char **&earg, *LAMMPS* *lmp, int **argmap = nullptr)

Expand list of arguments when containing fix/compute wildcards

This function searches the list of arguments in *arg* for strings of the kind c_ID[*], f_ID[*], v_ID[*], i2_ID[*], d2_ID[*], or c_ID:gname:dname[*] referring to computes, fixes, vector style variables, custom per-atom arrays, or grids, respectively. Any such strings are replaced by one or more strings with the '*' character replaced by the corresponding possible numbers as determined from the fix, compute, variable, property, or grid instance. Unrecognized strings are just copied. If the *mode* parameter is set to 0, expand global vectors, but not global arrays; if it is set to 1, expand global arrays (by column) but not global vectors.

If any expansion happens, the earg list and all its strings are new allocations and must be freed explicitly by the caller. Otherwise arg and earg will point to the same address and no explicit de-allocation is needed by the caller.

The *argmap* pointer to an int pointer may be used to accept an array of integers mapping the arguments after the expansion to their original index. If this pointer is NULL (the default) than this map is not created. Otherwise, it must be deallocated by the calling code.

**Parameters**

- **file** – name of source file for error message
- **line** – line number in source file for error message
- **narg** – number of arguments in current list
- **arg** – argument list, possibly containing wildcards
- **mode** – select between global vectors(=0) and arrays (=1)
- **earg** – new argument list with wildcards expanded
- **lmp** – pointer to top-level *LAMMPS* class instance
- **argmap** – pointer to integer pointer for mapping expanded indices to input (optional)

**Returns**

number of arguments in expanded list

std::vector<std::string> LAMMPS_NS::utils::**parse_grid_id**(const char *file, int line, const std::string &name, Error *error)

Parse grid reference into 3 sub-strings

Format of grid ID reference = id:gname:dname. Return vector with the 3 sub-strings.

**Parameters**

- **file** – name of source file for error message
- **line** – line number in source file for error message
- **name** – complete grid ID
- **error** – pointer to Error class

**Returns**

std::vector<std::string> containing the 3 sub-strings

char *LAMMPS_NS::utils::**expand_type**(const char *file, int line, const std::string &str, int mode, *LAMMPS* *lmp)

Expand type label string into its equivalent numeric type

This function checks if a given string may be a type label and then searches the labelmap type indicated by the *mode* argument for the corresponding numeric type. If this is found, a copy of the numeric type string is made

and returned. Otherwise a null pointer is returned. If a string is returned, the calling code must free it with delete[].

**Parameters**

- **file** – name of source file for error message
- **line** – line number in source file for error message
- **str** – type string to be expanded
- **mode** – select labelmap using constants from *Atom* class
- **lmp** – pointer to top-level *LAMMPS* class instance

**Returns**
pointer to expanded string or null pointer

---

### 4.15.6 Convenience functions

template<typename ...**Args**>
void `LAMMPS_NS::utils::`**logmesg**(*LAMMPS* *lmp, const std::string &format, *Args*&&... args)

Send formatted message to screen and logfile, if available

This function simplifies the repetitive task of outputting some message to both the screen and/or the log file. The template wrapper with {fmt} formatting and argument processing allows this function to work similar to :cpp:func:`utils::print() <LAMMPS_NS::utils::print>`.

**Parameters**

- **lmp** – pointer to *LAMMPS* class instance
- **format** – format string of message to be printed
- **args** – arguments to format string

void `LAMMPS_NS::utils::`**logmesg**(*LAMMPS* *lmp, const std::string &mesg)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

**Parameters**

- **lmp** – pointer to *LAMMPS* class instance
- **mesg** – string with message to be printed

template<typename ...**Args**>
void `LAMMPS_NS::utils::`**print**(FILE *fp, const std::string &format, *Args*&&... args)

Write formatted message to file

Added in version 4Feb2025.

This function implements a version of (f)printf() that uses {fmt} formatting

**Parameters**

- **fp** – stdio FILE pointer
- **format** – format string of message to be printed
- **args** – arguments to format string

void LAMMPS_NS::utils::**print**(FILE *fp, const std::string &mesg)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Print string message without format

**Parameters**

- **fp** – stdio FILE pointer
- **mesg** – string with message to be printed

std::string LAMMPS_NS::utils::**errorurl**(int errorcode)

Return text redirecting the user to a specific paragraph in the manual

The *LAMMPS* manual contains detailed explanations for errors and warnings where a simple error message may not be sufficient. These can be reached through URLs with a numeric code > 0. This function creates the corresponding text to be included into the error message that redirects the user to that URL. Using an error code of 0 returns a message pointing to a URL discussing error messages in general.

**Parameters**

**errorcode** – non-negative number pointing to a paragraph in the manual

void LAMMPS_NS::utils::**missing_cmd_args**(const std::string &file, int line, const std::string &cmd, Error *error)

Print error message about missing arguments for command

This function simplifies the repetitive reporting missing arguments to a command.

**Parameters**

- **file** – name of source file for error message
- **line** – line number in source file for error message
- **cmd** – name of the failing command
- **error** – pointer to Error class instance (for abort) or nullptr

std::string LAMMPS_NS::utils::**point_to_error**(*Input* *input, int failed)

Create string with last command and optionally pointing to arg with error

Added in version 4Feb2025.

This function is a helper function for error messages. It creates extra output in error messages. It will produce either two or three lines: the original last input line *before* variable substitutions, the corresponding pre-processed command (only when different) and one or more '^' characters pointing to the faulty argument as indicated by the *failed* argument. Any whitespace in the lines with the command output are compressed to a single blank by calling *strcompress()*

**Parameters**

- **input** – pointer to the *Input* class instance (for access to last command args)
- **failed** – index of the faulty argument (-1 to point to the command itself)

**Returns**

string with two or three lines to follow error messages

void LAMMPS_NS::utils::**flush_buffers**(*LAMMPS* *lmp)

Flush output buffers

This function calls fflush() on screen and logfile FILE pointers if available and thus tells the operating system to output all currently buffered data. This is local operation and independent from buffering by a file system or an MPI library.

std::string LAMMPS_NS::utils::**getsyserror**()

Return a string representing the current system error status

This is a wrapper around calling strerror(errno).

**Returns**
error string

std::string LAMMPS_NS::utils::**check_packages_for_style**(const std::string &style, const std::string &name, *LAMMPS* *lmp)

Report if a requested style is in a package or may have a typo

**Parameters**

- **style** – type of style that is to be checked for
- **name** – name of style that was not found
- **lmp** – pointer to top-level *LAMMPS* class instance

**Returns**
string usable for error messages

double LAMMPS_NS::utils::**timespec2seconds**(const std::string &timespec)

Convert a time string to seconds

The strings "off" and "unlimited" result in -1

**Parameters**
**timespec** – a string in the following format: ([[HH:]MM:]SS)

**Returns**
total in seconds

int LAMMPS_NS::utils::**date2num**(const std::string &date)

Convert a *LAMMPS* version date to a number

This will generate a number YYYYMMDD from a date string (with or without blanks) that is suitable for numerical comparisons, i.e. later dates will generate a larger number.

The day may or may not have a leading zero, the month is identified by the first 3 letters (so there may be more) and the year may be 2 or 4 digits (the missing 2 digits will be assumed as 20. That is 04 corresponds to 2004).

No check is made whether the date is valid.

**Parameters**
**date** – string in the format (Day Month Year)

**Returns**
date code

std::string LAMMPS_NS::utils::**current_date**()

Return current date as string

This will generate a string containing the current date in YYYY-MM-DD format.

**Returns**
string with current date

---

### 4.15.7 Customized standard functions

int LAMMPS_NS::utils::**binary_search**(const double needle, const int n, const double *haystack)

Binary search in a vector of ascending doubles of length N

If the value is smaller than the smallest value in the vector, 0 is returned. If the value is larger or equal than the largest value in the vector, N-1 is returned. Otherwise the index that satisfies the condition

haystack[index] <= value < haystack[index+1]

is returned, i.e. a value from 1 to N-2. Note that if there are tied values in the haystack, always the larger index is returned as only that satisfied the condition.

**Parameters**

- **needle** – search value for which are are looking for the closest index
- **n** – size of the haystack array
- **haystack** – array with data in ascending order.

**Returns**
index of value in the haystack array smaller or equal to needle

void LAMMPS_NS::utils::**merge_sort**(int *index, int num, void *ptr, int (*comp)(int, int, void*))

Custom merge sort implementation

This function provides a custom upward hybrid merge sort implementation with support to pass an opaque pointer to the comparison function, e.g. for access to class members. This avoids having to use global variables. For improved performance, it uses an in-place insertion sort on initial chunks of up to 64 elements and switches to merge sort from then on.

**Parameters**

- **index** – Array with indices to be sorted
- **num** – Length of the index array
- **ptr** – Pointer to opaque object passed to comparison function
- **comp** – Pointer to comparison function

---

## 4.16 Special Math functions

The MathSpecial namespace implements a selection of custom and optimized mathematical functions for a variety of applications.

double LAMMPS_NS::MathSpecial::**factorial**(const int n)

Fast tabulated factorial function

This function looks up pre-computed factorial values for arguments of n = 0 to a maximum of 167, which is the maximal value representable by a double precision floating point number. For other values of n a NaN value is returned.

**Parameters**
**n** – argument (valid: 0 <= n <= 167)

**Returns**
value of n! as double precision number or NaN

double LAMMPS_NS::MathSpecial::**exp2_x86**(double x)

Fast implementation of 2^x without argument checks for little endian CPUs

This function implements an optimized version of pow(2.0, x) that does not check for valid arguments and thus may only be used where arguments are well behaved. The implementation makes assumptions about the layout of double precision floating point numbers in memory and thus will only work on little endian CPUs. If little endian cannot be safely detected, the result of calling pow(2.0, x) will be returned. This function also is the basis for the fast exponential fm_exp(x).

**Parameters**
**x** – argument

**Returns**
value of 2^x as double precision number

double LAMMPS_NS::MathSpecial::**fm_exp**(double x)

Fast implementation of exp(x) for little endian CPUs

This function implements an optimized version of exp(x) for little endian CPUs. It calls the exp2_x86(x) function with a suitable prefactor to x to return exp(x). The implementation makes assumptions about the layout of double precision floating point numbers in memory and thus will only work on little endian CPUs. If little endian cannot be safely detected, the result of calling the exp(x) implementation in the standard math library will be returned.

**Parameters**
**x** – argument

**Returns**
value of e^x as double precision number

static inline double LAMMPS_NS::MathSpecial::**my_erfcx**(const double x)

Fast scaled error function complement exp(x*x)*erfc(x) for coul/long styles

This is a portable fast implementation of exp(x*x)*erfc(x) that can be used in coul/long pair styles as a replacement for the polynomial expansion that is/was widely used. Unlike the polynomial expansion, that is only accurate at the level of single precision floating point it provides full double precision accuracy, but at comparable speed (unlike the erfc() implementation shipped with GNU standard math library).

**Parameters**
**x** – argument

**Returns**
value of e^(x*x)*erfc(x)

static inline double LAMMPS_NS::MathSpecial::**expmsq**(double x)

Fast implementation of exp(-x*x) for little endian CPUs for coul/long styles

This function implements an optimized version of exp(-x*x) based on exp2_x86() for use with little endian CPUs. If little endian cannot be safely detected, the result of calling the exp(-x*x) implementation in the standard math library will be returned.

**Parameters**
**x** – argument

**Returns**
value of e^(-x*x) as double precision number

static inline double LAMMPS_NS::MathSpecial::**square**(const double &x)

Fast inline version of pow(x, 2.0)

**Parameters**
**x** – argument

**Returns**
x*x

static inline double LAMMPS_NS::MathSpecial::**cube**(const double &x)

Fast inline version of pow(x, 3.0)

**Parameters**
**x** – argument

**Returns**
x*x

static inline double LAMMPS_NS::MathSpecial::**powsign**(const int n)

static inline double LAMMPS_NS::MathSpecial::**powint**(const double &x, const int n)

static inline double LAMMPS_NS::MathSpecial::**powsinxx**(const double &x, int n)

---

## 4.17 Tokenizer classes

The purpose of the tokenizer classes is to simplify the recurring task of breaking lines of text down into words and/or numbers. Traditionally, LAMMPS code would be using the `strtok()` function from the C library for that purpose, but that function has two significant disadvantages: 1) it cannot be used concurrently from different LAMMPS instances since it stores its status in a global variable and 2) it modifies the string that it is processing. These classes were implemented to avoid both of these issues and also to reduce the amount of code that needs to be written.

The basic procedure is to create an instance of the tokenizer class with the string to be processed as an argument and then do a loop until all available tokens are read. The constructor has a default set of separator characters, but that can be overridden. The default separators are all "whitespace" characters, i.e. the space character, the tabulator character, the carriage return character, the linefeed character, and the form feed character.

Listing 1: Tokenizer class example listing entries of the PATH environment variable

```
#include "tokenizer.h"
#include <cstdlib>
#include <string>
#include <iostream>

using namespace LAMMPS_NS;

int main(int, char **)
{
    const char *path = getenv("PATH");

    if (path != nullptr) {
        Tokenizer p(path,":");
        while (p.has_next())
```

(continues on next page)

(continued from previous page)

```
            std::cout << "Entry: " << p.next() << "\n";
    }
    return 0;
}
```

Most tokenizer operations cannot fail except for *LAMMPS_NS::Tokenizer::next()* (when used without first checking with *LAMMPS_NS::Tokenizer::has_next()*) and *LAMMPS_NS::Tokenizer::skip()*. In case of failure, the class will throw an exception, so you may need to wrap the code using the tokenizer into a `try` / `catch` block to handle errors. The *LAMMPS_NS::ValueTokenizer* class may also throw an exception when a (type of) number is requested as next token that is not compatible with the string representing the next word.

Listing 2: ValueTokenizer class example with exception handling

```
#include "tokenizer.h"
#include <cstdlib>
#include <string>
#include <iostream>

using namespace LAMMPS_NS;

int main(int, char **)
{
    const char *text = "1 2 3 4 5 20.0 21 twentytwo 2.3";
    double num1(0),num2(0),num3(0),num4(0);

    ValueTokenizer t(text);
    // read 4 doubles after skipping over 5 numbers
    try {
        t.skip(5);
        num1 = t.next_double();
        num2 = t.next_double();
        num3 = t.next_double();
        num4 = t.next_double();
    } catch (TokenizerException &e) {
        std::cout << "Reading numbers failed: " << e.what() << "\n";
    }
    std::cout << "Values: " << num1 << " " << num2 << " " << num3 << " " << num4 << "\n";
    return 0;
}
```

This code example should produce the following output:

```
Reading numbers failed: Not a valid floating-point number: 'twentytwo'
Values: 20 21 0 0
```

---

class **Tokenizer**

### Public Functions

**Tokenizer**(std::string str, std::string separators = TOKENIZER_DEFAULT_SEPARATORS)

Class for splitting text into words

This tokenizer will break down a string into sub-strings (i.e words) separated by the given separator characters. If the string contains certain known UTF-8 characters they will be replaced by their ASCII equivalents processing the string.

*See also*
*ValueTokenizer*, *utils::split_words()*, *utils::utf8_subst()*

**Parameters**

- **str** – string to be processed
- **_separators** – string with separator characters (default: " \t\r\n\f")

void **reset**()

Re-position the tokenizer state to the first word, i.e. the first non-separator character

void **skip**(int n = 1)

Skip over a given number of tokens

**Parameters**
**n** – number of tokens to skip over

bool **has_next**() const

Indicate whether more tokens are available

**Returns**
true if there are more tokens, false if not

bool **contains**(const std::string &str) const

Search the text to be processed for a sub-string.

This method does a generic sub-string match.

**Parameters**
**str** – string to be searched for

**Returns**
true if string was found, false if not

bool **matches**(const std::string &str) const

Search the text to be processed for regular expression match.

This method matches the current string against a regular expression using the *utils::strmatch()* function.

**Parameters**
**str** – regular expression to be matched against

**Returns**
true if string was found, false if not

std::string **next**()

Retrieve next token.

**Returns**
string with the next token

size_t **count**()

Count number of tokens in text.

**Returns**
number of counted tokens

std::vector<std::string> **as_vector**()

Retrieve the entire text converted to an STL vector of tokens.

**Returns**
The STL vector

class **TokenizerException** : public exception

General Tokenizer exception class

Subclassed by *InvalidFloatException*, *InvalidIntegerException*

### Public Functions

**TokenizerException**() = delete

The default constructor is disabled

explicit **TokenizerException**(const std::string &msg, const std::string &token)

Thrown during retrieving or skipping tokens

**Parameters**

- **msg** – String with error message
- **token** – String of the token or word that caused the error

inline const char ***what**() const noexcept override

Retrieve message describing the thrown exception

This function provides the message that can be retrieved when the corresponding exception is caught.

**Returns**
String with error message

class **ValueTokenizer**

### Public Functions

**ValueTokenizer**(const std::string &str, const std::string &separators =
TOKENIZER_DEFAULT_SEPARATORS)

Class for reading text with numbers

***See also***
*Tokenizer*

**See also**

Tokenizer *InvalidIntegerException InvalidFloatException*

**Parameters**

- **str** – String to be processed
- **separators** – String with separator characters (default: " \t\r\n\f")

std::string **next_string**()

Retrieve next token

**Returns**
string with next token

tagint **next_tagint**()

Retrieve next token and convert to tagint

**Returns**
value of next token

bigint **next_bigint**()

Retrieve next token and convert to bigint

**Returns**
value of next token

int **next_int**()

Retrieve next token and convert to int

**Returns**
value of next token

double **next_double**()

Retrieve next token and convert to double

**Returns**
value of next token

bool **has_next**() const

Indicate whether more tokens are available

**Returns**
true if there are more tokens, false if not

bool **contains**(const std::string &value) const

Search the text to be processed for a sub-string.

This method does a generic sub-string match.

**Parameters**
**value** – string with value to be searched for

**Returns**
true if string was found, false if not

bool **matches**(const std::string &str) const

Search the text to be processed for regular expression match.

This method matches the current string against a regular expression using the *utils::strmatch()* function.

**Parameters**
**str** – regular expression to be matched against

**Returns**
true if string was found, false if not

void **skip**(int ntokens = 1)

Skip over a given number of tokens

**Parameters**
**n** – number of tokens to skip over

size_t **count**()

Count number of tokens in text.

**Returns**
number of counted tokens

class **InvalidIntegerException** : public *TokenizerException*

Exception thrown by ValueTokenizer when trying to convert an invalid integer string

### Public Functions

inline explicit **InvalidIntegerException**(const std::string &token)

Thrown during converting string to integer number

**Parameters**
**token** – String of the token/word that caused the error

class **InvalidFloatException** : public *TokenizerException*

Exception thrown by ValueTokenizer when trying to convert an floating point string

### Public Functions

inline explicit **InvalidFloatException**(const std::string &token)

Thrown during converting string to floating point number

**Parameters**
**token** – String of the token/word that caused the error

## 4.18 Argument parsing classes

The purpose of argument parsing classes it to simplify and unify how arguments of commands in LAMMPS are parsed and to make abstractions of repetitive tasks.

The *LAMMPS_NS::ArgInfo* class provides an abstraction for parsing references to compute or fix styles, variables or custom integer or double properties handled by *fix property/atom*. These would start with a "c_", "f_", "v_", "d_", "d2_", "i_", or "i2_" followed by the ID or name of than instance and may be postfixed with one or two array indices "[<number>]" with numbers > 0.

A typical code segment would look like this:

Listing 3: Usage example for ArgInfo class

```
int nvalues = 0;
for (iarg = 0; iarg < nargnew; iarg++) {
  ArgInfo argi(arg[iarg]);

  which[nvalues] = argi.get_type();
  argindex[nvalues] = argi.get_index1();
  ids[nvalues] = argi.copy_name();

  if ((which[nvalues] == ArgInfo::UNKNOWN)
      || (which[nvalues] == ArgInfo::NONE)
      || (argi.get_dim() > 1))
    error->all(FLERR,"Illegal compute XXX command");

  nvalues++;
}
```

---

class **ArgInfo**

### Public Types

enum **ArgTypes**

constants for argument types

*Values:*

enumerator **ERROR**

enumerator **UNKNOWN**

enumerator **NONE**

enumerator **X**

enumerator **V**

enumerator **F**

enumerator **COMPUTE**

enumerator **FIX**

enumerator **VARIABLE**

enumerator **KEYWORD**

enumerator **TYPE**

enumerator **MOLECULE**

enumerator **DNAME**

enumerator **INAME**

enumerator **DENSITY_NUMBER**

enumerator **DENSITY_MASS**

enumerator **MASS**

enumerator **TEMPERATURE**

enumerator **BIN1D**

enumerator **BIN2D**

enumerator **BIN3D**

enumerator **BINSPHERE**

enumerator **BINCYLINDER**

### Public Functions

**ArgInfo**(const std::string &arg, int allowed = *COMPUTE* | *FIX* | *VARIABLE*)

Class for processing references to fixes, computes and variables

This class provides an abstraction for the repetitive task of parsing arguments that may contain references to fixes, computes, variables, or custom per-atom properties. It will identify the name and the index value in the first and second dimension, if present.

**Parameters**

- **arg** – string with possible reference
- **allowed** – integer with bitmap of allowed types of references

inline int **get_type**() const

get type of reference

Return a type constant for the reference. This may be either COMPUTE, FIX, VARIABLE (if not restricted to a subset of those by the "allowed" argument of the constructor) or NONE, if it if not a recognized or allowed reference, or UNKNOWN, in case some error happened identifying or parsing the values of the indices

**Returns**
integer with a constant from ArgTypes enumerator

inline int **get_dim**() const

get dimension of reference

This will return either 0, 1, 2 depending on whether the reference has no, one or two "[{number}]" postfixes.

**Returns**
integer with the dimensionality of the reference

inline int **get_index1**() const

get index of first dimension

This will return the number in the first "[{number}]" postfix or 0 if there is no postfix.

**Returns**
integer with index or the postfix or 0

inline int **get_index2**() const

get index of second dimension

This will return the number in the second "[{number}]" postfix or -1 if there is no second postfix.

**Returns**
integer with index of the postfix or -1

inline const char ***get_name**() const

return reference to the ID or name of the reference

This string is pointing to an internal storage element and is only valid to use while the ArgInfo class instance is in scope. If you need a long-lived string make a copy with copy_name().

**Returns**
C-style char * string

char ***copy_name**()

make copy of the ID of the reference as C-style string

The ID is copied into a buffer allocated with "new" and thus must be later deleted with "delete []" to avoid a memory leak. Because it is a full copy in a newly allocated buffer, the lifetime of this string extends beyond the the time the ArgInfo class is in scope.

**Returns**
copy of string as char *

---

## 4.19 Safe pointer classes

These are custom classes to support the Resource Acquisition Is Initialization (RAII) programming idiom in LAMMPS for certain types of pointers. Currently there is:

class **SafeFilePtr**

Class to automatically close a FILE pointer when it goes out of scope

This is a drop-in replacement for declaring a `FILE *` variable and can be passed to functions or used in logical expressions the same way. This is particularly useful when a code block will throw an exception, or has to close and re-open a file and similar. It helps to simplify code and reduces the risk of memory and file descriptor leaks.

Below are some usage examples:

```
// Replace:
FILE *fp = nullptr;
// With:
SafeFilePtr fp;

// You can use "fp" as usual:
fp = fopen("some.file","r");
// a second assignment will automatically close the opened file
fp = fopen("other.file", "r");
// and assigning nullptr will just close it
fp = nullptr;

// There also is a custom constructor available as a shortcut
SafeFilePtr fp(fopen("some.file", "r"));

// You can indicate that a file was opened with popen() to call pclose() instead of␣
↪fclose()
SafeFilePtr fp;
if (platform::has_compress_extension(filename)) {
  fp.set_pclose();
  fp = platform::compressed_write(filename);
} else {
  fp = fopen(filename, "w");
}
if (!fp) error->one(FLERR, "Failed to open file {}: {}", filename,␣
↪utils::getsyserror());

// reading or writing works without needing to change the source code
fputs("write text to file\n", fp);
char buffer[100];
utils::sfgets(FLERR, buffer, 100, fp, filename, error);
```

### Public Functions

*SafeFilePtr* &**operator=**(FILE *_fp)

Assign new file pointer and close old one if still open.

The value of use_pclose determines whether `pclose()` is called or `fclose()`. Assigning `nullptr` closes the file and resets use_pclose

**Parameters**
**_fp** – new file pointer, may be `nullptr`

**Returns**
reference to updated class instance

inline void **set_pclose**()

Flag that the file pointer needs to be closed with `pclose()` instead of `fclose()`

inline **operator FILE***() const

Custom type cast operator so that *SafeFilePtr* can be used where FILE * was used

**Returns**
currently stored/monitored file pointer

## 4.20 File reader classes

The purpose of the file reader classes is to simplify the recurring task of reading and parsing files. They can use the *ValueTokenizer* class to process the read in text. The *TextFileReader* is a more general version while *PotentialFileReader* is specialized to implement the behavior expected for looking up and reading/parsing files with potential parameters in LAMMPS. The potential file reader class requires a LAMMPS instance, requires to be run on MPI rank 0 only, will use the *utils::get_potential_file_path* function to look up and open the file, and will call the `LAMMPS_NS::Error` class in case of failures to read or to convert numbers, so that LAMMPS will be aborted.

Listing 4: Use of PotentialFileReader class in pair style coul/streitz

```
PotentialFileReader reader(lmp, file, "coul/streitz");
char * line;

while((line = reader.next_line(NPARAMS_PER_LINE))) {
  try {
    ValueTokenizer values(line);
    std::string iname = values.next_string();

    int ielement;
    for (ielement = 0; ielement < nelements; ielement++)
      if (iname == elements[ielement]) break;

    if (nparams == maxparam) {
      maxparam += DELTA;
      params = (Param *) memory->srealloc(params,maxparam*sizeof(Param),
                                          "pair:params");
    }

    params[nparams].ielement = ielement;
    params[nparams].chi = values.next_double();
    params[nparams].eta = values.next_double();
    params[nparams].gamma = values.next_double();
    params[nparams].zeta = values.next_double();
    params[nparams].zcore = values.next_double();

  } catch (TokenizerException & e) {
    error->one(FLERR, e.what());
  }
  nparams++;
}
```

A file that would be parsed by the reader code fragment looks like this:

```
# DATE: 2015-02-19 UNITS: metal CONTRIBUTOR: Ray Shan CITATION: Streitz and Mintmire,␣
↪Phys Rev B, 50, 11996-12003 (1994)
#
# X (eV)                J (eV)          gamma (1/AA)   zeta (1/AA)    Z (e)

Al      0.000000        10.328655       0.000000        0.968438        0.763905
O       5.484763        14.035715       0.000000        2.143957        0.000000
```

class **TextFileReader**

### Public Functions

**TextFileReader**(const std::string &filename, const std::string &filetype)

Class for reading and parsing text files

The value of the class member variable *ignore_comments* controls whether any text following the pound sign (#) should be ignored (true) or not (false). Default: true, i.e. ignore.

***See also***
*TextFileReader*

**Parameters**

- **filename** – Name of file to be read
- **filetype** – Description of file type for error messages

**TextFileReader**(FILE *fp, std::string filetype)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

This function is useful in combination with *utils::open_potential()*.

**Note**

The FILE pointer is not closed in the destructor, but will be advanced when reading from it.

**Parameters**

- **fp** – File descriptor of the already opened file
- **filetype** – Description of file type for error messages

virtual **~TextFileReader**()

Closes the file

void **set_bufsize**(int)

adjust line buffer size

**Parameters**
**newsize** – New size of the internal line buffer

void **rewind**()

Reset file to the beginning

void **skip_line**()

Read the next line and ignore it

char ***next_line**(int nparams = 0)

Read the next line(s) until *nparams* words have been read.

This reads a line and counts the words in it, if the number is less than the requested number, it will read the next line, as well. Output will be a string with all read lines combined. The purpose is to somewhat replicate the reading behavior of formatted files in Fortran.

If the *ignore_comments* class member has the value *true*, then any text read in is truncated at the first '#' character.

**Parameters**
**nparams** – Number of words that must be read. Default: 0

**Returns**
String with the concatenated text

void **next_dvector**(double *list, int n)

Read lines until *n* doubles have been read and stored in array *list*

This reads lines from the file using the next_line() function, and splits them into floating-point numbers using the ValueTokenizer class and stores the number in the provided list.

**Parameters**

- **list** – Pointer to array with suitable storage for *n* doubles
- **n** – Number of doubles to be read

*ValueTokenizer* **next_values**(int nparams, const std::string &separators = TOKENIZER_DEFAULT_SEPARATORS)

Read text until *nparams* words are read and passed to a tokenizer object for custom parsing.

This reads lines from the file using the next_line() function, and splits them into floating-point numbers using the ValueTokenizer class and stores the number in the provided list.

**Parameters**

- **nparams** – Number of words to be read
- **separators** – String with list of separators.

**Returns**
ValueTokenizer object for read in text

### Public Members

bool **ignore_comments**

Controls whether comments are ignored.

class **PotentialFileReader** : protected *Pointers*

### Public Functions

**PotentialFileReader**(class *LAMMPS* *lmp, const std::string &filename, const std::string &potential_name, const std::string &name_suffix, const int auto_convert = 0)

Class for reading and parsing *LAMMPS* potential files

The value of the class member variable *ignore_comments* controls whether any text following the pound sign (#) should be ignored (true) or not (false). Default: true, i.e. ignore.

***See also***
*TextFileReader*

**Parameters**

- **lmp** – Pointer to *LAMMPS* instance
- **filename** – Name of file to be read

- **potential_name** – Name of potential style for error messages
- **name_suffix** – Suffix added to potential name in error messages
- **auto_convert** – Bitmask of supported unit conversions

**~PotentialFileReader**() override

Closes the file

void **set_bufsize**(int bufsize)

Set line buffer size of the internal TextFileReader class instance.

**Parameters**
**bufsize** – New size of the line buffer

void **ignore_comments**(bool value)

Set comment (= text after '#') handling preference for the file to be read

**Parameters**
**value** – Comment text is ignored if true, or not if false

void **rewind**()

Reset file to the beginning

void **skip_line**()

Read a line but ignore its content

char ***next_line**(int nparams = 0)

Read the next line(s) until *nparams* words have been read.

This reads a line and counts the words in it, if the number is less than the requested number, it will read the next line, as well. Output will be a string with all read lines combined. The purpose is to somewhat replicate the reading behavior of formatted files in Fortran.

**Parameters**
**nparams** – Number of words that must be read. Default: 0

**Returns**
String with the concatenated text

void **next_dvector**(double *list, int n)

Read lines until *n* doubles have been read and stored in array *list*

This reads lines from the file using the next_line() function, and splits them into floating-point numbers using the ValueTokenizer class and stores the number in the provided list.

**Parameters**

- **list** – Pointer to array with suitable storage for *n* doubles
- **n** – Number of doubles to be read

*ValueTokenizer* **next_values**(int nparams, const std::string &separators = TOKENIZER_DEFAULT_SEPARATORS)

Read text until *nparams* words are read and passed to a tokenizer object for custom parsing.

This reads lines from the file using the next_line() function, and splits them into floating-point numbers using the ValueTokenizer class and stores the number in the provided list.

**Parameters**

- **nparams** – Number of words to be read

- **separators** – String with list of separators.

**Returns**
ValueTokenizer object for read in text

double **next_double**()

Read next line and convert first word to a double

**Returns**
Value of first word in line as double

int **next_int**()

Read next line and convert first word to an int

**Returns**
Value of first word in line as int

tagint **next_tagint**()

Read next line and convert first word to a tagint

**Returns**
Value of first word in line as tagint

bigint **next_bigint**()

Read next line and convert first word to a bigint

**Returns**
Value of first word in line as bigint

std::string **next_string**()

Read next line and return first word

**Returns**
First word of read in line

---

## 4.21 Type label support

### 4.21.1 Overview

The *LabelMap* class provides a two way mapping between symbolic type labels in input and output files and numeric types as they are used by LAMMPS internally. Instead of changing the numeric types in files to satisfy the requirements from LAMMPS for a given application, the symbolic types can remain and only the label map needs to be adjusted. When following the convention that the labels for bonded interactions are created by joining the constituent atom types with hyphens, this can significantly improve readability, maintainability, and re-usability of inputs and reduces the chance of errors.

The LabelMap class also provides automatic type inference for bonded interactions based on their constituent atom types. For instance, based on the atom type labels, the corresponding bond, angle, dihedral, or improper types can be inferred provided the corresponding type labels follow the convention that they are composed of the symbolic atom types connected by hyphens.

### 4.21.2 Integration with utils namespace

Several utility functions in the `utils` namespace work with type labels and interact with the LabelMap class:

- *utils::is_type()* - Validates whether a string is a valid type label.
- *utils::expand_type()* - Converts a type label string to its numeric equivalent using the LabelMap.

- *utils::bounds_typelabel()* - Extended version of `utils::bounds()` that accepts type labels in addition to numeric ranges. Uses `expand_type()` internally to convert labels to numeric bounds before processing.

These functions enable seamless integration of type labels throughout LAMMPS, allowing commands that accept type specifications to work with both numeric indices and symbolic labels. Below are some code examples.

### Finding types from labels and vice versa

```
#include "label_map.h"
#include "atom.h"

// assuming this code is used inside a class that is derived from LAMMPS_NS::Pointers
LabelMap *lmap = atom->lmap;

// Forward lookup: Get numeric type from label
int ctype = lmap->find_type("C", Atom::ATOM);    // Returns atom type for "C"
int htype = lmap->find_type("H", Atom::ATOM);    // Returns atom type for "H"
int missing = lmap->find_type("X", Atom::ATOM);  // Returns -1 (not found)

// Reverse lookup: Get label from numeric type
const std::string &label1 = lmap->find_label(1, Atom::ATOM);  // Returns label for type 1
const std::string &label2 = lmap->find_label(2, Atom::BOND);  // Returns bond label for␣
↪type 2

// Check if all types have labels
bool complete = lmap->is_complete(Atom::ATOM);  // Returns true if all atom types labeled
```

### Inferring bonded types from atom types

```
#include "label_map.h"
#include "atom.h"

LabelMap *lmap = atom->lmap;

// Assume we have: labelmap atom 1 C 2 H 3 N
// And: labelmap bond 1 C-H 2 C-C 3 C-N

// Infer bond type from numeric atom types
int bt1 = lmap->infer_bondtype(1, 2);  // Returns 1 (C-H bond)
int bt2 = lmap->infer_bondtype(1, 1);  // Returns 2 (C-C bond)
int bt3 = lmap->infer_bondtype(3, 1);  // Returns 3 (C-N bond, symmetric match)

// Infer bond type from atom type labels (handles symmetry automatically)
int bt4 = lmap->infer_bondtype({"C", "H"});  // Returns 1 (C-H)
int bt5 = lmap->infer_bondtype({"H", "C"});  // Returns 1 (symmetric match)
int bt6 = lmap->infer_bondtype({"C", "N"});  // Returns 3 (C-N)
```

### Validating and expanding type labels

```
#include "utils.h"
#include "lammps.h"

using namespace LAMMPS_NS;
```

(continues on next page)

(continued from previous page)

```
LAMMPS *lmp = /* ... */;

// Validate type label strings
int result1 = utils::is_type("C");     // Returns 1 (valid label)
int result2 = utils::is_type("123");   // Returns 0 (numeric type)
int result3 = utils::is_type("*");     // Returns -1 (invalid - starts with *)
int result4 = utils::is_type("C H");   // Returns -1 (invalid - contains whitespace)

// Convert type label to numeric string
char *numstr = utils::expand_type(FLERR, "C", Atom::ATOM, lmp);
if (numstr) {
    // Use the numeric type string
    delete[] numstr;  // Must delete after use
}

// Convert type label to integer
int type = utils::expand_type_int(FLERR, "C", Atom::ATOM, lmp, true);
// The 'true' argument enables range verification

// Use bounds_typelabel for ranges with label support
int lo, hi;
utils::bounds_typelabel(FLERR, "C:H", 1, 10, lo, hi, lmp, Atom::ATOM);
// Expands "C:H" to numeric range, e.g., "1:2" -> lo=1, hi=2
```

### 4.21.3 LabelMap class reference

class **LabelMap** : protected *Pointers*

Manage type labels for atoms, bonds, angles, dihedrals, and impropers.

The *LabelMap* class provides functionality to map between string labels and numeric type indices for atoms, bonds, angles, dihedrals, and impropers in *LAMMPS*. This enables users to reference types by symbolic names (e.g., “C”, “H”, “C-H”) instead of numeric indices, improving readability and maintainability of input scripts.

Type labels for bonded interactions *may* (but are not required to) use a hyphen-delimited format indicating the types of the constituent atoms. Examples:

- Bond types: “atom1-atom2” (e.g., “C-H”, “N-O”)
- Angle types: “atom1-atom2-atom3” (e.g., “H-C-H”, “C-N-C”)
- Dihedral types: “atom1-atom2-atom3-atom4” (e.g., “C-C-N-H”)
- Improper types: “atom1-atom2-atom3-atom4” (e.g., “C-N-C-C”)

The class supports bidirectional lookup (label <-> type) and can infer bonded interaction types from constituent atom types when using a hyphen-delimited format convention.

#### Interaction type inference from hyphen-delimited labels

These methods infer bonded interaction types (bonds, angles, dihedrals, impropers) from constituent atom types using a hyphen-delimited format. This requires that type labels for bonded interactions were entered following this convention. The inference functions consider the symmetry of the interaction and thus atom types may be swapped accordingly and the bonded type will still be matched.

int **infer_bondtype**(int atype1, int atype2)

Infer bond type from two numeric atom types

Look up or create a bond type from two atom type indices by constructing a hyphen-delimited label (e.g., "C-H") and searching the bond type labels.

**Parameters**

- **atype1** – First atom type index
- **atype2** – Second atom type index

**Returns**

Bond type index if types match in the specified order, negative bond type index if types match in reverse order, 0 if there is no match found

int **infer_bondtype**(const std::vector<std::string> &labels)

Infer bond type from atom type labels

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Look up a bond type from two atom type labels.

**Parameters**

**labels** – Vector of two atom type label strings

**Returns**

Bond type index if types match in the specified order, negative bond type index if types match in reverse order, 0 if there is no match found

int **infer_angletype**(int atype1, int atype2, int atype3)

Infer angle type from three numeric atom types

Look up or create an angle type from three atom type indices by constructing a hyphen-delimited label (e.g., "H1-C1-H2").

**Parameters**

- **atype1** – First atom type index
- **atype2** – Second atom type index (center atom)
- **atype3** – Third atom type index

**Returns**

Angle type index if types match in the specified order, negative angle type index if types match in reverse order, 0 if there is no match found

int **infer_angletype**(const std::vector<std::string> &labels)

Infer angle type from three atom type labels

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Look up an angle type from three atom type labels.

**Parameters**

**labels** – Vector of three atom type label strings

**Returns**

Angle type index if types match in the specified order, negative angle type index if types match in reverse order, 0 if there is no match found

int **infer_dihedraltype**(int atype1, int atype2, int atype3, int atype4)

Infer dihedral type from four numeric atom types

Look up a dihedral type from four atom type indices by constructing a hyphen-delimited label (e.g., "C-C-N-H").

**Parameters**

- **atype1** – First atom type index
- **atype2** – Second atom type index
- **atype3** – Third atom type index
- **atype4** – Fourth atom type index

**Returns**

Dihedral type index if types match in the specified order, negative dihedral type index if types match in reverse order, 0 if there is no match found

int **infer_dihedraltype**(const std::vector<std::string> &labels)

Infer dihedral type from atom type labels

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Look up a dihedral type from four atom type labels.

**Parameters**

**labels** – Vector of four atom type label strings

**Returns**

Dihedral type index if types match in the specified order, negative dihedral type index if types match in reverse order, 0 if there is no match found

int **infer_impropertype**(int atype1, int atype2, int atype3, int atype4, std::array<int, 4> *iorder = nullptr)

Infer improper type from four numeric atom types

Look up an improper type from four atom type indices by constructing a hyphen-delimited label (e.g., "C-N-C-C").

**Parameters**

- **atype1** – First atom type index (center atom)
- **atype2** – Second atom type index
- **atype3** – Third atom type index
- **atype4** – Fourth atom type index
- **iorder** – Order in which types were matched to improper type label

**Returns**

Improper type index if types match in the specified order, negative improper type index if types match but different order, 0 if there is no match found

int **infer_impropertype**(const std::vector<std::string> &labels, std::array<int, 4> *iorder = nullptr)

Infer improper type from atom type labels

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Look up an improper type from four atom type labels.

**Parameters**

- **labels** – Vector of four atom type label strings
- **iorder** – Order in which types were matched to improper type label

**Returns**

Improper type index if types match in the specified order, negative improper type index if types match but different order, 0 if there is no match found

### I/O methods for label map persistence

void **write_data**(FILE *fp)

Write label map to data file

Output all type labels as sections to a *LAMMPS* data file.

**Parameters**

**fp** – File pointer for writing

void **read_restart**(FILE *fp)

Read label map from restart file

Restore label map data from a *LAMMPS* restart file.

**Parameters**

**fp** – File pointer for reading

void **write_restart**(FILE*)

Write label map to restart file

Save label map data to a *LAMMPS* restart file for later restoration.

**Parameters**

**fp** – File pointer for writing

void **check_labels**()

Check if type labels are self-consistent.

### Public Functions

**LabelMap**(*LAMMPS* *lmp, int natomtypes, int nbondtypes, int nangletypes, int ndihedraltypes, int nimpropertypes)

Construct a *LabelMap* instance

**Parameters**

- **lmp** – Pointer to *LAMMPS* instance
- **natomtypes** – Number of atom types in map
- **nbondtypes** – Number of bond types in map
- **nangletypes** – Number of angle types in map
- **ndihedraltypes** – Number of dihedral types in map
- **nimpropertypes** – Number of improper types in map

void **modify_lmap**(int narg, char **arg)

Process labelmap command from input script

Add or modify type label mappings from the LAMMPS *labelmap* input command.

**Parameters**

- **narg** – Number of arguments
- **arg** – Array of argument strings

void **merge_lmap**(*LabelMap* *lmap, int mode)

Copy another *LabelMap* into this one

Merge type labels from another LabelMap instance into the current one. Currently used when combining data from multiple sources with *read_data add* or when replicating the system with *replicate*.

**Parameters**

- **lmap** – Pointer to source *LabelMap*
- **mode** – Merge mode flag

void **create_lmap2lmap**(*LabelMap* *lmap, int mode)

Create index mapping between two LabelMaps

Build a mapping structure (lmap2lmap) that translates type indices from another *LabelMap* to the current one based on matching labels.

**Parameters**

- **lmap** – Pointer to source *LabelMap*
- **mode** – Mapping mode flag

int **find_type**(const std::string&, int) const

Find numeric type from type label

Look up the numeric type index corresponding to a type label string.

**Parameters**

- **mylabel** – Type label string to search for
- **mode** – Type category: Atom::ATOM, Atom::BOND, Atom::ANGLE, Atom::DIHEDRAL, or Atom::IMPROPER

**Returns**
Numeric type index (1-based), or -1 if not found

const std::string &**find_label**(int, int) const

Find type label from numeric type

Reverse lookup: retrieve the type label string for a given numeric type.

**Parameters**

- **i** – Numeric type index (1-based)
- **mode** – Type category: Atom::ATOM, Atom::BOND, Atom::ANGLE, Atom::DIHEDRAL, or Atom::IMPROPER

**Returns**
Reference to type label string, or empty string if not found

bool **is_complete**(int) const

Check if all types have assigned labels

Verify that every type in the specified category has a corresponding label.

**Parameters**
**mode** – Type category: Atom::ATOM, Atom::BOND, Atom::ANGLE, Atom::DIHEDRAL, or Atom::IMPROPER

**Returns**
True if all types have labels, false otherwise

int **parse_typelabel**(int ntypes, const std::string &label, std::vector<std::string> &types)

Parse hyphen-delimited type label into components

Split a hyphen-delimited label (e.g., "C-N-H") into individual type strings. Validates that the number of components matches the expected count.

**Parameters**

- **ntypes** – Expected number of components
- **label** – Hyphen-delimited label string
- **types** – **[out]** Output vector to store component strings

**Returns**
0 on success, -1 if component count doesn't match ntypes

### Public Members

int **checkflag**

Flag to check for self-consistent type labels.

---

## 4.22 Memory pool classes

The memory pool classes are used for cases where otherwise many small memory allocations would be needed and where the data would be either all used or all freed. One example for that is the storage of neighbor lists. The memory management strategy is based on the assumption that allocations will be in chunks of similar sizes. The allocation is then not done per individual call for a reserved chunk of memory, but for a "page" that can hold multiple chunks of data. A parameter for the maximum chunk size must be provided, as that is used to determine whether a new page of memory must be used.

The *MyPage* class offers two ways to reserve a chunk: 1) with *MyPage::get()* the chunk size needs to be known in advance, 2) with *MyPage::vget()* a pointer to the next chunk is returned, but its size is registered later with *MyPage::vgot()*.

Listing 5: Example of using *MyPage*

```
#include "my_page.h"
using namespace LAMMPS_NS;

MyPage<double> *dpage = new MyPage<double>;
// max size of chunk: 256, size of page: 10240 doubles (=81920 bytes)
dpage->init(256,10240);

double **build_some_lists(int num)
{
    dpage->reset();
    double **dlist = new double*[num];
```

(continues on next page)

(continued from previous page)

```
    for (int i=0; i < num; ++i) {
        double *dptr = dpage.vget();
        int jnum = 0;
        for (int j=0; j < jmax; ++j) {
            // compute some dvalue for eligible loop index j
            dptr[j] = dvalue;
            ++jnum;
        }
        if (dpage.status() != 0) {
            // handle out of memory or jnum too large errors
        }
        dpage.vgot(jnum);
        dlist[i] = dptr;
    }
    return dlist;
}
```

template<class **T**>

class **MyPage**

Templated class for storing chunks of datums in pages.

The size of the chunk may vary from call to call, but must be less or equal than the *maxchunk* setting. The chunks are not returnable like with malloc() (i.e. you cannot call free() on them individually). One can only reset and start over. The purpose of this class is to replace many small memory allocations via malloc() with a few large ones. Since the pages are never freed until the class is re-initialized, they can be re-used without having to re-allocate them by calling the *reset()* method.

The settings *maxchunk*, *pagesize*, and *pagedelta* control the memory allocation strategy. The *maxchunk* value represents the largest number of items per chunk allowed; using more will trigger an error. If there is less space than *maxchunk* left on the current page, a new page is allocated for the next chunk. The *pagesize* value represents how many items can fit on a single page. It should have space for multiple chunks of size *maxchunk*. The combination of these two parameters determines how much memory is wasted by either switching to the next page too soon or allocating too large pages that never get fully used. The *pagedelta* parameter determines how many pages are allocated in one go. In combination with the *pagesize* setting, this determines how often blocks of memory get allocated (fewer allocations will result in faster execution).

**Note**

This is a template class with explicit instantiation. If the class is used with a new data type, a new explicit instantiation may need to be added at the end of the file `src/my_page.cpp` to avoid symbol lookup errors.

### Public Functions

**MyPage**()

Create a class instance

Need to call *init()* before use to define allocation settings

int **init**(int user_maxchunk = 1, int user_pagesize = 1024, int user_pagedelta = 1)

(Re-)initialize the set of pages and allocation parameters.

This also frees all previously allocated storage and allocates the first page(s).

**Parameters**

- **user_maxchunk** – Maximum allowed number of items for one chunk
- **user_pagesize** – Number of items on a single memory page
- **user_pagedelta** – Number of pages to allocate with one malloc

**Returns**
1 if there were invalid parameters, 2 if there was an allocation error or 0 if successful

*T* ***get**(int n = 1)

Pointer to location that can store N items.

This will allocate more pages as needed. If the parameter *N* is larger than the *maxchunk* setting, an error is flagged.

**Parameters**
**n** – number of items for which storage is requested

**Returns**
memory location or null pointer, if error or allocation failed

inline *T* ***vget**()

Get pointer to location that can store *maxchunk* items.

This will return the same pointer as the previous call to this function unless *vgot()* is called afterwards to record how many items of the chunk were actually used.

**Returns**
pointer to chunk of memory or null pointer if run out of memory

inline void **vgot**(int n)

Mark *N* items as used of the chunk reserved with a preceding call to *vget()*.

This will advance the internal pointer inside the current memory page. It is not necessary to call this function for *N* = 0, implying the reserved storage was not used. A following call to *vget()* will then reserve the same location again. It is an error if *N* > *maxchunk*.

**Parameters**
**n** – Number of items used in previously reserved chunk

void **reset**()

Reset state of memory pool without freeing any memory

inline double **size**() const

Return total size of allocated pages

**Returns**
total storage used in bytes

inline int **status**() const

Return error status

**Returns**
0 if no error, 1 requested chunk size > maxchunk, 2 if malloc failed

template<class **T**>

class **MyPoolChunk**

Templated class for storing chunks of datums in pages.

The size of the chunk may vary from call to call between the *minchunk* and *maxchunk* setting. Chunks may be returned to the pool for re-use. Chunks can be reserved in *nbin* different sizes between *minchunk* and *maxchunk*. The *chunksperpage* setting specifies how many chunks are stored on any page and the *pagedelta* setting determines how many pages are allocated in one go. Pages are never freed, so they can be re-used without re-allocation.

**Note**

This is a template class with explicit instantiation. If the class is used with a new data type, a new explicit instantiation may need to be added at the end of the file `src/my_pool_chunk.cpp` to avoid symbol lookup errors.

### Public Functions

**MyPoolChunk**(int user_minchunk = 1, int user_maxchunk = 1, int user_nbin = 1, int user_chunkperpage = 1024, int user_pagedelta = 1)

Create a class instance and set memory pool parameters

**Parameters**

- **user_minchunk** – Minimal chunk size
- **user_maxchunk** – Maximal chunk size
- **user_nbin** – Number of bins of different chunk sizes
- **user_chunkperpage** – Number of chunks per page
- **user_pagedelta** – Number of pages to allocate in one go

**~MyPoolChunk**()

Destroy class instance and free all allocated memory

*T* ***get**(int &index)

Return pointer/index of unused chunk of size maxchunk

**Parameters**
**index** – Index of chunk in memory pool

**Returns**
Pointer to requested chunk of storage

*T* ***get**(int n, int &index)

Return pointer/index of unused chunk of size N

**Parameters**

- **n** – Size of chunk
- **index** – Index of chunk in memory pool

**Returns**
Pointer to requested chunk of storage

void **put**(int index)

Put indexed chunk back into memory pool via free list

**Parameters**
**index** – Memory chunk index returned by call to *get()*

double **size**() const

Return total size of allocated pages

**Returns**
total storage used in bytes

inline int **status**() const

Return error status

**Returns**
0 if no error, 1 if invalid input, 2 if malloc() failed, 3 if chunk > maxchunk

---

## 4.23 Eigensolver functions

The MathEigen sub-namespace of the LAMMPS_NS namespace contains functions and classes for eigensolvers. Currently only the *jacobi3 function* is used in various places in LAMMPS. That function is built on top of a group of more generic eigensolvers that are maintained in the math_eigen_impl.h header file. This header contains the implementation of three template classes:

1. "Jacobi" calculates all of the eigenvalues and eigenvectors of a dense, symmetric, real matrix.
2. The "PEigenDense" class only calculates the principal eigenvalue (i.e. the largest or smallest eigenvalue), and its corresponding eigenvector. However it is much more efficient than "Jacobi" when applied to large matrices (larger than 13x13). PEigenDense also can understand complex-valued Hermitian matrices.
3. The "LambdaLanczos" class is a generalization of "PEigenDense" which can be applied to arbitrary sparse matrices.

The "math_eigen_impl.h" code is an amalgamation of jacobi_pd by Andrew Jewett at Scripps Research (under CC0-1.0 license) and Lambda Lanczos by Yuya Kurebayashi at Tohoku University (under MIT license)

---

int MathEigen::**jacobi3**(double const *const *mat, double *eval, double **evec, int sort = -1)

A specialized function which finds the eigenvalues and eigenvectors of a 3x3 matrix (in double ** format).

**Parameters**

- **mat** – the 3x3 matrix you wish to diagonalize
- **eval** – store the eigenvalues here
- **evec** – store the eigenvectors here…
- **sort** – order eigenvalues and -vectors (-1 decreasing (default), 1 increasing, 0 unsorted)

**Returns**
0 if eigenvalue calculation converged, 1 if it failed

int MathEigen::**jacobi3**(double const mat[3][3], double *eval, double evec[3][3], int sort = -1)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

---

## 4.24 Communication buffer coding with *ubuf*

LAMMPS uses communication buffers where it collects data from various class instances and then exchanges the data with neighboring subdomains. For simplicity those buffers are defined as `double` buffers and used for doubles and integer numbers. This presents a unique problem when 64-bit integers are used. While the storage needed for a `double` is also 64-bit, it cannot be used by a simple assignment. To get around that limitation, LAMMPS uses the *ubuf* union. It is used in the various "pack" and "unpack" functions in the LAMMPS classes to store and retrieve integers that may be 64-bit from the communication buffers.

---

union **ubuf**

*#include <lmptype.h>* Data structure for packing 32-bit and 64-bit integers into double (communication) buffers

Using this union avoids aliasing issues by having member types (double, int) referencing the same buffer memory location.

The explicit constructor for 32-bit integers prevents compilers from (incorrectly) calling the double constructor when storing an int into a double buffer.

**Usage:**

Listing 6: To copy an integer into a double buffer:

```
double buf[2];
int    foo =   1;
tagint bar = 2<<40;
buf[1] = ubuf(foo).d;
buf[2] = ubuf(bar).d;
```

Listing 7: To copy from a double buffer back to an int:

```
foo = (int)    ubuf(buf[1]).i;
bar = (tagint) ubuf(buf[2]).i;
```

The typecasts prevent compiler warnings about possible truncation issues.

### Public Functions

inline **ubuf**(const double &arg)

inline **ubuf**(const int64_t &arg)

inline **ubuf**(const int &arg)

### Public Members

double **d**

int64_t **i**

## 4.25 Internal Styles

LAMMPS has a number of styles that are not meant to be used in an input file and thus are not documented in the *LAMMPS command documentation*. The differentiation between user commands and internal commands is through

the case of the command name: user commands and styles are all lower case, internal styles are all upper case. Internal styles are not called from the input file, but their classes are instantiated by other styles. Often they are created by other styles to store internal data or to perform actions regularly at specific steps of the simulation.

The paragraphs below document some of those styles that have general utility and may be used to avoid redundant implementation.

### 4.25.1 DEPRECATED Styles

The styles called DEPRECATED (e.g. pair, bond, fix, compute, region, etc.) have the purpose to inform users that a specific style has been removed or renamed. This is achieved by creating an alias for the deprecated style to the corresponding class. For example, the fix style DEPRECATED is aliased to fix style ave/spatial and fix style ave/spatial/sphere with the following code:

```
FixStyle(DEPRECATED,FixDeprecated);
FixStyle(ave/spatial,FixDeprecated);
FixStyle(ave/spatial/sphere,FixDeprecated);
```

The individual class will then determine based on the style name what action to perform:

- inform that the style has been removed and what style replaces it, if any, and then error out
- inform that the style has been renamed and then either execute the replacement or error out
- inform that the style is no longer required, and it is thus ignored and continue

There is also a section in the user's guide for *removed commands and packages* with additional explanations.

### 4.25.2 Internal fix styles

These provide an implementation of features that would otherwise have been replicated across multiple styles. The used fix ID is generally derived from the compute or fix ID creating the fix with some string appended. When needed, the fix can be looked up with `Modify::get_fix_by_id()`, which returns a pointer to the fix instance. The data managed by the fix can be accessed just as for other fixes that can be used in input files.

#### fix DUMMY

Most fix classes cannot be instantiated before the simulation box has been created since they access data that is only available then. However, in some cases it is required that a fix must be at or close to the top of the list of all fixes. In those cases an instance of the DUMMY fix style may be created by calling `Modify::add_fix()` and then later replaced by the intended fix through calling `Modify::replace_fix()`.

#### fix OXDNA/LRF

Fix OXDNA/LRF calculates the local reference frame of ellipsoids from the quaternions. All oxDNA bond and pair styles require the Cartesian local reference frame during the force, torque and energy calculation. The fix is invoked internally at the pre-force stage and not to be used in an input script.

#### fix STORE/ATOM

Fix STORE/ATOM can be used as persistent storage of per-atom data.

**Syntax**

```
fix ID group-ID STORE/ATOM N1 N2 gflag rflag
```

- ID, group-ID are documented in *fix* command, group-ID is *ignored*
- STORE/ATOM = style name of this fix command

- N1 = 1, N2 = 0 : data is per-atom vector = single value per atom
- N1 > 1, N2 = 0 : data is per-atom array = N1 values per atom
- N1 > 0, N2 > 0 : data is per-atom tensor = N1xN2 values per atom
- gflag = 1 communicate per-atom values with ghost atoms, 0 do not update ghost atom data
- rflag = 1 store per-atom value in restart file, 0 do not store data in restart

Similar functionality is also available through using custom per-atom properties with *fix property/atom*. The choice between the two fixes should be based on whether the user should be able to access this per-atom data: if yes, then fix property/atom is preferred, otherwise fix STORE/ATOM.

### fix STORE/GLOBAL

Fix STORE/GLOBAL can be used as persistent storage of global data with support for restarts

**Syntax**

```
fix ID group-ID STORE/GLOBAL N1 N2
```

- ID, group-ID are documented in *fix* command, group-ID is *ignored*
- STORE/GLOBAL = style name of this fix command
- N1 >=1 : number of global items to store
- N2 = 1 : data is global vector of length N1
- N2 > 1 : data is global N1xN2 array

### fix STORE/LOCAL

Fix STORE/LOCAL can be used as persistent storage for local data

**Syntax**

```
fix ID group-ID STORE/LOCAL Nreset Nvalues
```

- ID, group-ID are documented in *fix* command
- STORE/LOCAL = style name of this fix command
- Nreset = frequency at which local data is available
- Nvalues = number of values per local item, that is the number of columns

## 4.26 Use of distributed grids within style classes

Added in version 22Dec2022.

The LAMMPS source code includes two classes which facilitate the creation and use of distributed grids. These are the Grid2d and Grid3d classes in the src/grid2d.cpp.h and src/grid3d.cpp.h files respectively. As the names imply, they are used for 2d or 3d simulations, as defined by the *dimension* command.

The *Howto_grid* page gives an overview of how distributed grids are defined from a user perspective, lists LAMMPS commands which use them, and explains how grid cell data is referenced from an input script. Please read that page first as it motivates the coding details discussed here.

This doc page is for users who wish to write new styles (input script commands) which use distributed grids. There are a variety of material models and analysis methods which use atoms (or coarse-grained particles) and grids in tandem.

A *distributed* grid means each processor owns a subset of the grid cells. In LAMMPS, the subset for each processor will be a sub-block of grid cells with low and high index bounds in each dimension of the grid. The union of the sub-blocks across all processors is the global grid.

More specifically, a grid point is defined for each cell (by default the center point), and a processor owns a grid cell if its point is within the processor's spatial subdomain. The union of processor subdomains is the global simulation box. If a grid point is on the boundary of two subdomains, the lower processor owns the grid cell. A processor may also store copies of ghost cells which surround its owned cells.

---

### 4.26.1 Style commands

Style commands which can define and use distributed grids include the *compute*, *fix*, *pair*, and *kspace* styles. If you wish grid cell data to persist across timesteps, then use a fix. If you wish grid cell data to be accessible by other commands, then use a fix or compute. Currently in LAMMPS, the *pair_style amoeba*, *kspace_style pppm*, and *kspace_style msm* commands use distributed grids but do not require either of these capabilities; they thus create and use distributed grids internally. Note that a pair style which needs grid cell data to persist could be coded to work in tandem with a fix style which provides that capability.

The *size* of a grid is specified by the number of grid cells in each dimension of the simulation domain. In any dimension the size can be any value >= 1. Thus a 10x10x1 grid for a 3d simulation is effectively a 2d grid, where each grid cell spans the entire z-dimension. A 1x100x1 grid for a 3d simulation is effectively a 1d grid, where grid cells are a series of thin xz slabs in the y-dimension. It is even possible to define a 1x1x1 3d grid, though it may be inefficient to use it in a computational sense.

Note that the choice of grid size is independent of the number of processors or their layout in a grid of processor subdomains which overlays the simulations domain. Depending on the distributed grid size, a single processor may own many 1000s or no grid cells.

A command can define multiple grids, each of a different size. Each grid is an instantiation of the Grid2d or Grid3d class.

The command also defines what data it will store for each grid it creates and it allocates the multidimensional array(s) needed to store the data. No grid cell data is stored within the Grid2d or Grid3d classes.

If a single value per grid cell is needed, the data array will have the same dimension as the grid, i.e. a 2d array for a 2d grid, likewise for 3d. If multiple values per grid cell are needed, the data array will have one more dimension than the grid, i.e. a 3d array for a 2d grid, or 4d array for a 3d grid. A command can choose to define multiple data arrays for each grid it defines.

---

### 4.26.2 Grid data allocation and access

The simplest way for a command to allocate and access grid cell data is to use the *create_offset()* methods provided by the Memory class. Arguments for these methods can be values returned by the *setup_grid()* method (described below), which define the extent of the grid cells (owned+ghost) the processor owns. These 4 methods allocate memory for 2d (first two) and 3d (second two) grid data. The two methods that end in "_offset" allocate an array which stores a single value per grid cell. The two that end in "_last" allocate an array which stores *Nvalues* per grid cell.

```
// single value per cell for a 2d grid = 2d array
memory->create2d_offset(data2d_one, nylo_out, nyhi_out,
                        nxlo_out, nxhi_out, "data2d_one");

// nvalues per cell for a 2d grid = 3d array
memory->create3d_offset_last(data2d_multi, nylo_out, nyhi_out,
```

(continues on next page)

(continued from previous page)

```
                              nxlo_out, nxhi_out, nvalues, "data2d_multi");

// single value per cell for a 3d grid = 3d array
memory->create3d_offset(data3d_one, nzlo_out, nzhi_out, nylo_out,
                        nyhi_out, nxlo_out, nxhi_out, "data3d_one");

// nvalues per cell for a 3d grid = 4d array
memory->create4d_offset_last(data3d_multi, nzlo_out, nzhi_out, nylo_out,
                             nyhi_out, nxlo_out, nxhi_out, nvalues,
                             "data3d_multi");
```

Note that these multidimensional arrays are allocated as contiguous chunks of memory where the x-index of the grid varies fastest, then y, and the z-index slowest. For multiple values per grid cell, the Nvalues are contiguous, so their index varies even faster than the x-index.

The key point is that the "offset" methods create arrays which are indexed by the range of indices which are the bounds of the sub-block of the global grid owned by this processor. This means loops like these can be written in the caller code to loop over owned grid cells, where the "i" loop bounds are the range of owned grid cells for the processor. These are the bounds returned by the *setup_grid()* method:

```
for (int iy = iylo; iy <= iyhi; iy++)
  for (int ix = ixlo; ix <= ixhi; ix++)
    data2d_one[iy][ix] = 0.0;

for (int iy = iylo; iy <= iyhi; iy++)
  for (int ix = ixlo; ix <= ixhi; ix++)
    for (int m = 0; m < nvalues; m++)
      data2d_multi[iy][ix][m] = 0.0;

for (int iz = izlo; iz <= izhi; iz++)
  for (int iy = iylo; iy <= iyhi; iy++)
    for (int ix = ixlo; ix <= ixhi; ix++)
      data3d_one[iz][iy][ix] = 0.0;

for (int iz = izlo; iz <= izhi; iz++)
  for (int iy = iylo; iy <= iyhi; iy++)
    for (int ix = ixlo; ix <= ixhi; ix++)
       for (int m = 0; m < nvalues; m++)
          data3d_multi[iz][iy][ix][m] = 0.0;
```

Simply replacing the "i" bounds with "o" bounds, also returned by the *setup_grid()* method, would alter this code to loop over owned+ghost cells (the entire allocated grid).

### 4.26.3 Grid class constructors

The following subsections describe the public methods of the Grid3d class which a style command can invoke. The Grid2d methods are similar; simply remove arguments which refer to the z-dimension.

There are 2 constructors which can be used. They differ in the extra i/o xyz lo/hi arguments:

```
Grid3d(class LAMMPS *lmp, MPI_Comm gcomm, int gnx, int gny, int gnz)
Grid3d(class LAMMPS *lmp, MPI_Comm gcomm, int gnx, int gny, int gnz,
```

(continues on next page)

(continued from previous page)

```
      int ixlo, int ixhi, int iylo, int iyhi, int izlo, int izhi,
      int oxlo, int oxhi, int oylo, int oyhi, int ozlo, int ozhi)
```

Both constructors take the LAMMPS instance pointer and a communicator over which the grid will be distributed. Typically this is the *world* communicator the LAMMPS instance is using. The *kspace_style msm* command creates a series of grids, each of different size, which are partitioned across different sub-communicators of processors. Both constructors are also passed the global grid size: *gnx* by *gny* by *gnz*.

The first constructor is used when the caller wants the Grid class to partition the global grid across processors; the Grid class defines which grid cells each processor owns and also which it stores as ghost cells. A subsequent call to *setup_grid()*, discussed below, returns this info to the caller.

The second constructor allows the caller to define the extent of owned and ghost cells, and pass them to the Grid class. The 6 arguments which start with "i" are the inclusive lower and upper index bounds of the owned (inner) grid cells this processor owns in each of the 3 dimensions within the global grid. Owned grid cells are indexed from 0 to N-1 in each dimension.

The 6 arguments which start with "o" are the inclusive bounds of the owned+ghost (outer) grid cells it stores. If the ghost cells are on the other side of a periodic boundary, then these indices may be < 0 or >= N in any dimension, so that oxlo <= ixlo and ixhi >= ixhi is always the case.

For example, if Nx = 100, then a processor might pass ixlo=50, ixhi=60, oxlo=48, oxhi=62 to the Grid class. Or ixlo=0, ixhi=10, oxlo=-2, oxhi=13. If a processor owns no grid cells in a dimension, then the ihi value should be specified as one less than the ilo value.

Note that the only reason to use the second constructor is if the logic for assigning ghost cells is too complex for the Grid class to compute, using the various set() methods described next. Currently only the kspace_style pppm/electrode and kspace_style msm commands use the second constructor.

---

### 4.26.4 Grid class set methods

The following methods affect how the Grid class computes which owned and ghost cells are assigned to each processor. *Set_shift_grid()* is the only method which influences owned cell assignment; all the rest influence ghost cell assignment. These methods are only used with the first constructor; they are ignored if the second constructor is used. These methods must be called before the *setup_grid()* method is invoked, because they influence its operation.

```
void set_shift_grid(double shift);
void set_distance(double distance);
void set_stencil_atom(int lo, int hi);
void set_shift_atom(double shift_lo, double shift_hi);
void set_stencil_grid(int lo, int hi);
void set_zfactor(double factor);
```

Processors own a grid cell if a point within the grid cell is inside the processor's subdomain. By default this is the center point of the grid cell. The *set_shift_grid()* method can change this. The *shift* argument is a value from 0.0 to 1.0 (inclusive) which is the offset of the point within the grid cell in each dimension. The default is 0.5 for the center of the cell. A value of 0.0 is the lower left corner point; a value of 1.0 is the upper right corner point. There is typically no need to change the default as it is optimal for minimizing the number of ghost cells needed.

If a processor maps its particles to grid cells, it needs to allow for its particles being outside its subdomain between reneighboring. The *distance* argument of the *set_distance()* method sets the furthest distance outside a processor's subdomain which a particle can move. Typically this is half the neighbor skin distance, assuming reneighboring is done appropriately. This distance is used in determining how many ghost cells a processor needs to store to enable its particles to be mapped to grid cells. The default value is 0.0.

Some commands, like the *kspace_style pppm* command, map values (charge in the case of PPPM) to a stencil of grid cells beyond the grid cell the particle is in. The stencil extent may be different in the low and high directions. The *set_stencil_atom()* method defines the maximum values of those 2 extents, assumed to be the same in each of the 3 dimensions. Both the lo and hi values are specified as positive integers. The default values are both 0.

Some commands, like the *kspace_style pppm* command, shift the position of an atom when mapping it to a grid cell, based on the size of the stencil used to map values to the grid (charge in the case of PPPM). The lo and hi arguments of the *set_shift_atom()* method are the minimum shift in the low direction and the maximum shift in the high direction, assumed to be the same in each of the 3 dimensions. The shifts should be fractions of a grid cell size with values between 0.0 and 1.0 inclusive. The default values are both 0.0. See the src/pppm.cpp file for examples of these lo/hi values for regular and staggered grids.

Some methods like the *fix ttm/grid* command, perform finite difference kinds of operations on the grid, to diffuse electron heat in the case of the two-temperature model (TTM). This operation uses ghost grid values beyond the owned grid values the processor updates. The *set_stencil_grid()* method defines the extent of this stencil in both directions, assumed to be the same in each of the 3 dimensions. Both the lo and hi values are specified as positive integers. The default values are both 0.

The kspace_style pppm commands allow a grid to be defined which overlays a volume which extends beyond the simulation box in the z dimension. This is for the purpose of modeling a 2d-periodic slab (non-periodic in z) as if it were a larger 3d periodic system, extended (with empty space) in the z dimension. The *kspace_modify slab* command is used to specify the ratio of the larger volume to the simulation volume; a volume ratio of ~3 is typical. For this kind of model, the PPPM caller sets the global grid size *gnz* ~3x larger than it would be otherwise. This same ratio is passed by the PPPM caller as the *factor* argument to the Grid class via the *set_zfactor()* method (*set_yfactor()* for 2d grids). The Grid class will then assign ownership of the 1/3 of grid cells that overlay the simulation box to the processors which also overlay the simulation box. The remaining 2/3 of the grid cells are assigned to processors whose subdomains are adjacent to the upper z boundary of the simulation box.

---

### 4.26.5 Grid class setup_grid method

The *setup_grid()* method is called after the first constructor (above) to partition the grid across processors, which determines which grid cells each processor owns. It also calculates how many ghost grid cells in each dimension and each direction each processor needs to store.

Note that this method is NOT called if the second constructor above is used. In that case, the caller assigns owned and ghost cells to each processor.

Also note that this method must be invoked after any *set_*()* methods have been used, since they can influence the assignment of owned and ghost cells.

```
void setup_grid(int &ixlo, int &ixhi, int &iylo, int &iyhi, int &izlo, int &izhi,
                int &oxlo, int &oxhi, int &oylo, int &oyhi, int &ozlo, int &ozhi)
```

The 6 return arguments which start with "i" are the inclusive lower and upper index bounds of the owned (inner) grid cells this processor owns in each of the 3 dimensions within the global grid. Owned grid cells are indexed from 0 to N-1 in each dimension.

The 6 return arguments which start with "o" are the inclusive bounds of the owned+ghost cells it owns. If the ghost cells are on the other side of a periodic boundary, then these indices may be < 0 or >= N in any dimension, so that oxlo <= ixlo and ixhi >= ixhi is always the case.

---

### 4.26.6 More grid class set methods

The following 2 methods can be used to override settings made by the constructors above. If used, they must be called called before the *setup_comm()* method is invoked, since it uses the settings that these methods override. In LAMMPS these methods are called by by the *kspace_style msm* command for the grids it instantiates using the 2nd constructor above.

```
void set_proc_neighs(int pxlo, int pxhi, int pylo, int pyhi, int pzlo, int pzhi)
void set_caller_grid(int fxlo, int fxhi, int fylo, int fyhi, int fzlo, int fzhi)
```

The *set_proc_neighs()* method sets the processor IDs of the 6 neighboring processors for each processor. Normally these would match the processor grid neighbors which LAMMPS creates to overlay the simulation box (the default). However, MSM excludes non-participating processors from coarse grid communication when less processors are used. This method allows MSM to override the default values.

The *set_caller_grid()* method species the size of the data arrays the caller allocates. Normally these would match the extent of the ghost grid cells (the default). However the MSM caller allocates a larger data array (more ghost cells) for its finest-level grid, for use in other operations besides owned/ghost cell communication. This method allows MSM to override the default values.

---

### 4.26.7 Grid class get methods

The following methods allow the caller to query the settings for a specific grid, whether it created the grid or another command created it.

```
void get_size(int &nxgrid, int &nygrid, int &nzgrid);
void get_bounds_owned(int &xlo, int &xhi, int &ylo, int &yhi, int &zlo, int &zhi)
void get_bounds_ghost(int &xlo, int &xhi, int &ylo, int &yhi, int &zlo, int &zhi)
```

The *get_size()* method returns the size of the global grid in each dimension.

The *get_bounds_owned()* method return the inclusive index bounds of the grid cells this processor owns. The values range from 0 to N-1 in each dimension. These values are the same as the "i" values returned by *setup_grid()*.

The *get_bounds_ghost()* method return the inclusive index bounds of the owned+ghost grid cells this processor stores. The owned cell indices range from 0 to N-1, so these indices may be less than 0 or greater than or equal to N in each dimension. These values are the same as the "o" values returned by *setup_grid()*.

---

### 4.26.8 Grid class owned/ghost communication

If needed by the command, the following methods setup and perform communication of grid data to/from neighboring processors. The *forward_comm()* method sends owned grid cell data to the corresponding ghost grid cells on other processors. The *reverse_comm()* method sends ghost grid cell data to the corresponding owned grid cells on another processor. The caller can choose to sum ghost grid cell data to the owned grid cell or simply copy it.

```
void setup_comm(int &nbuf1, int &nbuf2)
void forward_comm(int caller, void *ptr, int which, int nper, int nbyte,
                  void *buf1, void *buf2, MPI_Datatype datatype);
void reverse_comm(int caller, void *ptr, int which, int nper, int nbyte,
                  void *buf1, void *buf2, MPI_Datatype datatype)
int ghost_adjacent();
```

The *setup_comm()* method must be called one time before performing *forward* or *reverse* communication (multiple times if needed). It returns two integers, which should be used to allocate two buffers. The *nbuf1* and *nbuf2* values are the number of grid cells whose data will be stored in two buffers by the Grid class when *forward* or *reverse* communication is performed. The caller should thus allocate them to a size large enough to hold all the data used in any single forward or reverse communication operation it performs. Note that the caller may allocate and communicate multiple data arrays for a grid it instantiates. This size includes the bytes needed for the data type of the grid data it stores, e.g. double precision values.

The *forward_comm()* and *reverse_comm()* methods send grid cell data from owned to ghost cells, or ghost to owned cells, respectively, as described above. The *caller* argument should be one of these values – Grid3d::COMPUTE, Grid3d::FIX, Grid3d::KSPACE, Grid3d::PAIR – depending on the style of the caller class. The *ptr* argument is the "this" pointer to the caller class. These two arguments are used to call back to pack()/unpack() functions in the caller class, as explained below.

The *which* argument is a flag the caller can set which is passed to the caller's pack()/unpack() methods. This allows a single callback method to pack/unpack data for several different flavors of forward/reverse communication, e.g. operating on different grids or grid data.

The *nper* argument is the number of values per grid cell to be communicated. The *nbyte* argument is the number of bytes per value, e.g. 8 for double-precision values. The *buf1* and *buf2* arguments are the two allocated buffers described above. So long as they are allocated for the maximum size communication, they can be re-used for any *forward_comm()/reverse_comm()* call. The *datatype* argument is the MPI_Datatype setting, which should match the buffer allocation and the *nbyte* argument. E.g. MPI_DOUBLE for buffers storing double precision values.

To use the *forward_grid()* method, the caller must provide two callback functions; likewise for use of the *reverse_grid()* methods. These are the 4 functions, their arguments are all the same.

```
void pack_forward_grid(int which, void *vbuf, int nlist, int *list);
void unpack_forward_grid(int which, void *vbuf, int nlist, int *list);
void pack_reverse_grid(int which, void *vbuf, int nlist, int *list);
void unpack_reverse_grid(int which, void *vbuf, int nlist, int *list);
```

The *which* argument is set to the *which* value of the *forward_comm()* or *reverse_comm()* calls. It allows the pack/unpack function to select what data values to pack/unpack. *Vbuf* is the buffer to pack/unpack the data to/from. It is a void pointer so that the caller can cast it to whatever data type it chooses, e.g. double precision values. *Nlist* is the number of grid cells to pack/unpack and *list* is a vector (nlist in length) of offsets to where the data for each grid cell resides in the caller's data arrays, which is best illustrated with an example from the src/EXTRA-FIX/fix_ttm_grid.cpp class which stores the scalar electron temperature for 3d system in a 3d grid (one value per grid cell):

```
void FixTTMGrid::pack_forward_grid(int /*which*/, void *vbuf, int nlist, int *list)
{
  auto buf = (double *) vbuf;
  double *src = &T_electron[nzlo_out][nylo_out][nxlo_out];
  for (int i = 0; i < nlist; i++) buf[i] = src[list[i]];
}
```

In this case, the *which* argument is not used, *vbuf* points to a buffer of doubles, and the electron temperature is stored by the FixTTMGrid class in a 3d array of owned+ghost cells called T_electron. That array is allocated by the *memory->create_3d_offset()* method described above so that the first grid cell it stores is indexed as T_electron[nzlo_out][nylo_out][nxlo_out]. The *nlist* values in *list* are integer offsets from that first grid cell. Setting *src* to the address of the first cell allows those offsets to be used to access the temperatures to pack into the buffer.

Here is a similar portion of code from the src/fix_ave_grid.cpp class which can store two kinds of data, a scalar count of atoms in a grid cell, and one or more grid-cell-averaged atom properties. The code from its *unpack_reverse_grid()* function for 2d grids and multiple per-atom properties per grid cell (*nvalues*) is shown here:

```
void FixAveGrid::unpack_reverse_grid(int /*which*/, void *vbuf, int nlist, int *list)
{
  auto buf = (double *) vbuf;
  double *count,*data,*values;
  count = &count2d[nylo_out][nxlo_out];
  data = &array2d[nylo_out][nxlo_out][0];
  m = 0;
  for (i = 0; i < nlist; i++) {
    count[list[i]] += buf[m++];
    values = &data[nvalues*list[i]];
    for (j = 0; j < nvalues; j++)
     values[j] += buf[m++];
  }
}
```

Both the count and the multiple values per grid cell are communicated in *vbuf*. Note that *data* is now a pointer to the first value in the first grid cell. And *values* points to where the first value in *data* is for an offset of grid cells, calculated by multiplying *nvalues* by *list[i]*. Finally, because this is reverse communication, the communicated buffer values are summed to the caller values.

The *ghost_adjacent()* method returns a 1 if every processor can perform the necessary owned/ghost communication with only its nearest neighbor processors (4 in 2d, 6 in 3d). It returns a 0 if any processor's ghost cells extend further than nearest neighbor processors.

This can be checked by callers who have the option to change the global grid size to ensure more efficient nearest-neighbor-only communication if they wish. In this case, they instantiate a grid of a given size (resolution), then invoke *setup_comm()* followed by *ghost_adjacent()*. If the ghost cells are not adjacent, they destroy the grid instance and start over with a higher-resolution grid. Several of the *kspace_style pppm* command variants have this option.

---

### 4.26.9 Grid class remap methods for load balancing

The following methods are used when a load-balancing operation, triggered by the *balance* or *fix balance* commands, changes the partitioning of the simulation domain into processor subdomains.

In order to work with load-balancing, any style command (compute, fix, pair, or kspace style) which allocates a grid and stores per-grid data should define a *reset_grid()* method; it takes no arguments. It will be called by the two balance commands after they have reset processor subdomains and migrated atoms (particles) to new owning processors. The *reset_grid()* method will typically perform some or all of the following operations. See the src/fix_ave_grid.cpp and src/EXTRA_FIX/fix_ttm_grid.cpp files for examples of *reset_grid()* methods, as well as the *pack_remap_grid()* and *unpack_remap_grid()* functions.

First, the *reset_grid()* method can instantiate new grid(s) of the same global size, then call *setup_grid()* to partition them via the new processor subdomains. At this point, it can invoke the *identical()* method which compares the owned and ghost grid cell index bounds between two grids, the old grid passed as a pointer argument, and the new grid whose *identical()* method is being called. It returns 1 if the indices match on all processors, otherwise 0. If they all match, then the new grids can be deleted; the command can continue to use the old grids.

If not, then the command should allocate new grid data array(s) which depend on the new partitioning. If the command does not need to persist its grid data from the old partitioning to the new one, then the command can simply delete the old data array(s) and grid instance(s). It can then return.

If the grid data does need to persist, then the data for each grid needs to be "remapped" from the old grid partitioning to the new grid partitioning. The *setup_remap()* and *remap()* methods are used for that purpose.

```
int identical(Grid3d *old);
void setup_remap(Grid3d *old, int &nremap_buf1, int &nremap_buf2)
void remap(int caller, void *ptr, int which, int nper, int nbyte,
           void *buf1, void *buf2, MPI_Datatype datatype)
```

The arguments to these methods are identical to those for the *setup_comm()* and *forward_comm()* or *reverse_comm()* methods. However the returned *nremap_buf1* and *nremap2_buf* values will be different than the *nbuf1* and *nbuf2* values. They should be used to allocate two different remap buffers, separate from the owned/ghost communication buffers.

To use the *remap()* method, the caller must provide two callback functions:

```
void pack_remap_grid(int which, void *vbuf, int nlist, int *list);
void unpack_remap_grid(int which, void *vbuf, int list, int *list);
```

Their arguments are identical to those for the *pack_forward_grid()* and *unpack_forward_grid()* callback functions (or the reverse variants) discussed above. Normally, both these methods pack/unpack all the data arrays for a given grid. The *which* argument of the *remap()* method sets the *which* value for the pack/unpack functions. If the command instantiates multiple grids (of different sizes), it can be used within the pack/unpack methods to select which grid's data is being remapped.

Note that the *pack_remap_grid()* function must copy values from the OLD grid data arrays into the *vbuf* buffer. The *unpack_remap_grid()* function must copy values from the *vbuf* buffer into the NEW grid data arrays.

After the remap operation for grid cell data has been performed, the *reset_grid()* method can deallocate the two remap buffers it created, and can then exit.

---

### 4.26.10 Grid class I/O methods

There are two I/O methods in the Grid classes which can be used to read and write grid cell data to files. The caller can decide on the precise format of each file, e.g. whether header lines are prepended or comment lines are allowed. Fundamentally, the file should contain one line per grid cell for the entire global grid. Each line should contain identifying info as to which grid cell it is, e.g. a unique grid cell ID or the ix,iy,iz indices of the cell within a 3d grid. The line should also contain one or more data values which are stored within the grid data arrays created by the command

For grid cell IDs, the LAMMPS convention is that the IDs run from 1 to N, where N = Nx * Ny for 2d grids and N = Nx * Ny * Nz for 3d grids. The x-index of the grid cell varies fastest, then y, and the z-index varies slowest. So for a 10x10x10 grid the cell IDs from 901-1000 would be in the top xy layer of the z dimension.

The *read_file()* method does something simple. It reads a chunk of consecutive lines from the file and passes them back to the caller to process. The caller provides a *unpack_read_grid()* function for this purpose. The function checks the grid cell ID or indices and only stores grid cell data for the grid cells it owns.

The *write_file()* method does something slightly more complex. Each processor packs the data for its owned grid cells into a buffer. The caller provides a *pack_write_grid()* function for this purpose. The *write_file()* method then loops over all processors and each sends its buffer one at a time to processor 0, along with the 3d (or 2d) index bounds of its grid cell data within the global grid. Processor 0 calls back to the *unpack_write_grid()* function provided by the caller with the buffer. The function writes one line per grid cell to the file.

See the src/EXTRA_FIX/fix_ttm_grid.cpp file for examples of now both these methods are used to read/write electron temperature values from/to a file, as well as for implementations of the the pack/unpack functions described below.

Here are the details of the two I/O methods and the 3 callback functions. See the src/fix_ave_grid.cpp file for examples of all of them.

```
void read_file(int caller, void *ptr, FILE *fp, int nchunk, int maxline)
void write_file(int caller, void *ptr, int which,
                int nper, int nbyte, MPI_Datatype datatype
```

The *caller* argument in both methods should be one of these values – Grid3d::COMPUTE, Grid3d::FIX, Grid3d::KSPACE, Grid3d::PAIR – depending on the style of the caller class. The *ptr* argument in both methods is the "this" pointer to the caller class. These 2 arguments are used to call back to pack()/unpack() functions in the caller class, as explained below.

For the *read_file()* method, the *fp* argument is a file pointer to the file to be read from, opened on processor 0 by the caller. *Nchunk* is the number of lines to read per chunk, and *maxline* is the maximum number of characters per line. The Grid class will allocate a buffer for storing chunks of lines based on these values.

For the *write_file()* method, the *which* argument is a flag the caller can set which is passed back to the caller's pack()/unpack() methods. If the command instantiates multiple grids (of different sizes), this flag can be used within the pack/unpack methods to select which grid's data is being written out (presumably to different files). the *nper* argument is the number of values per grid cell to be written out. The *nbyte* argument is the number of bytes per value, e.g. 8 for double-precision values. The *datatype* argument is the MPI_Datatype setting, which should match the *nbyte* argument. E.g. MPI_DOUBLE for double precision values.

To use the *read_grid()* method, the caller must provide one callback function. To use the *write_grid()* method, it provides two callback functions:

```
int unpack_read_grid(int nlines, char *buffer)
void pack_write_grid(int which, void *vbuf)
void unpack_write_grid(int which, void *vbuf, int *bounds)
```

For *unpack_read_grid()* the *nlines* argument is the number of lines of character data read from the file and contained in *buffer*. The lines each include a newline character at the end. When the function processes the lines, it may choose to skip some of them (header or comment lines). It returns an integer count of the number of grid cell lines it processed. This enables the Grid class *read_file()* method to know when it has read the correct number of lines.

For *pack_write_grid()* and *unpack_write_grid()*, the *vbuf* argument is the buffer to pack/unpack data to/from. It is a void pointer so that the caller can cast it to whatever data type it chooses, e.g. double precision values. the *which* argument is set to the *which* value of the *write_file()* method. It allows the caller to choose which grid data to operate on.

For *unpack_write_grid()*, the *bounds* argument is a vector of 4 or 6 integer grid indices (4 for 2d, 6 for 3d). They are the xlo,xhi,ylo,yhi,zlo,zhi index bounds of the portion of the global grid which the *vbuf* holds owned grid cell data values for. The caller should loop over the values in *vbuf* with a double loop (2d) or triple loop (3d), similar to the code snippets listed above. The x-index varies fastest, then y, and the z-index slowest. If there are multiple values per grid cell, the index for those values varies fastest of all. The caller can add the x,y,z indices of the grid cell (or the corresponding grid cell ID) to the data value(s) written as one line to the output file.

---

### 4.26.11 Style class grid access methods

A style command can enable its grid cell data to be accessible from other commands. For example *fix ave/grid* or *dump grid* or *dump grid/vtk*. Those commands access the grid cell data by using a *grid reference* in their input script syntax, as described on the *Howto_grid* doc page. They look like this:

- c_ID:gname:dname
- c_ID:gname:dname[I]
- f_ID:gname:dname
- f_ID:gname:dname[I]

Each grid command instantiates has a unique *gname*, defined by the command. Likewise each grid cell data structure (scalar or vector) associated with a grid has a unique *dname*, also defined by the command.

To provide access to its grid cell data, a style command needs to implement the following 4 methods:

```
int get_grid_by_name(const std::string &name, int &dim);
void *get_grid_by_index(int index);
int get_griddata_by_name(int igrid, const std::string &name, int &ncol);
void *get_griddata_by_index(int index);
```

Currently only computes and fixes can implement these methods. If it does so, the compute of fix should also set the variable *pergrid_flag* to 1. See any of the compute or fix commands which set "pergrid_flag = 1" for examples of how these 4 functions can be implemented.

The *get_grid_by_name()* method takes a grid name as input and returns two values. The *dim* argument is returned as 2 or 3 for the dimensionality of the grid. The function return is a grid index from 0 to G-1 where *G* is the number of grids the command instantiates. A value of -1 is returned if the grid name is not recognized.

The *get_grid_by_index()* method is called after the *get_grid_by_name()* method, using the grid index it returned as its argument. This method will return a pointer to the Grid2d or Grid3d class. The caller can use this to query grid attributes, such as the global size of the grid, to ensure it is of the expected size.

The *get_griddata_by_name()* method takes a grid index *igrid* and a data name as input. It returns two values. The *ncol* argument is returned as a 0 if the grid data is a single value (scalar) per grid cell, or an integer M > 0 if there are M values (vector) per grid cell. Note that even if M = 1, it is still a 1-length vector, not a scalar. The function return is a data index from 0 to D-1 where *D* is the number of data sets associated with that grid by the command. A value of -1 is returned if the data name is not recognized.

The *get_griddata_by_index()* method is called after the *get_griddata_by_name()* method, using the data index it returned as its argument. This method will return a pointer to the multidimensional array which stores the requested data.

As in the discussion above of the Memory class *create_offset()* methods, the dimensionality of the array associated with the returned pointer depends on whether it is a 2d or 3d grid and whether there is a single or multiple values stored for each grid cell:

- single value per cell for a 2d grid = 2d array pointer
- multiple values per cell for a 2d grid = 3d array pointer
- single value per cell for a 3d grid = 3d array pointer
- multiple values per cell for a 3d grid = 4d array pointer

The caller will typically access the data by casting the void pointer to the corresponding array pointer and using nested loops in x,y,z between owned or ghost index bounds returned by the *get_bounds_owned()* or *get_bounds_ghost()* methods to index into the array. Example code snippets with this logic were listed above,

---

### 4.26.12 Final notes

Finally, here are some additional issues to pay attention to for writing any style command which uses distributed grids via the Grid2d or Grid3d class.

The command destructor should delete all instances of the Grid class, any buffers it allocated for forward/reverse or remap communication, and any data arrays it allocated to store grid cell data.

If a command is intended to work for either 2d or 3d simulations, then it should have logic to instantiate either 2d or 3d grids and their associated data arrays, depending on the dimension of the simulation box. The *fix ave/grid* command is an example of such a command.

When a command maps its particles to the grid and updates grid cell values, it should check that it is not updating or accessing a grid cell value outside the range of its owned+ghost cells, and generate an error message if that is the case. This could happen, for example, if a particle has moved further than half the neighbor skin distance, because the neighbor list update criterion are not adequate to prevent it from happening. See the src/KSPACE/pppm.cpp file and its *particle_map()* method for an example of this kind of error check.

# Part III

# Command Reference

CHAPTER

ONE

# COMMANDS

## 1.1 angle_coeff command

### 1.1.1 Syntax

```
angle_coeff N args
```

- N = numeric angle type (see asterisk form below), or type label
- args = coefficients for one or more angle types

### 1.1.2 Examples

```
angle_coeff 1 300.0 107.0
angle_coeff * 5.0
angle_coeff 2*10 5.0

labelmap angle 1 hydroxyl
angle_coeff hydroxyl 300.0 107.0
```

### 1.1.3 Description

Specify the angle force field coefficients for one or more angle types. The number and meaning of the coefficients depends on the angle style. Angle coefficients can also be set in the data file read by the *read_data* command or in a restart file.

*N* can be specified in one of two ways. An explicit numeric value can be used, as in the first example above. Or *N* can be a type label, which is an alphanumeric string defined by the *labelmap* command or in a section of a data file read by the *read_data* command.

For numeric values only, a wild-card asterisk can be used to set the coefficients for multiple angle types. This takes the form "*" or "*n" or "n*" or "m*n". If *N* is the number of angle types, then an asterisk with no numeric values means all types from 1 to *N*. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to *N* (inclusive). A middle asterisk means all types from m to n (inclusive).

Note that using an *angle_coeff* command can override a previous setting for the same angle type. For example, these commands set the coeffs for all angle types, then overwrite the coeffs for just angle type 2:

```
angle_coeff * 200.0 107.0 1.2
angle_coeff 2 50.0 107.0
```

A line in a data file that specifies angle coefficients uses the exact same format as the arguments of the *angle_coeff* command in an input script, except that wild-card asterisks should not be used since coefficients for all *N* types must

be listed in the file. For example, under the "Angle Coeffs" section of a data file, the line that corresponds to the first example above would be listed as

```
1 300.0 107.0
```

The *angle_style class2* is an exception to this rule, in that an additional argument is used in the input script to allow specification of the cross-term coefficients. See its doc page for details.

---

The list of all angle styles defined in LAMMPS is given on the *angle_style* doc page. They are also listed in more compact form on the *Commands angle* doc page.

On either of those pages, click on the style to display the formula it computes and its coefficients as specified by the associated *angle_coeff* command.

---

### 1.1.4 Restrictions

This command must come after the simulation box is defined by a *read_data*, *read_restart*, or *create_box* command.

An angle style must be defined before any angle coefficients are set, either in the input script or in a data file.

### 1.1.5 Related commands

*angle_style*

### 1.1.6 Default

none

## 1.2 angle_style command

### 1.2.1 Syntax

```
angle_style style
```

- style = *none* or *zero* or *hybrid* or *amoeba* or *charmm* or *class2* or *class2xe* or *class2/p6* or *cosine* or *cosine/buck6d* or *cosine/delta* or *cosine/periodic* or *cosine/shift* or *cosine/shift/exp* or *cosine/squared* or *cosine/squared/restricted* or *cross* or *dipole* or *fourier* or *fourier/simple* or *gaussian* or *harmonic* or *lepton* or *mm3* or *quartic* or *spica* or *table*

### 1.2.2 Examples

```
angle_style harmonic
angle_style charmm
angle_style hybrid harmonic cosine
```

### 1.2.3 Description

Set the formula(s) LAMMPS uses to compute angle interactions between triplets of atoms, which remain in force for the duration of the simulation. The list of angle triplets is read in by a *read_data* or *read_restart* command from a data or restart file.

Hybrid models where angles are computed using different angle potentials can be setup using the *hybrid* angle style.

The coefficients associated with a angle style can be specified in a data or restart file or via the *angle_coeff* command.

All angle potentials store their coefficient data in binary restart files which means angle_style and *angle_coeff* commands do not need to be re-specified in an input script that restarts a simulation. See the *read_restart* command for details on how to do this. The one exception is that angle_style *hybrid* only stores the list of sub-styles in the restart file; angle coefficients need to be re-specified.

**Note**

When both an angle and pair style is defined, the *special_bonds* command often needs to be used to turn off (or weight) the pairwise interaction that would otherwise exist between 3 bonded atoms.

In the formulas listed for each angle style, *theta* is the angle between the three atoms in the angle.

---

Here is an alphabetic list of angle styles defined in LAMMPS. Click on the style to display the formula it computes and coefficients specified by the associated *angle_coeff* command.

Click on the style to display the formula it computes, any additional arguments specified in the angle_style command, and coefficients specified by the associated *angle_coeff* command.

There are also additional accelerated pair styles included in the LAMMPS distribution for faster performance on CPUs, GPUs, and KNLs. The individual style names on the *Commands angle* page are followed by one or more of (g,i,k,o,t) to indicate which accelerated styles exist.

- *none* - turn off angle interactions
- *zero* - topology but no interactions
- *hybrid* - define multiple styles of angle interactions
- *amoeba* - AMOEBA angle
- *charmm* - CHARMM angle
- *class2* - COMPASS (class 2) angle
- *class2xe* - ClassII-xe (class 2) angle
- *class2/p6* - COMPASS (class 2) angle expanded to 6th order
- *cosine* - angle with cosine term
- *cosine/buck6d* - same as cosine with Buckingham term between 1-3 atoms
- *cosine/delta* - angle with difference of cosines
- *cosine/periodic* - DREIDING angle
- *cosine/shift* - angle cosine with a shift
- *cosine/shift/exp* - cosine with shift and exponential term in spring constant
- *cosine/squared* - angle with cosine squared term
- *cosine/squared/restricted* - angle with restricted cosine squared term
- *cross* - cross term coupling angle and bond lengths
- *dipole* - angle that controls orientation of a point dipole
- *fourier* - angle with multiple cosine terms
- *fourier/simple* - angle with a single cosine term

- *gaussian* - multi-centered Gaussian-based angle potential
- *harmonic* - harmonic angle
- *lepton* - angle potential from evaluating a string
- *mesocnt* - piecewise harmonic and linear angle for bending-buckling of nanotubes
- *mm3* - anharmonic angle
- *mwlc* - meltable wormlike chain
- *quartic* - angle with cubic and quartic terms
- *spica* - harmonic angle with repulsive SPICA pair style between 1-3 atoms
- *table* - tabulated by angle

---

### 1.2.4 Restrictions

Angle styles can only be set for atom_styles that allow angles to be defined.

Most angle styles are part of the MOLECULE package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info. The doc pages for individual bond potentials tell if it is part of a package.

### 1.2.5 Related commands

*angle_coeff*

### 1.2.6 Default

```
angle_style none
```

## 1.3 angle_write command

### 1.3.1 Syntax

```
angle_write atype N file keyword
```

- atype = angle type
- N = # of values
- file = name of file to write values to
- keyword = section name in file for this set of tabulated values

### 1.3.2 Examples

```
angle_write 1  500 table.txt Harmonic_1
angle_write 3 1000 table.txt Harmonic_3
```

### 1.3.3 Description

Added in version 8Feb2023.

Write energy and force values to a file as a function of angle for the currently defined angle potential. Force in this context means the force with respect to the angle, not the force on individual atoms. This is useful for plotting the potential function or otherwise debugging its values. The resulting file can also be used as input for use with *angle style table*.

If the file already exists, the table of values is appended to the end of the file to allow multiple tables of energy and force to be included in one file. The individual sections may be identified by the *keyword*.

The energy and force values are computed for angles ranging from 0 degrees to 180 degrees for 3 interacting atoms forming an angle type atype, using the appropriate *angle_coeff* coefficients. N evenly spaced angles are used.

For example, for N = 6, values are computed at $\theta = 0, 36, 72, 108, 144, 180$.

The file is written in the format used as input for the *angle_style table* option with *keyword* as the section name. Each line written to the file lists an index number (1-N), an angle (in degrees), an energy (in energy units), and a force (in force units per radians^2). In case a new file is created, the first line will be a comment with a "DATE:" and "UNITS:" tag with the current date and *units* settings. For subsequent invocations of the *angle_write* command for the same file, data will be appended and the current units settings will be compared to the data from the header, if present. The *angle_write* will refuse to add a table to an existing file if the units are not the same.

### 1.3.4 Restrictions

All force field coefficients for angle and other kinds of interactions must be set before this command can be invoked.

The table of the angle energy and force data data is created by using a separate, internally created, new LAMMPS instance with a dummy system of 3 atoms for which the angle potential energy is computed after transferring the angle style and coefficients and arranging the three atoms into the corresponding geometries. The angle force is then determined from the potential energies through numerical differentiation. As a consequence of this approach, not all angle styles are compatible. The following conditions must be met:

- The angle style must be able to write its coefficients to a data file. This condition excludes for example *angle style hybrid* and *angle style table*.
- The potential function must not have any terms that depend on geometry properties other than the angle. This condition excludes for example *angle style class2* all angle types for *angle style charmm* that have non-zero Urey-Bradley terms. Please note that the *write_angle* command has no way of checking for this condition, so the resulting tables may be bogus if the requirement is not met. It is thus recommended to make careful tests for any created tables.

### 1.3.5 Related commands

*angle_style table*, *bond_write*, *dihedral_write*, *angle_style*, *angle_coeff*

### 1.3.6 Default

none

## 1.4 atom_modify command

### 1.4.1 Syntax

```
atom_modify keyword values ...
```

- one or more keyword/value pairs may be appended

- keyword = *id* or *map* or *first* or *sort*

```
id value = yes or no
map value = yes or array or hash
first value = group-ID = group whose atoms will appear first in internal atom lists
sort values = Nfreq binsize
  Nfreq = sort atoms spatially every this many time steps
  binsize = bin size for spatial sorting (distance units)
```

### 1.4.2 Examples

```
atom_modify map yes
atom_modify map hash sort 10000 2.0
atom_modify first colloid
```

### 1.4.3 Description

Modify certain attributes of atoms defined and stored within LAMMPS, in addition to what is specified by the *atom_style* command. The *id* and *map* keywords must be specified before a simulation box is defined; other keywords can be specified any time.

The *id* keyword determines whether non-zero atom IDs can be assigned to each atom. If the value is *yes*, which is the default, IDs are assigned, whether you use the *create atoms* or *read_data* or *read_restart* commands to initialize atoms. If the value is *no* the IDs for all atoms are assumed to be 0.

If atom IDs are used, they must all be positive integers. They should also be unique, though LAMMPS does not check for this. Typically they should also be consecutively numbered (from 1 to Natoms), though this is not required. Molecular *atom styles* are those that store bond topology information (styles bond, angle, molecular, full). These styles require atom IDs since the IDs are used to encode the topology. Some other LAMMPS commands also require the use of atom IDs. E.g. some many-body pair styles use them to avoid double computation of the I-J interaction between two atoms.

The only reason not to use atom IDs is if you are running an atomic simulation so large that IDs cannot be uniquely assigned. For a default LAMMPS build this limit is 2^31 or about 2 billion atoms. However, even in this case, you can use 64-bit atom IDs, allowing 2^63 or about 9e18 atoms, if you build LAMMPS with the - DLAMMPS_BIGBIG switch. This is described on the *Build_settings* doc page. If atom IDs are not used, they must be specified as 0 for all atoms, e.g. in a data or restart file.

**Note**

If a *triclinic simulation box* is used, atom IDs are required, due to how neighbor lists are built.

The *map* keyword determines how atoms with specific IDs are found when required. For example, the bond (angle, etc) methods need to find the local index of an atom with a specific global ID which is a bond (angle, etc) partner. LAMMPS performs this operation efficiently by creating a "map", which is either an *array* or *hash* table, as described below.

When the *map* keyword is not specified in your input script, LAMMPS only creates a map for *atom_styles* for molecular systems which have permanent bonds (angles, etc). No map is created for atomic systems, since it is normally not needed. However some LAMMPS commands require a map, even for atomic systems, and will generate an error if one does not exist. The *map* keyword thus allows you to force the creation of a map.

Specifying a value of *yes* will create either an array-style or hash-style map, depending on the size of the system. If no atom ID is larger than 1 million, then an array-style map is used, otherwise a hash-style map is used. Specifying a value of *array* or *hash* creates an array-style or hash-style map respectively, regardless of the size of the system.

For an array-style map, each processor stores a lookup table of length N, where N is the largest atom ID in the system. This is a fast, simple method for many simulations, but requires too much memory for large simulations. For a hash-style map, a hash table is created on each processor, which finds an atom ID in constant time (independent of the global number of atom IDs). It can be slightly slower than the *array* map, but its memory cost is proportional to the number of atoms owned by a processor, i.e. N/P when N is the total number of atoms in the system and P is the number of processors.

The *first* keyword allows a *group* to be specified whose atoms will be maintained as the first atoms in each processor's list of owned atoms. This in only useful when the specified group is a small fraction of all the atoms, and there are other operations LAMMPS is performing that will be sped-up significantly by being able to loop over the smaller set of atoms. Otherwise the reordering required by this option will be a net slow-down. The *neigh_modify include* and *comm_modify group* commands are two examples of commands that require this setting to work efficiently. Several *fixes*, most notably time integration fixes like *fix nve*, also take advantage of this setting if the group they operate on is the group specified by this command. Note that specifying "all" as the group-ID effectively turns off the *first* option.

It is OK to use the *first* keyword with a group that has not yet been defined, e.g. to use the atom_modify first command at the beginning of your input script. LAMMPS does not use the group until a simulation is run.

The *sort* keyword turns on a spatial sorting or reordering of atoms within each processor's subdomain every *Nfreq* timesteps. If *Nfreq* is set to 0, then sorting is turned off. Sorting can improve cache performance and thus speed-up a LAMMPS simulation, as discussed in a paper by *(Meloni)*. Its efficacy depends on the problem size (atoms/processor), how quickly the system becomes disordered, and various other factors. As a general rule, sorting is typically more effective at speeding up simulations of liquids as opposed to solids. In tests we have done, the speed-up can range from zero to 3-4x.

Reordering is performed every *Nfreq* timesteps during a dynamics run or iterations during a minimization. More precisely, reordering occurs at the first reneighboring that occurs after the target timestep. The reordering is performed locally by each processor, using bins of the specified *binsize*. If *binsize* is set to 0.0, then a binsize equal to half the *neighbor* cutoff distance (force cutoff plus skin distance) is used, which is a reasonable value. After the atoms have been binned, they are reordered so that atoms in the same bin are adjacent to each other in the processor's 1d list of atoms. When using neighbor style *multi*, the default is half of the largest cutoff. However, testing is encouraged as smaller values may improve performance.

The goal of this procedure is for atoms to put atoms close to each other in the processor's one-dimensional list of atoms that are also near to each other spatially. This can improve cache performance when pairwise interactions and neighbor lists are computed. Note that if bins are too small, there will be few atoms/bin. Likewise if bins are too large, there will be many atoms/bin. In both cases, the goal of cache locality will be undermined.

**Note**

Running a simulation with sorting on versus off should not change the simulation results in a statistical sense. However, a different ordering will induce round-off differences, which will lead to diverging trajectories over time when comparing two simulations. Various commands, particularly those which use random numbers (e.g. *velocity create*, and *fix langevin*), may generate (statistically identical) results which depend on the order in which atoms are processed. The order of atoms in a *dump* file will also typically change if sorting is enabled.

> **Note**
>
> When running simple pair-wise potentials like Lennard Jones on GPUs with the KOKKOS package, using a larger binsize (e.g. 2x larger than default) and a more frequent reordering than default (e.g. every 100 time steps) may improve performance.

### 1.4.4 Restrictions

The *first* and *sort* options cannot be used together. Since sorting is on by default, it will be turned off if the *first* keyword is used with a group-ID that is not "all".

### 1.4.5 Related commands

none

### 1.4.6 Default

By default, *id* is yes. By default, atomic systems (no bond topology info) do not use a map. For molecular systems (with bond topology info), the default is to use a map of either *array* or *hash* style depending on the size of the system, as explained above for the *map yes* keyword/value option. By default, a *first* group is not defined. By default, sorting is enabled with a frequency of 1000 and a binsize of 0.0, which means the maximum neighbor cutoff will be used to set the bin size. If no neighbor cutoff is defined, sorting will be turned off.

---

**(Meloni)** Meloni, Rosati and Colombo, J Chem Phys, 126, 121102 (2007).

## 1.5 atom_style command

### 1.5.1 Syntax

```
atom_style style args
```

- style = *amoeba* or *angle* or *apip* or *atomic* or *body* or *bond* or *charge* or *dielectric* or *dipole* or *dpd* or *edpd* or *electron* or *ellipsoid* or *full* or *line* or *mdpd* or *molecular* or *oxdna* or *peri* or *smd* or *sph* or *sphere* or *bpm/sphere* or *spin* or *tdpd* or *tri* or *template* or *hybrid*

```
args = none for any style except the following
  apip arg = conservative/thermostat (optional) for conservative APIP/lambda␣
↪thermostat APIP
  body args = bstyle bstyle-args
    bstyle = style of body particles
    bstyle-args = additional arguments specific to the bstyle
                  see the Howto body doc
                  page for details
  sphere arg = 0/1 (optional) for static/dynamic particle radii
  bpm/sphere arg = 0/1 (optional) for static/dynamic particle radii
  tdpd arg = Nspecies
    Nspecies = # of chemical species
  template arg = template-ID
    template-ID = ID of molecule template specified in a separate molecule command
  hybrid args = list of one or more sub-styles, each with their args
  ellipsoid arg = superellipsoid (optional) for superellipsoids instead of␣
↪ellipsoids
```

- accelerated styles (with same args) = *angle/kk* or *atomic/kk* or *bond/kk* or *charge/kk* or *full/kk* or *molecular/kk* or *spin/kk*

### 1.5.2 Examples

```
atom_style atomic
atom_style bond
atom_style full
atom_style body nparticle 2 10
atom_style hybrid charge bond
atom_style hybrid charge body nparticle 2 5
atom_style spin
atom_style template myMols
atom_style hybrid template twomols charge
atom_style tdpd 2
```

### 1.5.3 Description

The *atom_style* command selects which per-atom attributes are associated with atoms in a LAMMPS simulation and thus stored and communicated with those atoms as well as read from and stored in data and restart files. Different models (e.g. *pair styles*) require access to specific per-atom attributes and thus require a specific atom style. For example, to compute Coulomb interactions, the atom must have a "charge" (aka "q") attribute.

A number of distinct atom styles exist that combine attributes. Some atom styles are a superset of other atom styles. Further attributes may be added to atoms either via using a hybrid style which provides a union of the attributes of the sub-styles, or via the *fix property/atom* command. The *atom_style* command must be used before a simulation is setup via a *read_data*, *read_restart*, or *create_box* command.

**Note**

Many of the atom styles discussed here are only enabled if LAMMPS was built with a specific package, as listed below in the Restrictions section.

Once a style is selected and the simulation box defined, it cannot be changed but only augmented with the *fix property/atom* command. So one should select an atom style general enough to encompass all attributes required. E.g. with atom style *bond*, it is not possible to define angles and use angle styles.

It is OK to use a style more general than needed, though it may be slightly inefficient because it will allocate and communicate additional unused data.

### 1.5.4 Atom style attributes

The atom style *atomic* has the minimum subset of per-atom attributes and is also the default setting. It encompasses the following per-atom attributes (name of the vector or array in the *Atom class* is given in parenthesis): atom-ID (tag), type (type), position (x), velocities (v), forces (f), image flags (image), group membership (mask). Since all atom styles are a superset of atom style *atomic*, they all include these attributes.

This table lists all the available atom styles, which attributes they provide, which *package* is required to use them, and what the typical applications are that use them. See the *read_data*, *create_atoms*, and *set* commands for details on how to set these various quantities. More information about many of the styles is provided in the Additional Information section below.

| Atom style | Attributes | Required package | Applications |
|---|---|---|---|
| *amoeba* | *full* + "1-5 special neighbor data" | *AMOEBA* | AMOEBA/HIPPO force fields |
| *angle* | *bond* + "angle data" | *MOLECULE* | bead-spring polymers with stiffness |
| *apip thermostat* | *atomic* + apip_lambda, apip_lambda_required, apip_lambda_input, apip_lambda_const, apip_lambda_input_ta, apip_e_fast, apip_e_precise, apip_f_const_lambda, apip_f_dyn_lambda | *APIP* | adaptive-precision interatomic potentials(APIP) with a *lambda thermostat*, see *APIP howto* |
| *apip conservative* | *atomic* + apip_lambda, apip_lambda_required, apip_e_fast, apip_e_precise | *APIP* | conservative adaptive-precision interatomic potentials(APIP), see *APIP howto* |
| *atomic* | tag, type, x, v, f, image, mask | | atomic liquids, solids, metals |
| *body* | *atomic* + radius, rmass, angmom, torque, body | *BODY* | arbitrary bodies, see *body howto* |
| *bond* | *atomic* + molecule, nspecial, special + "bond data" | *MOLECULE* | bead-spring polymers |
| *bpm/sphere* | *bond* + radius, rmass, omega, torque, quat | *BPM* | granular bonded particle models, see *BPM howto* |
| *charge* | *atomic* + q | | atomic systems with charges |
| *dielectric* | *full* + mu, area, ed, em, epsilon, curvature, q_scaled | *DIELECTRIC* | systems with surface polarization |
| *dipole* | *charge* + mu | *DIPOLE* | atomic systems with charges and point dipoles |
| *dpd* | *atomic* + rho + "reactive DPD data" | *DPD-REACT* | reactive DPD |
| *edpd* | *atomic* + "eDPD data" | *DPD-MESO* | Energy conservative DPD (eDPD) |
| *electron* | *charge* + espin, eradius, ervel, erforce | *EFF* | Electron force field systems |
| *ellipsoid* | *atomic* + rmass, angmom, torque, ellipsoid | | aspherical particles |
| *full* | *molecular* + q | *MOLECULE* | molecular force fields |
| *line* | *atomic* + molecule, radius, rmass, omega, torque, line | | 2-d rigid body particles |
| *mdpd* | *atomic* + rho, drho, vest | *DPD-MESO* | Many-body DPD (mDPD) |
| *molecular* | *angle* + "dihedral and improper data" | *MOLECULE* | apolar and uncharged molecules |
| *oxdna* | *atomic* + id5p | *CG-DNA* | coarse-grained DNA and RNA models |
| *peri* | *atomic* + rmass, vfrac, s0, x0 | *PERI* | mesoscopic Peridynamics models |
| *smd* | *atomic* + molecule, radius, rmass + "smd data" | *MACHDYN* | Smooth Mach Dynamics models |
| *rheo* | *atomic* + rho, status | *RHEO* | solid and fluid RHEO particles |
| *rheo/thermal* | *atomic* + rho, status, energy, temperature | *RHEO* | RHEO particles with temperature |
| *sph* | *atomic* + "sph data" | *SPH* | Smoothed particle hydrodynamics models |
| *sphere* | *atomic* + radius, rmass, omega, torque | | finite size spherical particles, e.g. granular models |
| *spin* | *atomic* + "magnetic moment data" | *SPIN* | magnetic particles |

continues on next page

Table 1 – continued from previous page

| Atom style | Attributes | Required package | Applications |
|---|---|---|---|
| *tdpd* | *atomic* + cc, cc_flux, vest | *DPD-MESO* | Transport DPD (tDPD) |
| *template* | *atomic* + molecule, molindex, molatom | *MOLECULE* | molecular systems where attributes are taken from *molecule files* |
| *tri* | *sphere* + molecule, angmom, tri | | 3-d triangulated rigid body LJ particles |

**Note**

It is possible to add some attributes, such as a molecule ID and charge, to atom styles that do not have them built in using the *fix property/atom* command. This command also allows new custom-named attributes consisting of extra integer or floating-point values or vectors to be added to atoms. See the *fix property/atom* page for examples of cases where this is useful and details on how to initialize, access, and output these custom values.

### 1.5.5 Particle size and mass

All of the atom styles define point particles unless they (1) define finite-size spherical particles via the *radius* attribute, or (2) define finite-size aspherical particles (e.g. the *body*, *ellipsoid*, *line*, and *tri* styles). Most of these styles can also be used with mixtures of point and finite-size particles.

Note that the *radius* property may need to be provided as a *diameter* (e.g. in *molecule files* or *data files*). See the *Howto spherical* page for an overview of using finite-size spherical and aspherical particle models with LAMMPS.

Unless an atom style defines the per-atom *rmass* attribute, particle masses are defined on a per-type basis, using the *mass* command. This means each particle's mass is indexed by its atom *type*.

A few styles define the per-atom *rmass* attribute which can also be added using the *fix property/atom* command. In this case each particle stores its own mass. Atom styles that have a per-atom rmass may define it indirectly through setting particle diameter and density on a per-particle basis. If both per-type mass and per-atom *rmass* are defined (e.g. in a hybrid style), the per-atom mass will take precedence in any operation which which works with both flavors of mass.

### 1.5.6 Additional information about specific atom styles

Changed in version 30Mar2026.

For the *apip* style, one can choose between the style for conservative potentials and the style for the *lambda thermostat*. The *Howto apip* describes the differences between both use cases.

For the *body* style, the particles are arbitrary bodies with internal attributes defined by the "style" of the bodies, which is specified by the *bstyle* argument. Body particles can represent complex entities, such as surface meshes of discrete points, collections of sub-particles, deformable objects, etc.

The *Howto body* page describes the body styles LAMMPS currently supports, and provides more details as to the kind of body particles they represent. For all styles, each body particle stores moments of inertia and a quaternion 4-vector, so that its orientation and position can be time integrated due to forces and torques.

Note that there may be additional arguments required along with the *bstyle* specification, in the atom_style body command. These arguments are described on the *Howto body* doc page.

For the *dielectric* style, each particle can be either a physical particle (e.g. an ion), or an interface particle representing a boundary element between two regions of different dielectric constant. For interface particles, in addition to the properties associated with atom_style full, each particle also should be assigned a unit dipole vector (mu) representing

the direction of the induced dipole moment at each interface particle, an area (area/patch), the difference and mean of the dielectric constants of two sides of the interface along the direction of the normal vector (ed and em), the local dielectric constant at the boundary element (epsilon), and a mean local curvature (curv). Physical particles must be assigned these values, as well, but only their local dielectric constants will be used; see documentation for associated *pair styles* and *fixes*. The distinction between the physical and interface particles is only meaningful when *fix polarize* commands are applied to the interface particles. This style is part of the DIELECTRIC package.

For the *dipole* style, a point dipole vector mu is defined for each point particle. Note that if you wish the particles to be finite-size spheres as in a Stockmayer potential for a dipolar fluid, so that the particles can rotate due to dipole-dipole interactions, then you need to use the command *atom_style hybrid sphere dipole*, which will assign both a diameter and dipole moment to each particle. This also requires using an integrator with a "/sphere" suffix like *fix nve/sphere* or *fix nvt/sphere* and the "update dipole" or "update dlm" parameters to the fix commands.

The *dpd* style is for reactive dissipative particle dynamics (DPD) particles. Note that it is part of the DPD-REACT package, and is not required for use with the *pair_style dpd or dpd/stat* commands, which only require the attributes from atom_style *atomic*. Atom_style *dpd* extends DPD particle properties with internal temperature (dpdTheta), internal conductive energy (uCond), internal mechanical energy (uMech), and internal chemical energy (uChem).

The *edpd* style is for energy-conserving dissipative particle dynamics (eDPD) particles which store a temperature (edpd_temp), and heat capacity (edpd_cv).

For the *electron* style, the particles representing electrons are 3d Gaussians with a specified position and bandwidth or uncertainty in position, which is represented by the eradius = electron size.

For the *ellipsoid* style, particles can be ellipsoids which each stores a shape vector with the 3 diameters of the ellipsoid and a quaternion 4-vector with its orientation. Each particle stores a flag in the ellipsoid vector which indicates whether it is an ellipsoid (1) or a point particle (0).

Added in version 30Mar2026.

By adding the flag *superellipsoid* to the *ellipsoid* atom_style command, the particles can be superellipsoids, which are a generalization of ellipsoids with two additional blockiness parameters that control the shape. Superellipsoids also store the principal moments of inertia of the particle.

For the *line* style, particles can be are idealized line segments which store a per-particle mass and length and orientation (i.e. the end points of the line segment). Each particle stores a flag in the line vector which indicates whether it is a line segment (1) or a point particle (0).

The *mdpd* style is for many-body dissipative particle dynamics (mDPD) particles which store a density (rho) for considering density-dependent many-body interactions.

The *oxdna* style is for coarse-grained nucleotides and stores the 3'-to-5' polarity of the nucleotide strand, which is set through the bond topology in the data file. The first (second) atom in a bond definition is understood to point towards the 3'-end (5'-end) of the strand.

For the *peri* style, the particles are spherical and each stores a per-particle mass and volume.

The *smd* style is for Smooth Particle Mach dynamics. Both fluids and solids can be modeled. Particles store the mass and volume of an integration point, a kernel diameter used for calculating the field variables (e.g. stress and deformation) and a contact radius for calculating repulsive forces which prevent individual physical bodies from penetrating each other.

The *sph* style is for smoothed particle hydrodynamics (SPH) particles which store a density (rho), energy (esph), and heat capacity (cv).

For the *spin* style, a magnetic spin is associated with each atom. Those spins have a norm (their magnetic moment) and a direction.

The *tdpd* style is for transport dissipative particle dynamics (tDPD) particles which store a set of chemical concentration. An integer "cc_species" is required to specify the number of chemical species involved in a tDPD system.

The *wavepacket* style is similar to the *electron* style, but the electrons may consist of several Gaussian wave packets, summed up with coefficients cs= (cs_re,cs_im). Each of the wave packets is treated as a separate particle in LAMMPS, wave packets belonging to the same electron must have identical *etag* values.

The *sphere* and *bpm/sphere* styles allow particles to be either point particles or finite-size particles. If the *radius* attribute is > 0.0, the particle is a finite-size sphere. If the diameter = 0.0, it is a point particle. Note that by using the *disc* keyword with the *fix nve/sphere*, *fix nvt/sphere*, *fix nph/sphere*, *fix npt/sphere* commands for the *sphere* style, spheres can be effectively treated as 2d discs for a 2d simulation if desired. See also the *set density/disc* command. These styles also take an optional 0 or 1 argument. A value of 0 means the radius of each sphere is constant for the duration of the simulation (this is the default). A value of 1 means the radii may vary dynamically during the simulation, e.g. due to use of the *fix adapt* command.

The *template* style allows molecular topology (bonds,angles,etc) to be defined via a molecule template using the *molecule* command. The template stores one or more molecules with a single copy of the topology info (bonds,angles,etc) of each. Individual atoms only store a template index and template atom to identify which molecule and which atom-within-the-molecule they represent. Using the *template* style instead of the *bond*, *angle*, *molecular* styles can save memory for systems comprised of a large number of small molecules, all of a single type (or small number of types). See the paper by Grime and Voth, in *(Grime)*, for examples of how this can be advantageous for large-scale coarse-grained systems. The `examples/template` directory has a few demo inputs and examples showing the use of the *template* atom style versus *molecular*.

**Note**

When using the *template* style with a *molecule template* that contains multiple molecules, you should ensure the atom types, bond types, angle_types, etc in all the molecules are consistent. E.g. if one molecule represents H2O and another CO2, then you probably do not want each molecule file to define two atom types and a single bond type, because they will conflict with each other when a mixture system of H2O and CO2 molecules is defined, e.g. by the *read_data* command. Rather the H2O molecule should define atom types 1 and 2, and bond type 1. And the CO2 molecule should define atom types 3 and 4 (or atom types 3 and 2 if a single oxygen type is desired), and bond type 2.

For the *tri* style, particles can be planar triangles which each stores a per-particle mass and size and orientation (i.e. the corner points of the triangle). Each particle stores a flag in the tri vector which indicates whether it is a triangle (1) or a point particle (0).

---

Typically, simulations require only a single (non-hybrid) atom style. If some atoms in the simulation do not have all the properties defined by a particular style, use the simplest style that defines all the needed properties by any atom. For example, if some atoms in a simulation are charged, but others are not, use the *charge* style. If some atoms have bonds, but others do not, use the *bond* style.

The only scenario where the *hybrid* style is needed is if there is no single style which defines all needed properties of all atoms. For example, as mentioned above, if you want dipolar particles which will rotate due to torque, you need to use "atom_style hybrid sphere dipole". When a hybrid style is used, atoms store and communicate the union of all quantities implied by the individual styles.

When using the *hybrid* style, you cannot combine the *template* style with another molecular style that stores bond, angle, etc info on a per-atom basis.

LAMMPS can be extended with new atom styles as well as new body styles; see the corresponding manual page on *modifying & extending LAMMPS*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages*

page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 1.5.7 Restrictions

This command cannot be used after the simulation box is defined by a *read_data* or *create_box* command.

Many of the styles listed above are only enabled if LAMMPS was built with a specific package, as listed below. See the *Build package* page for more info. The table above lists which package is required for individual atom styles.

### 1.5.8 Related commands

*read_data*, *pair_style*, *fix property/atom*, *set*

### 1.5.9 Default

The default atom style is *atomic*. If atom_style *sphere* or *bpm/sphere* is used, its default argument is 0. If atom_style *apip* is used, its default argument is 'thermostat'.

---

**(Grime)** Grime and Voth, to appear in J Chem Theory & Computation (2014).

## 1.6 balance command

### 1.6.1 Syntax

```
balance thresh style args ... keyword args ...
```

- thresh = imbalance threshold that must be exceeded to perform a re-balance
- one style/arg pair can be used (or multiple for *x*,*y*,*z*)
- style = *x* or *y* or *z* or *shift* or *rcb*

```
  x args = uniform or Px-1 numbers between 0 and 1
    uniform = evenly spaced cuts between processors in x dimension
    numbers = Px-1 ascending values between 0 and 1, Px - # of processors in x␣
  ↪dimension
    x can be specified together with y or z
  y args = uniform or Py-1 numbers between 0 and 1
    uniform = evenly spaced cuts between processors in y dimension
    numbers = Py-1 ascending values between 0 and 1, Py - # of processors in y␣
  ↪dimension
    y can be specified together with x or z
  z args = uniform or Pz-1 numbers between 0 and 1
    uniform = evenly spaced cuts between processors in z dimension
```

```
    numbers = Pz-1 ascending values between 0 and 1, Pz - # of processors in z␣
  ↪dimension
    z can be specified together with x or y
  shift args = dimstr Niter stopthresh
    dimstr = sequence of letters containing "x" or "y" or "z", each not more than once
    Niter = # of times to iterate within each dimension of dimstr sequence
    stopthresh = stop balancing when this imbalance threshold is reached
  rcb args = none
```

- zero or more keyword/arg pairs may be appended
- keyword = *weight* or *out*

```
  weight style args = use weighted particle counts for the balancing
    style = group or neigh or time or var or store
      group args = Ngroup group1 weight1 group2 weight2 ...
        Ngroup = number of groups with assigned weights
        group1, group2, ... = group IDs
        weight1, weight2, ...   = corresponding weight factors
      neigh factor = compute weight based on number of neighbors
        factor = scaling factor (> 0)
      time factor = compute weight based on time spend computing
        factor = scaling factor (> 0)
      var name = take weight from atom-style variable
        name = name of the atom-style variable
      store name = store weight in custom atom property defined by fix property/atom␣
  ↪command
        name = atom property name (without d_ prefix)
  sort arg = no or yes
  out arg = filename
    filename = write each processor's subdomain to a file
```

### 1.6.2 Examples

```
balance 0.9 x uniform y 0.4 0.5 0.6
balance 1.2 shift xz 5 1.1
balance 1.0 shift xz 5 1.1
balance 1.1 rcb
balance 1.0 shift x 10 1.1 weight group 2 fast 0.5 slow 2.0
balance 1.0 shift x 10 1.1 weight time 0.8 weight neigh 0.5 weight store balance
balance 1.0 shift x 20 1.0 out tmp.balance
```

### 1.6.3 Description

This command adjusts the size and shape of processor subdomains within the simulation box, to attempt to balance the number of atoms or particles and thus indirectly the computational cost (load) more evenly across processors. The load balancing is "static" in the sense that this command performs the balancing once, before or between simulations. The processor subdomains will then remain static during the subsequent run. To perform "dynamic" balancing, see the *fix balance* command, which can adjust processor subdomain sizes and shapes on-the-fly during a *run*.

Load-balancing is typically most useful if the particles in the simulation box have a spatially-varying density distribution or when the computational cost varies significantly between different particles. E.g. a model of a vapor/liquid interface, or a solid with an irregular-shaped geometry containing void regions, or *hybrid pair style simulations* which combine pair styles with different computational cost. In these cases, the LAMMPS default of dividing the simulation box volume into a regular-spaced grid of 3d bricks, with one equal-volume subdomain per processor, may assign numbers

of particles per processor in a way that the computational effort varies significantly. This can lead to poor performance when the simulation is run in parallel.

The balancing can be performed with or without per-particle weighting. With no weighting, the balancing attempts to assign an equal number of particles to each processor. With weighting, the balancing attempts to assign an equal aggregate computational weight to each processor, which typically induces a different number of atoms assigned to each processor. Details on the various weighting options and examples for how they can be used are *given below*.

Note that the *processors* command allows some control over how the box volume is split across processors. Specifically, for a Px by Py by Pz grid of processors, it allows choice of Px, Py, and Pz, subject to the constraint that Px * Py * Pz = P, the total number of processors. This is sufficient to achieve good load-balance for some problems on some processor counts. However, all the processor subdomains will still have the same shape and same volume.

The requested load-balancing operation is only performed if the current "imbalance factor" in particles owned by each processor exceeds the specified *thresh* parameter. The imbalance factor is defined as the maximum number of particles (or weight) owned by any processor, divided by the average number of particles (or weight) per processor. Thus an imbalance factor of 1.0 is perfect balance.

As an example, for 10000 particles running on 10 processors, if the most heavily loaded processor has 1200 particles, then the factor is 1.2, meaning there is a 20% imbalance. Note that a re-balance can be forced even if the current balance is perfect (1.0) be specifying a *thresh* < 1.0.

**Note**

Balancing is performed even if the imbalance factor does not exceed the *thresh* parameter if a "grid" style is specified when the current partitioning is "tiled". The meaning of "grid" vs "tiled" is explained below. This is to allow forcing of the partitioning to "grid" so that the *comm_style brick* command can then be used to replace a current *comm_style tiled* setting.

When the balance command completes, it prints statistics about the result, including the change in the imbalance factor and the change in the maximum number of particles on any processor. For "grid" methods (defined below) that create a logical 3d grid of processors, the positions of all cutting planes in each of the 3 dimensions (as fractions of the box length) are also printed.

**Note**

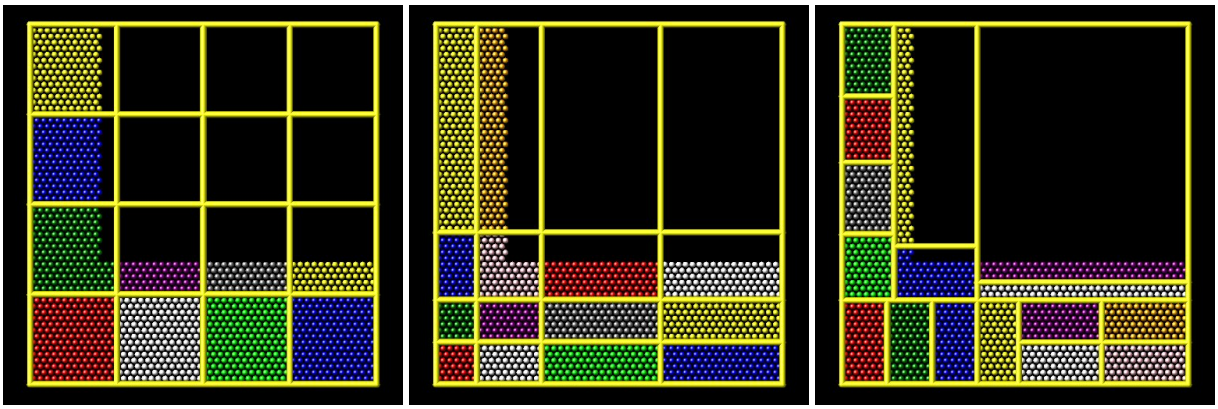
This command attempts to minimize the imbalance factor, as defined above. But depending on the method a perfect balance (1.0) may not be achieved. For example, "grid" methods (defined below) that create a logical 3d grid cannot achieve perfect balance for many irregular distributions of particles. Likewise, if a portion of the system is a perfect lattice, e.g. the initial system is generated by the *create_atoms* command, then "grid" methods may be unable to achieve exact balance. This is because entire lattice planes will be owned or not owned by a single processor.

**Note**

The imbalance factor is also an estimate of the maximum speed-up you can hope to achieve by running a perfectly balanced simulation versus an imbalanced one. In the example above, the 10000 particle simulation could run up to 20% faster if it were perfectly balanced, versus when imbalanced. However, computational cost is not strictly proportional to particle count, and changing the relative size and shape of processor subdomains may lead to additional computational and communication overheads, e.g. in the PPPM solver used via the *kspace_style* command. Thus you should benchmark the run times of a simulation before and after balancing.

The method used to perform a load balance is specified by one of the listed styles (or more in the case of *x*,*y*,*z*), which are described in detail below. There are 2 kinds of styles.

The *x*, *y*, *z*, and *shift* styles are "grid" methods which produce a logical 3d grid of processors. They operate by changing the cutting planes (or lines) between processors in 3d (or 2d), to adjust the volume (area in 2d) assigned to each processor, as in the following 2d diagram where processor subdomains are shown and particles are colored by the processor that owns them.



The leftmost diagram is the default partitioning of the simulation box across processors (one sub-box for each of 16 processors); the middle diagram is after a "grid" method has been applied. The *rcb* style is a "tiling" method which does not produce a logical 3d grid of processors. Rather it tiles the simulation domain with rectangular sub-boxes of varying size and shape in an irregular fashion so as to have equal numbers of particles (or weight) in each sub-box, as in the rightmost diagram above.

The "grid" methods can be used with either of the *comm_style* command options, *brick* or *tiled*. The "tiling" methods can only be used with *comm_style tiled*. Note that it can be useful to use a "grid" method with *comm_style tiled* to return the domain partitioning to a logical 3d grid of processors so that "comm_style brick" can afterwords be specified for subsequent *run* commands.

When a "grid" method is specified, the current domain partitioning can be either a logical 3d grid or a tiled partitioning. In the former case, the current logical 3d grid is used as a starting point and changes are made to improve the imbalance factor. In the latter case, the tiled partitioning is discarded and a logical 3d grid is created with uniform spacing in all dimensions. This becomes the starting point for the balancing operation.

When a "tiling" method is specified, the current domain partitioning ("grid" or "tiled") is ignored, and a new partitioning is computed from scratch.

---

The *x*, *y*, and *z* styles invoke a "grid" method for balancing, as described above. Note that any or all of these 3 styles can be specified together, one after the other, but they cannot be used with any other style. This style adjusts the position of cutting planes between processor subdomains in specific dimensions. Only the specified dimensions are altered.

The *uniform* argument spaces the planes evenly, as in the left diagrams above. The *numeric* argument requires listing Ps-1 numbers that specify the position of the cutting planes. This requires knowing Ps = Px or Py or Pz = the number of processors assigned by LAMMPS to the relevant dimension. This assignment is made (and the Px, Py, Pz values printed out) when the simulation box is created by the "create_box" or "read_data" or "read_restart" command and is influenced by the settings of the *processors* command.

Each of the numeric values must be between 0 and 1, and they must be listed in ascending order. They represent the fractional position of the cutting place. The left (or lower) edge of the box is 0.0, and the right (or upper) edge is 1.0. Neither of these values is specified. Only the interior Ps-1 positions are specified. Thus is there are 2 processors in the x dimension, you specify a single value such as 0.75, which would make the left processor's subdomain 3x larger than the right processor's subdomain.

---

The *shift* style invokes a "grid" method for balancing, as described above. It changes the positions of cutting planes between processors in an iterative fashion, seeking to reduce the imbalance factor, similar to how the *fix balance shift* command operates.

The *dimstr* argument is a string of characters, each of which must be an "x" or "y" or "z". Each character can appear zero or one time, since there is no advantage to balancing on a dimension more than once. You should normally only list dimensions where you expect there to be a density variation in the particles.

Balancing proceeds by adjusting the cutting planes in each of the dimensions listed in *dimstr*, one dimension at a time. For a single dimension, the balancing operation (described below) is iterated on up to *Niter* times. After each dimension finishes, the imbalance factor is re-computed, and the balancing operation halts if the *stopthresh* criterion is met.

A re-balance operation in a single dimension is performed using a recursive multisectioning algorithm, where the position of each cutting plane (line in 2d) in the dimension is adjusted independently. This is similar to a recursive bisectioning for a single value, except that the bounds used for each bisectioning take advantage of information from neighboring cuts if possible. At each iteration, the count of particles on either side of each plane is tallied. If the counts do not match the target value for the plane, the position of the cut is adjusted to be halfway between a low and high bound. The low and high bounds are adjusted on each iteration, using new count information, so that they become closer together over time. Thus as the recursion progresses, the count of particles on either side of the plane gets closer to the target value.

After the balanced plane positions are determined, if any pair of adjacent planes are closer together than the neighbor skin distance (as specified by the *neigh_modify* command), then the plane positions are shifted to separate them by at least this amount. This is to prevent particles being lost when dynamics are run with processor subdomains that are too narrow in one or more dimensions.

Once the re-balancing is complete and final processor subdomains assigned, particles are migrated to their new owning processor, and the balance procedure ends.

**Note**

At each re-balance operation, the bisectioning for each cutting plane (line in 2d) typically starts with low and high bounds separated by the extent of a processor's subdomain in one dimension. The size of this bracketing region shrinks by 1/2 every iteration. Thus if *Niter* is specified as 10, the cutting plane will typically be positioned to 1 part in 1000 accuracy (relative to the perfect target position). For *Niter* = 20, it will be accurate to 1 part in a million. Thus there is no need to set *Niter* to a large value. LAMMPS will check if the threshold accuracy is reached (in a dimension) is less iterations than *Niter* and exit early. However, *Niter* should also not be set too small, since it will take roughly the same number of iterations to converge even if the cutting plane is initially close to the target value.

---

The *rcb* style invokes a "tiled" method for balancing, as described above. It performs a recursive coordinate bisectioning (RCB) of the simulation domain. The basic idea is as follows.

The simulation domain is cut into 2 boxes by an axis-aligned cut in one of the dimensions, leaving one new sub-box on either side of the cut. Which dimension is chosen for the cut depends on the particle (weight) distribution within the parent box. Normally the longest dimension of the box is cut, but if all (or most) of the particles are at one end of the box, a cut may be performed in another dimension to induce sub-boxes that are more cube-ish (3d) or square-ish (2d) in shape.

After the cut is made, all the processors are also partitioned into 2 groups, half assigned to the box on the lower side of the cut, and half to the box on the upper side. (If the processor count is odd, one side gets an extra processor.) The cut is positioned so that the number of (weighted) particles in the lower box is exactly the number that the processors assigned to that box should own for load balance to be perfect. This also makes load balance for the upper box perfect. The positioning of the cut is done iteratively, by a bisectioning method (median search). Note that counting particles on either side of the cut requires communication between all processors at each iteration.

That is the procedure for the first cut. Subsequent cuts are made recursively, in exactly the same manner. The subset of processors assigned to each box make a new cut in one dimension of that box, splitting the box, the subset of processors, and the particles in the box in two. The recursion continues until every processor is assigned a sub-box of the entire simulation domain, and owns the (weighted) particles in that sub-box.

---

This subsection describes how to perform weighted load balancing using the *weight* keyword.

By default, all particles have a weight of 1.0, which means each particle is assumed to require the same amount of computation during a timestep. There are, however, scenarios where this is not a good assumption. Measuring the computational cost for each particle accurately would be impractical and slow down the computation. Instead the *weight* keyword implements several ways to influence the per-particle weights empirically by properties readily available or using the user's knowledge of the system. Note that the absolute value of the weights are not important; only their relative ratios affect which particle is assigned to which processor. A particle with a weight of 2.5 is assumed to require 5x more computational than a particle with a weight of 0.5. For all the options below the weight assigned to a particle must be a positive value; an error will be be generated if a weight is <= 0.0.

Below is a list of possible weight options with a short description of their usage and some example scenarios where they might be applicable. It is possible to apply multiple weight flags and the weightings they induce will be combined through multiplication. Most of the time, however, it is sufficient to use just one method.

The *group* weight style assigns weight factors to specified *groups* of particles. The *group* style keyword is followed by the number of groups, then pairs of group IDs and the corresponding weight factor. If a particle belongs to none of the specified groups, its weight is not changed. If it belongs to multiple groups, its weight is the product of the weight factors.

This weight style is useful in combination with pair style *hybrid*, e.g. when combining a more costly many-body potential with a fast pairwise potential. It is also useful when using *run_style respa* where some portions of the system have many bonded interactions and others none. It assumes that the computational cost for each group remains constant over time. This is a purely empirical weighting, so a series test runs to tune the assigned weight factors for optimal performance is recommended.

The *neigh* weight style assigns the same weight to each particle owned by a processor based on the total count of neighbors in the neighbor list owned by that processor. The motivation is that more neighbors means a higher computational cost. The style does not use neighbors per atom to assign a unique weight to each atom, because that value can vary depending on how the neighbor list is built.

The *factor* setting is applied as an overall scale factor to the *neigh* weights which allows adjustment of their impact on the balancing operation. The specified *factor* value must be positive. A value > 1.0 will increase the weights so that the ratio of max weight to min weight increases by *factor*. A value < 1.0 will decrease the weights so that the ratio of max weight to min weight decreases by *factor*. In both cases the intermediate weight values increase/decrease proportionally as well. A value = 1.0 has no effect on the *neigh* weights. As a rule of thumb, we have found a *factor* of about 0.8 often results in the best performance, since the number of neighbors is likely to overestimate the ideal weight.

This weight style is useful for systems where there are different cutoffs used for different pairs of interactions, or the density fluctuates, or a large number of particles are in the vicinity of a wall, or a combination of these effects. If a simulation uses multiple neighbor lists, this weight style will use the first suitable neighbor list it finds. It will not request or compute a new list. A warning will be issued if there is no suitable neighbor list available or if it is not current, e.g. if the balance command is used before a *run* or *minimize* command is used, in which case the neighbor list may not yet have been built. In this case no weights are computed. Inserting a *run 0 post no* command before issuing the *balance* command, may be a workaround for this case, as it will induce the neighbor list to be built.

The *time* weight style uses *timer data* to estimate weights. It assigns the same weight to each particle owned by a processor based on the total computational time spent by that processor. See details below on what time window is used. It uses the same timing information as is used for the *MPI task timing breakdown*, namely, for sections *Pair*, *Bond*, *Kspace*, and *Neigh*. The time spent in those portions of the timestep are measured for each MPI rank, summed, then divided by the number of particles owned by that processor. I.e. the weight is an effective CPU time/particle averaged over the particles on that processor.

The *factor* setting is applied as an overall scale factor to the *time* weights which allows adjustment of their impact on the balancing operation. The specified *factor* value must be positive. A value > 1.0 will increase the weights so that the ratio of max weight to min weight increases by *factor*. A value < 1.0 will decrease the weights so that the ratio of max weight to min weight decreases by *factor*. In both cases the intermediate weight values increase/decrease proportionally as well. A value = 1.0 has no effect on the *time* weights. As a rule of thumb, effective values to use are typically between 0.5 and 1.2. Note that the timer quantities mentioned above can be affected by communication which occurs in the middle of the operations, e.g. pair styles with intermediate exchange of data witin the force computation, and likewise for KSpace solves.

When using the *time* weight style with the *balance* command, the timing data is taken from the preceding run command, i.e. the timings are for the entire previous run. For the *fix balance* command the timing data is for only the timesteps since the last balancing operation was performed. If timing information for the required sections is not available, e.g. at the beginning of a run, or when the *timer* command is set to either *loop* or *off*, a warning is issued. In this case no weights are computed.

**Note**

The *time* weight style is the most generic option, and should be tried first, unless the *group* style is easily applicable. However, since the computed cost function is averaged over all particles on a processor, the weights may not be highly accurate. This style can also be effective as a secondary weight in combination with either *group* or *neigh* to offset some of inaccuracies in either of those heuristics.

The *var* weight style assigns per-particle weights by evaluating an *atom-style variable* specified by *name*. This is provided as a more flexible alternative to the *group* weight style, allowing definition of a more complex heuristics based on information (global and per atom) available inside of LAMMPS. For example, atom-style variables can reference the position of a particle, its velocity, the volume of its Voronoi cell, etc.

The *store* weight style does not compute a weight factor. Instead it stores the current accumulated weights in a custom per-atom vector specified by *name*. This must be a vector defined as *d_name* via the *fix property/atom* command. This means the values in the vector can be read as part of a data file with the *read_data* command or specified with the *set* command. These weights can also be output in a *dump* file, so this is a way to examine, debug, or visualize the per-particle weights used during the load-balancing operation.

Note that the name of the custom per-atom vector is specified just as *name*, not as *d_name* as it is for other commands that use different kinds of custom atom vectors or arrays as arguments.

---

The *sort* keyword determines whether the communication of per-atom data to other processors during load-balancing will be random or deterministic. Random is generally faster; deterministic will ensure the new ordering of atoms on each processor is the same each time the same simulation is run. This can be useful for debugging purposes. Since the balance command is a one-time operation, the default is *yes* to perform sorting.

The *out* keyword writes a text file to the specified *filename* with the results of the balancing operation. The file contains the bounds of the subdomain for each processor after the balancing operation completes. The format of the file is compatible with the Pizza.py *mdump* tool which has support for manipulating and visualizing mesh files. An example is shown here for a balancing by 4 processors for a 2d problem:

```
ITEM: TIMESTEP
0
ITEM: NUMBER OF NODES
16
ITEM: BOX BOUNDS
0 10
0 10
```

(continues on next page)

(continued from previous page)

```
0 10
ITEM: NODES
1 1 0 0 0
2 1 5 0 0
3 1 5 5 0
4 1 0 5 0
5 1 5 0 0
6 1 10 0 0
7 1 10 5 0
8 1 5 5 0
9 1 0 5 0
10 1 5 5 0
11 1 5 10 0
12 1 10 5 0
13 1 5 5 0
14 1 10 5 0
15 1 10 10 0
16 1 5 10 0
ITEM: TIMESTEP
0
ITEM: NUMBER OF SQUARES
4
ITEM: SQUARES
1 1 1 2 3 4
2 1 5 6 7 8
3 1 9 10 11 12
4 1 13 14 15 16
```

The coordinates of all the vertices are listed in the NODES section, 5 per processor. Note that the 4 subdomains share vertices, so there will be duplicate nodes in the list.

The "SQUARES" section lists the node IDs of the 4 vertices in a rectangle for each processor (1 to 4).

For a 3d problem, the syntax is similar with 8 vertices listed for each processor, instead of 4, and "SQUARES" replaced by "CUBES".

---

### 1.6.4 Restrictions

For 2d simulations, the *z* style cannot be used. Nor can a "z" appear in *dimstr* for the *shift* style.

Balancing through recursive bisectioning (*rcb* style) requires *comm_style tiled*

### 1.6.5 Related commands

*group*, *processors*, *fix balance*, *comm_style*

### 1.6.6 Default

The default setting is sort = yes.

# 1.7 bond_coeff command

## 1.7.1 Syntax

```
bond_coeff N args
```

- N = numeric bond type (see asterisk form below), or type label
- args = coefficients for one or more bond types

## 1.7.2 Examples

```
bond_coeff 5 80.0 1.2
bond_coeff * 30.0 1.5 1.0 1.0
bond_coeff 1*4 30.0 1.5 1.0 1.0
bond_coeff 1 harmonic 200.0 1.0 # (for bond_style hybrid)

labelmap bond 5 carbonyl
bond_coeff carbonyl 80.0 1.2
```

## 1.7.3 Description

Specify the bond force field coefficients for one or more bond types. The number and meaning of the coefficients depends on the bond style. Bond coefficients can also be set in the data file read by the *read_data* command or in a restart file.

*N* can be specified in one of several ways. An explicit numeric value can be used, as in the first example above. Or *N* can be a type label, which is an alphanumeric string defined by the *labelmap* command or in a section of a data file read by the *read_data* command.

For numeric values only, a wild-card asterisk can be used to set the coefficients for multiple bond types. This takes the form "*" or "*n" or "n*" or "m*n". If *N* is the number of bond types, then an asterisk with no numeric values means all types from 1 to *N*. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to *N* (inclusive). A middle asterisk means all types from m to n (inclusive).

Note that using a bond_coeff command can override a previous setting for the same bond type. For example, these commands set the coeffs for all bond types, then overwrite the coeffs for just bond type 2:

```
bond_coeff * 100.0 1.2
bond_coeff 2 200.0 1.2
```

A line in a data file that specifies bond coefficients uses the exact same format as the arguments of the bond_coeff command in an input script, except that wild-card asterisks should not be used since coefficients for all *N* types must be listed in the file. For example, under the "Bond Coeffs" section of a data file, the line that corresponds to the first example above would be listed as

```
5 80.0 1.2
```

---

The list of all bond styles defined in LAMMPS is given on the *bond_style* doc page. They are also listed in more compact form on the *Commands bond* doc page.

On either of those pages, click on the style to display the formula it computes and its coefficients as specified by the associated bond_coeff command.

---

### 1.7.4 Restrictions

This command must come after the simulation box is defined by a *read_data*, *read_restart*, or *create_box* command.

A bond style must be defined before any bond coefficients are set, either in the input script or in a data file.

### 1.7.5 Related commands

*bond_style*

### 1.7.6 Default

none

## 1.8 bond_style command

### 1.8.1 Syntax

```
bond_style style args
```

- style = *none* or *zero* or *hybrid* or *bpm/rotational* or *bpm/spring* or *bpm/spring/plastic* or *class2* or *fene* or *fene/expand* or *fene/nm* or *gaussian* or *gromos* or *harmonic* or *harmonic/restrain harmonic/shift* or *harmonic/shift/cut* or *lepton* or *morse* or *nonlinear* or *oxdna/fene* or *oxdena2/fene* or *oxrna2/fene* or *quartic* or *quartic/exp* or *special* or *table*
- args = none for any style except *hybrid*
  - *hybrid* args = list of one or more styles

### 1.8.2 Examples

```
bond_style harmonic
bond_style fene
bond_style hybrid harmonic fene
```

### 1.8.3 Description

Set the formula(s) LAMMPS uses to compute bond interactions between pairs of atoms. In LAMMPS, a bond differs from a pairwise interaction, which are set via the *pair_style* command. Bonds are defined between specified pairs of atoms and remain in force for the duration of the simulation (unless new bonds are created or existing bonds break, which is possible in some fixes and bond potentials). The list of bonded atoms is read in by a *read_data* or *read_restart* command from a data or restart file. By contrast, pair potentials are typically defined between all pairs of atoms within a cutoff distance and the set of active interactions changes over time.

Hybrid models where bonds are computed using different bond potentials can be setup using the *hybrid* bond style.

The coefficients associated with a bond style can be specified in a data or restart file or via the *bond_coeff* command.

All bond potentials store their coefficient data in binary restart files which means bond_style and *bond_coeff* commands do not need to be re-specified in an input script that restarts a simulation. See the *read_restart* command for details on how to do this. The one exception is that bond_style *hybrid* only stores the list of sub-styles in the restart file; bond coefficients need to be re-specified.

**Note**

When both a bond and pair style is defined, the *special_bonds* command often needs to be used to turn off (or weight) the pairwise interaction that would otherwise exist between two bonded atoms.

In the formulas listed for each bond style, $r$ is the distance between the two atoms in the bond.

---

Here is an alphabetic list of bond styles defined in LAMMPS. Click on the style to display the formula it computes and coefficients specified by the associated *bond_coeff* command.

Click on the style to display the formula it computes, any additional arguments specified in the bond_style command, and coefficients specified by the associated *bond_coeff* command.

There are also additional accelerated pair styles included in the LAMMPS distribution for faster performance on CPUs, GPUs, and KNLs. The individual style names on the *Commands bond* doc page are followed by one or more of (g,i,k,o,t) to indicate which accelerated styles exist.

- *none* - turn off bonded interactions
- *zero* - topology but no interactions
- *hybrid* - define multiple styles of bond interactions
- *bpm/rotational* - breakable bond with forces and torques based on deviation from reference state
- *bpm/spring* - breakable bond with forces based on deviation from reference length
- *bpm/spring/plastic* - a similar breakable bond with plastic yield
- *class2* - COMPASS (class 2) bond
- *fene* - FENE (finite-extensible non-linear elastic) bond
- *fene/expand* - FENE bonds with variable size particles
- *fene/nm* - FENE bonds with a generalized Lennard-Jones potential
- *gaussian* - multicentered Gaussian-based bond potential
- *gromos* - GROMOS force field bond
- *harmonic* - harmonic bond
- *harmonic/restrain* - harmonic bond to restrain to original bond distance
- *harmonic/shift* - shifted harmonic bond
- *harmonic/shift/cut* - shifted harmonic bond with a cutoff
- *lepton* - bond potential from evaluating a string
- *mesocnt* - Harmonic bond wrapper with parameterization presets for nanotubes
- *mm3* - MM3 anharmonic bond
- *morse* - Morse bond
- *nonlinear* - nonlinear bond
- *oxdna/fene* - modified FENE bond suitable for DNA modeling
- *oxdna2/fene* - same as oxdna but used with different pair styles
- *oxdna3/fene* - same as oxdna but used with different pair styles

- *oxrna2/fene* - modified FENE bond suitable for RNA modeling
- *quartic* - breakable quartic bond
- *quartic/exp* - quartic bond with optional exponential repulsion
- *rheo/shell* - shell bond for oxidation modeling in RHEO
- *special* - enable special bond exclusions for 1-5 pairs and beyond
- *table* - tabulated by bond length

---

### 1.8.4 Restrictions

Bond styles can only be set for atom styles that allow bonds to be defined.

Most bond styles are part of the MOLECULE package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info. The doc pages for individual bond potentials tell if it is part of a package.

### 1.8.5 Related commands

*bond_coeff*, *delete_bonds*

### 1.8.6 Default

```
bond_style none
```

## 1.9 bond_write command

### 1.9.1 Syntax

```
bond_write btype N inner outer file keyword itype jtype
```

- btype = bond type
- N = # of values
- inner,outer = inner and outer bond length (distance units)
- file = name of file to write values to
- keyword = section name in file for this set of tabulated values
- itype,jtype = two atom types (optional)

### 1.9.2 Examples

```
bond_write 1 500 0.5 3.5 table.txt Harmonic_1
bond_write 3 1000 0.1 6.0 table.txt Morse
```

### 1.9.3 Description

Write energy and force values to a file as a function of distance for the currently defined *bond style* for a selected bond type. This is useful for plotting the potential function or otherwise debugging its values. The resulting file can also be used as input for use with *bond style table*.

If the file already exists, the table of values is appended to the end of the file to allow multiple tables of energy and force to be included in one file. The individual sections may be identified by the *keyword*.

The energy and force values are computed at distances from *inner* to *outer* for two interacting atoms forming a bond of type *btype*, using the appropriate *bond_coeff* coefficients. N evenly spaced distances are used.

For example, for N = 7, inner = 1.0, and outer = 4.0, values are computed at r = 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0.

The file is written in the format used as input for the *bond_style table* option with *keyword* as the section name. Each line written to the file lists an index number (1-N), a distance (in distance units), an energy (in energy units), and a force (in force units). In case a new file is created, the first line will be a comment with a "DATE:" and "UNITS:" tag with the current date and *units* settings. For subsequent invocations of the *bond_write* command for the same file, data will be appended and the current units settings will be compared to the data from the header, if present. The *bond_write* command will refuse to add a table to an existing file if the units are not the same.

### 1.9.4 Restrictions

All force field coefficients for bond and other kinds of interactions must be set before this command can be invoked.

Due to how the bond force is computed, an inner value > 0.0 must be specified even if the potential has a finite value at r = 0.0.

### 1.9.5 Related commands

*bond_style table*, *angle_write*, *bond_style*, *bond_coeff*

### 1.9.6 Default

none

## 1.10 boundary command

### 1.10.1 Syntax

```
boundary x y z
```

- x,y,z = *p* or *s* or *f* or *m*, one or two letters

  ```
  p is periodic
  f is non-periodic and fixed
  s is non-periodic and shrink-wrapped
  m is non-periodic and shrink-wrapped with a minimum value
  ```

### 1.10.2 Examples

```
boundary p p f
boundary p fs p
boundary s f fm
```

### 1.10.3 Description

Set the style of boundaries for the global simulation box in each dimension. A single letter assigns the same style to both the lower and upper face of the box. Two letters assigns the first style to the lower face and the second style to the upper face. The initial size of the simulation box is set by the *read_data*, *read_restart*, or *create_box* commands.

The style *p* means the box is periodic, so that particles interact across the boundary, and they can exit one end of the box and re-enter the other end. A periodic dimension can change in size due to constant pressure boundary conditions or box deformation (see the *fix npt* and *fix deform* commands). The *p* style must be applied to both faces of a dimension. For 2d simulations the z dimension must be periodic (which is the default).

The styles *f*, *s*, and *m* mean the box is non-periodic, so that particles do not interact across the boundary and do not move from one side of the box to the other.

For style *f*, the position of the face is fixed. If an atom moves outside the face it will be deleted on the next timestep that reneighboring occurs. This will typically generate an error unless you have set the *thermo_modify lost* option to allow for lost atoms.

For style *s*, the position of the face is set so as to encompass the atoms in that dimension (shrink-wrapping), no matter how far they move. Note that when the difference between the current box dimensions and the shrink-wrap box dimensions is large, this can lead to lost atoms at the beginning of a run when running in parallel. This is due to the large change in the (global) box dimensions also causing significant changes in the individual subdomain sizes. If these changes are farther than the communication cutoff, atoms will be lost. This is best addressed by setting initial box dimensions to match the shrink-wrapped dimensions more closely, by using *m* style boundaries (see below).

For style *m*, shrink-wrapping occurs, but is bounded by the value specified in the data or restart file or set by the *create_box* command. For example, if the upper z face has a value of 50.0 in the data file, the face will always be positioned at 50.0 or above, even if the maximum z-extent of all the atoms becomes less than 50.0. This can be useful if you start a simulation with an empty box or if you wish to leave room on one side of the box, e.g. for atoms to evaporate from a surface.

LAMMPS also allows use of triclinic (non-orthogonal) simulation boxes. See the *Howto triclinic* page for a description of both general and restricted triclinic boxes and how to define them. General triclinic boxes (arbitrary edge vectors **A**, **B**, and **C**) are converted internally to restricted triclinic boxes with tilt factors (xy,xz,yz) which skew an otherwise orthogonal box.

The boundary <boundary> command settings explained above for the 6 faces of an orthogonal box also apply in similar manner to the 6 faces of a restricted triclinic box (and thus to the corresponding 6 faces of a general triclinic box), with the following context.

if the second dimension of a tilt factor (e.g. y for xy) is periodic, then the periodicity is enforced with the tilt factor offset. This means that for y periodicity a particle which exits the lower y boundary is displaced in the x-direction by xy before it re-enters the upper y boundary. And vice versa if a particle exits the upper y boundary. Likewise the ghost atoms surrounding a particle near the lower y boundary include images of particles near the upper y-boundary which are displaced in the x-direction by xy. Similar rules apply for z-periodicity and the xz and/or yz tilt factors.

If the first dimension of a tilt factor is shrink-wrapped, then the shrink wrapping is applied to the tilted box face, to encompass the atoms. E.g. for a positive xy tilt, the xlo and xhi faces of the box are planes tilting in the +y direction as y increases. The position of these tilted planes are adjusted dynamically to shrink-wrap around the atoms to determine the xlo and xhi extents of the box.

### 1.10.4 Restrictions

This command cannot be used after the simulation box is defined by a *read_data* or *create_box* command or *read_restart* command. See the *change_box* command for how to change the simulation box boundaries after it has been defined.

For 2d simulations, the z dimension must be periodic.

### 1.10.5 Related commands

See the *thermo_modify* command for a discussion of lost atoms.

### 1.10.6 Default

```
boundary p p p
```

## 1.11 change_box command

### 1.11.1 Syntax

```
change_box group-ID parameter args ... keyword args ...
```

- group-ID = ID of group of atoms to (optionally) displace
- one or more parameter/arg pairs may be appended

```
parameter = x or y or z or xy or xz or yz or boundary or ortho or triclinic or set␣
↪or remap
  x, y, z args = style value(s)
    style = final or delta or scale or volume
      final values = lo hi
        lo hi = box boundaries after displacement (distance units)
      delta values = dlo dhi
        dlo dhi = change in box boundaries after displacement (distance units)
      scale values = factor
        factor = multiplicative factor for change in box length after displacement
      volume value = none = adjust this dim to preserve volume of system
  xy, xz, yz args = style value
    style = final or delta
      final value = tilt
        tilt = tilt factor after displacement (distance units)
      delta value = dtilt
        dtilt = change in tilt factor after displacement (distance units)
  boundary args = x y z
    x,y,z = p or s or f or m, one or two letters
    p is periodic
    f is non-periodic and fixed
    s is non-periodic and shrink-wrapped
    m is non-periodic and shrink-wrapped with a minimum value
  ortho args = none = change box to orthogonal
  triclinic args = none = change box to triclinic
  set args = none = store state of current box
  remap args = none = remap atom coords from last saved state to current box
```

- zero or more keyword/value pairs may be appended
- keyword = *units*

```
units value = lattice or box
  lattice = distances are defined in lattice units
  box = distances are defined in simulation box units
```

### 1.11.2 Examples

```
change_box all xy final -2.0 z final 0.0 5.0 boundary p p f remap units box
change_box all x scale 1.1 y volume z volume remap
```

### 1.11.3 Description

Change the volume and/or shape and/or boundary conditions for the simulation box. Orthogonal simulation boxes have 3 adjustable size parameters (x,y,z). Triclinic (non-orthogonal) simulation boxes have 6 adjustable size/shape parameters (x,y,z,xy,xz,yz). Any or all of them can be adjusted independently by this command. Thus it can be used to expand or contract a box, or to apply a shear strain to a non-orthogonal box. It can also be used to change the boundary conditions for the simulation box, similar to the *boundary* command.

The size and shape of the initial simulation box are specified by the *create_box* or *read_data* or *read_restart* command used to setup the simulation. The size and shape may be altered by subsequent runs, e.g. by use of the *fix npt* or *fix deform* commands. The *create_box*, *read data*, and *read_restart* commands also determine whether the simulation box is orthogonal or triclinic and their doc pages explain the meaning of the xy,xz,yz tilt factors.

See the *Howto triclinic* page for a geometric description of triclinic boxes, as defined by LAMMPS, and how to transform these parameters to and from other commonly used triclinic representations.

The keywords used in this command are applied sequentially to the simulation box and the atoms in it, in the order specified.

Before the sequence of keywords are invoked, the current box size/shape is stored, in case a *remap* keyword is used to map the atom coordinates from a previously stored box size/shape to the current one.

After all the keywords have been processed, any shrink-wrap boundary conditions are invoked (see the *boundary* command) which may change simulation box boundaries, and atoms are migrated to new owning processors.

**Note**

This means that you cannot use the change_box command to enlarge a shrink-wrapped box, e.g. to make room to insert more atoms via the *create_atoms* command, because the simulation box will be re-shrink-wrapped before the change_box command completes. Instead you could do something like this, assuming the simulation box is non-periodic and atoms extend from 0 to 20 in all dimensions:

```
change_box all x final -10 20
create_atoms 1 single -5 5 5       # this will fail to insert an atom

change_box all x final -10 20 boundary f s s
create_atoms 1 single -5 5 5
change_box all boundary s s s      # this will work
```

**Note**

Unlike the earlier "displace_box" version of this command, atom remapping is NOT performed by default. This command allows remapping to be done in a more general way, exactly when you specify it (zero or more times) in the sequence of transformations. Thus if you do not use the *remap* keyword, atom coordinates will not be changed even if the box size/shape changes. If a uniformly strained state is desired, the *remap* keyword should be specified.

**Note**

It is possible to lose atoms with this command. E.g. by changing the box without remapping the atoms, and having atoms end up outside of non-periodic boundaries. It is also possible to alter bonds between atoms straddling a boundary in bad ways. E.g. by converting a boundary from periodic to non-periodic. It is also possible when remapping atoms to put them (nearly) on top of each other. E.g. by converting a boundary from non-periodic to periodic. All of these will typically lead to bad dynamics and/or generate error messages.

Changed in version 4Jul2026.

Atoms that end up outside of non-periodic boundaries as a result of this command are now handled according to the *thermo_modify lost* setting, the same policy used to detect lost atoms during a run: with the default *error* setting the command stops with a "lost atoms" error, *warn* prints a warning and deletes the atoms, and *ignore* deletes them silently. Previously such atoms were only reported with a warning and could be left in an inconsistent state (for example causing a subsequent *write_data* command to write a corrupted data file).

**Note**

The simulation box size/shape can be changed by arbitrarily large amounts by this command. This is not a problem, except that the mapping of processors to the simulation box is not changed from its initial 3d configuration; see the *processors* command. Thus, if the box size/shape changes dramatically, the mapping of processors to the simulation box may not end up as optimal as the initial mapping attempted to be. You may wish to re-balance the atoms by using the *balance* command if that is the case.

**Note**

You cannot use this command after reading a restart file (and before a run is performed) if the restart file stored per-atom information from a fix and any of the specified keywords change the box size or shape or boundary conditions. This is because atoms may be moved to new processors and the restart info will not migrate with them. LAMMPS will generate an error if this could happen. Only the *ortho* and *triclinic* keywords do not trigger this error. One solution is to perform a "run 0" command before using the change_box command. This clears the per-atom restart data, whether it has been re-assigned to a new fix or not.

**Note**

Because the keywords used in this command are applied one at a time to the simulation box and the atoms in it, care must be taken with triclinic cells to avoid exceeding the limits on skew after each transformation in the sequence. If skew is exceeded before the final transformation this can be avoided by changing the order of the sequence, or breaking the transformation into two or more smaller transformations. For more information on the allowed limits for box skew see the discussion on triclinic boxes on *Howto triclinic* doc page.

---

For the *x*, *y*, and *z* parameters, this is the meaning of their styles and values.

For style *final*, the final lo and hi box boundaries of a dimension are specified. The values can be in lattice or box distance units. See the discussion of the units keyword below.

For style *delta*, plus or minus changes in the lo/hi box boundaries of a dimension are specified. The values can be in lattice or box distance units. See the discussion of the units keyword below.

For style *scale*, a multiplicative factor to apply to the box length of a dimension is specified. For example, if the initial box length is 10, and the factor is 1.1, then the final box length will be 11. A factor less than 1.0 means compression.

The *volume* style changes the specified dimension in such a way that the overall box volume remains constant with respect to the operation performed by the preceding keyword. The *volume* style can only be used following a keyword that changed the volume, which is any of the *x*, *y*, *z* keywords. If the preceding keyword "key" had a *volume* style, then both it and the current keyword apply to the keyword preceding "key". I.e. this sequence of keywords is allowed:

```
change_box all x scale 1.1 y volume z volume
```

The *volume* style changes the associated dimension so that the overall box volume is unchanged relative to its value before the preceding keyword was invoked.

If the following command is used, then the z box length will shrink by the same 1.1 factor the x box length was increased by:

```
change_box all x scale 1.1 z volume
```

If the following command is used, then the y,z box lengths will each shrink by sqrt(1.1) to keep the volume constant. In this case, the y,z box lengths shrink so as to keep their relative aspect ratio constant:

```
change_box all x scale 1.1 y volume z volume
```

If the following command is used, then the final box will be a factor of 10% larger in x and y, and a factor of 21% smaller in z, so as to keep the volume constant:

```
change_box all x scale 1.1 z volume y scale 1.1 z volume
```

**Note**

For solids or liquids, when one dimension of the box is expanded, it may be physically undesirable to hold the other 2 box lengths constant since that implies a density change. For solids, adjusting the other dimensions via the *volume* style may make physical sense (just as for a liquid), but may not be correct for materials and potentials whose Poisson ratio is not 0.5.

For the *scale* and *volume* styles, the box length is expanded or compressed around its mid point.

---

For the *xy*, *xz*, and *yz* parameters, this is the meaning of their styles and values. Note that changing the tilt factors of a triclinic box does not change its volume.

For style *final*, the final tilt factor is specified. The value can be in lattice or box distance units. See the discussion of the units keyword below.

For style *delta*, a plus or minus change in the tilt factor is specified. The value can be in lattice or box distance units. See the discussion of the units keyword below.

All of these styles change the xy, xz, yz tilt factors. In LAMMPS, tilt factors (xy,xz,yz) for triclinic boxes are required to be no more than half the distance of the parallel box length. For example, if xlo = 2 and xhi = 12, then the x box length is 10 and the xy tilt factor must be between -5 and 5. Similarly, both xz and yz must be between -(xhi-xlo)/2 and +(yhi-ylo)/2. Note that this is not a limitation, since if the maximum tilt factor is 5 (as in this example), then configurations with tilt = . . . , -15, -5, 5, 15, 25, . . . are all equivalent. Any tilt factor specified by this command must be within these limits.

---

The *boundary* keyword takes arguments that have exactly the same meaning as they do for the *boundary* command. In each dimension, a single letter assigns the same style to both the lower and upper face of the box. Two letters assigns the first style to the lower face and the second style to the upper face.

The style *p* means the box is periodic; the other styles mean non-periodic. For style *f*, the position of the face is fixed. For style *s*, the position of the face is set so as to encompass the atoms in that dimension (shrink-wrapping), no matter how far they move. For style *m*, shrink-wrapping occurs, but is bounded by the current box edge in that dimension, so that the box will become no smaller. See the *boundary* command for more explanation of these style options.

Note that the "boundary" command itself can only be used before the simulation box is defined via a *read_data* or *create_box* or *read_restart* command. This command allows the boundary conditions to be changed later in your input

script. Also note that the *read_restart* will change boundary conditions to match what is stored in the restart file. So if you wish to change them, you should use the change_box command after the read_restart command.

**Note**

Changing a periodic boundary to a non-periodic one will also cause the image flag for that dimension of all atoms to be reset to 0. LAMMPS will print a warning message, if that happens. Please note that this reset can lead to undesired changes when atoms are involved in bonded interactions that straddle periodic boundaries and thus the values of the image flag differs for those atoms.

---

The *ortho* and *triclinic* keywords convert the simulation box to be orthogonal or triclinic (non-orthogonal).

The simulation box is defined as either orthogonal or triclinic when it is created via the *create_box*, *read_data*, or *read_restart* commands.

These keywords allow you to toggle the existing simulation box from orthogonal to triclinic and vice versa. For example, an initial equilibration simulation can be run in an orthogonal box, the box can be toggled to triclinic, and then a *non-equilibrium MD (NEMD) simulation* can be run with deformation via the *fix deform* command.

If the simulation box is currently triclinic and has non-zero tilt in xy, yz, or xz, then it cannot be converted to an orthogonal box.

---

The *set* keyword saves the current box size/shape. This can be useful if you wish to use the *remap* keyword more than once or if you wish it to be applied to an intermediate box size/shape in a sequence of keyword operations. Note that the box size/shape is saved before any of the keywords are processed, i.e. the box size/shape at the time the create_box command is encountered in the input script.

The *remap* keyword remaps atom coordinates from the last saved box size/shape to the current box state. For example, if you stretch the box in the x dimension or tilt it in the xy plane via the *x* and *xy* keywords, then the *remap* command will dilate or tilt the atoms to conform to the new box size/shape, as if the atoms moved with the box as it deformed.

Note that this operation is performed without regard to periodic boundaries. Also, any shrink-wrapping of non-periodic boundaries (see the *boundary* command) occurs after all keywords, including this one, have been processed.

Only atoms in the specified group are remapped.

---

The *units* keyword determines the meaning of the distance units used to define various arguments. A *box* value selects standard distance units as defined by the *units* command, e.g. Angstroms for units = real or metal. A *lattice* value means the distance units are in lattice spacings. The *lattice* command must have been previously used to define the lattice spacing.

---

### 1.11.4 Restrictions

If you use the *ortho* or *triclinic* keywords, then at the point in the input script when this command is issued, no *dumps* can be active, nor can a *fix deform* be active. This is because these commands test whether the simulation box is orthogonal when they are first issued. Note that these commands can be used in your script before a change_box command is issued, so long as an *undump* or *unfix* command is also used to turn them off.

### 1.11.5 Related commands

*fix deform*, *boundary*

### 1.11.6 Default

The option default is units = lattice.

## 1.12 clear command

### 1.12.1 Syntax

```
clear
```

### 1.12.2 Examples

```
# (commands for 1st simulation)
clear
# (commands for 2nd simulation)
```

### 1.12.3 Description

This command deletes all atoms, restores all settings to their default values, and frees all memory allocated by LAMMPS. Once a clear command has been executed, it is almost as if LAMMPS is completely reset, with some exceptions noted below. The command thus allows to run multiple jobs sequentially from a single input script, often with a loop.

The following settings are not affected by a clear command:

- working directory (*shell* command)
- log file status (*log* command)
- echo status (*echo* command)
- input script variables except for *atomfile* style variables (*variable* command).

### 1.12.4 Restrictions

none

### 1.12.5 Related commands

*label*, *jump*, *next*

### 1.12.6 Default

none

## 1.13 comm_modify command

### 1.13.1 Syntax

```
comm_modify keyword value ...
```

- one or more keyword/value pairs may be appended
- keyword = *mode* or *cutoff* or *cutoff/multi* or *group* or *reduce/multi* or *vel*

```
mode value = single or multi = communicate atoms within a single or multiple␣
↪distances
cutoff value = Rcut (distance units) = communicate atoms from this far away
cutoff/multi collection value
   collection = atom collection or collection range (supports asterisk notation)
   value = Rcut (distance units) = communicate atoms for selected types from this␣
↪far away
reduce/multi arg = none = reduce number of communicated ghost atoms for multi style
group value = group-ID = only communicate atoms in the group
vel value = yes or no = do or do not communicate velocity info with ghost atoms
```

### 1.13.2 Examples

```
comm_modify mode multi reduce/multi
comm_modify mode multi group solvent
comm_modify mode multi cutoff/multi 1 10.0 cutoff/multi 2*4 15.0
comm_modify vel yes
comm_modify mode single cutoff 5.0 vel yes
comm_modify cutoff/multi * 0.0
```

### 1.13.3 Description

This command sets parameters that affect the inter-processor communication of atom information that occurs each timestep as coordinates and other properties are exchanged between neighboring processors and stored as properties of ghost atoms.

**Note**

These options apply to the currently defined comm style. When you specify a *comm_style* or *read_restart* command, all communication settings are restored to their default or stored values, including those previously reset by a comm_modify command. Thus if your input script specifies a comm_style or read_restart command, you should use the comm_modify command after it.

The *mode* keyword determines whether a single or multiple cutoff distances are used to determine which atoms to communicate.

The default mode is *single* which means each processor acquires information for ghost atoms that are within a single distance from its subdomain. The distance is by default the maximum of the neighbor cutoff across all atom type pairs.

For many systems this is an efficient algorithm, but for systems with widely varying cutoffs for different type pairs, the *multi* mode can be faster. In *multi*, each atom is assigned to a collection which should correspond to a set of atoms with similar interaction cutoffs. See the *neighbor* command for a detailed description of collections. In this case, each atom collection is assigned its own distance cutoff for communication purposes, and fewer atoms will be communicated. See the *neighbor multi* command for neighbor list construction options that may also be beneficial for simulations of this kind. The *multi* communication mode is only compatible with the *multi* neighbor style.

The *cutoff* keyword allows you to extend the ghost cutoff distance for communication mode *single*, which is the distance from the borders of a processor's subdomain at which ghost atoms are acquired from other processors. By default the ghost cutoff = neighbor cutoff = pairwise force cutoff + neighbor skin. See the *neighbor* command for more information about the skin distance. If the specified Rcut is greater than the neighbor cutoff, then extra ghost atoms will be acquired. If the provided cutoff is smaller, the provided value will be ignored, the ghost cutoff is set to the neighbor cutoff and

a warning will be printed. Specifying a cutoff value of 0.0 will reset any previous value to the default. If bonded interactions exist and equilibrium bond length information is available, then also a heuristic based on that bond length is computed. It is used as communication cutoff, if there is no pair style present and no *comm_modify cutoff* command used. Otherwise a warning is printed, if this bond based estimate is larger than the communication cutoff used.

The *cutoff/multi* option is equivalent to *cutoff*, but applies to communication mode *multi* instead. Since the communication cutoffs are determined per atom collections, a collection specifier is needed and cutoff for one or multiple collections can be extended. Also ranges of collections using the usual asterisk notation can be given. Collections are indexed from 1 to N where N is the total number of collections. Note that the arguments for *cutoff/multi* are parsed right before each simulation to account for potential changes in the number of collections. Custom cutoffs are preserved between runs but if collections are redefined, one may want to re-specify the communication cutoffs. For granular pair styles,the default cutoff is set to the sum of the current maximum atomic radii for each collection.

The *reduce/multi* option applies to *multi* and sets the communication cutoff for a particle equal to the maximum interaction distance between particles in the same collection. This reduces the number of ghost atoms that need to be communicated. This method is only compatible with the *multi* neighbor style and requires a half neighbor list and Newton on. See the *neighbor multi* command for more information.

These are simulation scenarios in which it may be useful or even necessary to set a ghost cutoff > neighbor cutoff:

- a single polymer chain with bond interactions, but no pairwise interactions
- bonded interactions (e.g. dihedrals) extend further than the pairwise cutoff
- ghost atoms beyond the pairwise cutoff are needed for some computation

In the first scenario, a pairwise potential is not defined. Thus the pairwise neighbor cutoff will be 0.0. But ghost atoms are still needed for computing bond, angle, etc interactions between atoms on different processors, or when the interaction straddles a periodic boundary.

The appropriate ghost cutoff depends on the *newton bond* setting. For newton bond *off*, the distance needs to be the furthest distance between any two atoms in the bond, angle, etc. E.g. the distance between 1-4 atoms in a dihedral. For newton bond *on*, the distance between the central atom in the bond, angle, etc and any other atom is sufficient. E.g. the distance between 2-4 atoms in a dihedral.

In the second scenario, a pairwise potential is defined, but its neighbor cutoff is not sufficiently long enough to enable bond, angle, etc terms to be computed. As in the previous scenario, an appropriate ghost cutoff should be set.

In the last scenario, a *fix* or *compute* or *pairwise potential* needs to calculate with ghost atoms beyond the normal pairwise cutoff for some computation it performs (e.g. locate neighbors of ghost atoms in a manybody pair potential). Setting the ghost cutoff appropriately can ensure it will find the needed atoms.

**Note**

In these scenarios, if you do not set the ghost cutoff long enough, and if there is only one processor in a periodic dimension (e.g. you are running in serial), then LAMMPS may "find" the atom it is looking for (e.g. the partner atom in a bond), that is on the far side of the simulation box, across a periodic boundary. This will typically lead to bad dynamics (i.e. the bond length is now the simulation box length). To detect if this is happening, see the *neigh_modify cluster* command.

The *group* keyword will limit communication to atoms in the specified group. This can be useful for models where no ghost atoms are needed for some kinds of particles. All atoms (not just those in the specified group) will still migrate to new processors as they move. The group specified with this option must also be specified via the *atom_modify first* command.

The *vel* keyword enables velocity information to be communicated with ghost particles. Depending on the *atom_style*, velocity info includes the translational velocity, angular velocity, and angular momentum of a particle. If the *vel* option is set to *yes*, then ghost atoms store these quantities; if *no* then they do not. The *yes* setting is needed by some pair

styles which require the velocity state of both the I and J particles to compute a pairwise I,J interaction, as well as by some compute and fix commands.

Note that if the *fix deform* command is being used with its "remap v" option enabled, then the velocities for ghost atoms (in the fix deform group) mirrored across a periodic boundary will also include components due to any velocity shift that occurs across that boundary (e.g. due to dilation or shear).

### 1.13.4 Restrictions

Communication mode *multi* is currently only available for *comm_style* *brick*.

### 1.13.5 Related commands

*comm_style*, *neighbor*

### 1.13.6 Default

The option defaults are mode = single, group = all, cutoff = 0.0, vel = no. The cutoff default of 0.0 means that ghost cutoff = neighbor cutoff = pairwise force cutoff + neighbor skin.

## 1.14 comm_style command

### 1.14.1 Syntax

```
comm_style style
```

- style = *brick* or *tiled*

### 1.14.2 Examples

```
comm_style brick
comm_style tiled
```

### 1.14.3 Description

This command sets the style of inter-processor communication of atom information that occurs each timestep as coordinates and other properties are exchanged between neighboring processors and stored as properties of ghost atoms.

For the default *brick* style, the domain decomposition used by LAMMPS to partition the simulation box must be a regular 3d grid of bricks, one per processor. Each processor communicates with its 6 Cartesian neighbors in the grid to acquire information for nearby atoms.

For the *tiled* style, a more general domain decomposition can be used, as triggered by the *balance* or *fix balance* commands. The simulation box can be partitioned into non-overlapping rectangular-shaped "tiles" or varying sizes and shapes. Again there is one tile per processor. To acquire information for nearby atoms, communication must now be done with a more complex pattern of neighboring processors.

Note that this command does not actually define a partitioning of the simulation box (a domain decomposition), rather it determines what kinds of decompositions are allowed and the pattern of communication used to enable the decomposition. A decomposition is created when the simulation box is first created, via the *create_box* or *read_data* or *read_restart* commands. For both the *brick* and *tiled* styles, the initial decomposition will be the same, as described by *create_box* and *processors* commands. The decomposition can be changed via the *balance* or *fix balance* commands.

### 1.14.4 Restrictions

none

### 1.14.5 Related commands

*comm_modify*, *processors*, *balance*, *fix balance*

### 1.14.6 Default

The default style is brick.

## 1.15 compute command

### 1.15.1 Syntax

```
compute ID group-ID style args
```

- ID = user-assigned name for the computation
- group-ID = ID of the group of atoms to perform the computation on
- style = one of a list of possible style names (see below)
- args = arguments used by a particular style

### 1.15.2 Examples

```
compute 1 all temp
compute newtemp flow temp/partial 1 1 0
compute 3 all ke/atom
```

### 1.15.3 Description

Define a diagnostic computation that will be performed on a group of atoms. Quantities calculated by a compute are instantaneous values, meaning they are calculated from information about atoms on the current timestep or iteration, though internally a compute may store some information about a previous state of the system. Defining a compute does not perform the computation. Instead computes are invoked by other LAMMPS commands as needed (e.g., to calculate a temperature needed for a thermostat fix or to generate thermodynamic or dump file output). See the *Howto output* page for a summary of various LAMMPS output options, many of which involve computes.

The ID of a compute can only contain alphanumeric characters and underscores.

---

Computes calculate and store any of four *styles* of quantities: global, per-atom, local, or per-grid.

A global quantity is one or more system-wide values, e.g. the temperature of the system. A per-atom quantity is one or more values per atom, e.g. the kinetic energy of each atom. Per-atom values are set to 0.0 for atoms not in the specified compute group. Local quantities are calculated by each processor based on the atoms it owns, but there may be zero or more per atom, e.g. a list of bond distances. Per-grid quantities are calculated on a regular 2d or 3d grid which overlays a 2d or 3d simulation domain. The grid points and the data they store are distributed across processors; each processor owns the grid points which fall within its subdomain.

As a general rule of thumb, computes that produce per-atom quantities have the word "atom" at the end of their style, e.g. *ke/atom*. Computes that produce local quantities have the word "local" at the end of their style, e.g. *bond/local*.

Computes that produce per-grid quantities have the word "grid" at the end of their style, e.g. *property/grid*. And styles with neither "atom" or "local" or "grid" at the end of their style name produce global quantities.

Global, per-atom, local, and per-grid quantities can also be of three *kinds*: a single scalar value (global only), a vector of values, or a 2d array of values. For per-atom, local, and per-grid quantities, a "vector" means a single value for each atom, each local entity (e.g. bond), or grid cell. Likewise an "array", means multiple values for each atom, each local entity, or each grid cell.

Note that a single compute can produce any combination of global, per-atom, local, or per-grid values. Likewise it can produce any combination of scalar, vector, or array output for each style. The exception is that for per-atom, local, and per-grid output, either a vector or array can be produced, but not both. The doc page for each compute explains the values it produces.

When a compute output is accessed by another input script command it is referenced via the following bracket notation, where ID is the ID of the compute:

| | |
|---|---|
| c_ID | entire scalar, vector, or array |
| c_ID[I] | one element of vector, one column of array |
| c_ID[I][J] | one element of array |

In other words, using one bracket reduces the dimension of the quantity once (vector → scalar, array → vector). Using two brackets reduces the dimension twice (array → scalar). Thus, for example, a command that uses global scalar compute values as input can also process elements of a vector or array. Depending on the command, this can either be done directly using the syntax in the table, or by first defining a *variable* of the appropriate style to store the quantity, then using the variable as an input to the command.

Note that commands and *variables* which take compute outputs as input typically do not allow for all styles and kinds of data (e.g., a command may require global but not per-atom values, or it may require a vector of values, not a scalar). This means there is typically no ambiguity about referring to a compute output as c_ID even if it produces, for example, both a scalar and vector. The doc pages for various commands explain the details, including how any ambiguities are resolved.

---

In LAMMPS, the values generated by a compute can be used in several ways:

- The results of computes that calculate a global temperature or pressure can be used by fixes that do thermostatting or barostatting or when atom velocities are created.
- Global values can be output via the *thermo_style custom* or *fix ave/time* command. Or the values can be referenced in a *variable equal* or *variable atom* command.
- Per-atom values can be output via the *dump custom* command. Or they can be time-averaged via the *fix ave/atom* command or reduced by the *compute reduce* command. Or the per-atom values can be referenced in an *atom-style variable*.
- Local values can be reduced by the *compute reduce* command, or histogrammed by the *fix ave/histo* command, or output by the *dump local* command.

The results of computes that calculate global quantities can be either "intensive" or "extensive" values. Intensive means the value is independent of the number of atoms in the simulation (e.g., temperature). Extensive means the value scales with the number of atoms in the simulation (e.g., total rotational kinetic energy). *Thermodynamic output* will normalize extensive values by the number of atoms in the system, depending on the "thermo_modify norm" setting. It will not normalize intensive values. If a compute value is accessed in another way (e.g., by a *variable*), you may want to know whether it is an intensive or extensive value. See the page for individual computes for further info.

---

LAMMPS creates its own computes internally for thermodynamic output. Three computes are always created, named "thermo_temp", "thermo_press", and "thermo_pe", as if these commands had been invoked in the input script:

```
compute thermo_temp all temp
compute thermo_press all pressure thermo_temp
compute thermo_pe all pe
```

Additional computes for other quantities are created if the thermo style requires it. See the documentation for the *thermo_style* command.

Fixes that calculate temperature or pressure, i.e. for thermostatting or barostatting, may also create computes. These are discussed in the documentation for specific *fix* commands.

In all these cases, the default computes LAMMPS creates can be replaced by computes defined by the user in the input script, as described by the *thermo_modify* and *fix modify* commands.

Properties of either a default or user-defined compute can be modified via the *compute_modify* command.

Computes can be deleted with the *uncompute* command.

Code for new computes can be added to LAMMPS; see the *Modify* page for details. The results of their calculations accessed in the various ways described above.

---

Each compute style has its own page which describes its arguments and what it does. Here is an alphabetic list of compute styles available in LAMMPS. They are also listed in more compact form on the *Commands compute* doc page.

There are also additional accelerated compute styles included in the LAMMPS distribution for faster performance on CPUs, GPUs, and KNLs. The individual style names on the *Commands compute* page are followed by one or more of (g,i,k,o,t) to indicate which accelerated styles exist.

- *ackland/atom* - determines the local lattice structure based on the Ackland formulation
- *adf* - angular distribution function of triples of atoms
- *aggregate/atom* - aggregate ID for each atom
- *angle* - energy of each angle sub-style
- *angle/local* - theta and energy of each angle
- *angmom/chunk* - angular momentum for each chunk
- *ave/sphere/atom* - compute local density and temperature around each atom
- *basal/atom* - calculates the hexagonal close-packed “c” lattice vector of each atom
- *body/local* - attributes of body sub-particles
- *bond* - energy of each bond sub-style
- *bond/local* - distance and energy of each bond
- *born/matrix* - second derivative or potential with respect to strain
- *centro/atom* - centro-symmetry parameter for each atom
- *centroid/stress/atom* - centroid based stress tensor for each atom
- *chunk/atom* - assign chunk IDs to each atom
- *chunk/spread/atom* - spreads chunk values to each atom in chunk
- *cluster/atom* - cluster ID for each atom
- *cna/atom* - common neighbor analysis (CNA) for each atom
- *cnp/atom* - common neighborhood parameter (CNP) for each atom

- *com* - center of mass of group of atoms
- *com/chunk* - center of mass for each chunk
- *composition/atom* - local composition for each atom
- *contact/atom* - contact count for each spherical particle
- *coord/atom* - coordination number for each atom
- *count/type* - count of atoms or bonds by type
- *damage/atom* - Peridynamic damage for each atom
- *dihedral* - energy of each dihedral sub-style
- *dihedral/local* - angle of each dihedral
- *dilatation/atom* - Peridynamic dilatation for each atom
- *dipole* - dipole vector and total dipole
- *dipole/chunk* - dipole vector and total dipole for each chunk
- *dipole/tip4p* - dipole vector and total dipole with TIP4P pair style
- *dipole/tip4p/chunk* - dipole vector and total dipole for each chunk with TIP4P pair style
- *displace/atom* - displacement of each atom
- *dpd* - total values of internal conductive energy, internal mechanical energy, chemical energy, and harmonic average of internal temperature
- *dpd/atom* - per-particle values of internal conductive energy, internal mechanical energy, chemical energy, and internal temperature
- *edpd/temp/atom* - per-atom temperature for each eDPD particle in a group
- *efield/atom* - electric field at each atom
- *efield/wolf/atom* - electric field at each atom
- *entropy/atom* - pair entropy fingerprint of each atom
- *erotate/asphere* - rotational energy of aspherical particles
- *erotate/rigid* - rotational energy of rigid bodies
- *erotate/sphere* - rotational energy of spherical particles
- *erotate/sphere/atom* - rotational energy for each spherical particle
- *event/displace* - detect event on atom displacement
- *fabric* - calculates fabric tensors from pair interactions
- *fep* - compute free energies for alchemical transformation from perturbation theory
- *fep/ta* - compute free energies for a test area perturbation
- *force/tally* - force between two groups of atoms via the tally callback mechanism
- *fragment/atom* - fragment ID for each atom
- *gaussian/grid/local* - local array of Gaussian atomic contributions on a regular grid
- *global/atom* - assign global values to each atom from arrays of global values
- *group/group* - energy/force between two groups of atoms
- *gyration* - radius of gyration of group of atoms

- *gyration/chunk* - radius of gyration for each chunk
- *gyration/shape* - shape parameters from gyration tensor
- *gyration/shape/chunk* - shape parameters from gyration tensor for each chunk
- *hbond/local* - identify hydrogen bonds
- *heat/flux* - heat flux through a group of atoms
- *heat/flux/tally* - heat flux through a group of atoms via the tally callback mechanism
- *heat/flux/virial/tally* - virial heat flux between two groups via the tally callback mechanism
- *hexorder/atom* - bond orientational order parameter q6
- *hma* - harmonically mapped averaging for atomic crystals
- *improper* - energy of each improper sub-style
- *improper/local* - angle of each improper
- *inertia/chunk* - inertia tensor for each chunk
- *ke* - translational kinetic energy
- *ke/atom* - kinetic energy for each atom
- *ke/atom/eff* - per-atom translational and radial kinetic energy in the electron force field model
- *ke/eff* - kinetic energy of a group of nuclei and electrons in the electron force field model
- *ke/rigid* - translational kinetic energy of rigid bodies
- *mliap* - gradients of energy and forces with respect to model parameters and related quantities for training machine learning interatomic potentials
- *momentum* - translational momentum
- *msd* - mean-squared displacement of group of atoms
- *msd/chunk* - mean-squared displacement for each chunk
- *msd/nongauss* - MSD and non-Gaussian parameter of group of atoms
- *nbond/atom* - calculates number of bonds per atom
- *omega/chunk* - angular velocity for each chunk
- *orientorder/atom* - Steinhardt bond orientational order parameters Ql
- *pace* - atomic cluster expansion descriptors and related quantities
- *pair* - values computed by a pair style
- *pair/local* - distance/energy/force of each pairwise interaction
- *pe* - potential energy
- *pe/atom* - potential energy for each atom
- *pe/mol/tally* - potential energy between two groups of atoms separated into intermolecular and intramolecular components via the tally callback mechanism
- *pe/tally* - potential energy between two groups of atoms via the tally callback mechanism
- *plasticity/atom* - Peridynamic plasticity for each atom
- *pod/atom* - POD descriptors for each atom
- *podd/atom* - derivative of POD descriptors for each atom

- *pod/local* - local POD descriptors and their derivatives
- *pod/global* - global POD descriptors and their derivatives
- *pressure* - total pressure and pressure tensor
- *pressure/alchemy* - mixed system total pressure and pressure tensor for *fix alchemy* runs
- *pressure/uef* - pressure tensor in the reference frame of an applied flow field
- *property/atom* - convert atom attributes to per-atom vectors/arrays
- *property/chunk* - extract various per-chunk attributes
- *property/grid* - convert per-grid attributes to per-grid vectors/arrays
- *property/local* - convert local attributes to local vectors/arrays
- *ptm/atom* - determines the local lattice structure based on the Polyhedral Template Matching method
- *rattlers/atom* - identify under-coordinated rattler atoms
- *rdf* - radial distribution function $g(r)$ histogram of group of atoms
- *reaxff/atom* - extract ReaxFF bond information
- *reduce* - combine per-atom quantities into a single global value
- *reduce/chunk* - reduce per-atom quantities within each chunk
- *reduce/region* - same as compute reduce, within a region
- *rheo/property/atom* - convert atom attributes in RHEO package to per-atom vectors/arrays
- *rigid/local* - extract rigid body attributes
- *saed* - electron diffraction intensity on a mesh of reciprocal lattice nodes
- *slcsa/atom* - perform Supervised Learning Crystal Structure Analysis (SL-CSA)
- *slice* - extract values from global vector or array
- *smd/contact/radius* - contact radius for Smooth Mach Dynamics
- *smd/damage* - damage status of SPH particles in Smooth Mach Dynamics
- *smd/hourglass/error* - error associated with approximated relative separation in Smooth Mach Dynamics
- *smd/internal/energy* - per-particle enthalpy in Smooth Mach Dynamics
- *smd/plastic/strain* - equivalent plastic strain per particle in Smooth Mach Dynamics
- *smd/plastic/strain/rate* - time rate of the equivalent plastic strain in Smooth Mach Dynamics
- *smd/rho* - per-particle mass density in Smooth Mach Dynamics
- *smd/tlsph/defgrad* - deformation gradient in Smooth Mach Dynamics
- *smd/tlsph/dt* - CFL-stable time increment per particle in Smooth Mach Dynamics
- *smd/tlsph/num/neighs* - number of particles inside the smoothing kernel radius for Smooth Mach Dynamics
- *smd/tlsph/shape* - current shape of the volume of a particle for Smooth Mach Dynamics
- *smd/tlsph/strain* - Green–Lagrange strain tensor for Smooth Mach Dynamics
- *smd/tlsph/strain/rate* - rate of strain for Smooth Mach Dynamics
- *smd/tlsph/stress* - per-particle Cauchy stress tensor for SPH particles

- *smd/triangle/vertices* - coordinates of vertices corresponding to the triangle elements of a mesh for Smooth Mach Dynamics
- *smd/ulsph/effm* - per-particle effective shear modulus
- *smd/ulsph/num/neighs* - number of neighbor particles inside the smoothing kernel radius for Smooth Mach Dynamics
- *smd/ulsph/strain* - logarithmic strain tensor for Smooth Mach Dynamics
- *smd/ulsph/strain/rate* - logarithmic strain rate for Smooth Mach Dynamics
- *smd/ulsph/stress* - per-particle Cauchy stress tensor and von Mises equivalent stress in Smooth Mach Dynamics
- *smd/vol* - per-particle volumes and their sum in Smooth Mach Dynamics
- *snap* - gradients of SNAP energy and forces with respect to linear coefficients and related quantities for fitting SNAP potentials
- *sna/atom* - bispectrum components for each atom
- *sna/grid* - global array of bispectrum components on a regular grid
- *sna/grid/local* - local array of bispectrum components on a regular grid
- *snad/atom* - derivative of bispectrum components for each atom
- *snav/atom* - virial contribution from bispectrum components for each atom
- *sph/e/atom* - per-atom internal energy of Smooth-Particle Hydrodynamics atoms
- *sph/rho/atom* - per-atom density of Smooth-Particle Hydrodynamics atoms
- *sph/t/atom* - per-atom internal temperature of Smooth-Particle Hydrodynamics atoms
- *spin* - magnetic quantities for a system of atoms having spins
- *stress/atom* - stress tensor for each atom
- *stress/cartesian* - stress tensor in cartesian coordinates
- *stress/cylinder* - stress tensor in cylindrical coordinates
- *stress/mop* - normal components of the local stress tensor using the method of planes
- *stress/mop/profile* - profile of the normal components of the local stress tensor using the method of planes
- *stress/spherical* - stress tensor in spherical coordinates
- *stress/tally* - stress between two groups of atoms via the tally callback mechanism
- *tdpd/cc/atom* - per-atom chemical concentration of a specified species for each tDPD particle
- *temp* - temperature of group of atoms
- *temp/asphere* - temperature of aspherical particles
- *temp/body* - temperature of body particles
- *temp/chunk* - temperature of each chunk
- *temp/com* - temperature after subtracting center-of-mass velocity
- *temp/cs* - temperature based on the center-of-mass velocity of atom pairs that are bonded to each other
- *temp/deform* - temperature excluding box deformation velocity
- *temp/deform/eff* - temperature excluding box deformation velocity in the electron force field model
- *temp/drude* - temperature of Core–Drude pairs

- *temp/eff* - temperature of a group of nuclei and electrons in the electron force field model
- *temp/partial* - temperature excluding one or more dimensions of velocity
- *temp/profile* - temperature excluding a binned velocity profile
- *temp/ramp* - temperature excluding ramped velocity component
- *temp/region* - temperature of a region of atoms
- *temp/region/eff* - temperature of a region of nuclei and electrons in the electron force field model
- *temp/rotate* - temperature of a group of atoms after subtracting out their center-of-mass and angular velocities
- *temp/sphere* - temperature of spherical particles
- *temp/uef* - kinetic energy tensor in the reference frame of an applied flow field
- *ti* - thermodynamic integration free energy values
- *torque/chunk* - torque applied on each chunk
- *vacf* - velocity auto-correlation function of group of atoms
- *vacf/chunk* - velocity auto-correlation for the center of mass velocities of chunks of atoms
- *vcm/chunk* - velocity of center-of-mass for each chunk
- *viscosity/cos* - velocity profile under cosine-shaped acceleration
- *voronoi/atom* - Voronoi volume and neighbors for each atom
- *xrd* - X-ray diffraction intensity on a mesh of reciprocal lattice nodes

### 1.15.4 Restrictions

none

### 1.15.5 Related commands

*uncompute*, *compute_modify*, *fix ave/atom*, *fix ave/time*, *fix ave/histo*

### 1.15.6 Default

none

## 1.16 compute_modify command

### 1.16.1 Syntax

```
compute_modify compute-ID keyword value ...
```

- compute-ID = ID of the compute to modify
- one or more keyword/value pairs may be listed
- keyword = *extra/dof* or *dynamic/dof* or *temp*

```
extra/dof value = N
  N = # of extra degrees of freedom to subtract
dynamic/dof value = yes or no
  yes/no = do or do not re-compute the number of degrees of freedom (DOF)␣
↪contributing to the temperature
```

*temp* value = compute ID that calculates a temperature

### 1.16.2 Examples

```
compute_modify myTemp extra/dof 0
compute_modify newtemp dynamic/dof yes extra/dof 600
```

### 1.16.3 Description

Modify one or more parameters of a previously defined compute. Not all compute styles support all parameters.

The *extra/dof* keyword refers to how many degrees of freedom are subtracted (typically from 3*N*) as a normalizing factor in a temperature computation. Only computes that compute a temperature use this option. The default is 2 or 3 for *2d or 3d systems* which is a correction factor for an ensemble of velocities with zero total linear momentum. For compute temp/partial, if one or more velocity components are excluded, the value used for *extra/dof* is scaled accordingly. You can use a negative number for the *extra/dof* parameter if you need to add degrees-of-freedom. See the *compute temp/asphere* command for an example.

The *dynamic/dof* keyword determines whether the number of atoms *N* in the compute group and their associated degrees of freedom (DOF) are re-computed each time a temperature is computed. Only compute styles that calculate a temperature use this option. By default, *N* and their DOF are assumed to be constant. If you are adding atoms or molecules to the system (see the *fix pour*, *fix deposit*, and *fix gcmc* commands) or expect atoms or molecules to be lost (e.g. due to exiting the simulation box or via *fix evaporate*), then this option should be used to ensure the temperature is correctly normalized.

Added in version 11Feb2026.

The *temp* keyword is used with *compute temp/deform* to change the internal temperature compute.

### 1.16.4 Restrictions

none

### 1.16.5 Related commands

*compute*

### 1.16.6 Default

The option defaults are extra/dof = 2 or 3 for 2d or 3d systems, respectively, and dynamic/dof = *no*.

## 1.17 create_atoms command

### 1.17.1 Syntax

```
create_atoms type style args keyword values ...
```

- type = atom type (1-Ntypes or type label) of atoms to create (offset for molecule creation)
- style = *box* or *region* or *single* or *mesh* or *random*

  *box* args = none
  *region* args = region-ID
    region-ID = particles will only be created if contained in the region
  *single* args = x y z
    x,y,z = coordinates of a single particle (distance units)

```
mesh args = STL-file
  STL-file = file with triangle mesh in STL format
random args = N seed region-ID
  N = number of particles to create
  seed = random # seed (positive integer)
  region-ID = create atoms within this region, use NULL for entire simulation box
```

- zero or more keyword/value pairs may be appended
- keyword = *mol* or *basis* or *ratio* or *subset* or *group* or *remap* or *var* or *set* or *radscale* or *meshmode* or *rotate* or *overlap* or *maxtry* or *units*

```
mol values = template-ID seed
  template-ID = ID of molecule template specified in a separate molecule command
  seed = random # seed (positive integer)
basis values = M itype
  M = which basis atom
  itype = atom type (1-Ntypes or type label) to assign to this basis atom
ratio values = frac seed
  frac = fraction of lattice sites (0 to 1) to populate randomly
  seed = random # seed (positive integer)
subset values = Nsubset seed
  Nsubset = # of lattice sites to populate randomly
  seed = random # seed (positive integer)
group value = group name
remap value = yes or no
var value = name = variable name to evaluate for test of atom creation
set values = dim name
  dim = x or y or z
  name = name of variable to set with x, y, or z atom position
radscale value = factor
  factor = scale factor for setting atom radius
meshmode values = mode arg
  mode = bisect or qrand
  bisect arg = radthresh
    radthresh = threshold value for mesh to determine when to split triangles␣
↪(distance units)
  qrand arg = density
    density = minimum number density for atoms place on mesh triangles (inverse␣
↪distance squared units)
rotate values = theta Rx Ry Rz
  theta = rotation angle for single molecule (degrees)
  Rx,Ry,Rz = rotation vector for single molecule
overlap value = Doverlap
  Doverlap = only insert if at least this distance from all existing atoms
maxtry value = Ntry
  Ntry = number of attempts to insert a particle before failure
units value = lattice or box
  lattice = the geometry is defined in lattice units
  box = the geometry is defined in simulation box units
```

### 1.17.2 Examples

```
create_atoms 1 box

labelmap atom 1 Pt
create_atoms Pt box

labelmap atom 1 C 2 Si
create_atoms C region regsphere basis 1 Si

create_atoms 2 region regsphere basis 2 3
create_atoms 1 region regsphere basis 2 2 ratio 0.5 74637
create_atoms 3 single 0 0 5 group newatom
create_atoms 1 box var v set x xpos set y ypos
create_atoms 2 random 50 12345 NULL overlap 2.0 maxtry 50
create_atoms 1 mesh open_box.stl meshmode qrand 0.1 units box
create_atoms 1 mesh funnel.stl meshmode bisect 4.0 units box radscale 0.9
```

### 1.17.3 Description

This command creates atoms (or molecules) within the simulation box, either on a lattice, or at random points, or on a surface defined by a triangulated mesh. Or it creates a single atom (or molecule) at a specified point. It is an alternative to reading in atom coordinates explicitly via a *read_data* or *read_restart* command.

To use this command a simulation box must already exist, which is typically created via the *create_box* command. Before using this command, a lattice must typically also be defined using the *lattice* command, unless you specify the *single* or *mesh* style with units = box or the *random* style. To create atoms on a lattice for general triclinic boxes, see the discussion below.

For the remainder of this doc page, a created atom or molecule is referred to as a "particle".

If created particles are individual atoms, they are assigned the specified atom *type*, though this can be altered via the *basis* keyword as discussed below. If molecules are being created, the type of each atom in the created molecule is specified in a specified file read by the *molecule* command, and those values are added to the specified atom *type* (e.g., if *type* = 2 and the file specifies atom types 1, 2, and 3, then each created molecule will have atom types 3, 4, and 5).

**Atoms outside the box**

You cannot use this command to create atoms that are outside the simulation box; they will just be ignored by LAMMPS. This is true even if you are using shrink-wrapped box boundaries, as specified by the *boundary* command. However, you can first use the *change_box* command to temporarily expand the box, then add atoms via create_atoms, then finally use change_box command again if needed to re-shrink-wrap the new atoms. See the *change_box* doc page for an example of how to do this, using the create_atoms *single* style to insert a new atom outside the current simulation box.

---

For the *box* style, the create_atoms command fills the entire simulation box with particles on the lattice. If your simulation box is periodic, you should ensure its size is a multiple of the lattice spacings, to avoid unwanted atom overlaps at the box boundaries. If your box is periodic and a multiple of the lattice spacing in a particular dimension, LAMMPS is careful to put exactly one particle at the boundary (on either side of the box), not zero or two.

For the *region* style, a geometric volume is filled with particles on the lattice. This volume is what is both inside the simulation box and also consistent with the region volume. See the *region* command for details. Note that a region can be specified so that its "volume" is either inside or outside its geometric boundary. Also note that if a region is the same size as a periodic simulation box (in some dimension), LAMMPS does NOT implement the same logic described

above for the *box* style, to ensure exactly one particle at periodic boundaries. If this is desired, you should either use the *box* style, or tweak the region size to get precisely the particles you want.

---

If the simulation box is formulated as a general triclinic box defined by arbitrary edge vectors **A**, **B**, **C**, then the *box* and *region* styles will create atoms on a lattice commensurate with those edge vectors. See the *Howto_triclinic* doc page for a detailed explanation of orthogonal, restricted triclinic, and general triclinic simulation boxes. As with the *create_box* command, the *lattice* command used by this command must be of style *custom* and use its *triclinic/general* option. The *a1, *a2*, *a3* settings of the *lattice* command define the edge vectors of a unit cell of the general triclinic lattice. The *create_box* command creates a simulation box which replicates that unit cell along each of the **A**, **B**, **C** edge vectors.

**Note**

LAMMPS allows specification of general triclinic simulation boxes as a convenience for users who may be converting data from solid-state crystallographic representations or from DFT codes for input to LAMMPS. However, as explained on the *Howto_triclinic* doc page, internally, LAMMPS only uses restricted triclinic simulation boxes. This means the box created by the *create_box* command as well as the atoms created by this command with their per-atom information (e.g. coordinates, velocities) are converted (rotated) from general to restricted triclinic form when the two commands are invoked. The *Howto_triclinic* doc page also discusses other LAMMPS commands which can input/output general triclinic representations of the simulation box and per-atom data.

The *box* style will fill the entire general triclinic box with particles on the lattice, as explained above.

**Note**

The *region* style also operates as explained above, but the check for particles inside the region is performed *after* the particle coordinates have been converted to the restricted triclinic box. This means the region must also be defined with respect to the restricted triclinic box, not the general triclinic box.

If the simulation box is general triclinic, the *single*, *random*, and *mesh* styles described next operate on the box *after* it has been converted to restricted triclinic. So all the settings for those styles should be made in that context.

---

For the *single* style, a single particle is added to the system at the specified coordinates. This can be useful for debugging purposes or to create a tiny system with a handful of particles at specified positions. For a 2d simulation the specified z coordinate must be 0.0.

Changed in version 2Jun2022.

The *porosity* style has been renamed to *random* with added functionality.

For the *random* style, *N* particles are added to the system at randomly generated coordinates, which can be useful for generating an amorphous system. For 2d simulations, the z coordinates of all added atoms will be 0.0.

The particles are created one by one using the specified random number *seed*, resulting in the same set of particle coordinates, independent of how many processors are being used in the simulation. Unless the *overlap* keyword is specified, particles created by the *random* style will typically be highly overlapped. Various additional criteria can be used to accept or reject a random particle insertion; see the keyword discussion below. Multiple attempts per particle are made (see the *maxtry* keyword) until the insertion is either successful or fails. If this command fails to add all requested *N* particles, a warning will be output.

If the *region-ID* argument is specified as NULL, then the randomly created particles will be anywhere in the simulation box. If a *region-ID* is specified, a geometric volume is filled that is both inside the simulation box and is also consistent

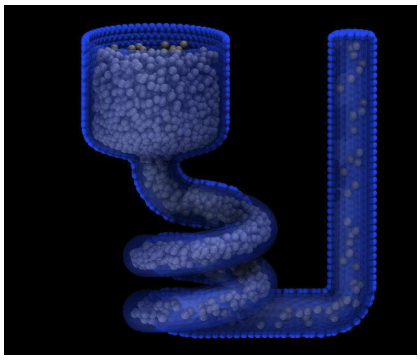
with the region volume. See the *region* command for details. Note that a region can be specified so that its "volume" is either inside or outside its geometric boundary.

Note that the create_atoms command adds particles to those that already exist. This means it can be used to add particles to a system previously read in from a data or restart file. Or the create_atoms command can be used multiple times, to add multiple sets of particles to the simulation. For example, grain boundaries can be created, by interleaving the create_atoms command with *lattice* commands specifying different orientations.

When this command is used, care should be taken to ensure the resulting system does not contain particles that are highly overlapped. Such overlaps will cause many interatomic potentials to compute huge energies and forces, leading to bad dynamics. There are several strategies to avoid this problem:

- Use the *delete_atoms overlap* command after create_atoms. For example, this strategy can be used to overlay and surround a large protein molecule with a volume of water molecules, then delete water molecules that overlap with the protein atoms.
- For the *random* style, use the optional *overlap* keyword to avoid overlaps when each new particle is created.
- Before running dynamics on an overlapped system, perform an *energy minimization*. Or run initial dynamics with *pair_style soft* or with *fix nve/limit* to un-overlap the particles, before running normal dynamics.

Added in version 2Jun2022.



For the *mesh* style, a file with a triangle mesh in STL format is read and one or more particles are placed into the area of each triangle. The reader supports both ASCII and binary files conforming to the format on the Wikipedia page. Binary STL files (e.g. as frequently offered for 3d-printing) can also be first converted to ASCII for editing with the *stl_bin2txt tool*. The use of the *units box* option is required. There are two algorithms available for placing atoms: *bisect* and *qrand*. They can be selected via the *meshmode* option; *bisect* is the default. If the atom style allows it, the radius will be set to a value depending on the algorithm and the value of the *radscale* parameter (see below), and the atoms created from the mesh are assigned a new molecule ID.

In *bisect* mode a particle is created at the center of each triangle unless the average distance of the triangle vertices from its center is larger than the *radthresh* value (default is lattice spacing in x-direction). In case the average distance is over the threshold, the triangle is recursively split into two halves along the the longest side until the threshold is reached. There will be at least one sphere per triangle. The value of *radthresh* is set as an argument to *meshmode bisect*. The average distance of the vertices from the center is also used to set the radius.

In *qrand* mode a quasi-random sequence is used to distribute particles on mesh triangles using an approach by *(Roberts)*. Particles are added to the triangle until the minimum number density is met or exceeded such that every triangle will have at least one particle. The minimum number density is set as an argument to the *qrand* option. The radius will be set so that the sum of the area of the radius of the particles created in place of a triangle will be equal to the area of that triangle.

**Note**

The atom placement algorithms in the *mesh* style benefit from meshes where triangles are close to equilateral. It is therefore recommended to pre-process STL files to optimize the mesh accordingly. There are multiple open source and commercial software tools available with the capability to generate optimized meshes.

**Note**

In most cases the atoms created in *mesh* style will become an immobile or rigid object that would not be time integrated or moved by *fix move* or *fix rigid*. For computational efficiency *and* to avoid undesired contributions to pressure and potential energy due to close contacts, it is usually beneficial to exclude computing interactions between the created particles using *neigh_modify exclude*.

---

Individual atoms are inserted by this command, unless the *mol* keyword is used. It specifies a *template-ID* previously defined using the *molecule* command, which reads a file that defines the molecule. The coordinates, atom types, charges, etc, as well as any bond/angle/etc and special neighbor information for the molecule can be specified in the molecule file. See the *molecule* command for details. The only settings required to be in this file are the coordinates and types of atoms in the molecule.

**Note**

If you are using the *mol* keyword in combination with the *atom style template* command, they must use the same molecule template-ID.

Using a lattice to add molecules, e.g. via the *box* or *region* or *single* styles, is exactly the same as adding atoms on lattice points, except that entire molecules are added at each point, i.e. on the point defined by each basis atom in the unit cell as it tiles the simulation box or region. This is done by placing the geometric center of the molecule at the lattice point, and (by default) giving the molecule a random orientation about the point. The random *seed* specified with the *mol* keyword is used for this operation, and the random numbers generated by each processor are different. This means the coordinates of individual atoms (in the molecules) will be different when running on different numbers of processors, unlike when atoms are being created in parallel.

Note that with random rotations, it may be important to use a lattice with a large enough spacing that adjacent molecules will not overlap, regardless of their relative orientations. See the description of the *rotate* keyword below, which overrides the default random orientation and inserts all molecules at a specified orientation.

**Note**

If the *create_box* command is used to create the simulation box, followed by the create_atoms command with its *mol* option for adding molecules, then you typically need to use the optional keywords allowed by the *create_box* command for extra bonds (angles,etc) or extra special neighbors. This is because by default, the *create_box* command sets up a non-molecular system that does not allow molecules to be added.

---

This is the meaning of the other optional keywords.

The *basis* keyword is only used when atoms (not molecules) are being created. It specifies an atom type that will be assigned to specific basis atoms as they are created that *overrides* the default atom type. See the *lattice* command for specifics on how basis atoms are defined for the unit cell of the lattice. By default, all created atoms are assigned the argument *type* as their atom type. To illustrate this please see the following 4 example *create_atom* command lines that all create the same system:

```
lattice diamond 4.36
create_atoms 1 box basis 1 1 basis 2 2 basis 3 1 basis 4 2 basis 5 1 basis 6 2 basis 7 1␣
↪basis 8 2
create_atoms 2 box basis 1 1 basis 2 2 basis 3 1 basis 4 2 basis 5 1 basis 6 2 basis 7 1␣
↪basis 8 2
```

(continues on next page)

(continued from previous page)

```
create_atoms 1 box basis 2 2 basis 4 2 basis 6 2 basis 8 2
create_atoms 2 box basis 1 1 basis 3 1 basis 5 1 basis 7 1
```

The *ratio* and *subset* keywords can be used in conjunction with the *box* or *region* styles to limit the total number of particles inserted. The lattice defines a set of *Nlatt* eligible sites for inserting particles, which may be limited by the *region* style or the *var* and *set* keywords. For the *ratio* keyword, only the specified fraction of them ($0 \leq f \leq 1$) will be assigned particles. For the *subset* keyword only the specified *Nsubset* of them will be assigned particles. In both cases the assigned lattice sites are chosen randomly. An iterative algorithm is used that ensures the correct number of particles are inserted, in a perfectly random fashion. Which lattice sites are selected will change with the number of processors used.

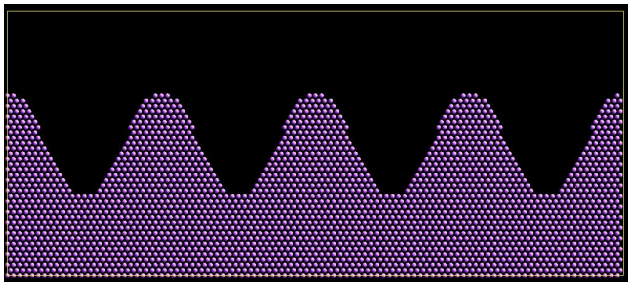
Added in version 12Jun2025.

The *group* keyword adds the newly created atoms to the named *group*. If the group does not yet exist it will be created. There can be only one such group, thus if the *group* keyword is used multiple times, only the last one will be used. All created atoms are always added to the group "all".

The *remap* keyword only applies to the *single* style. If it is set to *yes*, then if the specified position is outside the simulation box, it will mapped back into the box, assuming the relevant dimensions are periodic. If it is set to *no*, no remapping is done and no particle is created if its position is outside the box.

The *var* and *set* keywords can be used together to provide a criterion for accepting or rejecting the addition of an individual atom, based on its coordinates. They apply to all styles except *single*. The *name* specified for the *var* keyword is the name of an *equal-style variable* that should evaluate to a zero or non-zero value based on one or two or three variables that will store the $x$, $y$, or $z$ coordinates of an atom (one variable per coordinate). If used, these other variables must be specified by the *set* keyword. They are internal-style variable, because this command resets their values directly. The internal-style variables do not need to be defined in the input script (though they can be); if one (or more) is not defined, then the *set* option creates an *internal-style variable* with the specified name.

When an atom is about to be created, its $(x, y, z)$ coordinates become the values for any *set* variable that is defined. The *var* variable is then evaluated. If the returned value is 0.0, the atom is not created. If it is non-zero, the atom is created.



As an example, these commands can be used in a 2d simulation, to create a sinusoidal surface. Note that the surface is "rough" due to individual lattice points being "above" or "below" the mathematical expression for the sinusoidal curve. If a finer lattice were used, the sinusoid would appear to be "smoother". Also note the use of the "xlat" and "ylat" *thermo_style* keywords, which converts lattice spacings to distance.

```
dimension   2
variable    x equal 100
variable    y equal 25
lattice     hex 0.8442
region      box block 0 $x 0 $y -0.5 0.5
create_box  1 box

variable    v equal "(0.2*v_y*ylat * cos(v_xx/xlat * 2.0*PI*4.0/v_x) + 0.5*v_y*ylat - v_
→yy) > 0.0"
create_atoms  1 box var v set x xx set y yy
write_dump  all atom sinusoid.lammpstrj
```

The *rotate* keyword allows specification of the orientation at which molecules are inserted. The axis of rotation is determined by the rotation vector $(R_x, R_y, R_z)$ that goes through the insertion point. The specified *theta* determines the angle of rotation around that axis. Note that the direction of rotation for the atoms around the rotation axis is consistent with the right-hand rule: if your right-hand's thumb points along $R$, then your fingers wrap around the axis in the direction of rotation.

The *radscale* keyword only applies to the *mesh* style and adjusts the radius of created particles (see above), provided this is supported by the atom style. Its value is a prefactor (must be $> 0.0$, default is 1.0) that is applied to the atom radius inferred from the size of the individual triangles in the triangle mesh that the particle corresponds to.

Added in version 2Jun2022.

The *overlap* keyword only applies to the *random* style. It prevents newly created particles from being created closer than the specified *Doverlap* distance from any other particle. If particles have finite size (see *atom_style sphere* for example) *Doverlap* should be specified large enough to include the particle size in the non-overlapping criterion. If molecules are being randomly inserted, then an insertion is only accepted if each particle in the molecule meets the overlap criterion with respect to other particles (not including particles in the molecule itself).

**Note**

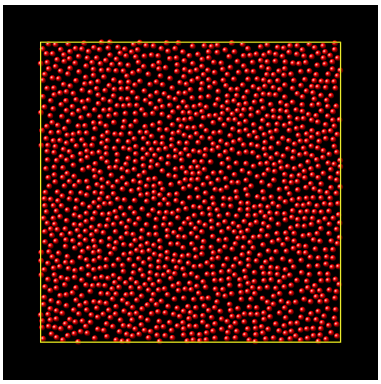
Checking for overlaps is a costly $\mathcal{O}(N(N+M))$ operation for inserting $N$ new particles into a system with $M$ existing particles. This is because distances to all $M$ existing particles are computed for each new particle that is added. Thus the intended use of this keyword is to add relatively small numbers of particles to systems that remain at a relatively low density even after the new particles are created. Careful use of the *maxtry* keyword in combination with *overlap* is recommended. See the discussion above about systems with overlapped particles for alternate strategies that allow for overlapped insertions.

Added in version 2Jun2022.

The *maxtry* keyword only applies to the *random* style. It limits the number of attempts to generate valid coordinates for a single new particle that satisfy all requirements imposed by the *region*, *var*, and *overlap* keywords. The default is 10 attempts per particle before the loop over the requested $N$ particles advances to the next particle. Note that if insertion success is unlikely (e.g., inserting new particles into a dense system using the *overlap* keyword), setting the *maxtry* keyword to a large value may result in this command running for a long time.

Here is an example for the *random* style using these commands

```
units         lj
dimension     2
region        box block 0 50 0 50 -0.5 0.5
create_box    1 box
create_atoms  1 random 2000 13487 NULL overlap 1.0 maxtry 50
pair_style    lj/cut 2.5
pair_coeff    1 1 1.0 1.0 2.5
```



to produce a system as shown in the image with 1520 particles (out of 2000 requested) that are moderately dense and which have no overlaps sufficient to prevent the LJ pair_style from running properly (because the overlap criterion is 1.0). The create_atoms command ran for 0.3 s on a single CPU core.

---

The *units* keyword determines the meaning of the distance units used by parameters for various styles. A *box* value selects standard distance units as defined by the *units* command (e.g., Å for units = *real* or *metal*. A *lattice* value means the distance units are in lattice spacings. These are affected settings:

- for *single* style: coordinates of the particle created

- for *random* style: overlap distance *Doverlap* by the *overlap* keyword
- for *mesh* style: *bisect* threshold value for *meshmode* = *bisect*
- for *mesh* style: *radthresh* value for *meshmode* = *bisect*
- for *mesh* style: *density* value for *meshmode* = *qrand*

Since *density* represents an area (distance ^2), the lattice spacing factor is also squared.

---

Atom IDs are assigned to created atoms in the following way. The collection of created atoms are assigned consecutive IDs that start immediately following the largest atom ID existing before the create_atoms command was invoked. This is done by the processor's communicating the number of atoms they each own, the first processor numbering its atoms from 1 to $N_1$, the second processor from $N_1$ + 1 to $N_2$, and so on, where $N_1$ is the number of atoms owned by the first processor, $N_2$ is the number owned by the second processor, and so forth. Thus, when the same simulation is performed on different numbers of processors, there is no guarantee a particular created atom will be assigned the same ID in both simulations. If molecules are being created, molecule IDs are assigned to created molecules in a similar fashion.

Aside from their ID, atom type, and *xyz* position, other properties of created atoms are set to default values, depending on which quantities are defined by the chosen *atom style*. See the *atom style* command for more details. See the *set* and *velocity* commands for info on how to change these values.

- charge = 0.0
- dipole moment magnitude = 0.0
- diameter = 1.0
- shape = 0.0 0.0 0.0
- density = 1.0
- volume = 1.0
- velocity = 0.0 0.0 0.0
- angular velocity = 0.0 0.0 0.0
- angular momentum = 0.0 0.0 0.0
- quaternion = (1,0,0,0)
- bonds, angles, dihedrals, impropers = none

If molecules are being created, these defaults can be overridden by values specified in the file read by the *molecule* command. That is, the file typically defines bonds (angles, etc.) between atoms in the molecule, and can optionally define charges on each atom.

Note that the *sphere* atom style sets the default particle diameter to 1.0 as well as the density. This means the mass for the particle is not 1.0, but is $\frac{\pi}{6}d^3 = 0.5236$, where $d$ is the diameter. When using the *mesh* style, the particle diameter is adjusted from the size of the individual triangles in the triangle mesh.

Note that the *ellipsoid* atom style sets the default particle shape to (0.0 0.0 0.0) and the density to 1.0, which means it is a point particle, not an ellipsoid, and has a mass of 1.0.

Note that the *peri* style sets the default volume and density to 1.0 and thus also set the mass for the particle to 1.0.

The *set* command can be used to override many of these default settings.

---

### 1.17.4 Restrictions

An *atom_style* must be previously defined to use this command.

A rotation vector specified for a single molecule must be in the z-direction for a 2d model.

For *molecule templates* that are created from multiple files, i.e. contain multiple molecule *sets*, only the first set is used. To create multiple molecules the files currently need to be merged and different molecule IDs assigned with a Molecules section.

### 1.17.5 Related commands

*lattice*, *region*, *create_box*, *read_data*, *read_restart*

### 1.17.6 Default

The default for the *basis* keyword is that all created atoms are assigned the argument *type* as their atom type (when single atoms are being created). The other defaults are *remap* = no, *rotate* = random, *radscale* = 1.0, *radthresh* = x-lattice spacing, *overlap* not checked, *maxtry* = 10, and *units* = lattice.

---

**(Roberts)** R. Roberts (2019) "Evenly Distributing Points in a Triangle." Extreme Learning. https://extremelearning.com.au/evenly-distributing-points-in-a-triangle/

## 1.18 create_bonds command

### 1.18.1 Syntax

```
create_bonds style args ... keyword value ...
```

- style = *many* or *single/bond* or *single/angle* or *single/dihedral* or *single/improper*

```
many args = group-ID group2-ID btype rmin rmax
  group-ID = ID of first group
  group2-ID = ID of second group, bonds will be between atoms in the 2 groups
  btype = bond type of created bonds
  rmin = minimum distance between pair of atoms to bond together
  rmax = maximum distance between pair of atoms to bond together
single/bond args = btype batom1 batom2
  btype = bond type of new bond
  batom1,batom2 = atom IDs for two atoms in bond
single/angle args = atype aatom1 aatom2 aatom3
  atype = angle type of new angle
  aatom1,aatom2,aatom3 = atom IDs for three atoms in angle
single/dihedral args = dtype datom1 datom2 datom3 datom4
  dtype = dihedral type of new dihedral
  datom1,datom2,datom3,datom4 = atom IDs for four atoms in dihedral
single/improper args = itype iatom1 iatom2 iatom3 iatom4
  itype = improper type of new improper
  iatom1,iatom2,iatom3,iatom4 = atom IDs for four atoms in improper
```

- zero or more keyword/value pairs may be appended
- keyword = *special*

```
special value = yes or no
```

### 1.18.2 Examples

```
create_bonds many all all 1 1.0 1.2
create_bonds many surf solvent 3 2.0 2.4
create_bonds single/bond 1 1 2
create_bonds single/angle 5 52 98 107 special no
create_bonds single/dihedral 2 4 19 27 101
create_bonds single/improper 3 23 26 31 57
```

### 1.18.3 Description

Create bonds between pairs of atoms that meet a specified distance criteria. Or create a single bond, angle, dihedral or improper between 2, 3, or 4 specified atoms.

The new bond (angle, dihedral, improper) interactions will then be computed during a simulation by the bond (angle, dihedral, improper) potential defined by the *bond_style*, *bond_coeff*, *angle_style*, *angle_coeff*, *dihedral_style*, *dihedral_coeff*, *improper_style*, *improper_coeff* commands.

The *many* style is useful for adding bonds to a system (e.g., between nearest neighbors in a lattice of atoms) without having to enumerate all the bonds in the data file read by the *read_data* command.

The *single* styles are useful for adding bonds, angles, dihedrals, and impropers to a system incrementally, then continuing a simulation.

Note that this command does not auto-create any angle, dihedral, or improper interactions when a bond is added, nor does it auto-create any bonds when an angle, dihedral, or improper is added. It also will not auto-create any angles when a dihedral or improper is added. Thus, the flexibility of this command is limited. It can be used several times to create different types of bond at different distances, but it cannot typically auto-create all the bonds or angles or dihedrals or impropers that would normally be defined in a data file for a complex system of molecules.

**Note**

If the system has no bonds (angles, dihedrals, impropers) to begin with, or if more bonds per atom are being added than currently exist, then you must ensure that the number of bond types and the maximum number of bonds per atom are set to large enough values, and similarly for angles, dihedrals, impropers, and special neighbors, otherwise an error may occur when too many bonds (angles, dihedrals, impropers) are added to an atom. If the *read_data* command is used to define the system, these parameters can be set via its optional *extra/bond/types*, *extra/bond/per/atom*, and similar keywords to the command. If the *create_box* command is used to define the system, these two parameters can be set via its optional *bond/types* and *extra/bond/per/atom* arguments, and similarly for angles, dihedrals, and impropers. See the corresponding documentation pages for these two commands for details.

---

The *many* style will create bonds between pairs of atoms $I, J$, where $I$ is in one of the two specified groups and $J$ is in the other. The two groups can be the same (e.g., group "all"). The created bonds will be of bond type *btype*, where *btype* must be a value between 1 and the number of bond types defined.

For a bond to be created, an $I, J$ pair of atoms must be a distance $D$ apart such that $r_{\min} \leq D \leq r_{\max}$.

The following settings must have been made in an input script before the *many* style is used:

- special_bonds weight for 1–2 interactions must be 0.0
- a *pair_style* must be defined
- no *kspace_style* defined
- minimum *pair_style* cutoff + *neighbor* skin $\geq r_{\max}$

These settings are required so that a neighbor list can be created to search for nearby atoms. Pairs of atoms that are already bonded cannot appear in the neighbor list, to avoid creation of duplicate bonds. The neighbor list for all atom type pairs must also extend to a distance that encompasses the *rmax* for new bonds to create. When using periodic boundary conditions, the box length in each periodic dimension must be larger than *rmax*, so that no bonds are created between the system and its own periodic image.

**Note**

If you want to create bonds between pairs of 1–3 or 1–4 atoms in the current bond topology, then you need to use *special_bonds lj 0 1 1* to ensure those pairs appear in the neighbor list. They will not appear with the default special_bonds settings, which are zero for 1–2, 1–3, and 1–4 atoms. 1–3 or 1–4 atoms are those which are two hops or three hops apart in the bond topology.

An additional requirement for this style is that your system must be ready to perform a simulation. This means, for example, that all *pair_style* coefficients be set via the *pair_coeff* command. A *bond_style* command and all bond coefficients must also be set, even if no bonds exist before this command is invoked. This is because the building of neighbor list requires initialization and setup of a simulation, similar to what a *run* command would require.

Note that you can change any of these settings after this command executes (e.g., if you wish to use long-range Coulombic interactions) via the *kspace_style* command for your subsequent simulation.

---

The *single/bond* style creates a single bond of type *btype* between two atoms with IDs *batom1* and *batom2*. *Btype* must be a value between 1 and the number of bond types defined.

The *single/angle* style creates a single angle of type *atype* between three atoms with IDs *aatom1*, *aatom2*, and *aatom3*. The ordering of the atoms is the same as in the *Angles* section of a data file read by the *read_data* command (i.e., the three atoms are ordered linearly within the angle; the central atom is *aatom2*). *Atype* must be a value between 1 and the number of angle types defined.

The *single/dihedral* style creates a single dihedral of type *dtype* between four atoms with IDs *datom1*, *datom2*, *datom3*, and *datom4*. The ordering of the atoms is the same as in the *Dihedrals* section of a data file read by the *read_data* command. I.e. the 4 atoms are ordered linearly within the dihedral. *dtype* must be a value between 1 and the number of dihedral types defined.

The *single/improper* style creates a single improper of type *itype* between four atoms with IDs *iatom1*, *iatom2*, *iatom3*, and *iatom4*. The ordering of the atoms is the same as in the *Impropers* section of a data file read by the *read_data* command. I.e. the 4 atoms are ordered linearly within the improper. *itype* must be a value between 1 and the number of improper types defined.

---

The keyword *special* controls whether an internal list of special bonds is created after one or more bonds, or a single angle, dihedral, or improper is added to the system.

The default value is *yes*. A value of *no* cannot be used with the *many* style.

This is an expensive operation since the bond topology for the system must be walked to find all 1–2, 1–3, and 1–4 interactions to store in an internal list, which is used when pairwise interactions are weighted; see the *special_bonds* command for details.

Thus if you are adding a few bonds or a large list of angles all at the same time, by using this command repeatedly, it is more efficient to only trigger the internal list to be created once, after the last bond (or angle, or dihedral, or improper) is added:

```
create_bonds single/bond 5 52 98 special no
create_bonds single/bond 5 73 74 special no
...
create_bonds single/bond 5 17 386 special no
create_bonds single/bond 4 112 183 special yes
```

Note that you **must** ensure the internal list is rebuilt after the last bond (angle, dihedral, improper) is added, *before* performing a simulation. Otherwise, pairwise interactions will not be properly excluded or weighted. LAMMPS does **not** check that you have done this correctly.

---

### 1.18.4 Restrictions

This command cannot be used with molecular systems defined using molecule template files via the *molecule* and *atom_style template* commands.

For style *many*, no *kspace style* must be defined. Also, the *rmax* value must be smaller than any periodic box length and the neighbor list cutoff (largest pair cutoff plus neighbor skin).

### 1.18.5 Related commands

*create_atoms*, *delete_bonds*

### 1.18.6 Default

The keyword default is special = yes.

## 1.19 create_box command

### 1.19.1 Syntax

```
create_box N region-ID keyword value ...
create_box N NULL alo ahi blo bhi clo chi keyword value ...
```

- N = # of atom types to use in this simulation
- region-ID = ID of region to use as simulation domain or NULL for general triclinic box
- alo,ahi,blo,bhi,clo,chi = multipliers on a1,a2,a3 vectors defined by *lattice* command (only when region-ID = NULL)
- zero or more keyword/value pairs may be appended
- keyword = *bond/types* or *angle/types* or *dihedral/types* or *improper/types* or *extra/bond/per/atom* or *extra/angle/per/atom* or *extra/dihedral/per/atom* or *extra/improper/per/atom* or *extra/special/per/atom*

  *bond/types* value = # of bond types
  *angle/types* value = # of angle types
  *dihedral/types* value = # of dihedral types
  *improper/types* value = # of improper types
  *extra/bond/per/atom* value = # of bonds per atom
  *extra/angle/per/atom* value = # of angles per atom
  *extra/dihedral/per/atom* value = # of dihedrals per atom
  *extra/improper/per/atom* value = # of impropers per atom
  *extra/special/per/atom* value = # of special neighbors per atom

### 1.19.2 Examples

```
# orthogonal or restricted triclinic box using regionID = mybox
create_box 2 mybox
create_box 2 mybox bond/types 2 extra/bond/per/atom 1
```

```
# 2d general triclinic box using primitive cell for 2d hex lattice
lattice       custom 1.0 a1 1.0 0.0 0.0 a2 0.5 0.86602540378 0.0 &
              a3 0.0 0.0 1.0 basis 0.0 0.0 0.0 triclinic/general
create_box    1 NULL 0 5 0 5 -0.5 0.5
```

```
# 3d general triclinic box using primitive cell for 3d fcc lattice
lattice custom 1.0 a2 0.0 0.5 0.5 a1 0.5 0.0 0.5 a3 0.5 0.5 0.0 basis 0.0 0.0 0.0␣
↪triclinic/general
create box 1 NULL -5 5 -10 10 0 20
```

### 1.19.3 Description

This command creates a simulation box. It also partitions the box into a regular 3d grid of smaller sub-boxes, one per processor (MPI task). The geometry of the partitioning is based on the size and shape of the simulation box, the number of processors being used and the settings of the *processors* command. The partitioning can later be changed by the *balance* or *fix balance* commands.

Simulation boxes in LAMMPS can be either orthogonal or triclinic in shape. Orthogonal boxes are a brick in 3d (rectangle in 2d) with 6 faces that are each perpendicular to one of the standard xyz coordinate axes. Triclinic boxes are a parallelepiped in 3d (parallelogram in 2d) with opposite pairs of faces parallel to each other. LAMMPS supports two forms of triclinic boxes, restricted and general, which differ in how the box is oriented with respect to the xyz coordinate axes. See the *Howto triclinic* for a detailed description of all 3 kinds of simulation boxes.

The argument $N$ is the number of atom types that will be used in the simulation.

Orthogonal and restricted triclinic boxes are created by specifying a region ID previously defined by the *region* command. General triclinic boxes are discussed below.

If the region is not of style *prism*, then LAMMPS encloses the region (block, sphere, etc.) with an axis-aligned orthogonal bounding box which becomes the simulation domain. For a 2d simulation, the zlo and zhi values of the simulation box must straddle zero.

If the region is of style *prism*, LAMMPS creates a non-orthogonal simulation domain shaped as a parallelepiped with triclinic symmetry. As defined by the *region prism* command, the parallelepiped has an "origin" at (xlo,ylo,zlo) and three edge vectors starting from the origin given by $\vec{a} = (x_{\rm hi} - x_{\rm lo}, 0, 0)$; $\vec{b} = (xy, y_{\rm hi} - y_{\rm lo}, 0)$; and $\vec{c} = (xz, yz, z_{\rm hi} - z_{\rm lo})$. In LAMMPS lingo, this is a restricted triclinic box because the three edge vectors cannot be defined in arbitrary (general) directions. The parameters $xy$, $xz$, and $yz$ can be 0.0 or positive or negative values and are called "tilt factors" because they are the amount of displacement applied to faces of an originally orthogonal box to transform it into the parallelepiped. For a 2d simulation, the zlo and zhi values of the simulation box must straddle zero.

Typically a *prism* region used with the create_box command should have tilt factors $(xy, xz, yz)$ that do not skew the box more than half the distance of the parallel box length. For example, if $x_{\rm lo} = 2$ and $x_{\rm hi} = 12$, then the $x$ box length is 10 and the $xy$ tilt factor must be between $-5$ and 5. Similarly, both $xz$ and $yz$ must be between $-(x_{\rm hi} - x_{\rm lo})/2$ and $+(y_{\rm hi} - y_{\rm lo})/2$. Note that this is not a limitation, since if the maximum tilt factor is 5 (as in this example), then configurations with tilt = $\ldots, -15, -5, 5, 15, 25, \ldots$ are all geometrically equivalent.

LAMMPS will issue a warning if the tilt factors of the created box do not meet this criterion. This is because simulations with large tilt factors may run inefficiently, since they require more ghost atoms and thus more communication. With very large tilt factors, LAMMPS may eventually produce incorrect trajectories and stop with errors due to lost atoms or similar issues.

See the *Howto triclinic* page for geometric descriptions of triclinic boxes and tilt factors, as well as how to transform the restricted triclinic parameters to and from other commonly used triclinic representations.

When a prism region is used, the simulation domain should normally be periodic in the dimension that the tilt is applied to, which is given by the second dimension of the tilt factor (e.g., *y* for *xy* tilt). This is so that pairs of atoms interacting across that boundary will have one of them shifted by the tilt factor. Periodicity is set by the *boundary* command. For example, if the *xy* tilt factor is non-zero, then the *y* dimension should be periodic. Similarly, the *z* dimension should be periodic if *xz* or *yz* is non-zero. LAMMPS does not require this periodicity, but you may lose atoms if this is not the case.

Note that if your simulation will tilt the box (e.g., via the *fix deform* command), the simulation box must be created as triclinic, even if the tilt factors are initially 0.0. You can also change an orthogonal box to a triclinic box or vice versa by using the *change box* command with its *ortho* and *triclinic* options.

**Note**

If the system is non-periodic (in a dimension), then you should not make the lo/hi box dimensions (as defined in your *region* command) radically smaller/larger than the extent of the atoms you eventually plan to create (e.g., via the *create_atoms* command). For example, if your atoms extend from 0 to 50, you should not specify the box bounds as −10000 and 10000. This is because as described above, LAMMPS uses the specified box size to lay out the 3d grid of processors. A huge (mostly empty) box will be sub-optimal for performance when using "fixed" boundary conditions (see the *boundary* command). When using "shrink-wrap" boundary conditions (see the *boundary* command), a huge (mostly empty) box may cause a parallel simulation to lose atoms the first time that LAMMPS shrink-wraps the box around the atoms.

---

As noted above, general triclinic boxes in LAMMPS allow the box to have arbitrary edge vectors **A**, **B**, **C**. The only restrictions are that the three vectors be distinct, non-zero, and not co-planar. They must also define a right-handed system such that (**A** x **B**) points in the direction of **C**. Note that a left-handed system can be converted to a right-handed system by simply swapping the order of any pair of the **A**, **B**, **C** vectors.

To create a general triclinic boxes, the region is specified as NULL and the next 6 parameters (alo,ahi,blo,bhi,clo,chi) define the three edge vectors **A**, **B**, **C** using additional information previously defined by the *lattice* command.

The lattice must be of style *custom* and use its *triclinic/general* option. This insures the lattice satisfies the restrictions listed above. The *a1, *a2, a3* settings of the *lattice* command define the edge vectors of a unit cell of the general triclinic lattice. This command uses them to define the three edge vectors and origin of the general triclinic box as:

- **A** = (ahi-alo) * *a1*
- **B** = (bhi-blo) * *a2*
- **C** = (chi-clo) * *a3*
- origin = (alo*a1 + blo*a2 + clo*a3)

For 2d general triclinic boxes, clo = -0.5 and chi = 0.5 is required.

**Note**

LAMMPS allows specification of general triclinic simulation boxes as a convenience for users who may be converting data from solid-state crystallographic representations or from DFT codes for input to LAMMPS. However, as explained on the *Howto_triclinic* doc page, internally, LAMMPS only uses restricted triclinic simulation boxes. This means the box defined by this command and per-atom information (e.g. coordinates, velocities) defined by the *create_atoms* command are converted (rotated) from general to restricted triclinic form when the two commands

are invoked. The <Howto_triclinic>` doc page also discusses other LAMMPS commands which can input/output general triclinic representations of the simulation box and per-atom data.

---

The optional keywords can be used to create a system that allows for bond (angle, dihedral, improper) interactions, or for molecules with special 1–2, 1–3, or 1–4 neighbors to be added later. These optional keywords serve the same purpose as the analogous keywords that can be used in a data file which are recognized by the *read_data* command when it sets up a system.

Note that if these keywords are not used, then the create_box command creates an atomic (non-molecular) simulation that does not allow bonds between pairs of atoms to be defined, or a *bond potential* to be specified, or for molecules with special neighbors to be added to the system by commands such as *create_atoms mol*, *fix deposit* or *fix pour*.

As an example, see the examples/deposit/in.deposit.molecule script, which deposits molecules onto a substrate. Initially there are no molecules in the system, but they are added later by the *fix deposit* command. The create_box command in the script uses the bond/types and extra/bond/per/atom keywords to allow this. If the added molecule contained more than one special bond (allowed by default), an extra/special/per/atom keyword would also need to be specified.

---

### 1.19.4 Restrictions

An *atom_style* and *region* must have been previously defined to use this command.

### 1.19.5 Related commands

*read_data*, *create_atoms*, *region*

### 1.19.6 Default

none

## 1.20 delete_atoms command

### 1.20.1 Syntax

```
delete_atoms style args keyword value ...
```

- style = *group* or *region* or *overlap* or *random* or *variable*

```
group args = group-ID
region args = region-ID
overlap args = cutoff group1-ID group2-ID
  cutoff = delete one atom from pairs of atoms within the cutoff (distance units)
  group1-ID = one atom in pair must be in this group
  group2-ID = other atom in pair must be in this group
random args = ranstyle value eflag group-ID region-ID seed
  ranstyle = fraction or count
    for fraction:
      value = fraction (0.0 to 1.0) of eligible atoms to delete
      eflag =  no for fast approximate deletion, yes for exact deletion
    for count:
      value = number of atoms to delete
      eflag = no for warning if count > eligible atoms, yes for error
```

```
  group-ID = group within which to perform deletions
  region-ID = region within which to perform deletions
              or NULL to only impose the group criterion
  seed = random number seed (positive integer)
variable args = variable-name
```

- zero or more keyword/value pairs may be appended
- keyword = *compress* or *condense* or *bond* or *mol*

```
compress value = no or yes
condense value = no or yes
bond value = no or yes
mol value = no or yes
```

### 1.20.2 Examples

```
delete_atoms group edge
delete_atoms region sphere compress no
delete_atoms region sphere condense yes
delete_atoms overlap 0.3 all all
delete_atoms overlap 0.5 solvent colloid
delete_atoms random fraction 0.1 yes all cube 482793 bond yes
delete_atoms random fraction 0.3 no polymer NULL 482793 bond yes
delete_atoms random count 500 no ions NULL 482793
delete_atoms variable checkers
```

### 1.20.3 Description

Delete the specified atoms. This command can be used, for example, to carve out voids from a block of material or to delete created atoms that are too close to each other (e.g., at a grain boundary).

For style *group*, all atoms belonging to the group are deleted.

For style *region*, all atoms in the region volume are deleted. Additional atoms can be deleted if they are in a molecule for which one or more atoms were deleted within the region; see the *mol* keyword discussion below.

For style *overlap* pairs of atoms whose distance of separation is within the specified cutoff distance are searched for, and one of the two atoms is deleted. Only pairs where one of the two atoms is in the first group specified and the other atom is in the second group are considered. The atom that is in the first group is the one that is deleted.

Note that it is OK for the two group IDs to be the same (e.g., group *all*), or for some atoms to be members of both groups. In these cases, either atom in the pair may be deleted. Also note that if there are atoms which are members of both groups, the only guarantee is that at the end of the deletion operation, enough deletions will have occurred that no atom pairs within the cutoff will remain (subject to the group restriction). There is no guarantee that the minimum number of atoms will be deleted, or that the same atoms will be deleted when running on different numbers of processors.

For style *random* a subset of eligible atoms are deleted. Which atoms to delete are chosen randomly using the specified random number *seed*. Which atoms are deleted may vary when running on different numbers of processors.

For *ranstyle* = *fraction*, the specified fractional *value* (0.0 to 1.0) of eligible atoms are deleted. If *eflag* is set to *no*, then the number of deleted atoms will be approximate, but the operation will be fast. If *eflag* is set to *yes*, then the number deleted will match the requested fraction, but for large systems the selection of deleted atoms may take additional time to determine.

For *ranstyle* = *count*, the specified integer *value* is the number of eligible atoms are deleted. If *eflag* is set to *no*, then if the requested number is larger then the number of eligible atoms, a warning is issued and only the eligible atoms are deleted instead of the requested *value*. If *eflag* is set to *yes*, an error is triggered instead and LAMMPS will exit.

For large systems the selection of atoms to delete may take additional time to determine, the same as for requesting an exact fraction with *pstyle* = *fraction*.

Which atoms are eligible for deletion for style *random* is determined by the specified *group-ID* and *region-ID*. To be eligible, an atom must be in both the specified group and region. If *group-ID* = all, there is effectively no group criterion. If *region-ID* is specified as NULL, no region criterion is imposed.

Added in version 4May2022.

For style *variable*, all atoms for which the atom-style variable with the given name evaluates to non-zero will be deleted. Additional atoms can be deleted if they are in a molecule for which one or more atoms were deleted within the region; see the *mol* keyword discussion below. This option allows complex selections of atoms not covered by the other options listed above.

---

Here is the meaning of the optional keywords.

If the *compress* keyword is set to *yes*, then after atoms are deleted, then atom IDs are re-assigned so that they run from 1 to the number of atoms in the system. This option is enabled by default for atomic systems. Note that in this case, the re-assignment of IDs is not really a compression, where gaps in atom IDs are removed by decrementing atom IDs that are larger. Instead the IDs for all atoms are erased, and new IDs are assigned so that the atoms owned by individual processors have consecutive IDs, as the *create_atoms* command explains. This is efficient, but incompatible with molecular systems.

Changed in version 10Sep2025.

For molecular systems (see the *atom_style* command), the atom ID re-assignment now calls the *reset_atoms id* command internally. For backward compatibility, the default setting is *no* in this case. This process does *not* preserve the order of atoms with respect to their atom IDs. See the *condense* keyword below.

Added in version 10Sep2025.

If the *condense* keyword set to *yes*, then after atoms are deleted, the atom IDs are re-assigned in such a way that the order of atom-IDs is preserved. This process is not efficient and cannot be used for very large systems and requires local storage that scales with the number of total atoms in the system. Also, the *compress* and the *condense* keywords cannot be used at the same time. Whichever of the two is used last will be applied.

A molecular system with fixed bonds, angles, dihedrals, or improper interactions, is one where the topology of the interactions is typically defined in the data file read by the *read_data* command, and where the interactions themselves are defined with the *bond_style*, *angle_style*, etc. commands.

**Warning**

If you delete atoms from a molecular system, you must be careful not to end up with bonded interactions that are stored by remaining atoms but which include deleted atoms. This will cause LAMMPS to generate a "missing atoms" error when the bonded interaction is computed. The *bond yes* and *mol yes* settings are recommended to avoid such inconsistencies.

If the *bond* keyword is set to *yes* then any bond or angle or dihedral or improper interaction that includes a deleted atom is also removed from the lists of such interactions stored by non-deleted atoms. Note that simply deleting interactions due to dangling bonds (e.g., at a surface) may result in a inaccurate or invalid model for the remaining atoms.

It the *mol* keyword is set to *yes*, then for every atom that is deleted, all other atoms in the same molecule (with the same molecule ID) will also be deleted. This is not done for atoms with molecule ID = 0, since such an ID is assumed to flag isolated atoms that are not part of molecules.

**Note**

The molecule deletion operation is invoked after all individual atoms have been deleted using the rules described above for each style. This means additional atoms may be deleted that are not in the group or region, that are not required by the overlap cutoff criterion, or that will create a higher fraction of porosity than was requested.

### 1.20.4 Restrictions

The *overlap* styles requires inter-processor communication to acquire ghost atoms and build a neighbor list. This means that your system must be ready to perform a simulation before using this command (force fields setup, atom masses set, etc.). Since a neighbor list is used to find overlapping atom pairs, it also means that you must define a *pair style* with the minimum force cutoff distance between any pair of atoms types (plus the *neighbor* skin) $\geq$ the specified overlap cutoff.

If the *special_bonds* command is used with a setting of 0, then a pair of bonded atoms (1–2, 1–3, or 1–4) will not appear in the neighbor list, and thus will not be considered for deletion by the *overlap* styles. You probably do not want to delete one atom in a bonded pair anyway.

The *bond yes* option cannot be used with molecular systems defined using molecule template files via the *molecule* and *atom_style template* commands.

### 1.20.5 Related commands

*create_atoms*, *reset_atoms id*

### 1.20.6 Default

The option defaults are compress = yes for atomic systems, otherwise compress = no; also bond = no and mol = no.

## 1.21 delete_bonds command

### 1.21.1 Syntax

```
delete_bonds group-ID style arg keyword ...
```

- group-ID = group ID
- style = *multi* or *atom* or *bond* or *angle* or *dihedral* or *improper* or *stats*

  ```
  multi arg = none
  atom arg = an atom type or range of types (see below)
  bond arg = a bond type or range of types (see below)
  angle arg = an angle type or range of types (see below)
  dihedral arg = a dihedral type or range of types (see below)
  improper arg = an improper type or range of types (see below)
  stats arg = none
  ```

- zero or more keywords may be appended
- keyword = *any* or *undo* or *remove* or *special*

  ```
  any arg = none = turn off interactions if any atoms are in the group (or on if undo␣
  →is also used)
  undo arg = none = turn specified bonds on instead of off
  remove arg = permanently remove bonds that have been turned off
  special arg = recompute pairwise 1-2, 1-3, and 1-4 lists
  ```

### 1.21.2 Examples

```
delete_bonds frozen multi remove
delete_bonds all atom 4 special
delete_bonds all bond 0*3 special
delete_bonds all stats

labelmap atom 4 hc
delete_bonds all atom hc special
```

### 1.21.3 Description

Turn off (or on) molecular topology interactions (i.e., bonds, angles, dihedrals, and/or impropers). This command is useful for deleting interactions that have been previously turned off by bond-breaking potentials. It is also useful for turning off topology interactions between frozen or rigid atoms. Pairwise interactions can be turned off via the *neigh_modify exclude* command. The *fix shake* command also effectively turns off certain bond and angle interactions.

For all styles, by default, an interaction is only turned off (or on) if all the atoms involved are in the specified group. See the *any* keyword to change the behavior.

**Possible errors caused by using *delete_bonds***

Since this command by default only *turns off* bonded interactions, their definitions are still present and subject to the limitations due to LAMMPS' domain decomposition based parallelization. That is, when a bond is turned off, the two constituent atoms may move apart and may reach a distance where they can lead to a "bond atoms missing" error and crash the simulation. Adding the *remove* keyword (see below) is required to fully remove those interactions and prevent the error.

Several of the styles (*atom*, *bond*, *angle*, *dihedral*, *improper*) take a *type* as an argument. The specified *type* can be a *type label*. Otherwise, the type should be an integer from 0 to *N*, where *N* is the number of relevant types (atom types, bond types, etc.). A value of 0 is only relevant for style *bond*; see details below. For numeric types, a wildcard asterisk can be used in place of or in conjunction with the *type* argument to specify a range of types. This takes the form "*" or "*n" or "m*" or "m*n". If *N* is the number of types, then an asterisk with no numeric values means all types from 0 to *N*. A leading asterisk means all types from 0 to n (inclusive). A trailing asterisk means all types from m to N (inclusive). A middle asterisk means all types from m to n (inclusive). Note that it is fine to include a type of 0 for non-bond styles; it will simply be ignored.

For style *multi* all bond, angle, dihedral, and improper interactions of any type, involving atoms in the group, are turned off.

Style *atom* is the same as style *multi* except that in addition, one or more of the atoms involved in the bond, angle, dihedral, or improper interaction must also be of the specified atom type.

For style *bond*, only bonds are candidates for turn-off, and the bond must also be of the specified type. Styles *angle*, *dihedral*, and *improper* are treated similarly.

For style *bond*, you can set the type to 0 to delete bonds that have been previously broken by a bond-breaking potential (which sets the bond type to 0 when a bond is broken); for example, see the *bond_style quartic* command.

For style *stats* no interactions are turned off (or on); the status of all interactions in the specified group is simply reported. This is useful for diagnostic purposes if bonds have been turned off by a bond-breaking potential during a previous run.

**Impact on special_bonds processing and exclusions**

The default behavior of the delete_bonds command is to turn off interactions by toggling their type to a negative value, but not to permanently remove the interaction. For example, a bond_type of 2 is set to −2. The neighbor list creation routines will not include such an interaction in their interaction lists. The default is also to not alter the list of 1–2, 1–3, or 1–4 neighbors computed by the *special_bonds* command and used to weight pairwise force and energy calculations. This means that pairwise computations will proceed as if the bond (or angle, etc.) were still turned on.

Several keywords can be appended to the argument list to alter the default behaviors.

The *any* keyword changes the requirement that all atoms in the bond (angle, etc.) must be in the specified group in order to turn off the interaction. Instead, if any of the atoms in the interaction are in the specified group, it will be turned off (or on if the *undo* keyword is used).

The *undo* keyword inverts the delete_bonds command so that the specified bonds, angles, etc. are turned on if they are currently turned off. This means a negative value is toggled to positive. For example, for style *angle*, if *type* is specified as 2, then all angles with current type = −2 are reset to type = 2. Note that the *fix shake* command also sets bond and angle types negative, so this option should not be used on those interactions.

The *remove* keyword is invoked at the end of the delete_bonds operation. It causes turned-off bonds (angles, etc.) to be removed from each atom's data structure and then adjusts the global bond (angle, etc.) counts accordingly. Removal is a permanent change; removed bonds cannot be turned back on via the *undo* keyword. Removal does not alter the pairwise 1–2, 1–3, or 1–4 weighting list.

The *special* keyword is invoked at the end of the delete_bonds operation, after (optional) removal. It re-computes the pairwise 1–2, 1–3, 1–4 weighting list. The weighting list computation treats turned-off bonds the same as turned-on. Thus, turned-off bonds must be removed if you wish to change the weighting list.

**Note**

The choice of *remove* and *special* options affects how 1–2, 1–3, 1–4 pairwise interactions will be computed across bonds that have been modified by the delete_bonds command.

### 1.21.4 Restrictions

This command requires inter-processor communication to acquire ghost atoms, to coordinate the deleting of bonds, angles, etc. between atoms shared by multiple processors. This means that your system must be ready to perform a simulation before using this command (force fields setup, atom masses set, etc.). Just as would be needed to run dynamics, the force field you define should define a cutoff (e.g., through a *pair_style* command) which is long enough for a processor to acquire the ghost atoms its needs to compute bond, angle, etc. interactions.

If deleted bonds (or angles, etc.) are removed but the 1–2, 1–3, and 1–4 weighting list is not recomputed, this can cause a later *fix shake* command to fail due to an atom's bonds being inconsistent with the weighting list. This should only happen if the group used in the fix command includes both atoms in the bond, in which case you probably should be recomputing the weighting list.

### 1.21.5 Related commands

*neigh_modify* exclude, *special_bonds*, *fix shake*

### 1.21.6 Default

none

# 1.22 dielectric command

## 1.22.1 Syntax

```
dielectric value
```

- value = dielectric constant

## 1.22.2 Examples

```
dielectric 2.0
```

## 1.22.3 Description

Set the dielectric constant for Coulombic interactions (pairwise and long-range) to this value. The constant is unitless, since it is used to reduce the strength of the interactions. The value is used in the denominator of the formulas for Coulombic interactions (e.g., a value of 4.0 reduces the Coulombic interactions to 25% of their default strength). See the *pair_style* command for more details.

## 1.22.4 Restrictions

none

## 1.22.5 Related commands

*pair_style*

## 1.22.6 Default

```
dielectric 1.0
```

# 1.23 dihedral_coeff command

## 1.23.1 Syntax

```
dihedral_coeff N args
```

- N = numeric dihedral type (see asterisk form below) or alphanumeric type label
- args = coefficients for one or more dihedral types

## 1.23.2 Examples

```
dihedral_coeff 1 80.0 1 3
dihedral_coeff * 80.0 1 3 0.5
dihedral_coeff 2* 80.0 1 3 0.5

labelmap dihedral 1 backbone
dihedral_coeff backbone 80.0 1 3
```

### 1.23.3 Description

Specify the dihedral force field coefficients for one or more dihedral types. The number and meaning of the coefficients depends on the dihedral style. Dihedral coefficients can also be set in the data file read by the *read_data* command or in a restart file.

*N* can be specified in one of two ways. An explicit numeric value can be used, as in the first example above. Or *N* can be an alphanumeric type label, which is a string defined by the *labelmap* command or in a corresponding section of a data file read by the *read_data* command.

For numeric values only, a wild-card asterisk can be used to set the coefficients for multiple dihedral types. This takes the form "*" or "*n" or "n*" or "m*n". If *N* is the number of dihedral types, then an asterisk with no numeric values means all types from 1 to *N*. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to *N* (inclusive). A middle asterisk means all types from m to n (inclusive).

Note that using a dihedral_coeff command can override a previous setting for the same dihedral type. For example, these commands set the coeffs for all dihedral types, then overwrite the coeffs for just dihedral type 2:

```
dihedral_coeff * 80.0 1 3
dihedral_coeff 2 200.0 1 3
```

A line in a data file that specifies dihedral coefficients uses the exact same format as the arguments of the dihedral_coeff command in an input script, except that wild-card asterisks should not be used since coefficients for all *N* types must be listed in the file. For example, under the "Dihedral Coeffs" section of a data file, the line that corresponds to the first example above would be listed as

```
1 80.0 1 3
```

The *dihedral_style class2* is an exception to this rule, in that an additional argument is used in the input script to allow specification of the cross-term coefficients. See its doc page for details.

**Note**

When comparing the formulas and coefficients for various LAMMPS dihedral styles with dihedral equations defined by other force fields, note that some force field implementations divide/multiply the energy prefactor *K* by the multiple number of torsions that contain the *J*–*K* bond in an *I*-*J*-*K*-*L* torsion. LAMMPS does not do this (i.e., the listed dihedral equation applies to each individual dihedral). Thus, you need to define *K* appropriately to account for this difference, if necessary.

---

The list of all dihedral styles defined in LAMMPS is given on the *dihedral_style* doc page. They are also listed in more compact form on the *Commands dihedral* doc page.

On either of those pages, click on the style to display the formula it computes and its coefficients as specified by the associated dihedral_coeff command.

---

### 1.23.4 Restrictions

This command must come after the simulation box is defined by a *read_data*, *read_restart*, or *create_box* command.

A dihedral style must be defined before any dihedral coefficients are set, either in the input script or in a data file.

### 1.23.5 Related commands

*dihedral_style*

### 1.23.6 Default

none

## 1.24 dihedral_style command

### 1.24.1 Syntax

```
dihedral_style style
```

- style = *none* or *zero* or *hybrid* or *charmm* or *charmmfsw* or *class2* or *class2xe* or *cosine/shift/exp* or *cosine/squared/restricted* or *fourier* or *harmonic* or *helix* or *lepton* or *multi/harmonic* or *nharmonic* or *opls* or *spherical* or *table* or *table/cut*

### 1.24.2 Examples

```
dihedral_style harmonic
dihedral_style multi/harmonic
dihedral_style hybrid harmonic charmm
```

### 1.24.3 Description

Set the formula(s) LAMMPS uses to compute dihedral interactions between quadruplets of atoms, which remain in force for the duration of the simulation. The list of dihedral quadruplets is read in by a *read_data* or *read_restart* command from a data or restart file.

Hybrid models where dihedrals are computed using different dihedral potentials can be setup using the *hybrid* dihedral style.

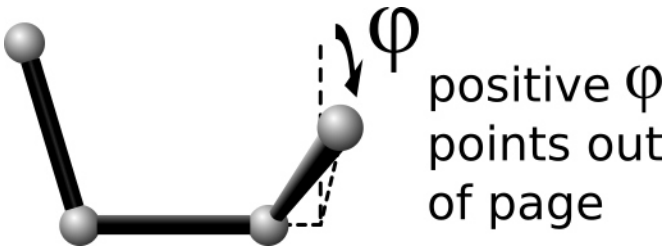
The coefficients associated with a dihedral style can be specified in a data or restart file or via the *dihedral_coeff* command.

All dihedral potentials store their coefficient data in binary restart files which means dihedral_style and *dihedral_coeff* commands do not need to be re-specified in an input script that restarts a simulation. See the *read_restart* command for details on how to do this. The one exception is that dihedral_style *hybrid* only stores the list of sub-styles in the restart file; dihedral coefficients need to be re-specified.

**Note**

When both a dihedral and pair style is defined, the *special_bonds* command often needs to be used to turn off (or weight) the pairwise interaction that would otherwise exist between four bonded atoms.

In the formulas listed for each dihedral style, *phi* is the torsional angle defined by the quadruplet of atoms. This angle has a sign convention as shown in this diagram:




where the $I, J, K, L$ ordering of the four atoms that define the dihedral is from left to right.

This sign convention effects several of the dihedral styles listed below (e.g., charmm, helix) in the sense that the energy formula depends on the sign of phi, which may be reflected in the value of the coefficients you specify.

**Note**

When comparing the formulas and coefficients for various LAMMPS dihedral styles with dihedral equations defined by other force fields, note that some force field implementations divide/multiply the energy prefactor $K$ by the multiple number of torsions that contain the $J$–$K$ bond in an $I$–$J$–$K$–$L$ torsion. LAMMPS does not do this (i.e., the listed dihedral equation applies to each individual dihedral). Thus, you need to define $K$ appropriately via the *dihedral_coeff* command to account for this difference if necessary.

---

Here is an alphabetic list of dihedral styles defined in LAMMPS. Click on the style to display the formula it computes and coefficients specified by the associated *dihedral_coeff* command.

Click on the style to display the formula it computes, any additional arguments specified in the dihedral_style command, and coefficients specified by the associated *dihedral_coeff* command.

There are also additional accelerated pair styles included in the LAMMPS distribution for faster performance on CPUs, GPUs, and KNLs. The individual style names on the *Commands dihedral* page are followed by one or more of (g,i,k,o,t) to indicate which accelerated styles exist.

- *none* - turn off dihedral interactions
- *zero* - topology but no interactions
- *hybrid* - define multiple styles of dihedral interactions
- *charmm* - CHARMM dihedral
- *charmmfsw* - CHARMM dihedral with force switching
- *class2* - COMPASS (class 2) dihedral
- *class2xe* - ClassII-xe (class 2) dihedral
- *cosine/shift/exp* - dihedral with exponential in spring constant
- *cosine/squared/restricted* - squared cosine dihedral with restricted term

- *fourier* - dihedral with multiple cosine terms
- *harmonic* - harmonic dihedral
- *helix* - helix dihedral
- *lepton* - dihedral potential from evaluating a string
- *multi/harmonic* - dihedral with 5 harmonic terms
- *nharmonic* - same as multi-harmonic with N terms
- *opls* - OPLS dihedral
- *quadratic* - dihedral with quadratic term in angle
- *spherical* - dihedral which includes angle terms to avoid singularities
- *table* - tabulated dihedral
- *table/cut* - tabulated dihedral with analytic cutoff

---

### 1.24.4 Restrictions

Dihedral styles can only be set for atom styles that allow dihedrals to be defined.

Most dihedral styles are part of the MOLECULE package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info. The doc pages for individual dihedral potentials tell if it is part of a package.

### 1.24.5 Related commands

*dihedral_coeff*

### 1.24.6 Default

dihedral_style none

## 1.25 dihedral_write command

### 1.25.1 Syntax

```
dihedral_write dtype N file keyword
```

- dtype = dihedral type
- N = # of values
- file = name of file to write values to
- keyword = section name in file for this set of tabulated values

### 1.25.2 Examples

```
dihedral_write 1  500 table.txt Harmonic_1
dihedral_write 3 1000 table.txt Harmonic_3
```

### 1.25.3 Description

Added in version 8Feb2023.

Write energy and force values to a file as a function of the dihedral angle for the currently defined dihedral potential. Force in this context means the force with respect to the dihedral angle, not the force on individual atoms. This is useful for plotting the potential function or otherwise debugging its values. The resulting file can also be used as input for use with *dihedral style table*.

If the file already exists, the table of values is appended to the end of the file to allow multiple tables of energy and force to be included in one file. The individual sections may be identified by the *keyword*.

The energy and force values are computed for dihedrals ranging from 0 degrees to 360 degrees for 4 interacting atoms forming an dihedral type dtype, using the appropriate *dihedral_coeff* coefficients. N evenly spaced dihedrals are used. Since 0 and 360 degrees are the same dihedral angle, the latter entry is skipped.

For example, for N = 6, values would be computed at $\phi = 0, 60, 120, 180, 240, 300$.

The file is written in the format used as input for the *dihedral_style table* option with *keyword* as the section name. Each line written to the file lists an index number (1-N), an dihedral angle (in degrees), an energy (in energy units), and a force (in force units per radians^2). In case a new file is created, the first line will be a comment with a "DATE:" and "UNITS:" tag with the current date and *units* settings. For subsequent invocations of the *dihedral_write* command for the same file, data will be appended and the current units settings will be compared to the data from the header, if present. The *dihedral_write* will refuse to add a table to an existing file if the units are not the same.

### 1.25.4 Restrictions

All force field coefficients for dihedrals and other kinds of interactions must be set before this command can be invoked.

The table of the dihedral energy and force data data is created by using a separate, internally created, new LAMMPS instance with a dummy system of 4 atoms for which the dihedral potential energy is computed after transferring the dihedral style and coefficients and arranging the 4 atoms into the corresponding geometries. The dihedral force is then determined from the potential energies through numerical differentiation. As a consequence of this approach, not all dihedral styles are compatible. The following conditions must be met:

- The dihedral style must be able to write its coefficients to a data file. This condition excludes for example *dihedral style hybrid* and *dihedral style table*.
- The potential function must not have any terms that depend on geometry properties other than the dihedral. This condition excludes for example *dihedral style class2*. Please note that the *write_dihedral* command has no way of checking for this condition. It will check the style name against an internal list of known to be incompatible styles. The resulting tables may be bogus for unlisted dihedral styles if the requirement is not met. It is thus recommended to make careful tests for any created tables.

### 1.25.5 Related commands

*dihedral_style table*, *bond_write*, *angle_write*, *dihedral_style*, *dihedral_coeff*

### 1.25.6 Default

none

## 1.26 dimension command

### 1.26.1 Syntax

```
dimension N
```

- N = 2 or 3

### 1.26.2 Examples

```
dimension 2
```

### 1.26.3 Description

Set the dimensionality of the simulation. By default LAMMPS runs 3d simulations. To run a 2d simulation, this command should be used prior to setting up a simulation box via the *create_box* or *read_data* commands. Restart files also store this setting.

See the discussion on the *Howto 2d* page for additional instructions on how to run 2d simulations.

**Note**

Some models in LAMMPS treat particles as finite-size spheres or ellipsoids, as opposed to point particles. In 2d, the particles will still be spheres or ellipsoids, not circular disks or ellipses, meaning their moment of inertia will be the same as in 3d.

### 1.26.4 Restrictions

This command must be used before the simulation box is defined by a *read_data* or *create_box* command.

### 1.26.5 Related commands

*fix enforce2d*

### 1.26.6 Default

```
dimension 3
```

## 1.27 displace_atoms command

### 1.27.1 Syntax

```
displace_atoms group-ID style args keyword value ...
```

- group-ID = ID of group of atoms to displace
- style = *move* or *ramp* or *random* or *rotate*

```
move args = delx dely delz
  delx,dely,delz = distance to displace in each dimension (distance units)
  any of delx,dely,delz can be a variable (see below)
ramp args = ddim dlo dhi dim clo chi
  ddim = x or y or z
  dlo,dhi = displacement distance between dlo and dhi (distance units)
  dim = x or y or z
  clo,chi = lower and upper bound of domain to displace (distance units)
random args = dx dy dz seed
  dx,dy,dz = random displacement magnitude in each dimension (distance units)
  seed = random # seed (positive integer)
```

```
rotate args = Px Py Pz Rx Ry Rz theta
  Px,Py,Pz = origin point of axis of rotation (distance units)
  Rx,Ry,Rz = axis of rotation vector
  theta = angle of rotation (degrees)
```

- zero or more keyword/value pairs may be appended

```
keyword = units
  units value = box or lattice
```

### 1.27.2 Examples

```
displace_atoms top move 0 -5 0 units box
displace_atoms flow ramp x 0.0 5.0 y 2.0 20.5
```

### 1.27.3 Description

Displace a group of atoms. This can be used to move atoms a large distance before beginning a simulation or to randomize atoms initially on a lattice. For example, in a shear simulation, an initial strain can be imposed on the system. Or two groups of atoms can be brought into closer proximity.

The *move* style displaces the group of atoms by the specified 3d displacement vector. Any of the three quantities defining the vector components can be specified as an equal-style or atom-style *variable*. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated, and its value(s) used for the displacement(s). The scale factor implied by the *units* keyword will also be applied to the variable result.

Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Atom-style variables can specify the same formulas as equal-style variables but can also include per-atom values, such as atom coordinates or per-atom values read from a file. Note that if the variable references other *compute* or *fix* commands, those values must be up-to-date for the current timestep. See the "Variable Accuracy" section of the *variable* doc page for more details.

The *ramp* style displaces atoms a variable amount in one dimension depending on the atom's coordinate in a (possibly) different dimension. For example, the second example command displaces atoms in the $x$-direction an amount between 0.0 and 5.0 distance units. Each atom's displacement depends on the fractional distance its $y$ coordinate is between 2.0 and 20.5. Atoms with $y$-coordinates outside those bounds will be moved the minimum (0.0) or maximum (5.0) amount.

The *random* style independently moves each atom in the group by a random displacement, uniformly sampled from a value between $-dx$ and $+dx$ in the $x$ dimension, and similarly for $y$ and $z$. Random numbers are used in such a way that the displacement of a particular atom is the same, regardless of how many processors are being used.

The *rotate* style rotates each atom in the group by the angle *theta* around a rotation axis $R = (R_x, R_y, R_z)$ that goes through a point $P = (P_x, P_y, P_z)$. The direction of rotation for the atoms around the rotation axis is consistent with the right-hand rule: if your right-hand thumb points along $R$, then your fingers wrap around the axis in the direction of positive theta.

If the defined *atom_style* assigns an orientation to each atom (*atom styles* ellipsoid, line, tri, body), then that property is also updated appropriately to correspond to the atom's rotation.

Distance units for displacements and the origin point of the *rotate* style are determined by the setting of *box* or *lattice* for the *units* keyword. *Box* means distance units as defined by the *units* command (e.g., Å for *real* or *metal* units). *Lattice* means distance units are in lattice spacings. The *lattice* command must have been previously used to define the lattice spacing.

**Note**

Care should be taken not to move atoms on top of other atoms. After the move, atoms are remapped into the periodic simulation box if needed, and any shrink-wrap boundary conditions (see the *boundary* command) are enforced which may change the box size. Other than this effect, this command does not change the size or shape of the simulation box. See the *change_box* command if that effect is desired.

**Note**

Atoms can be moved arbitrarily long distances by this command. If the simulation box is non-periodic and shrink-wrapped (see the *boundary* command), this can change its size or shape. This is not a problem, except that the mapping of processors to the simulation box is not changed by this command from its initial 3d configuration; see the *processors* command. Thus, if the box size/shape changes dramatically, the mapping of processors to the simulation box may not end up as optimal as the initial mapping attempted to be.

---

### 1.27.4 Restrictions

For a 2d simulation, only rotations around the a vector parallel to the $z$-axis are allowed.

### 1.27.5 Related commands

*lattice*, *change_box*, *fix move*

### 1.27.6 Default

The option defaults are units = lattice.

## 1.28 dynamical_matrix command

Accelerator Variant: dynamical_matrix/kk

### 1.28.1 Syntax

```
dynamical_matrix group-ID style gamma args keyword value ...
```

- group-ID = ID of group of atoms to displace
- style = *regular* or *eskm*
- gamma = finite different displacement length (distance units)
- one or more keyword/arg pairs may be appended

```
keyword = file or binary
  file name = name of output file for the dynamical matrix
  binary arg = yes or no or gzip
```

### 1.28.2 Examples

```
dynamical_matrix 1 regular 0.000001
dynamical_matrix 1 eskm 0.000001
dynamical_matrix 3 regular 0.00004 file dynmat.dat
dynamical_matrix 5 eskm 0.00000001 file dynamical.dat binary yes
```

### 1.28.3 Description

Calculate the dynamical matrix by finite difference of the selected group,

$$D = \frac{\Phi_{ij}^{\alpha\beta}}{\sqrt{M_i M_j}}$$

where D is the dynamical matrix and $\Phi$ is the force constant matrix defined by

$$\Phi_{ij}^{\alpha\beta} = \frac{\partial^2 U}{\partial x_{i,\alpha} \partial x_{j,\beta}}$$

The output for the dynamical matrix is printed three elements at a time. The three elements are the three $\beta$ elements for a respective i/$\alpha$/j combination. Each line is printed in order of j increasing first, $\alpha$ second, and i last.

If the style eskm is selected, the dynamical matrix will be in units of inverse squared femtoseconds. These units will then conveniently leave frequencies in THz.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 1.28.4 Restrictions

The command collects an array of nine times the number of atoms in a group on every single MPI rank, so the memory requirements can be very significant for large systems.

This command is part of the PHONON package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 1.28.5 Related commands

*fix phonon*, *fix numdiff*,

*compute hma* uses an analytic formulation of the Hessian provided by a pair_style's Pair::single_hessian() function, if implemented.

### 1.28.6 Default

The default settings are file = "dynmat.dyn", binary = no

## 1.29 echo command

### 1.29.1 Syntax

```
echo style
```

- style = *none* or *screen* or *log* or *both*

### 1.29.2 Examples

```
echo both
echo log
```

### 1.29.3 Description

This command determines whether LAMMPS echoes each input script command to the screen and/or log file as it is read and processed. If an input script has errors, it can be useful to look at echoed output to see the last command processed.

The *command-line switch* -echo can be used in place of this command.

### 1.29.4 Restrictions

none

### 1.29.5 Related commands

none

### 1.29.6 Default

```
echo log
```

## 1.30 fix command

### 1.30.1 Syntax

```
fix ID group-ID style args
```

- ID = user-assigned name for the fix
- group-ID = ID of the group of atoms to apply the fix to
- style = one of a long list of possible style names (see below)
- args = arguments used by a particular style

### 1.30.2 Examples

```
fix 1 all nve
fix 3 all nvt temp 300.0 300.0 0.01
fix mine top setforce 0.0 NULL 0.0
```

### 1.30.3 Description

Set a fix that will be applied to a group of atoms. In LAMMPS, a "fix" is any operation that is applied to the system during timestepping or minimization. Examples include updating of atom positions and velocities due to time integration, controlling temperature, applying constraint forces to atoms, enforcing boundary conditions, computing diagnostics, etc. There are hundreds of fixes defined in LAMMPS and new ones can be added; see the *Modify* page for details.

Fixes perform their operations at different stages of the timestep. If two or more fixes operate at the same stage of the timestep, they are invoked in the order they were specified in the input script.

The ID of a fix can only contain alphanumeric characters and underscores.

Fixes can be deleted with the *unfix* command.

**Note**

The *unfix* command is the only way to turn off a fix; simply specifying a new fix with a similar style will not turn off the first one. This is especially important to realize for integration fixes. For example, using a *fix nve* command for a second run after using a *fix nvt* command for the first run will not cancel out the NVT time integration invoked by the "fix nvt" command. Thus, two time integrators would be in place!

If you specify a new fix with the same ID and style as an existing fix, the old fix is deleted and the new one is created (presumably with new settings). This is the same as if an "unfix" command were first performed on the old fix, except that the new fix is kept in the same order relative to the existing fixes as the old one originally was. Note that this operation also wipes out any additional changes made to the old fix via the *fix_modify* command.

The *fix modify* command allows settings for some fixes to be reset. See the page for individual fixes for details.

Some fixes store an internal "state" which is written to binary restart files via the *restart* or *write_restart* commands. This allows the fix to continue on with its calculations in a restarted simulation. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file. See the doc pages for individual fixes for info on which ones can be restarted.

---

Some fixes calculate and store any of four *styles* of quantities: global, per-atom, local, or per-grid.

A global quantity is one or more system-wide values, e.g. the energy of a wall interacting with particles. A per-atom quantity is one or more values per atom, e.g. the original coordinates of each atom at time 0. Per-atom values are set to 0.0 for atoms not in the specified fix group. Local quantities are calculated by each processor based on the atoms it owns, but there may be zero or more per atom, e.g. values for each bond. Per-grid quantities are calculated on a regular 2d or 3d grid which overlays a 2d or 3d simulation domain. The grid points and the data they store are distributed across processors; each processor owns the grid points which fall within its subdomain.

As a general rule of thumb, fixes that produce per-atom quantities have the word "atom" at the end of their style, e.g. *ave/atom*. Fixes that produce local quantities have the word "local" at the end of their style, e.g. *store/local*. Fixes that produce per-grid quantities have the word "grid" at the end of their style, e.g. *ave/grid*.

Global, per-atom, local, and per-grid quantities can also be of three *kinds*: a single scalar value (global only), a vector of values, or a 2d array of values. For per-atom, local, and per-grid quantities, a "vector" means a single value for each atom, each local entity (e.g. bond), or grid cell. Likewise an "array", means multiple values for each atom, each local entity, or each grid cell.

Note that a single fix can produce any combination of global, per-atom, local, or per-grid values. Likewise it can produce any combination of scalar, vector, or array output for each style. The exception is that for per-atom, local, and per-grid output, either a vector or array can be produced, but not both. The doc page for each fix explains the values it produces, if any.

When a fix output is accessed by another input script command it is referenced via the following bracket notation, where ID is the ID of the fix:

| | |
|---|---|
| f_ID | entire scalar, vector, or array |
| f_ID[I] | one element of vector, one column of array |
| f_ID[I][J] | one element of array |

In other words, using one bracket reduces the dimension of the quantity once (vector → scalar, array → vector). Using two brackets reduces the dimension twice (array → scalar). Thus, for example, a command that uses global scalar fix values as input can also process elements of a vector or array. Depending on the command, this can either be done directly using the syntax in the table, or by first defining a *variable* of the appropriate style to store the quantity, then using the variable as an input to the command.

Note that commands and *variables* which take fix outputs as input typically do not allow for all styles and kinds of data (e.g., a command may require global but not per-atom values, or it may require a vector of values, not a scalar). This means there is typically no ambiguity about referring to a fix output as c_ID even if it produces, for example, both a scalar and vector. The doc pages for various commands explain the details, including how any ambiguities are resolved.

---

In LAMMPS, the values generated by a fix can be used in several ways:

- Global values can be output via the *thermo_style custom* or *fix ave/time* command. Alternatively, the values can be referenced in an *equal-style variable* command.
- Per-atom values can be output via the *dump custom* command, or they can be time-averaged via the *fix ave/atom* command or reduced by the *compute reduce* command. Alternatively, per-atom values can be referenced in an *atom-style variable*.
- Local values can be reduced by the *compute reduce* command or histogrammed by the *fix ave/histo* command. They can also be output by the *dump local* command.

See the *Howto output* page for a summary of various LAMMPS output options, many of which involve fixes.

The results of fixes that calculate global quantities can be either "intensive" or "extensive" values. Intensive means the value is independent of the number of atoms in the simulation (e.g., temperature). Extensive means the value scales with the number of atoms in the simulation (e.g., total rotational kinetic energy). *Thermodynamic output* will normalize extensive values by the number of atoms in the system, depending on the "thermo_modify norm" setting. It will not normalize intensive values. If a fix value is accessed in another way (e.g., by a *variable*), you may want to know whether it is an intensive or extensive value. See the page for individual fix styles for further info.

---

Each fix style has its own page that describes its arguments and what it does, as listed below. Here is an alphabetical list of fix styles available in LAMMPS. They are also listed in more compact form on the *Commands fix* doc page.

There are also additional accelerated fix styles included in the LAMMPS distribution for faster performance on CPUs, GPUs, and KNLs. The individual style names on the *Commands fix* doc page are followed by one or more of (g,i,k,o,t) to indicate which accelerated styles exist.

- *accelerate/cos* - apply cosine-shaped acceleration to atoms
- *acks2/reaxff* - apply ACKS2 charge equilibration
- *adapt* - change a simulation parameter over time

- *adapt/fep* - enhanced version of fix adapt
- *addforce* - add a force to each atom
- *add/heat* - add a heat flux to each atom
- *addtorque/atom* - add a torque to a finite-size particles
- *addtorque/group* - add a torque to a group of atoms
- *alchemy* - perform an "alchemical transformation" between two partitions
- *align/self* - add torque to groups of atoms due to a self-alignment
- *amoeba/bitorsion* - torsion/torsion terms in AMOEBA force field
- *amoeba/pitorsion* - 6-body terms in AMOEBA force field
- *append/atoms* - append atoms to a running simulation
- *atom/swap* - Monte Carlo atom type swapping
- *atom_weight/apip* - compute atomic load of an *APIP potential* for load balancing
- *ave/atom* - compute per-atom time-averaged quantities
- *ave/chunk* - compute per-chunk time-averaged quantities
- *ave/correlate* - compute/output time correlations
- *ave/correlate/long* - alternative to *ave/correlate* that allows efficient calculation over long time windows
- *ave/grid* - compute per-grid time-averaged quantities
- *ave/histo* - compute/output time-averaged histograms
- *ave/histo/weight* - weighted version of fix ave/histo
- *ave/moments* - compute moments of scalar quantities
- *ave/time* - compute/output global time-averaged quantities
- *aveforce* - add an averaged force to each atom
- *balance* - perform dynamic load-balancing
- *baoab* - BAOAB Langevin dynamics integrator
- *brownian* - overdamped translational brownian motion
- *brownian/asphere* - overdamped translational and rotational brownian motion for ellipsoids
- *brownian/sphere* - overdamped translational and rotational brownian motion for spheres
- *bocs* - NPT style time integration with pressure correction
- *bond/break* - break bonds on the fly
- *bond/create* - create bonds on the fly
- *bond/create/angle* - create bonds on the fly with angle constraints
- *bond/react* - apply topology changes to model reactions
- *bond/swap* - Monte Carlo bond swapping
- *box/relax* - relax box size during energy minimization
- *charge/regulation* - Monte Carlo sampling of charge regulation
- *cmap* - CMAP torsion/torsion terms in CHARMM force field

- *colvars* - interface to the collective variables "Colvars" library
- *controller* - apply control loop feedback mechanism
- *damping/cundall* - Cundall non-viscous damping for granular simulations
- *deform* - change the simulation box size/shape
- *deform/pressure* - change the simulation box size/shape with additional loading conditions
- *deposit* - add new atoms above a surface
- *dpd/energy* - constant energy dissipative particle dynamics
- *drag* - drag atoms towards a defined coordinate
- *drude* - part of Drude oscillator polarization model
- *drude/transform/direct* - part of Drude oscillator polarization model
- *drude/transform/inverse* - part of Drude oscillator polarization model
- *dt/reset* - reset the timestep based on velocity, forces
- *edpd/source* - add heat source to eDPD simulations
- *efield* - impose electric field on system
- *efield/lepton* - impose electric field on system using a Lepton expression for the potential
- *efield/tip4p* - impose electric field on system with TIP4P molecules
- *ehex* - enhanced heat exchange algorithm
- *electrode/conp* - impose electric potential
- *electrode/conq* - impose total electric charge
- *electrode/thermo* - apply thermo-potentiostat
- *electron/stopping* - electronic stopping power as a friction force
- *electron/stopping/fit* - electronic stopping power as a friction force
- *enforce2d* - zero out $z$-dimension velocity and force
- *eos/cv* - applies a mesoparticle equation of state to relate the particle internal energy to the particle internal temperature
- *eos/table* - applies a tabulated mesoparticle equation of state to relate the particle internal energy to the particle internal temperature
- *eos/table/rx* - applies a tabulated mesoparticle equation of state to relate the concentration-dependent particle internal energy to the particle internal temperature
- *evaporate* - remove atoms from simulation periodically
- *external* - callback to an external driver program
- *ffl* - apply a Fast-Forward Langevin equation thermostat
- *filter/corotate* - implement corotation filter to allow larger timesteps with r-RESPA
- *flow/gauss* - Gaussian dynamics for constant mass flux
- *freeze* - freeze atoms in a granular simulation
- *gcmc* - grand canonical Monte Carlo insertions/deletions/translations/rotations
- *gemc* - Gibbs ensemble Monte Carlo exchanges/volumes/translations/rotations

- *gjf* - statistically correct Langevin temperature control using the GJ methods
- *gld* - generalized Langevin dynamics integrator
- *gle* - generalized Langevin equation thermostat
- *graphics/arrows* - add arrow graphics objects to *dump image* output
- *graphics/chunk* - add convex hull graphics for *chunks* to *dump image* output
- *graphics/isosurface* - add an isosurface for a group of atoms to *dump image* output
- *graphics/labels* - add images or text as graphics objects to *dump image* output
- *graphics/lines* - add a trace of atom positions to *dump image* output
- *graphics/objects* - add graphics objects to *dump image* output
- *graphics/periodic* - add selected periodic images of atoms and bonds to *dump image* output
- *graphics/replica* - add atoms from other replica to *dump image* output
- *gravity* - add gravity to atoms in a granular simulation
- *grem* - implements the generalized replica exchange method
- *halt* - terminate a dynamics run or minimization
- *heat* - add/subtract momentum-conserving heat
- *heat/flow* - plain time integration of heat flow with per-atom temperature updates
- *hmc* - Hybrid/Hamiltonian Monte Carlo (HMC) particle propagation
- *hyper/global* - global hyperdynamics
- *hyper/local* - local hyperdynamics
- *ilves* - ILVES constraints on bonds and/or angles
- *imd* - implements the "Interactive MD" (IMD) protocol
- *indent* - impose force due to an indenter
- *ipi* - enable LAMMPS to run as a client for i-PI path-integral simulations
- *lambda/apip* - compute switching parameter, that controls the precision of an *APIP potential*
- *lambda/la/csp/apip* - compute a conservative switching parameter, that controls the precision of an *APIP potential*
- *langevin* - Langevin temperature control
- *langevin/drude* - Langevin temperature control of Drude oscillators
- *langevin/eff* - Langevin temperature control for the electron force field model
- *langevin/spin* - Langevin temperature control for a spin or spin-lattice system
- *lb/fluid* - lattice-Boltzmann fluid on a uniform mesh
- *lb/momentum* - *fix momentum* replacement for use with a lattice-Boltzmann fluid
- *lb/viscous* - *fix viscous* replacement for use with a lattice-Boltzmann fluid
- *lineforce* - constrain atoms to move in a line
- *lambda_thermostat/apip* - apply energy conserving correction for an *APIP potential*
- *manifoldforce* - restrain atoms to a manifold during minimization

- *mdi/qm* - LAMMPS operates as a client for a quantum code via the MolSSI Driver Interface (MDI)
- *mdi/qmmm* - LAMMPS operates as client for QM/MM simulation with a quantum code via the MolSSI Driver Interface (MDI)
- *meso/move* - move mesoscopic SPH/SDPD particles in a prescribed fashion
- *mol/swap* - Monte Carlo atom type swapping with a molecule
- *momentum* - zero the linear and/or angular momentum of a group of atoms
- *momentum/chunk* - zero the linear and/or angular momentum of a chunk of atoms
- *move* - move atoms in a prescribed fashion
- *msst* - multi-scale shock technique (MSST) integration
- *mvv/dpd* - DPD using the modified velocity-Verlet integration algorithm
- *mvv/edpd* - constant energy DPD using the modified velocity-Verlet algorithm
- *mvv/tdpd* - constant temperature DPD using the modified velocity-Verlet algorithm
- *neb* - nudged elastic band (NEB) spring forces
- *neb/spin* - nudged elastic band (NEB) spring forces for spins
- *neighbor/swap* - kinetic Monte Carlo (kMC) atom swapping
- *nonaffine/displacement* - calculate nonaffine displacement of atoms
- *nph* - constant NPH time integration via Nose/Hoover
- *nph/asphere* - NPH for aspherical particles
- *nph/body* - NPH for body particles
- *nph/eff* - NPH for nuclei and electrons in the electron force field model
- *nph/sphere* - NPH for spherical particles
- *nphug* - constant-stress Hugoniostat integration
- *npt* - constant NPT time integration via Nose/Hoover
- *npt/asphere* - NPT for aspherical particles
- *npt/body* - NPT for body particles
- *npt/cauchy* - NPT with Cauchy stress
- *npt/eff* - NPT for nuclei and electrons in the electron force field model
- *npt/sphere* - NPT for spherical particles
- *npt/uef* - NPT style time integration with diagonal flow
- *numdiff* - numerically approximate atomic forces using finite energy differences
- *numdiff/virial* - numerically approximate virial stress tensor using finite energy differences
- *nve* - constant NVE time integration
- *nve/asphere* - NVE for aspherical particles
- *nve/asphere/noforce* - NVE for aspherical particles without forces
- *nve/body* - NVE for body particles
- *nve/dot* - rigid body constant energy time integrator for coarse grain models

- *nve/dotc/langevin* - Langevin style rigid body time integrator for coarse grain models
- *nve/eff* - NVE for nuclei and electrons in the electron force field model
- *nve/limit* - NVE with limited step length
- *nve/line* - NVE for line segments
- *nve/manifold/rattle* - NVE time integration for atoms constrained to a curved surface (manifold)
- *nve/noforce* - NVE without forces (update positions only)
- *nve/sphere* - NVE for spherical particles
- *nve/bpm/sphere* - NVE for spherical particles used in the BPM package
- *nve/spin* - NVE for a spin or spin-lattice system
- *nve/tri* - NVE for triangles
- *nvk* - constant kinetic energy time integration
- *nvt* - NVT time integration via Nose/Hoover
- *nvt/asphere* - NVT for aspherical particles
- *nvt/body* - NVT for body particles
- *nvt/eff* - NVE for nuclei and electrons in the electron force field model
- *nvt/manifold/rattle* - NVT time integration for atoms constrained to a curved surface (manifold)
- *nvt/sllod* - NVT for NEMD with SLLOD equations
- *nvt/sllod/eff* - NVT for NEMD with SLLOD equations for the electron force field model
- *nvt/sphere* - NVT for spherical particles
- *nvt/uef* - NVT style time integration with diagonal flow
- *oneway* - constrain particles on move in one direction
- *orient/bcc* - add grain boundary migration force for BCC
- *orient/fcc* - add grain boundary migration force for FCC
- *orient/eco* - add generalized grain boundary migration force
- *pafi* - constrained force averages on hyper-planes to compute free energies (PAFI)
- *pair* - access per-atom info from pair styles
- *phonon* - calculate dynamical matrix from MD simulations
- *pimd/langevin* - Feynman path-integral molecular dynamics with stochastic thermostat
- *pimd/nvt* - Feynman path-integral molecular dynamics with Nose-Hoover thermostat
- *pimd/langevin/bosonic* - Bosonic Feynman path-integral molecular dynamics for with stochastic thermostat
- *pimd/nvt/bosonic* - Bosonic Feynman path-integral molecular dynamics with Nose-Hoover thermostat
- *planeforce* - constrain atoms to move in a plane
- *plumed* - wrapper on PLUMED free energy library
- *polarize/bem/gmres* - compute induced charges at the interface between impermeable media with different dielectric constants with generalized minimum residual (GMRES)
- *polarize/bem/icc* - compute induced charges at the interface between impermeable media with different dielectric constants with the successive over-relaxation algorithm

- *polarize/functional* - compute induced charges at the interface between impermeable media with different dielectric constants with the energy variational approach
- *pour* - pour new atoms/molecules into a granular simulation domain
- *precession/spin* - apply a precession torque to each magnetic spin
- *press/berendsen* - pressure control by Berendsen barostat
- *press/langevin* - pressure control by Langevin barostat
- *print* - print text and variables during a simulation
- *propel/self* - model self-propelled particles
- *property/atom* - add customized per-atom values
- *python/invoke* - call a Python function during a simulation
- *python/move* - move particles using a Python function during a simulation run
- *qbmsst* - quantum bath multi-scale shock technique time integrator
- *qeq/comb* - charge equilibration for COMB potential
- *qeq/ctip* - charge equilibration for CTIP potential
- *qeq/dynamic* - charge equilibration via dynamic method
- *qeq/fire* - charge equilibration via FIRE minimizer
- *qeq/point* - charge equilibration via point method
- *qeq/reaxff* - charge equilibration for ReaxFF potential
- *qeq/rel/reaxff* - charge equilibration for ReaxFF potential with alternate efield implementation
- *qeq/shielded* - charge equilibration via shielded method
- *qeq/slater* - charge equilibration via Slater method
- *qmmm* - functionality to enable a quantum mechanics/molecular mechanics coupling
- *qtb* - implement quantum thermal bath scheme
- *qtpie/reaxff* - apply QTPIE charge equilibration
- *rattle* - RATTLE constraints on bonds and/or angles
- *reaxff/bonds* - write out ReaxFF bond information
- *reaxff/species* - write out ReaxFF molecule information
- *recenter* - constrain the center-of-mass position of a group of atoms
- *restrain* - constrain a bond, angle, dihedral
- *rheo* - integrator for the RHEO package
- *rheo/thermal* - thermal integrator for the RHEO package
- *rheo/oxidation* - create oxidation bonds for the RHEO package
- *rheo/pressure* - pressure calculation for the RHEO package
- *rheo/viscosity* - viscosity calculation for the RHEO package
- *rhok* - add bias potential for long-range ordered systems
- *rigid* - constrain one or more clusters of atoms to move as a rigid body with NVE integration

- *rigid/meso* - constrain clusters of mesoscopic SPH/SDPD particles to move as a rigid body
- *rigid/nph* - constrain one or more clusters of atoms to move as a rigid body with NPH integration
- *rigid/nph/small* - constrain many small clusters of atoms to move as a rigid body with NPH integration
- *rigid/npt* - constrain one or more clusters of atoms to move as a rigid body with NPT integration
- *rigid/npt/small* - constrain many small clusters of atoms to move as a rigid body with NPT integration
- *rigid/nve* - constrain one or more clusters of atoms to move as a rigid body with alternate NVE integration
- *rigid/nve/small* - constrain many small clusters of atoms to move as a rigid body with alternate NVE integration
- *rigid/nvt* - constrain one or more clusters of atoms to move as a rigid body with NVT integration
- *rigid/nvt/small* - constrain many small clusters of atoms to move as a rigid body with NVT integration
- *rigid/small* - constrain many small clusters of atoms to move as a rigid body with NVE integration
- *rx* - solve reaction kinetic ODEs for a defined reaction set
- *saed/vtk* - time-average the intensities from *compute saed*
- *set* - reset an atom property via an atom-style variable every N steps
- *setforce* - set the force on each atom
- *setforce/spin* - set magnetic precession vectors on each atom
- *settorque/atom* - set the torque on each finite-size atom
- *sgcmc* - fix for hybrid semi-grand canonical MD/MC simulations
- *shake* - SHAKE constraints on bonds and/or angles
- *shardlow* - integration of DPD equations of motion using the Shardlow splitting
- *smd* - applied a steered MD force to a group
- *smd/adjust_dt* - calculate a new stable time increment for use with SMD integrators
- *smd/integrate_tlsph* - explicit time integration with total Lagrangian SPH pair style
- *smd/integrate_ulsph* - explicit time integration with updated Lagrangian SPH pair style
- *smd/move_tri_surf* - update position and velocity near rigid surfaces using SPH integrators
- *smd/setvel* - sets each velocity component, ignoring forces, for Smooth Mach Dynamics
- *smd/wall_surface* - create a rigid wall with a triangulated surface for use in Smooth Mach Dynamics
- *sph* - time integration for SPH/DPDE particles
- *sph/stationary* - update energy and density but not position or velocity in Smooth Particle Hydrodynamics
- *spring* - apply harmonic spring force to group of atoms
- *spring/chunk* - apply harmonic spring force to each chunk of atoms
- *spring/rg* - spring on radius of gyration of group of atoms
- *spring/self* - spring from each atom to its origin
- *srd* - stochastic rotation dynamics (SRD)
- *store/force* - store force on each atom
- *store/state* - store attributes for each atom
- *surface/global* - global granular surface model, each proc has copy

- *surface/local* - local granular surface model, distributed across procs
- *tdpd/source* - add external concentration source
- *temp/berendsen* - temperature control by Berendsen thermostat
- *temp/csld* - canonical sampling thermostat with Langevin dynamics
- *temp/csvr* - canonical sampling thermostat with Hamiltonian dynamics
- *temp/rescale* - temperature control by velocity rescaling
- *temp/rescale/eff* - temperature control by velocity rescaling in the electron force field model
- *tfmc* - perform force-bias Monte Carlo with time-stamped method
- *tgnvt/drude* - NVT time integration for Drude polarizable model via temperature-grouped Nose-Hoover
- *tgnpt/drude* - NPT time integration for Drude polarizable model via temperature-grouped Nose-Hoover
- *thermal/conductivity* - Mueller-Plathe kinetic energy exchange for thermal conductivity calculation
- *ti/spring* - perform thermodynamic integration between a solid and an Einstein crystal
- *tmd* - guide a group of atoms to a new configuration
- *ttm* - two-temperature model for electronic/atomic coupling (replicated grid)
- *ttm/grid* - two-temperature model for electronic/atomic coupling (distributed grid)
- *ttm/mod* - enhanced two-temperature model with additional options
- *ttm/thermal* - a two-temperature model for thermal transport
- *tune/kspace* - auto-tune *k*-space parameters
- *vector* - accumulate a global vector every *N* timesteps
- *viscosity* - Mueller-Plathe momentum exchange for viscosity calculation
- *viscous* - viscous damping for granular simulations
- *viscous/nonlinear* - nonlinear (Schiller-Naumann) viscous drag for granular simulations
- *viscous/sphere* - viscous damping on angular velocity for granular simulations
- *wall/body/polygon* - time integration for body particles of style *rounded/polygon*
- *wall/body/polyhedron* - time integration for body particles of style *rounded/polyhedron*
- *wall/colloid* - Lennard-Jones wall interacting with finite-size particles
- *wall/ees* - wall for ellipsoidal particles
- *wall/flow* - flow boundary conditions
- *wall/gran* - frictional wall(s) for granular simulations
- *wall/gran/region* - *fix wall/region* equivalent for use with granular particles
- *wall/harmonic* - harmonic spring wall
- *wall/harmonic/outside* - harmonic spring wall for containing particles
- *wall/lj1043* - Lennard-Jones 10–4–3 wall
- *wall/lj126* - Lennard-Jones 12–6 wall
- *wall/lj93* - Lennard-Jones 9–3 wall
- *wall/lepton* - Custom Lepton expression wall

- *wall/morse* - Morse potential wall
- *wall/piston* - moving reflective piston wall
- *wall/reflect* - reflecting wall(s)
- *wall/reflect/stochastic* - reflecting wall(s) with finite temperature
- *wall/region* - use region surface as wall
- *wall/region/ees* - use region surface as wall for ellipsoidal particles
- *wall/srd* - slip/no-slip wall for SRD particles
- *wall/table* - Tabulated potential wall wall
- *widom* - Widom insertions of atoms or molecules

### 1.30.4 Restrictions

Some fix styles are part of specific packages. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info. The doc pages for individual fixes tell if it is part of a package.

### 1.30.5 Related commands

*unfix*, *fix_modify*

### 1.30.6 Default

none

## 1.31 fix_modify command

### 1.31.1 Syntax

```
fix_modify fix-ID keyword value ...
```

- fix-ID = ID of the fix to modify
- one or more keyword/value pairs may be appended
- keyword = *bodyforces* or *dynamic/dof* or *energy* or *pad* or *press* or *respa* or *temp* or *virial* or *kick*

```
bodyforces value = early or late
  early/late = compute rigid-body forces/torques early or late in the timestep
dynamic/dof value = yes or no
  yes/no = do or do not re-compute the number of degrees of freedom (DOF)␣
↪contributing to the temperature
energy value = yes or no
pad    arg = Nchar = # of characters to convert timestep to
press value = compute ID that calculates a pressure
respa value = 1 to max respa level or 0 (for outermost level)
temp value = compute ID that calculates a temperature
virial value = yes or no
vizsteps value = number of MD steps that generated graphics objects should remain␣
↪visible for fixes that support it
kick value = yes or no
```

### 1.31.2 Examples

```
fix_modify 3 temp myTemp press myPress
fix_modify 1 energy yes
fix_modify tether respa 2
fix_modify myrxns vizsteps 100
```

### 1.31.3 Description

Modify one or more parameters of a previously defined fix. Specific fix styles support specific parameters. See the doc pages of an individual fix command for info on which fix_modify keywords it supports. This doc page only describes fix_modify keywords which are supported by multiple fixes. Additional keywords, specific to a single fix, are only described on that fix's doc page.

The *temp* keyword is used to determine how a fix computes temperature. The specified compute ID must have been previously defined by the user via the *compute* command and it must be a style of compute that calculates a temperature. All fixes that compute temperatures define their own compute by default, as described in their documentation. Thus this option allows the user to override the default method for computing T.

The *press* keyword is used to determine how a fix computes pressure. The specified compute ID must have been previously defined by the user via the *compute* command and it must be a style of compute that calculates a pressure. All fixes that compute pressures define their own compute by default, as described in their documentation. Thus this option allows the user to override the default method for computing P.

The *energy* keyword can be used with fixes that support it, which is explained at the bottom of their doc page. *Energy yes* will add a contribution to the potential energy of the system. More specifically, the fix's global or per-atom energy is included in the calculation performed by the *compute pe* or *compute pe/atom* commands. The former is what is used the *thermo_style* command for output of any quantity that includes the global potential energy of the system. Note that the *compute pe* and *compute pe/atom* commands also have an option to include or exclude the contribution from fixes. For fixes that tally a global energy, it can also be printed with thermodynamic output by using the keyword f_ID in the thermo_style custom command, where ID is the fix-ID of the appropriate fix.

**Note**

If you are performing an *energy minimization* with one of these fixes and want the energy and forces it produces to be part of the optimization criteria, you must specify the *energy yes* setting.

**Note**

For most fixes that support the *energy* keyword, the default setting is *no*. For a few it is *yes*, when a user would expect that to be the case. The page of each fix gives the default.

The *virial* keyword can be used with fixes that support it, which is explained at the bottom of their doc page. *Virial yes* will add a contribution to the virial of the system. More specifically, the fix's global or per-atom virial is included in the calculation performed by the *compute pressure* or *compute stress/atom* commands. The former is what is used the *thermo_style* command for output of any quantity that includes the global pressure of the system. Note that the *compute pressure* and *compute stress/atom* commands also have an option to include or exclude the contribution from fixes.

**Note**

If you are performing an *energy minimization* with *box relaxation* and one of these fixes and want the virial contribution of the fix to be part of the optimization criteria, you must specify the *virial yes* setting.

**Note**

For most fixes that support the *virial* keyword, the default setting is *no*. For a few it is *yes*, when a user would expect that to be the case. The page of each fix gives the default.

For fixes that set or modify forces, it may be possible to select at which *r-RESPA* level the fix operates via the *respa* keyword. The RESPA level at which the fix is active can be selected. This is a number ranging from 1 to the number of levels. If the RESPA level is larger than the current maximum, the outermost level will be used, which is also the default setting. This default can be restored using a value of *0* for the RESPA level. The affected fix has to be enabled to support this feature; if not, *fix_modify* will report an error. Active fixes with a custom RESPA level setting are reported with their specified level at the beginning of a r-RESPA run.

The *dynamic/dof* keyword determines whether the number of atoms N in the fix group and their associated degrees of freedom are re-computed each time a temperature is computed. Only fix styles that calculate their own internal temperature use this option. Currently this is only the *fix rigid/nvt/small* and *fix rigid/npt/small* commands for the purpose of thermostatting rigid body translation and rotation. By default, N and their DOF are assumed to be constant. If you are adding atoms or molecules to the system (see the *fix pour*, *fix deposit*, and *fix gcmc* commands) or expect atoms or molecules to be lost (e.g. due to exiting the simulation box or via *fix evaporate*), then this option should be used to ensure the temperature is correctly normalized.

**Note**

Other thermostatting fixes, such as *fix nvt*, do not use the *dynamic/dof* keyword because they use a temperature compute to calculate temperature. See the *compute_modify dynamic/dof* command for a similar way to ensure correct temperature normalization for those thermostats.

The *bodyforces* keyword determines whether the forces and torques acting on rigid bodies are computed *early* at the post-force stage of each timestep (right after per-atom forces have been computed and communicated among processors), or *late* at the final-integrate stage of each timestep (after any other fixes have finished their post-force tasks). Only the rigid-body integration fixes use this option, which includes *fix rigid* and *fix rigid/small*, and their variants.

The default is *late*. If there are other fixes that add forces to individual atoms, then the rigid-body constraints will include these forces when time-integrating the rigid bodies. If *early* is specified, then new fixes can be written that use or modify the per-body force and torque, before time-integration of the rigid bodies occurs. Note however this has the side effect, that fixes such as *fix addforce*, *fix setforce*, *fix spring*, which add forces to individual atoms will have no effect on the motion of the rigid bodies if they are specified in the input script after the fix rigid command. LAMMPS will give a warning if that is the case.

Added in version 2Apr2025.

The *pad* keyword only applies when a fix produces a file and the output filename is specified with a wildcard "*" character which becomes the timestep. If *pad* is 0, which is the default, the timestep is converted into a string of unpadded length (e.g., 100 or 12000 or 2000000). When *pad* is specified with *Nchar* $>$ 0, the string is padded with leading zeroes so they are all the same length = *Nchar*. For example, pad 7 would yield 0000100, 0012000, 2000000. This can be useful so that post-processing programs can easily read the files in ascending timestep order. Please see the documentation of the individual fix styles if this keyword is supported.

Added in version 11Feb2026.

The *vizsteps* keyword only applies when a fix produces event based graphics objects, e.g. atoms that were involved in

a reaction or a Monte Carlo swap, move, or insert. It determines for how many time steps the graphics objects will remain visible in the corresponding *dump image output*.

The *kick* keyword can only be used with *fix nvt/sllod* and *fix nvt/sllod/eff*. If *kick* is *yes* and velocity is stored in the laboratory frame, the velocity profile consistent with *fix deform* will be superimposed at the start of the next run. If velocity is stored in the peculiar frame, the *kick* flag is ignored.

### 1.31.4 Restrictions

none

### 1.31.5 Related commands

*fix*, *compute temp*, *compute pressure*, *thermo_style*

### 1.31.6 Default

The option defaults are temp = ID defined by fix, press = ID defined by fix, energy = no, virial = different for each fix style, respa = 0, bodyforce = late.

## 1.32 fitpod command

### 1.32.1 Syntax

```
fitpod Ta_param.pod Ta_data.pod Ta_coefficients.pod
```

- fitpod = style name of this command
- Ta_param.pod = an input file that describes proper orthogonal descriptors (PODs)
- Ta_data.pod = an input file that specifies DFT data used to fit a POD potential
- Ta_coefficients.pod (optional) = an input file that specifies trainable coefficients of a POD potential

### 1.32.2 Examples

```
fitpod Ta_param.pod Ta_data.pod
fitpod Ta_param.pod Ta_data.pod Ta_coefficients.pod
```

### 1.32.3 Description

Added in version 22Dec2022.

Fit a machine-learning interatomic potential (ML-IAP) based on proper orthogonal descriptors (POD); please see *(Nguyen and Rohskopf)*, *(Nguyen2023)*, *(Nguyen2024)*, and *(Nguyen and Sema)* for details. The fitted POD potential can be used to run MD simulations via *pair_style pod*.

Two input files are required for this command. The first input file describes a POD potential parameter settings, while the second input file specifies the DFT data used for the fitting procedure. All keywords except *species* have default values. If a keyword is not set in the input file, its default value is used. The table below has one-line descriptions of all the keywords that can be used in the first input file (i.e. `Ta_param.pod`)

| Keyword | Default | Type | Description |
|---|---|---|---|
| species | (none) | STRING | Chemical symbols for all elements in the system and have to match XYZ training files. |
| pbc | 1 1 1 | INT | three integer constants specify boundary conditions |
| rin | 0.5 | REAL | a real number specifies the inner cut-off radius |
| rcut | 5.0 | REAL | a real number specifies the outer cut-off radius |
| bessel_polynomial_degree | 4 | INT | the maximum degree of Bessel polynomials |
| inverse_polynomial_degree | 8 | INT | the maximum degree of inverse radial basis functions |
| number_of_environment_clusters | 1 | INT | the number of clusters for environment-adaptive potentials |
| number_of_principal_components | 2 | INT | the number of principal components for dimensionality reduction |
| onebody | 1 | BOOL | turns on/off one-body potential |
| twobody_number_radial_basis_functions | 8 | INT | number of radial basis functions for two-body potential |
| threebody_number_radial_basis_functions | 6 | INT | number of radial basis functions for three-body potential |
| threebody_angular_degree | 5 | INT | angular degree for three-body potential |
| fourbody_number_radial_basis_functions | 4 | INT | number of radial basis functions for four-body potential |
| fourbody_angular_degree | 3 | INT | angular degree for four-body potential |
| fivebody_number_radial_basis_functions | 0 | INT | number of radial basis functions for five-body potential |
| fivebody_angular_degree | 0 | INT | angular degree for five-body potential |
| sixbody_number_radial_basis_functions | 0 | INT | number of radial basis functions for six-body potential |
| sixbody_angular_degree | 0 | INT | angular degree for six-body potential |
| sevenbody_number_radial_basis_functions | 0 | INT | number of radial basis functions for seven-body potential |
| sevenbody_angular_degree | 0 | INT | angular degree for seven-body potential |

Note that both the number of radial basis functions and angular degree must decrease as the body order increases. The next table describes all keywords that can be used in the second input file (i.e. `Ta_data.pod` in the example above):

| Keyword | Default | Type | Description |
|---|---|---|---|
| file_format | extxyz | STRING | only the extended xyz format (extxyz) is currently supported |
| file_extension | xyz | STRING | extension of the data files |
| path_to_training_data_set | (none) | STRING | specifies the path to training data files in double quotes |
| path_to_test_data_set | "" | STRING | specifies the path to test data files in double quotes |
| path_to_environment_configuration_set | "" | STRING | specifies the path to environment configuration files in double quotes |
| fraction_training_data_set | 1.0 | REAL | a real number (<= 1.0) specifies the fraction of the training set used to fit POD |
| randomize_training_data_set | 0 | BOOL | turns on/off randomization of the training set |
| fraction_test_data_set | 1.0 | REAL | a real number (<= 1.0) specifies the fraction of the test set used to validate POD |
| randomize_test_data_set | 0 | BOOL | turns on/off randomization of the test set |
| fitting_weight_energy | 100.0 | REAL | a real constant specifies the weight for energy in the least-squares fit |
| fitting_weight_force | 1.0 | REAL | a real constant specifies the weight for force in the least-squares fit |
| fitting_regularization_parameter | 1.0e-10 | REAL | a real constant specifies the regularization parameter in the least-squares fit |
| error_analysis_for_training_data_set | 0 | BOOL | turns on/off error analysis for the training data set |
| error_analysis_for_test_data_set | 0 | BOOL | turns on/off error analysis for the test data set |
| basename_for_output_files | pod | STRING | a basename string added to the output files |
| precision_for_pod_coefficients | 8 | INT | number of digits after the decimal points for numbers in the coefficient file |
| group_weights | global | STRING | `table` uses group weights defined for each group named by filename |

All keywords except *path_to_training_data_set* have default values. If a keyword is not set in the input file, its default value is used. After successful training, a number of output files are produced, if enabled:

- <basename>`_training_errors.pod` reports the errors in energy and forces for the training data set
- <basename>`_training_analysis.pod` reports detailed errors for all training configurations
- <basename>`_test_errors.pod` reports errors for the test data set
- <basename>`_test_analysis.pod` reports detailed errors for all test configurations
- <basename>`_coefficients.pod` contains the coefficients of the POD potential

After training the POD potential, `Ta_param.pod` and <basename>`_coefficients.pod` are the two files needed to use the POD potential in LAMMPS. See *pair_style pod* for using the POD potential. Examples about training and using POD potentials are found in the directory lammps/examples/PACKAGES/pod and the Github repo https://github.com/cesmix-mit/pod-examples.

### Loss Function Group Weights

The *group_weights* keyword in the `data.pod` file is responsible for weighting certain groups of configurations in the loss function. For example:

```
group_weights table
Displaced_A15 100.0 1.0
Displaced_BCC 100.0 1.0
Displaced_FCC 100.0 1.0
Elastic_BCC   100.0 1.0
Elastic_FCC   100.0 1.0
GSF_110       100.0 1.0
GSF_112       100.0 1.0
Liquid        100.0 1.0
Surface       100.0 1.0
Volume_A15    100.0 1.0
Volume_BCC    100.0 1.0
Volume_FCC    100.0 1.0
```

This will apply an energy weight of `100.0` and a force weight of `1.0` for all groups in the `Ta` example. The groups are named by their respective filename. If certain groups are left out of this table, then the globally defined weights from the `fitting_weight_energy` and `fitting_weight_force` keywords will be used.

## 1.32.4 POD Potential

We consider a multi-element system of $N$ atoms with $N_e$ unique elements. We denote by $r_n$ and $Z_n$ position vector and type of an atom $n$ in the system, respectively. Note that we have $Z_n \in \{1, \ldots, N_e\}$, $R = (r_1, r_2, \ldots, r_N) \in \mathbb{R}^{3N}$, and $Z = (Z_1, Z_2, \ldots, Z_N) \in \mathbb{N}^N$. The total energy of the POD potential is expressed as $E(R, Z) = \sum_{i=1}^{N} E_i(R_i, Z_i)$, where

$$E_i(R_i, Z_i) \;=\; \sum_{m=1}^{M} c_m \mathcal{D}_{im}(R_i, Z_i)$$

Here $c_m$ are trainable coefficients and $\mathcal{D}_{im}(R_i, Z_i)$ are per-atom POD descriptors. Summing the per-atom descriptors over $i$ yields the global descriptors $d_m(R, Z) = \sum_{i=1}^{N} \mathcal{D}_{im}(R_i, Z_i)$. It thus follows that $E(R, Z) = \sum_{m=1}^{M} c_m d_m(R, Z)$.

The per-atom POD descriptors include one, two, three, four, five, six, and seven-body descriptors, which can be specified in the first input file. Furthermore, the per-atom POD descriptors also depend on the number of environment clusters specified in the first input file. Please see *(Nguyen2024)* and *(Nguyen and Sema)* for the detailed description of the per-atom POD descriptors.

## 1.32.5 Training

A POD potential is trained using the least-squares regression against density functional theory (DFT) data. Let $J$ be the number of training configurations, with $N_j$ being the number of atoms in the j-th configuration. The training configurations are extracted from the extended XYZ files located in a directory (i.e., path_to_training_data_set in the second input file). Let $\{E_j^\star\}_{j=1}^J$ and $\{F_j^\star\}_{j=1}^J$ be the DFT energies and forces for $J$ configurations. Next, we calculate the global descriptors and their derivatives for all training configurations. Let $d_{jm}$, $1 \le m \le M$, be the global descriptors associated with the j-th configuration, where $M$ is the number of global descriptors. We then form a matrix $A \in \mathbb{R}^{J \times M}$ with entries $A_{jm} = d_{jm}/N_j$ for $j = 1, \ldots, J$ and $m = 1, \ldots, M$. Moreover, we form a matrix $B \in \mathbb{R}^{\mathcal{N} \times M}$ by stacking the derivatives of the global descriptors for all training configurations from top to bottom, where $\mathcal{N} = 3\sum_{j=1}^{J} N_j$.

The coefficient vector $c$ of the POD potential is found by solving the following least-squares problem

$$\min_{c \in \mathbb{R}^M} w_E \|Ac - \bar{E}^\star\|^2 + w_F \|Bc + F^\star\|^2 + w_R \|c\|^2,$$

where $w_E$ and $w_F$ are weights for the energy (*fitting_weight_energy*) and force (*fitting_weight_force*), respectively; and $w_R$ is the regularization parameter (*fitting_regularization_parameter*). Here $\bar{E}^\star \in \mathbb{R}^J$ is a vector of with entries $\bar{E}_j^\star = E_j^\star/N_j$ and $F^\star$ is a vector of $\mathcal{N}$ entries obtained by stacking $\{F_j^\star\}_{j=1}^J$ from top to bottom.

### 1.32.6 Validation

POD potential can be validated on a test dataset in a directory specified by setting path_to_test_data_set in the second input file. It is possible to validate the POD potential after the training is complete. This is done by providing the coefficient file as an input to *fitpod*, for example,

```
fitpod Ta_param.pod Ta_data.pod Ta_coefficients.pod
```

### 1.32.7 Restrictions

This command is part of the ML-POD package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 1.32.8 Related commands

*pair_style pod*, *compute pod/atom*, *compute podd/atom*, *compute pod/local*, *compute pod/global*

### 1.32.9 Default

The keyword defaults are also given in the description of the input files.

---

**(Nguyen and Rohskopf)** Nguyen and Rohskopf, Journal of Computational Physics, 480, 112030, (2023).

**(Nguyen2023)** Nguyen, Physical Review B, 107(14), 144103, (2023).

**(Nguyen2024)** Nguyen, Journal of Computational Physics, 113102, (2024).

**(Nguyen and Sema)** Nguyen and Sema, https://arxiv.org/abs/2405.00306, (2024).

## 1.33 geturl command

### 1.33.1 Syntax

```
geturl url keyword args ...
```

- url = URL of the file to download
- zero or more keyword argument pairs may be provided
- keyword = *output* or *overwrite* or *timeout* or *verify* or *verbose*

  *output* filename = write to *filename* instead of inferring the name from the URL
  *overwrite* yes/no = if *yes* overwrite the output file in case it exists, do not if *no*
  *timeout* time = stop download if not completed within given time in seconds
  *verify* yes/no = verify SSL certificate and hostname if *yes*, do not if *no*
  *verbose* yes/no = if *yes* write verbose debug output from libcurl to screen, do not if *no*

### 1.33.2 Examples

```
geturl https://www.ctcms.nist.gov/potentials/Download/1990--Ackland-G-J-Vitek-V--Cu/2/Cu2.eam.fs
geturl https://github.com/lammps/lammps/blob/develop/bench/in.lj output in.bench-lj
```

### 1.33.3 Description

Added in version 29Aug2024.

Download a file from an URL to the local disk. This is implemented with the libcurl library which supports a large variety of protocols including "http", "https", "ftp", "scp", "sftp", "file". The transfer will only be performed on MPI rank 0.

The *output* keyword can be used to set the filename. By default, the last part of the URL is used.

The *overwrite* keyword determines whether a file should be overwritten if it already exists. If the argument is *no*, then the download will be skipped if the file exists.

The *timeout* keyword can be used to modify the timeout for downloads in seconds. After the timeout, the download will stop, even if incomplete. The default time value is 300, i.e. 5 minutes. Setting the timeout to 0 means to wait forever.

The *verify* keyword determines whether `libcurl` will validate the SSL certificate and hostname for encrypted connections. Turning this off may be required when using a proxy or connecting to a server with a self-signed SSL certificate.

The *verbose* keyword determines whether a detailed protocol of the steps performed by libcurl is written to the screen. Using the argument *yes* can be used to debug connection issues when the *geturl* command does not behave as expected. If the argument is *no*, geturl will operate silently and only report the error status number provided by libcurl, in case of a failure.

**Using *geturl* with proxies for http or https**

The libcurl library supports routing traffic through proxies by setting suitable environment variables (e.g. `http_proxy` or `https_proxy`) as required by some institutional or corporate security protocols. In that case you probably also want to use the *verify no* setting.

Using a proxy may also be needed if you are running on an HPC cluster where only the login or head nodes have access to the internet, but not the compute nodes. In this case the following input can be adapted and used for your local HPC environment:

```
variable headnode getenv PBS_O_HOST     # use SLURM_SUBMIT_HOST when using SLURM␣
↪instead of Torque/PBS
shell ssh -N -f -D 8001 ${headnode}     # start SOCKS5 proxy with backgrounded ssh␣
↪connection to cluster head node
shell putenv http_proxy=socks5://localhost:8001 https_proxy=socks5://localhost:8001
geturl https://download.lammps.org/tars/SHA256SUMS # download a file using proxy
shell head SHA256SUMS               # check if the download was successful
```

### 1.33.4 Restrictions

This command is part of the EXTRA-COMMAND package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info. It also requires that LAMMPS was built with support for the libcurl library. See the page about *Compiling LAMMPS with libcurl support* for further info. If support for libcurl is not included, using *geturl* will trigger an error.

### 1.33.5 Related commands

*shell*

### 1.33.6 Default

*overwrite* = yes, *timeout* = 300, *verify* = yes, *verbose* = no

## 1.34 group command

### 1.34.1 Syntax

```
group ID style args
```

- ID = user-defined name of the group
- style = *delete* or *clear* or *empty* or *region* or *type* or *id* or *molecule* or *variable* or *include* or *subtract* or *union* or *intersect* or *dynamic* or *static*

```
delete = no args
clear = no args
empty = no args
region args = region-ID
type or id or molecule
  args = list of one or more atom types (1-Ntypes or type label), atom IDs, or␣
↪molecule IDs
    any numeric entry in list can be a sequence formatted as A:B or A:B:C where
    A = starting index, B = ending index,
    C = increment between indices, 1 if not specified
  args = logical value
    logical = "<" or "<=" or ">" or ">=" or "==" or "!="
    value = an atom type (1-Ntypes or type label) or atom ID or molecule ID␣
↪(depending on style)
  args = logical value1 value2
    logical = "<>"
    value1,value2 = atom types or atom IDs or molecule IDs (depending on style)
variable args = variable-name
include args = molecule
  molecule = add atoms to group with same molecule ID as atoms already in group
subtract args = two or more group IDs
union args = one or more group IDs
intersect args = two or more group IDs
dynamic args = parent-ID keyword value ...
  one or more keyword/value pairs may be appended
  keyword = region or var or property or every or exclude or include or within
    region value = region-ID
    var value = name of variable
    property value = name of custom integer or floating point vector
    every value = N = update group every this many timesteps
    exclude value = group-ID = exclude atoms in this group from dynamic group
    include args = molecule
      molecule = add atoms to the dynamic group with same molecule ID as atoms␣
↪already selected for the dynamic group
    within value = cutoff = add atoms to the dynamic group that are within the␣
↪given cutoff distance of the atoms already selected for the dynamic group
static = no args
```

### 1.34.2 Examples

```
group edge region regstrip
group water type 3 4
group water type OW HT
group sub id 10 25 50
group sub id 10 25 50 500:1000
group sub id 100:10000:10
group sub id <= 150
group polyA molecule <> 50 250
group hienergy variable eng
group hienergy include molecule
group boundary subtract all a2 a3
group boundary union lower upper
group boundary intersect upper flow
group boundary delete
group mine dynamic all region myRegion every 100
group solvation dynamic peptide within 3.5 include molecule every 100 exclude peptide
```

### 1.34.3 Description

Identify a collection of atoms as belonging to a group. The group ID can then be used in other commands such as *fix*, *compute*, *dump*, or *velocity* to act on those atoms together.

If the group ID already exists, the group command adds the specified atoms to the group.

**Note**

By default groups are static, meaning the atoms are permanently assigned to the group. For example, if the *region* style is used to assign atoms to a group, the atoms will remain in the group even if they later move out of the region. As explained below, the *dynamic* style can be used to make a group dynamic so that a periodic determination is made as to which atoms are in the group. Since many LAMMPS commands operate on groups of atoms, you should think carefully about whether making a group dynamic makes sense for your model.

A group with the ID *all* is predefined. All atoms belong to this group. This group cannot be deleted, or made dynamic.

The *delete* style removes the named group and un-assigns all atoms that were assigned to that group. Since there is a restriction (see below) that no more than 32 groups can be defined at any time, the *delete* style allows you to remove groups that are no longer needed, so that more can be specified. You cannot delete a group if it has been used to define a current *fix* or *compute* or *dump*.

The *clear* style un-assigns all atoms that were assigned to that group. This may be dangerous to do during a simulation run (e.g., using the *run every* command if a fix or compute or other operation expects the atoms in the group to remain constant), but LAMMPS does not check for this.

The *empty* style creates an empty group, which is useful for commands like *fix gcmc* or with complex scripts that add atoms to a group.

The *region* style puts all atoms in the region volume into the group. Note that this is a static one-time assignment. The atoms remain assigned (or not assigned) to the group even if they later move out of the region volume. For extended triangle or line-segment particle, the center point of the triangle or line-segment is used to determine whether the particle is in the region or not.

The *type*, *id*, and *molecule* styles put all atoms with the specified atom types, atom IDs, or molecule IDs into the group. These three styles can use arguments specified in one of two formats.

The first format is a list of values (types or IDs). For example, the second command in the examples above puts all atoms of type 3 or 4 into the group named *water*. Each numeric entry in the list can be a colon-separated sequence `A:B` or `A:B:C`, as in two of the examples above. A "sequence" generates a sequence of values (types or IDs), with an optional increment. The first example with `500:1000` has the default increment of 1 and would add all atom IDs from 500 to 1000 (inclusive) to the group sub, along with 10, 25, and 50 since they also appear in the list of values. The second example with `100:10000:10` uses an increment of 10 and would thus add atoms IDs 100, 110, 120, . . . , 9990, 10000 to the group sub.

The second format is a *logical* followed by one or two values (type or ID). The 7 valid logicals are listed above. All the logicals except <> take a single argument. The third example above adds all atoms with IDs from 1 to 150 to the group named *sub*. The logical <> means "between" and takes 2 arguments. The fourth example above adds all atoms belonging to molecules with IDs from 50 to 250 (inclusive) to the group named polyA. For the *type* style, type labels are converted into numeric types before being evaluated.

The *variable* style evaluates a variable to determine which atoms to add to the group. It must be an *atom-style variable* previously defined in the input script. If the variable evaluates to a non-zero value for a particular atom, then that atom is added to the specified group.

Atom-style variables can specify formulas that include thermodynamic quantities, per-atom values such as atom coordinates, or per-atom quantities calculated by computes, fixes, or other variables. They can also include Boolean logic where two numeric values are compared to yield a 1 or 0 (effectively a true or false). Thus, using the *variable* style is a general way to flag specific atoms to include or exclude from a group.

For example, these lines define a variable "eatom" that calculates the potential energy of each atom and includes it in the group if its potential energy is above the threshold value −3.0.

```
compute         1 all pe/atom
compute         2 all reduce sum c_1
thermo_style    custom step temp pe c_2
run             0 post no

variable        eatom atom "c_1 > -3.0"
group           hienergy variable eatom
```

Note that these lines

```
compute         2 all reduce sum c_1
thermo_style    custom step temp pe c_2
run             0 post no
```

are necessary to ensure that the "eatom" variable is current when the group command invokes it. Because the eatom variable computes the per-atom energy via the pe/atom compute, it will only be current if a run has been performed which evaluated pairwise energies, and the pe/atom compute was actually invoked during the run. Printing the thermodynamic info for compute 2 ensures that this is the case, since it sums the pe/atom compute values (in the reduce compute) to output them to the screen. See the "Variable Accuracy" section of the *variable* page for more details on ensuring that variables are current when they are evaluated between runs.

The *include* style with its arg *molecule* adds atoms to a group that have the same molecule ID as atoms already in the group. The molecule ID = 0 is ignored in this operation, since it is assumed to flag isolated atoms that are not part of molecules. An example of where this operation is useful is if the *region* style has been used previously to add atoms to a group that are within a geometric region. If molecules straddle the region boundary, then atoms outside the region that are part of molecules with atoms inside the region will not be in the group. Using the group command a second time with *include molecule* will add those atoms that are outside the region to the group.

**Note**

The *include molecule* operation is relatively expensive in a parallel sense. This is because it requires communication of relevant molecule IDs between all the processors and each processor to loop over its atoms once per processor, to compare its atoms to the list of molecule IDs from every other processor. Hence it scales as N, rather than N/P as most of the group operations do, where N is the number of atoms, and P is the number of processors.

The *subtract* style takes a list of two or more existing group names as arguments. All atoms that belong to the first group, but not to any of the other groups are added to the specified group.

The *union* style takes a list of one or more existing group names as arguments. All atoms that belong to any of the listed groups are added to the specified group.

The *intersect* style takes a list of two or more existing group names as arguments. Atoms that belong to every one of the listed groups are added to the specified group.

---

The *dynamic* style flags an existing or new group as dynamic. This means atoms will be (re)assigned to the group periodically as a simulation runs. This is in contrast to static groups where atoms are permanently assigned to the group. The way the assignment occurs is as follows. Only atoms in the group specified as the parent group via the parent-ID are initially assigned to the dynamic group before the following selection conditions are applied. The *include* and *within* keywords can then trigger including additional atoms which may be outside the parent group.

If the *region* keyword is used, atoms not in the specified region are removed from the dynamic group.

If the *var* keyword is used, the variable name must be an atom-style or atomfile-style variable. The variable is evaluated and atoms whose per-atom values are 0.0, are removed from the dynamic group.

If the *property* keyword is used, the name refers to a custom integer or floating point per-atom vector defined via the *fix property/atom* command. This means the values in the vector can be read as part of a data file with the *read_data* command or specified with the *set* command. Or accessed and changed via the *library interface to LAMMPS*, or by styles you add to LAMMPS (pair, fix, compute, etc) which access the custom vector and modify its values. Which means the values can be modified between or during simulations. Atoms whose values in the custom vector are zero are removed from the dynamic group. Note that the name of the custom per-atom vector is specified just as *name*, not as *i_name* or *d_name* as it is for other commands that use different kinds of custom atom vectors or arrays as arguments.

The assignment of atoms to a dynamic group is done at the beginning of each run and on every timestep that is a multiple of *N*, which is the argument for the *every* keyword (*N* = 1 is the default). For an energy minimization, via the *minimize* command, an assignment is made at the beginning of the minimization, but not during the iterations of the minimizer.

Added in version 30Mar2026.

The optional *exclude* keyword has a group ID as argument and after all other atom selections for the dynamic group have been performed, it removes selected atoms that are in the specified group from the dynamic group. This can for example be used to first select additional solvent atoms up to a given distance from a group using the *within* keyword (see below), but then exclude the non-solvent atoms.

Added in version 11Feb2026.

The optional *include* keyword with its argument *molecule* adds atoms to a dynamic group that have the same molecule ID as atoms already in the group. Atoms with the molecule ID = 0 are ignored in this operation, since that ID is generally assumed to flag isolated atoms that are not part of molecules. An example of where this operation is useful is if the dynamic group is defined using a *region*. If molecules straddle the region boundary, then atoms outside the region that are part of molecules with atoms inside the region will be added to the group (see above).

Added in version 30Mar2026.

The optional *within* keyword with its *cutoff* argument adds atoms to a dynamic group that are within the given cutoff distance from any of the atoms selected for the dynamic group. Since this requires processing a neighbor list and computing distances for potentially many pairs of atoms plus and additional forward communication step, this can be an expensive operation that should be used in combination with the *every* option and a value of (e.g. 10 or larger) to reduce the overhead for updating the selection. Outside of flow simulations and similar, atoms will mostly oscillate around their positions and diffuse on a much larger timescale, so infrequent group updates will not be a problem. Also when selecting atoms for output to a *dump file* it is not necessary to update the selection more often than the dump interval.

The point in the timestep at which atoms are assigned to a dynamic group is **after** interatomic forces have been computed, but **before** any fixes which alter forces or otherwise update the system have been invoked. This means that atom positions have been updated, neighbor lists and ghost atoms are current, and both intermolecular and intramolecular forces have been calculated based on the new coordinates. Thus the region criterion, if applied, should be accurate. Also, any computes invoked by an atom-style variable should use updated information for that timestep (e.g., potential energy/atom or coordination number/atom). Similarly, fixes or computes which are invoked after that point in the timestep, should operate on the new group of atoms.

**Note**

If the *region* keyword is used to determine what atoms are in the dynamic group, atoms can move outside of the simulation box between reneighboring events. Thus if you want to include all atoms on the left side of the simulation box, you probably want to set the left boundary of the region to be outside the simulation box by some reasonable amount (e.g., up to the cutoff of the potential), else they may be excluded from the dynamic region.

Here is an example of using a dynamic group to shrink the set of atoms being integrated by using a spherical region with a variable radius (shrinking from 18 to 5 over the course of the run). This could be used to model a quench of the system, freezing atoms outside the shrinking sphere, then converting the remaining atoms to a static group and running further.

```
variable        nsteps equal 5000
variable        rad equal 18-(step/v_nsteps)*(18-5)
region          ss sphere 20 20 0 v_rad
group           mobile dynamic all region ss
fix             1 mobile nve
run             ${nsteps}
group           mobile static
run             ${nsteps}
```

**Note**

All fixes and computes take a group ID as an argument, but they do not all allow for use of a dynamic group. If you get an error message that this is not allowed, but feel that it should be for the fix or compute in question, then please post your reasoning to the LAMMPS forum at MatSci and we can look into changing it. The same applies if you come across inconsistent behavior when dynamic groups are allowed.

The *static* style removes the setting for a dynamic group, converting it to a static group (the default). The atoms in the static group are those currently in the dynamic group.

### 1.34.4 Restrictions

There can be no more than 32 groups defined at one time, including "all".

The parent group of a dynamic group cannot itself be a dynamic group.

### 1.34.5 Related commands

*dump*, *fix*, *region*, *velocity*

### 1.34.6 Default

All atoms belong to the "all" group.

## 1.35 group2ndx command

## 1.36 ndx2group command

### 1.36.1 Syntax

```
group2ndx file args
ndx2group file args
```

- file = name of index file to write out or read in
- args = zero or more group IDs may be appended

### 1.36.2 Examples

```
group2ndx allindex.ndx
group2ndx someindex.ndx upper lower mobile
ndx2group someindex.ndx
ndx2group someindex.ndx mobile
```

### 1.36.3 Description

Write or read a Gromacs style index file in text format that associates atom IDs with the corresponding group definitions. This index file can be used with in combination with Gromacs analysis tools or to import group definitions into the *fix colvars* input file.

It can also be used to save and restore group definitions for static groups using the individual atom IDs. This may be important if the original group definition depends on a region or otherwise on the geometry and thus cannot be easily recreated.

Another application would be to import atom groups defined for Gromacs simulation into LAMMPS. When translating Gromacs topology and geometry data to LAMMPS.

The *group2ndx* command will write group definitions to an index file. Without specifying any group IDs, all groups will be written to the index file. When specifying group IDs, only those groups will be written to the index file. In order to follow the Gromacs conventions, the group *all* will be renamed to *System* in the index file.

The *ndx2group* command will create or update group definitions from those stored in an index file. Without specifying any group IDs, all groups except *System* will be read from the index file and the corresponding groups recreated. If a group of the same name already exists, it will be completely reset. When specifying group IDs, those groups, if present, will be read from the index file and restored.

### 1.36.4 File Format

The file format is equivalent and compatible with what is produced by the Gromacs make_ndx command. and follows the Gromacs definition of an ndx file

Each group definition begins with the group name in square brackets with blanks, e.g. [ water ] and is then followed by the list of atom indices, which may be spread over multiple lines. Here is a small example file:

```
[ Oxygen ]
1  4  7
[ Hydrogen ]
2  3  5  6
8  9
[ Water ]
1 2 3 4 5 6 7 8 9
```

The index file defines 3 groups: Oxygen, Hydrogen, and Water and the latter happens to be the union of the first two.

---

### 1.36.5 Restrictions

These commands require that atoms have atom IDs, since this is the information that is written to the index file.

These commands are part of the EXTRA-COMMAND package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 1.36.6 Related commands

*group*, *dump*, *fix colvars*

### 1.36.7 Default

none

## 1.37 hyper command

### 1.37.1 Syntax

```
hyper N Nevent fix-ID compute-ID keyword values ...
```

- N = # of timesteps to run
- Nevent = check for events every this many steps
- fix-ID = ID of a fix that applies a global or local bias potential, can be NULL
- compute-ID = ID of a compute that identifies when an event has occurred
- zero or more keyword/value pairs may be appended
- keyword = *min* or *dump* or *rebond*

```
min values = etol ftol maxiter maxeval
  etol = stopping tolerance for energy, used in quenching
  ftol = stopping tolerance for force, used in quenching
  maxiter = max iterations of minimize, used in quenching
  maxeval = max number of force/energy evaluations, used in quenching
dump value = dump-ID
```

```
    dump-ID = ID of dump to trigger whenever an event takes place
  rebond value = Nrebond
    Nrebond = frequency at which to reset bonds, even if no event has occurred
```

### 1.37.2 Examples

```
compute event all event/displace 1.0
fix HG mobile hyper/global 3.0 0.3 0.4 800.0
hyper 5000 100 HG event min 1.0e-6 1.0e-6 100 100 dump 1 dump 5
```

### 1.37.3 Description

Run a bond-boost hyperdynamics (HD) simulation where time is accelerated by application of a bias potential to one or more pairs of nearby atoms in the system. This command can be used to run both global and local hyperdynamics. In global HD a single bond within the system is biased on each timestep. In local HD multiple bonds (separated by a sufficient distance) can be biased simultaneously at each timestep. In the bond-boost hyperdynamics context, a "bond" is not a covalent bond between a pair of atoms in a molecule. Rather it is simply a pair of nearby atoms as discussed below.

Both global and local HD are described in *(Voter2013)* by Art Voter and collaborators. Similar to parallel replica dynamics (PRD), global and local HD are methods for performing accelerated dynamics that are suitable for infrequent-event systems that obey first-order kinetics. A good overview of accelerated dynamics methods (AMD) for such systems in given in *(Voter2002)* from the same group. To quote from the review paper: "The dynamical evolution is characterized by vibrational excursions within a potential basin, punctuated by occasional transitions between basins. The transition probability is characterized by p(t) = k*exp(-kt) where k is the rate constant."

Both HD and PRD produce a time-accurate trajectory that effectively extends the timescale over which a system can be simulated, but they do it differently. HD uses a single replica of the system and accelerates time by biasing the interaction potential in a manner such that each timestep is effectively longer. PRD creates Nr replicas of the system and runs dynamics on each independently with a normal unbiased potential until an event occurs in one of the replicas. The time between events is reduced by a factor of Nr replicas. For both methods, per CPU second, more physical time elapses and more events occur. See the *prd* page for more info about PRD.

An HD run has several stages, which are repeated each time an event occurs, as explained below. The logic for an HD run is as follows:

```
quench
create initial list of bonds

while (time remains):
  run dynamics for Nevent steps
  quench
  check for an event
  if event occurred: reset list of bonds
  restore pre-quench state
```

The list of bonds is the list of atom pairs of atoms that are within a short cutoff distance of each other after the system energy is minimized (quenched). This list is created and reset by a *fix hyper/global* or *fix hyper/local* command specified as *fix-ID*. At every dynamics timestep, the same fix selects one of more bonds to apply a bias potential to.

**Note**

The style of fix associated with the specified *fix-ID* determines whether you are running the global versus local hyperdynamics algorithm.

Dynamics (with the bias potential) is run continuously, stopping every *Nevent* steps to check if a transition event has occurred. The specified *N* for total steps must be a multiple of *Nevent*. check is performed by quenching the system and comparing the resulting atom coordinates to the coordinates from the previous basin.

A quench is an energy minimization and is performed by whichever algorithm has been defined by the *min_style* command. Minimization parameters may be set via the *min_modify* command and by the *min* keyword of the hyper command. The latter are the settings that would be used with the *minimize* command. Note that typically, you do not need to perform a highly-converged minimization to detect a transition event, though you may need to in order to prevent a set of atoms in the system from relaxing to a saddle point.

The event check is performed by a compute with the specified *compute-ID*. Currently there is only one compute that works with the hyper command, which is the *compute event/displace* command. Other event-checking computes may be added. *Compute event/displace* checks whether any atom in the compute group has moved further than a specified threshold distance. If so, an event has occurred.

If this happens, the list of bonds is reset, since some bond pairs are likely now too far apart, and new pairs are likely close enough to be considered a bond. The pre-quenched state of the system (coordinates and velocities) is restored, and dynamics continue.

At the end of the hyper run, a variety of statistics are output to the screen and logfile. These include info relevant to both global and local hyperdynamics, such as the number of events and the elapsed hyper time (accelerated time), And it includes info specific to one or the other, depending on which style of fix was specified by *fix-ID*.

---

The optional keywords operate as follows.

As explained above, the *min* keyword can be used to specify parameters for the quench. Their meaning is the same as for the *minimize* command

The *dump* keyword can be used to trigger a specific dump command with the specified *dump-ID* to output a snapshot each time an event is detected. It can be specified multiple times with different *dump-ID* values, as in the example above. These snapshots will be for the quenched state of the system on a timestep that is a multiple of *Nevent*, i.e. a timestep after the event has occurred. Note that any dump command in the input script will also output snapshots at whatever timestep interval it defines via its *N* argument; see the *dump* command for details. This means if you only want a particular dump to output snapshots when events are detected, you should specify its *N* as a value larger than the length of the hyperdynamics run.

As in the code logic above, the bond list is normally only reset when an event occurs. The *rebond* keyword will force a reset of the bond list every *Nrebond* steps, even if an event has not occurred. *Nrebond* must be a multiple of *Nevent*. This can be useful to check if more frequent resets alter event statistics, perhaps because the parameters chosen for defining what is a bond and what is an event are producing bad dynamics in the presence of the bias potential.

---

### 1.37.4 Restrictions

This command can only be used if LAMMPS was built with the REPLICA package. See the *Build package* doc page for more info.

### 1.37.5 Related commands

*fix hyper/global*, *fix hyper/local*, *compute event/displace*, *prd*

### 1.37.6 Default

The option defaults are min = 0.1 0.1 40 50 and time = steps.

---

**(Voter2013)** S. Y. Kim, D. Perez, A. F. Voter, J Chem Phys, 139, 144110 (2013).

**(Voter2002)** Voter, Montalenti, Germann, Annual Review of Materials Research 32, 321 (2002).

# 1.38 if command

## 1.38.1 Syntax

```
if boolean then t1 t2 ... elif boolean f1 f2 ... elif boolean f1 f2 ... else e1 e2 ...
```

- boolean = a Boolean expression evaluated as TRUE or FALSE (see below)
- then = required word
- t1,t2,...,tN = one or more LAMMPS commands to execute if condition is met, each enclosed in quotes
- elif = optional word, can appear multiple times
- f1,f2,...,fN = one or more LAMMPS commands to execute if elif condition is met, each enclosed in quotes (optional arguments)
- else = optional argument
- e1,e2,...,eN = one or more LAMMPS commands to execute if no condition is met, each enclosed in quotes (optional arguments)

## 1.38.2 Examples

```
if "${steps} > 1000" then quit
if "${myString} == a10" then quit
if "$x <= $y" then "print 'X is smaller = $x'" else "print 'Y is smaller = $y'"
if "(${eng} > 0.0) || ($n < 1000)" then &
  "timestep 0.005" &
elif $n<10000 &
  "timestep 0.01" &
else &
  "timestep 0.02" &
  "print 'Max step reached'"
if "${eng} > ${eng_previous}" then "jump file1" else "jump file2"
```

## 1.38.3 Description

This command provides an if-then-else capability within an input script. A Boolean expression is evaluated and the result is TRUE or FALSE. Note that as in the examples above, the expression can contain variables, as defined by the *variable* command, which will be evaluated as part of the expression. Thus a user-defined formula that reflects the current state of the simulation can be used to issue one or more new commands.

If the result of the Boolean expression is TRUE, then one or more commands (t1, t2, ..., tN) are executed. If it is FALSE, then Boolean expressions associated with successive elif keywords are evaluated until one is found to be true, in which case its commands (f1, f2, ..., fN) are executed. If no Boolean expression is TRUE, then the commands associated with the else keyword, namely (e1, e2, ..., eN), are executed. The elif and else keywords and their associated commands are optional. If they are not specified and the initial Boolean expression is FALSE, then no commands are executed.

The syntax for Boolean expressions is described below.

Each command (t1, f1, e1, etc.) can be any valid LAMMPS input script command. If the command is more than one word, it must enclosed in quotes, so it will be treated as a single argument, as in the examples above.

**Note**

If a command itself requires a quoted argument (e.g., a *print* command), then double and single quotes can be used and nested in the usual manner, as in the examples above and below. The *Commands parse* page has more details on using quotes in arguments. Only one of level of nesting is allowed, but that should be sufficient for most use cases.

Note that by using the line continuation character "&", the if command can be spread across many lines, though it is still a single command:

```
if "$a < $b" then &
  "print 'Minimum value = $a'" &
  "run 1000" &
else &
  'print "Minimum value = $b"' &
  "minimize 0.001 0.001 1000 10000"
```

Note that if one of the commands to execute is *quit*, as in the first example above, then executing the command will cause LAMMPS to halt.

Note that by jumping to a label in the same input script, the if command can be used to break out of a loop. See the *variable delete* command for info on how to delete the associated loop variable, so that it can be re-used later in the input script.

Here is an example of a loop which checks every 1000 steps if the system temperature has reached a certain value, and if so, breaks out of the loop to finish the run. Note that any variable could be checked, so long as it is current on the timestep when the run completes. As explained on the *variable* doc page, this can be ensured by including the variable in thermodynamic output.

```
variable myTemp equal temp
label loop
variable a loop 1000
run 1000
if "${myTemp} < 300.0" then "jump SELF break"
next a
jump SELF loop
label break
print "ALL DONE"
```

Here is an example of a double loop which uses the if and *jump* commands to break out of the inner loop when a condition is met, then continues iterating through the outer loop.

```
label       loopa
variable    a loop 5
  label     loopb
  variable  b loop 5
    print     "A,B = $a,$b"
    run       10000
    if        "$b > 2" then "jump SELF break"
  next      b
  jump      in.script loopb
  label       break
  variable    b delete
next        a
```

(continues on next page)

(continued from previous page)

```
jump        SELF loopa
```

The Boolean expressions for the if and elif keywords have a C-like syntax. Note that each expression is a single argument within the if command. Thus if you want to include spaces in the expression for clarity, you must enclose the entire expression in quotes.

An expression is built out of numbers (which start with a digit or period or minus sign) or strings (which start with a letter and can contain alphanumeric characters, underscores, or forward slashes):

```
0.2, 100, 1.0e20, -15.4, ...
InP, myString, a123, ab_23_cd, lj/cut, ...
```

and Boolean operators:

```
A == B, A != B, A < B, A <= B, A > B, A >= B, A && B, A || B, A |^ B, !A
```

Each A and B is a number or string or a variable reference like `$a` or `${abc}`, or A or B can be another Boolean expression.

Note that all variables used will be substituted for before the Boolean expression in evaluated. A variable can produce a number, like an *equal-style variable*, or it can produce a string, like an *index-style variable*.

The Boolean operators == and != can operate on a pair or strings or numbers. They cannot compare a number to a string. All the other Boolean operations can only operate on numbers.

Expressions are evaluated left to right and have the usual C-style precedence: the unary logical NOT operator ! has the highest precedence, the 4 relational operators <, <=, >, and >= are next; the two remaining relational operators == and != are next; then the logical AND operator &&; and finally the logical OR operator || and logical XOR (exclusive or) operator |^ have the lowest precedence. Parenthesis can be used to group one or more portions of an expression and/or enforce a different order of evaluation than what would occur with the default precedence.

When the six relational operators (first six in list above) compare two numbers, they return either a 1.0 or 0.0 depending on whether the relationship between A and B is TRUE or FALSE.

When the three logical operators (last three in list above) compare two numbers, they also return either a 1.0 or 0.0 depending on whether the relationship between A and B is TRUE or FALSE (or just A). The logical AND operator will return 1.0 if both its arguments are non-zero, else it returns 0.0. The logical OR operator will return 1.0 if either of its arguments is non-zero, else it returns 0.0. The logical XOR operator will return 1.0 if one of its arguments is zero and the other non-zero, else it returns 0.0. The logical NOT operator returns 1.0 if its argument is 0.0, else it returns 0.0. The 3 logical operators can only be used to operate on numbers, not on strings.

The overall Boolean expression produces a TRUE result if the numeric result is non-zero. If the result is zero, the expression result is FALSE.

**Note**

If the Boolean expression is a single numeric value with no Boolean operators, it will be FALSE if the value = 0.0, otherwise TRUE. If the Boolean expression is a single string, an error message will be issued.

### 1.38.4 Restrictions

none

### 1.38.5 Related commands

*variable*, *print*

### 1.38.6 Default

none

## 1.39 improper_coeff command

### 1.39.1 Syntax

```
improper_coeff N args
```

- N = numeric improper type (see asterisk form below), or type label
- args = coefficients for one or more improper types

### 1.39.2 Examples

```
improper_coeff 1 300.0 0.0
improper_coeff * 80.2 -1 2
improper_coeff *4 80.2 -1 2

labelmap improper 1 benzene
improper_coeff benzene 300.0 0.0
```

### 1.39.3 Description

Specify the improper force field coefficients for one or more improper types. The number and meaning of the coefficients depends on the improper style. Improper coefficients can also be set in the data file read by the *read_data* command or in a restart file.

*N* can be specified in one of two ways. An explicit numeric value can be used, as in the first example above. Or *N* can be a type label, which is an alphanumeric string defined by the *labelmap* command or in a section of a data file read by the *read_data* command.

For numeric values only, a wild-card asterisk can be used to set the coefficients for multiple improper types. This takes the form "*" or "*n" or "n*" or "m*n". If *N* = the number of improper types, then an asterisk with no numeric values means all types from 1 to *N*. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to *N* (inclusive). A middle asterisk means all types from m to n (inclusive).

Note that using an improper_coeff command can override a previous setting for the same improper type. For example, these commands set the coeffs for all improper types, then overwrite the coeffs for just improper type 2:

```
improper_coeff * 300.0 0.0
improper_coeff 2 50.0 0.0
```

A line in a data file that specifies improper coefficients uses the exact same format as the arguments of the improper_coeff command in an input script, except that wild-card asterisks should not be used since coefficients for all *N* types must be listed in the file. For example, under the "Improper Coeffs" section of a data file, the line that corresponds to the first example above would be listed as

```
1 300.0 0.0
```

The *improper_style class2* is an exception to this rule, in that an additional argument is used in the input script to allow specification of the cross-term coefficients. See its doc page for details.

---

The list of all improper styles defined in LAMMPS is given on the *improper_style* doc page. They are also listed in more compact form on the *Commands improper* doc page.

On either of those pages, click on the style to display the formula it computes and its coefficients as specified by the associated improper_coeff command.

---

### 1.39.4 Restrictions

This command must come after the simulation box is defined by a *read_data*, *read_restart*, or *create_box* command.

An improper style must be defined before any improper coefficients are set, either in the input script or in a data file.

### 1.39.5 Related commands

*improper_style*

### 1.39.6 Default

none

## 1.40 improper_style command

### 1.40.1 Syntax

```
improper_style style
```

- style = *none* or *hybrid* or *class2* or *cvff* or *harmonic*

### 1.40.2 Examples

```
improper_style harmonic
improper_style cvff
improper_style hybrid cvff harmonic
```

### 1.40.3 Description

Set the formula(s) LAMMPS uses to compute improper interactions between quadruplets of atoms, which remain in force for the duration of the simulation. The list of improper quadruplets is read in by a *read_data* or *read_restart* command from a data or restart file. Note that the ordering of the 4 atoms in an improper quadruplet determines the definition of the improper angle used in the formula for each style. See the doc pages of individual styles for details.

Hybrid models where impropers are computed using different improper potentials can be setup using the *hybrid* improper style.

The coefficients associated with an improper style can be specified in a data or restart file or via the *improper_coeff* command.

All improper potentials store their coefficient data in binary restart files which means improper_style and *improper_coeff* commands do not need to be re-specified in an input script that restarts a simulation. See the *read_restart* command for details on how to do this. The one exception is that improper_style *hybrid* only stores the list of sub-styles in the restart file; improper coefficients need to be re-specified.

**Note**

When both an improper and pair style is defined, the *special_bonds* command often needs to be used to turn off (or weight) the pairwise interaction that would otherwise exist between a group of 4 bonded atoms.

---

Here is an alphabetic list of improper styles defined in LAMMPS. Click on the style to display the formula it computes and coefficients specified by the associated *improper_coeff* command.

Click on the style to display the formula it computes, any additional arguments specified in the improper_style command, and coefficients specified by the associated *improper_coeff* command.

There are also additional accelerated pair styles included in the LAMMPS distribution for faster performance on CPUs, GPUs, and KNLs. The individual style names on the *Commands improper* page are followed by one or more of (g,i,k,o,t) to indicate which accelerated styles exist.

- *none* - turn off improper interactions
- *zero* - topology but no interactions
- *hybrid* - define multiple styles of improper interactions
- *amoeba* - AMOEBA out-of-plane improper
- *class2* - COMPASS (class 2) improper
- *cossq* - improper with a cosine squared term
- *cvff* - CVFF improper
- *distance* - improper based on distance between atom planes
- *distharm* - improper that is harmonic in the out-of-plane distance
- *fourier* - improper with multiple cosine terms
- *harmonic* - harmonic improper
- *inversion/harmonic* - harmonic improper with Wilson-Decius out-of-plane definition
- *ring* - improper which prevents planar conformations
- *umbrella* - DREIDING improper
- *sqdistharm* - improper that is harmonic in the square of the out-of-plane distance

---

### 1.40.4 Restrictions

Improper styles can only be set for atom_style choices that allow impropers to be defined.

Most improper styles are part of the MOLECULE package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info. The doc pages for individual improper potentials tell if it is part of a package.

### 1.40.5 Related commands

*improper_coeff*

### 1.40.6 Default

```
improper_style none
```

## 1.41 include command

### 1.41.1 Syntax

```
include file
```

- file = filename of new input script to switch to

### 1.41.2 Examples

```
include newfile
include in.run2
```

### 1.41.3 Description

This command opens a new input script file and begins reading LAMMPS commands from that file. When the new file is finished, the original file is returned to. Include files can be nested as deeply as desired. If input script A includes script B, and B includes A, then LAMMPS could run for a long time.

If the filename is a variable (see the *variable* command), different processor partitions can run different input scripts.

### 1.41.4 Restrictions

none

### 1.41.5 Related commands

*variable*, *jump*

### 1.41.6 Default

none

## 1.42 info command

### 1.42.1 Syntax

```
info args
```

- args = one or more of the following keywords: *out*, *all*, *system*, *memory*, *communication*, *computes*, *dumps*, *fixes*, *groups*, *regions*, *variables*, *coeffs*, *styles*, *time*, *accelerator*, *fft* or *configuration*
- *out* values = *screen*, *log*, *append* filename, *overwrite* filename
- *styles* values = *all*, *angle*, *atom*, *bond*, *compute*, *command*, *dump*, *dihedral*, *fix*, *improper*, *integrate*, *kspace*, *minimize*, *pair*, *region*

### 1.42.2 Examples

```
info system
info groups computes variables
info all out log
info all out append info.txt
info styles all
info styles atom styles command
```

### 1.42.3 Description

Print out information about the current internal state of the running LAMMPS process. This can be helpful when debugging or validating complex input scripts. Several output categories are available and one or more output categories may be requested. All category keywords take no arguments, only *out* and *styles* take arguments as shown below. The keywords are cumulative, may be abbreviated, and unknown keywords are ignored.

The *out* flag controls where the output is sent. It can only be sent to one target. By default this is the screen, if it is active. The *log* argument selects the log file instead. With the *append* and *overwrite* option, followed by a filename, the output is written to that file, which is either appended to or overwritten, respectively.

The *all* flag activates printing all categories listed below.

The *configuration* category prints some information about the LAMMPS version as well as architecture and OS it is run on.

The *memory* category prints some information about the current memory allocation of MPI rank 0 (this the amount of dynamically allocated memory reported by LAMMPS classes). Where supported, also some OS specific information about the size of the reserved memory pool size (this is where malloc() and the new operator request memory from) and the maximum resident set size is reported (this is the maximum amount of physical memory occupied so far).

The *system* category prints a general system overview listing. This includes the unit style, atom style, number of atoms, bonds, angles, dihedrals, and impropers and the number of the respective types, box dimensions and properties, force computing styles and more.

The *communication* category prints a variety of information about communication and parallelization: the MPI library version level, the number of MPI ranks and OpenMP threads, the communication style and layout, the processor grid dimensions, ghost atom communication mode, cutoff, and related settings.

The *computes* category prints a list of all currently defined computes, their IDs and styles and groups they operate on.

The *dumps* category prints a list of all currently active dumps, their IDs, styles, filenames, groups, and dump frequencies.

The *fixes* category prints a list of all currently defined fixes, their IDs and styles and groups they operate on.

The *groups* category prints a list of all currently defined groups.

The *regions* category prints a list of all currently defined regions, their IDs and styles and whether "inside" or "outside" atoms are selected.

The *variables* category prints a list of all currently defined variables, their names, styles, definition and last computed value, if available.

The *coeffs* category prints a list for each defined force style (pair, bond, angle, dihedral, improper) indicating which of the corresponding coefficients have been set. This can be very helpful to debug error messages like "All pair coeffs are not set".

The *accelerator* category prints out information about compile time settings of included accelerator support for the GPU, KOKKOS, INTEL, and OPENMP packages.

Added in version 7Feb2024.

The *fft* category prints out information about the included 3d-FFT support. This lists the 3d-FFT engine, FFT precision, FFT library used by the FFT engine. If the KOKKOS package is included, the settings used for the KOKKOS package are displayed as well.

The *styles* category prints the list of styles available in the current LAMMPS binary. The *styles* keyword without option is the same as using the "all" option. One of the following options may be used to control which category of styles is printed out. To select multiple categories, the styles keyword needs to be used multiple times with the desired categories:

- all
- angle
- atom
- bond
- compute
- command
- dump
- dihedral
- fix
- improper
- integrate
- kspace
- minimize
- pair
- region

The *time* category prints the accumulated CPU and wall time for the process that writes output (usually MPI rank 0).

### 1.42.4 Restrictions

none

### 1.42.5 Related commands

*print*

### 1.42.6 Default

The *out* option has the default *screen*.

The *styles* option has the default *all*.

## 1.43 jump command

### 1.43.1 Syntax

```
jump file label
```

- file = filename of new input script to switch to

- label = optional label within file to jump to

### 1.43.2 Examples

```
jump newfile
jump in.run2 runloop
jump SELF runloop
```

### 1.43.3 Description

This command closes the current input script file, opens the file with the specified name, and begins reading LAMMPS commands from that file. Unlike the *include* command, the original file is not returned to, although by using multiple jump commands it is possible to chain from file to file or back to the original file.

If the word "SELF" is used for the filename, then the current input script is re-opened and read again.

**Note**

The SELF option is not guaranteed to work when the current input script is being read through stdin (standard input), e.g.

```
lmp_g++ < in.script
```

since the SELF option invokes the C-library rewind() call, which may not be supported for stdin on some systems or by some MPI implementations. This can be worked around by using the *-in command-line switch*, e.g.

```
lmp_g++ -in in.script
```

or by using the *-var command-line switch* to pass the script name as a variable to the input script. In the latter case, a *variable* called "fname" could be used in place of SELF, e.g.

```
lmp_g++ -var fname in.script < in.script
```

The second argument to the jump command is optional. If specified, it is treated as a label and the new file is scanned (without executing commands) until the label is found, and commands are executed from that point forward. This can be used to loop over a portion of the input script, as in this example. These commands perform 10 runs, each of 10000 steps, and create 10 dump files named file.1, file.2, etc. The *next* command is used to exit the loop after 10 iterations. When the "a" variable has been incremented for the tenth time, it will cause the next jump command to be skipped.

```
variable a loop 10
label loop
dump 1 all atom 100 file.$a
run 10000
undump 1
next a
jump in.lj loop
```

If the jump *file* argument is a variable, the jump command can be used to cause different processor partitions to run different input scripts. In this example, LAMMPS is run on 40 processors, with 4 partitions of 10 procs each. An in.file containing the example variable and jump command will cause each partition to run a different simulation.

```
mpirun -np 40 lmp_ibm -partition 4x10 -in in.file
```

(continues on next page)

(continued from previous page)

```
variable f world script.1 script.2 script.3 script.4
jump $f
```

Here is an example of a loop which checks every 1000 steps if the system temperature has reached a certain value, and if so, breaks out of the loop to finish the run. Note that any variable could be checked, so long as it is current on the timestep when the run completes. As explained on the *variable* doc page, this can be ensured by including the variable in thermodynamic output.

```
variable myTemp equal temp
label loop
variable a loop 1000
run 1000
if "${myTemp} < 300.0" then "jump SELF break"
next a
jump SELF loop
label break
print "ALL DONE"
```

Here is an example of a double loop which uses the if and *jump* commands to break out of the inner loop when a condition is met, then continues iterating through the outer loop.

```
label       loopa
variable    a loop 5
  label     loopb
  variable  b loop 5
    print     "A,B = $a,$b"
    run       10000
    if        "$b > 2" then "jump SELF break"
  next      b
  jump      in.script loopb
  label       break
  variable    b delete
next        a
jump        SELF loopa
```

### 1.43.4 Restrictions

If you jump to a file and it does not contain the specified label, LAMMPS will come to the end of the file and exit.

### 1.43.5 Related commands

*variable*, *include*, *label*, *next*

### 1.43.6 Default

none

# 1.44 kim command

## 1.44.1 Syntax

```
kim sub-command args
```

- sub-command = *init* or *interactions* or *query* or *param* or *property*
- args = arguments used by a particular sub-command

## 1.44.2 Examples

```
kim init args
kim interactions args
kim query args
kim param args
kim property args
```

## 1.44.3 Description

The *kim command* includes a set of sub-commands that allow LAMMPS users to use interatomic models (IM) (potentials and force fields) and their predictions for various physical properties archived in the Open Knowledgebase of Interatomic Models (OpenKIM) repository.

Using OpenKIM provides LAMMPS users with immediate access to a large number of verified IMs and their predictions. OpenKIM IMs have multiple benefits including reliability, reproducibility and convenience.

There are two types of IMs archived in OpenKIM:

1. The first type is called a *KIM Portable Model* (PM). A KIM PM is an independent computer implementation of an IM written in one of the languages supported by KIM (C, C++, Fortran) that conforms to the KIM Application Programming Interface (KIM API) Portable Model Interface (PMI) standard. A KIM PM will work seamlessly with any simulation code that supports the KIM API/PMI standard (including LAMMPS; see complete list of supported codes).
2. The second type is called a *KIM Simulator Model* (SM). A KIM SM is an IM that is implemented natively within a simulation code (*simulator*) that supports the KIM API Simulator Model Interface (SMI); in this case LAMMPS. A separate SM package is archived in OpenKIM for each parameterization of the IM, which includes all of the necessary parameter files, LAMMPS commands, and metadata (supported species, units, etc.) needed to run the IM in LAMMPS.

With these two IM types, OpenKIM can archive and test almost all IMs that can be used by LAMMPS. (It is easy to contribute new IMs to OpenKIM, see the upload instructions.)

OpenKIM IMs are uniquely identified by a KIM ID. The extended KIM ID consists of a human-readable prefix identifying the type of IM, authors, publication year, and supported species, separated by two underscores from the KIM ID itself, which begins with an IM code (*MO* for a KIM Portable Model, and *SM* for a KIM Simulator Model) followed by a unique 12-digit code and a 3-digit version identifier. By convention SM prefixes begin with *Sim_* to readily identify them.

```
SW_StillingerWeber_1985_Si__MO_405512056662_005
Sim_LAMMPS_ReaxFF_StrachanVanDuinChakraborty_2003_CHNO__SM_107643900657_001
```

Each OpenKIM IM has a dedicated "Model Page" on OpenKIM providing all the information on the IM including a title, description, authorship and citation information, test and verification check results, visualizations of results, a wiki with documentation and user comments, and access to raw files, and other information. The URL for the Model Page is constructed from the extended KIM ID of the IM:

https://openkim.org/doc/schema/kim\unhbox\voidb@x\kern\z@\char‘\protect\
discretionary{\char\defaulthyphenchar}{}{}ids/#extended\unhbox\voidb@x\kern\z@\char‘\
protect\discretionary{\char\defaulthyphenchar}{}{}kim\unhbox\voidb@x\kern\z@\char‘\
protect\discretionary{\char\defaulthyphenchar}{}{}ids

For example, for the Stillinger-Weber potential listed above the Model Page is located at:

https://openkim.org/id/SW_StillingerWeber_1985_Si__MO_405512056662_005

See the current list of KIM PMs and SMs archived in OpenKIM. This list is sorted by species and can be filtered to display only IMs for certain species combinations.

See Obtaining KIM Models to learn how to install a pre-built binary of the OpenKIM Repository of Models.

**Note**

It is also possible to locally install IMs not archived in OpenKIM, in which case their names do not have to conform to the KIM ID format.

### 1.44.4 Using OpenKIM IMs with LAMMPS (*kim init*, *kim interactions*)

Two sub-commands are employed when using OpenKIM IMs in LAMMPS, one to select the IM and perform necessary initialization (*kim init*), and the second to set up the IM for use by executing any necessary LAMMPS commands (*kim interactions*). Both are required.

#### Syntax

```
kim init model user_units unitarg
kim interactions typeargs
```

- model = name of the KIM interatomic model (the KIM ID for models archived in OpenKIM)
- user_units = the LAMMPS *units* style assumed in the LAMMPS input script
- unitarg = *unit_conversion_mode* (optional)
- typeargs = atom type to species mapping (one entry per atom type) or *fixed_types* for models with a preset fixed mapping

#### Examples

```
kim init SW_StillingerWeber_1985_Si__MO_405512056662_005 metal
kim interactions Si

kim init Sim_LAMMPS_ReaxFF_StrachanVanDuinChakraborty_2003_CHNO__SM_107643900657_001 real
kim init Sim_LAMMPS_ReaxFF_StrachanVanDuinChakraborty_2003_CHNO__SM_107643900657_001␣
↪metal unit_conversion_mode
kim interactions C H O

kim init Sim_LAMMPS_IFF_PCFF_HeinzMishraLinEmami_2015Ver1v5_
↪FccmetalsMineralsSolventsPolymers__SM_039297821658_000 real
kim interactions fixed_types
```

See the *examples/kim* directory for example input scripts that use KIM PMs and KIM SMs.

### OpenKIM IM Initialization (*kim init*)

The *kim* command followed by *init* sub-command must be issued **before** the simulation box is created (normally at the top of the file). This command sets the OpenKIM IM that will be used and may issue additional commands changing LAMMPS default settings that are required for using the selected IM (such as *units* or *atom_style*). If needed, those settings can be overridden, however, typically a script containing a *kim init* command would not include *units* and *atom_style* commands.

The required arguments of *kim init* are the *model* name of the IM to be used in the simulation (for an IM archived in OpenKIM this is its extended KIM ID, and the *user_units*, which are the LAMMPS *units style* used in the input script. (Any dimensioned numerical values in the input script and values read in from files are expected to be in the *user_units* system.)

The selected IM can be either a *KIM PM or a KIM SM*. For a KIM SM, the *kim init* command verifies that the SM is designed to work with LAMMPS (and not another simulation code). In addition, the LAMMPS version used for defining the SM and the LAMMPS version being currently run are printed to help diagnose any incompatible changes to input script or command syntax between the two LAMMPS versions.

Based on the selected model *kim init* may modify the *atom_style*. Some SMs have requirements for this setting. If this is the case, then *atom_style* will be set to the required style. Otherwise, the value is left unchanged (which in the absence of an *atom_style* command in the input script is the *default atom_style value*).

Regarding units, the *kim init* behaves in different ways depending on whether or not *unit conversion mode* is activated as indicated by the optional *unitarg* argument. If unit conversion mode is **not** active, then *user_units* must either match the required units of the IM or the IM must be able to adjust its units to match. (The latter is only possible with some KIM PMs; SMs can never adjust their units.) If a match is possible, the LAMMPS *units* command is called to set the units to *user_units*. If the match fails, the simulation is terminated with an error. The *kim init* command also sets the default value for the *skin* (extra distance beyond force cutoff) as 2.0 Angstroms and sets the default value for the *timestep* size as 1.0 femtosecond.

Here is an example of a LAMMPS script to compute the cohesive energy of a face-centered cubic (fcc) lattice for the MEAM potential by Pascuet and Fernandez (2015) for Al.

```
kim           init Sim_LAMMPS_MEAM_PascuetFernandez_2015_Al__SM_811588957187_000 metal
boundary      p p p
lattice       fcc 4.049
region        simbox block 0 1 0 1 0 1 units lattice
create_box    1 simbox
create_atoms  1 box
mass          1 26.981539
kim           interactions Al
run           0
variable      Ec equal (pe/count(all))
print         "Cohesive Energy = ${Ec} eV"
```

The above script will end with an error in the *kim init* line if the IM is changed to another potential for Al that does not work with *metal* units. To address this, *kim init* offers the *unit_conversion_mode* as shown below.

If unit conversion mode *is* active, then *kim init* calls the LAMMPS *units* command to set the units to the IM's required or preferred units. Conversion factors between the IM's units and the *user_units* are defined for all *physical quantities* (mass, distance, etc.). (Note that converting to or from the "lj" unit style is not supported.) These factors are stored as *internal style variables* with the following standard names:

```
_u_mass
_u_distance
_u_time
_u_energy
```

(continues on next page)

(continued from previous page)

```
_u_velocity
_u_force
_u_torque
_u_temperature
_u_pressure
_u_viscosity
_u_charge
_u_dipole
_u_efield
_u_density
```

If desired, the input script can be designed to work with these conversion factors so that the script will work without change with any OpenKIM IM. (This approach is used in the OpenKIM Testing Framework.)

For example, the script given above for the cohesive energy of fcc Al can be rewritten to work with any IM regardless of units. The following script constructs an fcc lattice with a lattice parameter defined in meters, computes the total energy, and prints the cohesive energy in Joules regardless of the units of the IM.

```
kim          init Sim_LAMMPS_MEAM_PascuetFernandez_2015_Al__SM_811588957187_000 si unit_
↪conversion_mode
boundary     p p p
lattice      fcc $(4.049e-10*v__u_distance)
region       simbox block 0 1 0 1 0 1 units lattice
create_box   1 simbox
create_atoms 1 box
mass         1 $(4.480134e-26*v__u_mass)
kim          interactions Al
neighbor     $(2e-10*v__u_distance) bin
run          0
variable     Ec_in_J equal (pe/count(all))/v__u_energy
print        "Cohesive Energy = ${Ec_in_J} J"
```

Note the multiplication by *v__u_distance* and *v__u_mass* to convert from SI units (specified in the *kim init* command) to whatever units the IM uses (metal in this case), and the division by *v__u_energy* to convert from the IM's energy units to SI units (Joule). This script will work correctly for any IM for Al (KIM PM or SM) selected by the *kim init* command.

Care must be taken to apply unit conversion to dimensional variables read in from a file. For example, if a configuration of atoms is read in from a dump file using the *read_dump* command, the following can be done to convert the box and all atomic positions to the correct units:

```
change_box all x scale ${_u_distance} &
               y scale ${_u_distance} &
               z scale ${_u_distance} &
               xy final $(xy*v__u_distance) &
               xz final $(xz*v__u_distance) &
               yz final $(yz*v__u_distance) &
               remap
```

**Note**

Unit conversion will only work if the conversion factors are placed in all appropriate places in the input script. It is up to the user to do this correctly.

### OpenKIM IM Execution (*kim interactions*)

The second and final step in using an OpenKIM IM is to execute the *kim interactions* command. This command must be preceded by a *kim init* command and a command that defines the number of atom types *N* (such as *create_box*). The *kim interactions* command has one argument *typeargs*. This argument contains either a list of *N* chemical species, which defines a mapping between atom types in LAMMPS to the available species in the OpenKIM IM, or the keyword *fixed_types* for models that have a preset fixed mapping (i.e. the mapping between LAMMPS atom types and chemical species is defined by the model and cannot be changed). In the latter case, the user must consult the model documentation to see how many atom types there are and how they map to the chemical species.

For example, consider an OpenKIM IM that supports Si and C species. If the LAMMPS simulation has four atom types, where the first three are Si, and the fourth is C, the following *kim interactions* command would be used:

```
kim interactions Si Si Si C
```

Alternatively, for a model with a fixed mapping the command would be:

```
kim interactions fixed_types
```

The *kim interactions* command performs all the necessary steps to set up the OpenKIM IM selected in the *kim init* command. The specific actions depend on whether the IM is a KIM PM or a KIM SM. For a KIM PM, a *pair_style kim* command is executed followed by the appropriate *pair_coeff* command. For example, for the Ercolessi and Adams (1994) KIM PM for Al set by the following commands:

```
kim init EAM_Dynamo_ErcolessiAdams_1994_Al__MO_123629422045_005 metal
...
...  box specification lines skipped
...
kim interactions Al
```

the *kim interactions* command executes the following LAMMPS input commands:

```
pair_style kim EAM_Dynamo_ErcolessiAdams_1994_Al__MO_123629422045_005
pair_coeff * * Al
```

For a KIM SM, the generated input commands may be more complex and require that LAMMPS is built with the required packages included for the type of potential being used. The set of commands to be executed is defined in the SM specification file, which is part of the SM package. For example, for the Strachan et al. (2003) ReaxFF SM set by the following commands:

```
kim init Sim_LAMMPS_ReaxFF_StrachanVanDuinChakraborty_2003_CHNO__SM_107643900657_000 real
...
...  box specification lines skipped
...
kim interactions C H N O
```

the *kim interactions* command executes the following LAMMPS input commands:

```
pair_style reaxff lmp_control safezone 2.0 mincap 100
pair_coeff * * ffield.reax.rdx C H N O
fix reaxqeq all qeq/reaxff 1 0.0 10.0 1.0e-6 param.qeq
```

**Note**

The files *lmp_control*, *ffield.reax.rdx* and *param.qeq* are specific to the Strachan et al. (2003) ReaxFF parameterization and are archived as part of the SM package in OpenKIM.

**Note**

Parameters like cutoff radii and charge tolerances, which have an effect on IM predictions, are also included in the SM definition ensuring reproducibility.

**Note**

When using *kim init* and *kim interactions* to select and set up an OpenKIM IM, other LAMMPS commands for the same functions (such as pair_style, pair_coeff, bond_style, bond_coeff, fixes related to charge equilibration, etc.) should normally not appear in the input script.

**Note**

*kim interactions* must be called each time after the *change_box* command to provide the correct settings (it should be called with the same *typeargs* as the first call.) The reason is that changing a periodic boundary to a non-periodic one, or in general, using the *change_box* command after the interactions are set via *kim interactions* or *pair_coeff* commands might affect some of the settings. For example, SM models containing Coulombic terms in the interactions require different settings if a periodic boundary changes to a non-periodic one. In other cases, the second call to *kim interactions* does not affect any other settings.

### 1.44.5 Using OpenKIM Web Queries in LAMMPS (*kim query*)

The *kim query* command performs a web query to retrieve the predictions of an IM set by *kim init* for material properties archived in OpenKIM, or to obtain a list of available models that support a given set of atomic species.

#### Syntax

```
kim query variable formatarg query_function queryargs
```

- variable(s) = single name or list of names of (string style) LAMMPS variable(s) where a query result or parameter get result is stored. Variables that do not exist will be created by the command
- formatarg = *list* or *split* or *index* (optional)

```
list = returns a single string with a list of space separated values
       (e.g. "1.0 2.0 3.0"), which is placed in a LAMMPS variable as
       defined by the variable argument. [default]
split = returns the values separately in new variables with names based
       on the prefix specified in variable and a number appended to
       indicate which element in the list of values is in the variable
index = returns a variable style index that can be incremented via the
       next command.  This enables the construction of simple loops
```

- query_function = name of the OpenKIM web API query function to be used
- queryargs = a series of *keyword=value* pairs that represent the web query; supported keywords depend on the query function

### Examples

```
kim query a0 get_lattice_constant_cubic crystal=[fcc] species=[Al] units=[angstrom]
kim query model index get_available_models species=[Al] potential_type=[eam]
```

The result of the query is stored in one or more *string style variables* as determined by the optional *formatarg* argument. For the "list" setting of *formatarg* (or if *formatarg* is not specified), the result is returned as a space-separated list of values in *variable*. The *formatarg* keyword "split" separates the result values into individual variables of the form *prefix_I*, where *prefix* is set to the *kim query variable* argument and *I* ranges from 1 to the number of returned values. The number and order of the returned values is determined by the type of query performed. The *formatarg* keyword "index" returns a *variable style index* that can be incremented via the *next* command. This enables the construction of simple loops over the returned values by the type of query performed.

**Note**

*kim query* only supports queries that return a single result or an array of values. More complex queries that return a JSON structure are not currently supported. An attempt to use *kim query* in such cases will generate an error.

The second required argument *query_function* is the name of the query function to be called (e.g. *get_lattice_constant_cubic*). All following *arguments* are parameters handed over to the web query in the format *keyword=value*, where *value* is always an array of one or more comma-separated items in brackets. The list of supported keywords and the type and format of their values depend on the query function used. The current list of query functions is available on the OpenKIM webpage at https://openkim.org/doc/usage/kim-query.

**Note**

The *get_available_models* function's output is post-processed by LAMMPS so that only the names of models that are currently installed are stored in the variable. Matching models that are not installed are instead written to the LAMMPS log file.

**Note**

All query functions, except *get_available_models*, require the *model* keyword, which identifies the IM whose predictions are being queried. *kim query* automatically generates the *model* keyword based on the IM set in by *kim init*, and it can be overwritten if specified as an argument to the *kim query*. Where *kim init* is not specified, the *model* keyword must be provided as an argument to the *kim query*.

> **Note**
>
> Each *query_function* is associated with a default method (implemented as a KIM Test) used to compute this property. In cases where there are multiple methods in OpenKIM for computing a property, a *method* keyword can be provided to select the method of choice. See the query documentation to see which methods are available for a given *query_function*.

### *kim query* Usage Examples and Further Clarifications

The data obtained by *kim query* commands can be used as part of the setup or analysis phases of LAMMPS simulations. Some examples are given below.

**Define an equilibrium fcc crystal**

```
kim      init EAM_Dynamo_ErcolessiAdams_1994_Al__MO_123629422045_005 metal
boundary p p p
kim      query a0 get_lattice_constant_cubic crystal=[fcc] species=[Al] units=[angstrom]
lattice  fcc ${a0}
...
```

```
units   metal
kim     query a0 get_lattice_constant_cubic crystal=[fcc] species=[Al] units=[angstrom]␣
↪model=[EAM_Dynamo_ErcolessiAdams_1994_Al__MO_123629422045_005]
lattice fcc ${a0}
...
```

The *kim query* command retrieves from OpenKIM the equilibrium lattice constant predicted by the Ercolessi and Adams (1994) potential for the fcc structure and places it in variable *a0*. This variable is then used on the next line to set up the crystal. By using *kim query*, the user is saved the trouble and possible error of tracking this value down, or of having to perform an energy minimization to find the equilibrium lattice constant.

> **Note**
>
> In *unit_conversion_mode* the results obtained from a *kim query* would need to be converted to the appropriate units system. For example, in the above script, the lattice command would need to be changed to: "lattice fcc $(v_a0*v__u_distance)".

**Define an equilibrium hcp crystal**

```
kim      init EAM_Dynamo_MendelevAckland_2007v3_Zr__MO_004835508849_000 metal
boundary p p p
kim      query latconst split get_lattice_constant_hexagonal crystal=[hcp] species=[Zr]␣
↪units=[angstrom]
lattice  custom ${latconst_1} a1 0.5 -0.866025 0 a2 0.5 0.866025 0 a3 0 0 $(latconst_2/
↪latconst_1) &
         basis 0.333333 0.666666 0.25 basis 0.666666 0.333333 0.75
...
```

In this case the *kim query* returns two arguments (since the hexagonal close packed (hcp) structure has two independent lattice constants). The *formatarg* keyword "split" places the two values into the variables *latconst_1* and *latconst_2*. (These variables are created if they do not already exist.)

**Define a crystal at finite temperature accounting for thermal expansion**

```
kim      init EAM_Dynamo_ErcolessiAdams_1994_Al__MO_123629422045_005 metal
boundary p p p
kim      query a0 get_lattice_constant_cubic crystal=[fcc] species=[Al] units=[angstrom]
kim      query alpha get_linear_thermal_expansion_coefficient_cubic crystal=[fcc]␣
↪species=[Al] units=[1/K] temperature=[293.15] temperature_units=[K]
variable DeltaT equal 300
lattice  fcc $(v_a0*v_alpha*v_DeltaT)
...
```

As in the previous example, the equilibrium lattice constant is obtained for the Ercolessi and Adams (1994) potential. However, in this case the crystal is scaled to the appropriate lattice constant at room temperature (293.15 K) by using the linear thermal expansion constant predicted by the potential.

**Note**

When passing numerical values as arguments (as in the case of the temperature in the above example) it is also possible to pass a tolerance indicating how close to the value is considered a match. If no tolerance is passed a default value is used. If multiple results are returned (indicating that the tolerance is too large), *kim query* will return an error. See the query documentation to see which numerical arguments and tolerances are available for a given *query_function*.

**Compute defect formation energy**

```
kim      init EAM_Dynamo_ErcolessiAdams_1994_Al__MO_123629422045_005 metal
...
... Build fcc crystal containing some defect and compute the total energy
... which is stored in the variable *Etot*
...
kim      query Ec get_cohesive_energy_cubic crystal=[fcc] species=[Al] units=[eV]
variable Eform equal ${Etot} - count(all)*${Ec}
...
```

The defect formation energy *Eform* is computed by subtracting the ideal fcc cohesive energy of the atoms in the system from *Etot*. The ideal fcc cohesive energy of the atoms is obtained from OpenKIM for the Ercolessi and Adams (1994) potential.

**Retrieve equilibrium fcc crystal of all EAM potentials that support a specific species**

```
kim   query model index get_available_models species=[Al] potential_type=[eam]
label model_loop
kim   query latconst get_lattice_constant_cubic crystal=[fcc] species=[Al]␣
↪units=[angstrom] model=[${model}]
print "FCC lattice constant (${model} potential) = ${latconst}"
...
...   do something with current value of latconst
...
next  model
jump  SELF model_loop
```

In this example, the *index* mode of *formatarg* is used. The first *kim query* returns the list of all available EAM potentials that support the *Al* species and archived in OpenKIM. The result of the query operation is stored in the LAMMPS variable *model* as an index *variable*. This variable is used later to access the values one at a time within a loop as shown in the example. The second *kim query* command retrieves from OpenKIM the equilibrium lattice constant

predicted by each potential for the fcc structure and places it in variable *latconst*.

**Note**

*kim query* commands return results archived in OpenKIM. These results are obtained using programs for computing material properties (KIM Tests and KIM Test Drivers) that were contributed to OpenKIM. In order to give credit to Test developers, the number of times results from these programs are queried is tracked. No other information about the nature of the query or its source is recorded.

### 1.44.6 Accessing KIM Model Parameters from LAMMPS (*kim param*)

All IMs are functional forms containing a set of parameters. These parameters' values are typically selected to best reproduce a training set of quantum mechanical calculations or available experimental data. For example, a Lennard-Jones potential intended to model argon might have the values of its two parameters, epsilon, and sigma, fit to the dimer dissociation energy or thermodynamic properties at a critical point of the phase diagram.

Normally a user employing an IM should not modify its parameters since, as noted above, these are selected to reproduce material properties. However, there are cases where accessing and modifying IM parameters is desired, such as for assessing uncertainty, fitting an IM, or working with an ensemble of IMs. As explained *above*, IMs archived in OpenKIM are either Portable Models (PMs) or Simulator Models (SMs). KIM PMs are complete independent implementations of an IM, whereas KIM SMs are wrappers to an IM implemented within LAMMPS. Two different mechanisms are provided for accessing IM parameters in these two cases:

- For a KIM PM, the *kim param* command can be used to *get* and *set* the values of the PM's parameters as explained below.
- For a KIM SM, the user should consult the documentation page for the specific IM and follow instructions there for how to modify its parameters (if possible).

The *kim param get* and *kim param set* commands provide an interface to access and change the parameters of a KIM PM that "publishes" its parameters and makes them publicly available (see the KIM API documentation for details).

**Note**

The *kim param set/get* command must be preceded by a *kim interactions* command (or alternatively by a *pair_style kim* and *pair_coeff* commands). The *kim param set* command may be used wherever a *pair_coeff* command may occur.

#### Syntax

```
kim param get param_name index_range variable formatarg
kim param set param_name index_range values
```

- param_name = name of a KIM portable model parameter (which is published by the PM and available for access). The specific string used to identify a parameter is defined by the PM. For example, for the Stillinger-Weber (SW) potential in OpenKIM, the parameter names are *A, B, p, q, sigma, gamma, cutoff, lambda, costheta0*
- index_range = KIM portable model parameter index range (an integer for a single element, or pair of integers separated by a colon for a range of elements)
- variable(s) = single name or list of names of (string style) LAMMPS variable(s) where a query result or parameter get result is stored. Variables that do not exist will be created by the command
- formatarg = *list* or *split* or *explicit* (optional)

  `list = returns a single string with a list of space separated values`

```
       (e.g. "1.0 2.0 3.0"), which is placed in a LAMMPS variable as
       defined by the variable argument
split = returns the values separately in new variables with names based
       on the prefix specified in variable and a number appended to
       indicate which element in the list of values is in the variable
explicit = returns the values separately in one more more variable names
       provided as arguments that precede formatarg (default)
```

- values = new value(s) to replace the current value(s) of a KIM portable model parameter

> **Note**
>
> The list of all the parameters that a PM exposes for access/mutation are automatically written to the lammps log file when *kim init* is called.

Each published parameter of a KIM PM takes the form of an array of numerical values. The array can contain one element for a single-valued parameter, or a set of values. For example, the multispecies SW potential for the Zn-Cd-Hg-S-Se-Te system has the same parameter names as the single-species SW potential, but each parameter array contains 21 entries that correspond to the parameter values used for each pairwise combination of the model's six supported species (this model does not have parameters specific to individual ternary combinations of its supported species).

The *index_range* argument may either be an integer referring to a specific element within the array associated with the parameter specified by *param_name*, or a pair of integers separated by a colon that refer to a slice of this array. In both cases, one-based indexing is used to refer to the entries of the array.

The result of a *get* operation for a specific *index_range* is stored in one or more *LAMMPS string style variables* as determined by the optional *formatarg* argument *documented above.* If not specified, the default for *formatarg* is "explicit" for the *kim param* command.

For the case where the result is an array with multiple values (i.e. *index_range* contains a range), the optional "split" or "explicit" *formatarg* keywords can be used to separate the results into multiple variables; see the examples below. Multiple parameters can be retrieved with a single call to *kim param get* by repeating the argument list following *get*.

For a *set* operation, the *values* argument contains the new value(s) for the element(s) of the parameter specified by *index_range*. For the case where multiple values are being set, *values* contains a set of values separated by spaces. Multiple parameters can be set with a single call to *kim param set* by repeating the argument list following *set*.

### *kim param* Usage Examples and Further Clarifications

Examples of getting and setting KIM PM parameters with further clarifications are provided below.

**Getting a scalar parameter**

```
kim init SW_StillingerWeber_1985_Si__MO_405512056662_005 metal
...
kim interactions Si
kim param get A 1 VARA
```

or

```
...
pair_style kim SW_StillingerWeber_1985_Si__MO_405512056662_005
pair_coeff * * Si
kim param get A 1 VARA
```

In these cases, the value of the SW *A* parameter is retrieved and placed in the LAMMPS variable *VARA*. The variable *VARA* can be used in the remainder of the input script in the same manner as any other LAMMPS variable.

**Getting multiple scalar parameters with a single call**

```
...
kim interactions Si
kim param get A 1 VARA B 1 VARB
```

In this example, it is shown how to retrieve the *A* and *B* parameters of the SW potential and store them in the LAMMPS variables *VARA* and *VARB*.

**Getting a range of values from a parameter**

There are several options when getting a range of values from a parameter determined by the *formatarg* argument.

```
kim init SW_ZhouWardMartin_2013_CdTeZnSeHgS__MO_503261197030_002 metal
...
kim interactions Te Zn Se
kim param get lambda 7:9 LAM_TeTe LAM_TeZn LAM_TeSe
```

In this case, *formatarg* is not specified and therefore the default "explicit" mode is used. (The behavior would be the same if the word *explicit* were added after *LAM_TeSe*.) Elements 7, 8 and 9 of parameter lambda retrieved by the *get* operation are placed in the LAMMPS variables *LAM_TeTe*, *LAM_TeZn* and *LAM_TeSe*, respectively.

**Note**

In the above example, elements 7-9 of the lambda parameter correspond to Te-Te, Te-Zm and Te-Se interactions. This can be determined by visiting the model page for the specified potential and looking at its parameter file linked to at the bottom of the page (file with .param ending) and consulting the README documentation provided with the driver for the PM being used. A link to the driver is provided at the top of the model page.

```
...
kim      interactions Te Zn Se
kim      param get lambda 15:17 LAMS list
variable LAM_VALUE index ${LAMS}
label    loop_on_lambda
...
...      do something with the current value of lambda
...
next     LAM_VALUE
jump     SELF loop_on_lambda
```

In this case, the "list" mode of *formatarg* is used. The result of the *get* operation is stored in the LAMMPS variable *LAMS* as a string containing the three retrieved values separated by spaces, e.g "1.0 2.0 3.0". This can be used in LAMMPS with an *index* variable to access the values one at a time within a loop as shown in the example. At each iteration of the loop *LAM_VALUE* contains the current value of lambda.

```
...
kim interactions Te Zn Se
kim param get lambda 15:17 LAM split
```

In this case, the "split" mode of *formatarg* is used. The three values retrieved by the *get* operation are stored in the three LAMMPS variables *LAM_15*, *LAM_16* and *LAM_17*. The provided name "LAM" is used as prefix and the location in the lambda array is appended to create the variable names.

**Setting a scalar parameter**

```
kim init SW_StillingerWeber_1985_Si__MO_405512056662_005 metal
...
kim interactions Si
kim param set gamma 1 2.6
```

Here, the SW potential's gamma parameter is set to 2.6. Note that the *get* and *set* commands work together, so that a *get* following a *set* operation will return the new value that was set. For example,

```
...
kim interactions Si
kim param get gamma 1 ORIG_GAMMA
kim param set gamma 1 2.6
kim param get gamma 1 NEW_GAMMA
...
print "original gamma = ${ORIG_GAMMA}, new gamma = ${NEW_GAMMA}"
```

Here, *ORIG_GAMMA* will contain the original gamma value for the SW potential, while *NEW_GAMMA* will contain the value 2.6.

**Setting multiple scalar parameters with a single call**

```
kim      init SW_ZhouWardMartin_2013_CdTeZnSeHgS__MO_503261197030_002 metal
...
kim      interactions Cd Te
variable VARG equal 2.6
variable VARS equal 2.0951
kim      param set gamma 1 ${VARG} sigma 3 ${VARS}
```

In this case, the first element of the *gamma* parameter and third element of the *sigma* parameter are set to 2.6 and 2.0951, respectively. This example also shows how LAMMPS variables can be used when setting parameters.

**Setting a range of values of a parameter**

```
kim init SW_ZhouWardMartin_2013_CdTeZnSeHgS__MO_503261197030_002 metal
...
kim interactions Cd Te Zn Se Hg S
kim param set sigma 2:6 2.35214 2.23869 2.04516 2.43269 1.80415
```

In this case, elements 2 through 6 of the parameter *sigma* are set to the values 2.35214, 2.23869, 2.04516, 2.43269 and 1.80415 in order.

### 1.44.7 Writing material properties in standard KIM Property Instance format (*kim property*)

The OpenKIM system includes a collection of Tests (material property calculation codes), Models (interatomic potentials), Predictions, and Reference Data (DFT or experiments). Specifically, a KIM Test is a computation that when coupled with a KIM Model generates the prediction of that model for a specific material property rigorously defined by a KIM Property Definition (see the KIM Properties Framework for further details). A prediction of a material property for a given model is a specific numerical realization of a property definition, referred to as a "Property Instance." The objective of the *kim property* command is to make it easy to output material properties in a standardized, machine readable, format that can be easily ingested by other programs. Additionally, it aims to make it as easy as possible to convert a LAMMPS script that computes a material property into a KIM Test that can then be uploaded to openkim.org

A developer interested in creating a KIM Test using a LAMMPS script should first determine whether a property definition that applies to their calculation already exists in OpenKIM by searching the properties page. If none exists,

it is possible to use a locally defined property definition contained in a file until it can be uploaded to the official repository (see below). Once one or more applicable property definitions have been identified, the *kim property create*, *kim property modify*, *kim property remove*, and *kim property destroy*, commands provide an interface to create, set, modify, remove, and destroy instances of them within a LAMMPS script.

### Syntax

```
kim property create  instance_id property_id
kim property modify  instance_id key key_name key_name_key key_name_value
kim property remove  instance_id key key_name
kim property destroy instance_id
kim property dump    file
```

- instance_id = a positive integer identifying the KIM property instance; (note that the results file can contain multiple property instances)
- property_id = identifier of a KIM Property Definition, which can be (1) a property short name, (2) the full unique ID of the property (including the contributor and date), (3) a file name corresponding to a local property definition file
- key_name = one of the keys belonging to the specified KIM property definition
- key_name_key = a key belonging to a key-value pair (standardized in the KIM Properties Framework)
- key_name_value = value to be associated with a key_name_key in a key-value pair
- file = name of a file to write the currently defined set of KIM property instances to

Examples of each of the three *property_id* cases are shown below,

```
kim property create 1 atomic-mass
kim property create 2 cohesive-energy-relation-cubic-crystal
```

```
kim property create 1 tag:brunnels@noreply.openkim.org,2016-05-11:property/atomic-mass
kim property create 2 tag:staff@noreply.openkim.org,2014-04-15:property/cohesive-energy-
↪relation-cubic-crystal
```

```
kim property create 1 new-property.edn
kim property create 2 /home/mary/marys-kim-properties/dissociation-energy.edn
```

In the last example, "new-property.edn" and "/home/mary/marys-kim-properties/dissociation-energy.edn" are the names of files that contain user-defined (local) property definitions.

A KIM property instance takes the form of a "map", i.e. a set of key-value pairs akin to Perl's hash, Python's dictionary, or Java's Hashtable. It consists of a set of property key names, each of which is referred to here by the *key_name* argument, that are defined as part of the relevant KIM Property Definition and include only lowercase alphanumeric characters and dashes. The value paired with each property key is itself a map whose possible keys are defined as part of the KIM Properties Framework; these keys are referred to by the *key_name_key* argument and their associated values by the *key_name_value* argument. These values may either be scalars or arrays, as stipulated in the property definition.

**Note**

Each map assigned to a *key_name* must contain the *key_name_key* "source-value" and an associated *key_name_value* of the appropriate type (as defined in the relevant KIM Property Definition). For keys that are defined as having physical units, the "source-unit" *key_name_key* must also be given a string value recognized by GNU units.

Once a *kim property create* command has been given to instantiate a property instance, maps associated with the property's keys can be edited using the *kim property modify* command. In using this command, the special keyword "key" should be given, followed by the property key name and the key-value pair in the map associated with the key that is to be set. For example, the atomic-mass property definition consists of two property keys named "mass" and "species." An instance of this property could be created like so:

```
kim property create 1 atomic-mass
kim property modify 1 key species source-value Al
kim property modify 1 key mass    source-value 26.98154
kim property modify 1 key mass    source-unit amu
```

or, equivalently,

```
kim property create 1 atomic-mass
kim property modify 1 key species source-value Al       &
                      key mass    source-value 26.98154 &
                                  source-unit  amu
```

### *kim property* Usage Examples and Further Clarifications

**Create**

```
kim property create instance_id property_id
```

The *kim property create* command takes as input a property instance ID and the property definition name, and creates an initial empty property instance data structure. For example,

```
kim property create 1 atomic-mass
kim property create 2 cohesive-energy-relation-cubic-crystal
```

creates an empty property instance of the "atomic-mass" property definition with instance ID 1 and an empty instance of the "cohesive-energy-relation-cubic-crystal" property with ID 2. A list of published property definitions in OpenKIM can be found on the properties page.

One can also provide the name of a file in the current working directory or the path of a file containing a valid property definition. For example,

```
kim property create 1 new-property.edn
```

where "new-property.edn" refers to a file name containing a new property definition that does not exist in OpenKIM.

If the *property_id* given cannot be found in OpenKIM and no file of this name containing a valid property definition can be found, this command will produce an error with an appropriate message. Calling *kim property create* with the same instance ID multiple times will also produce an error.

**Modify**

```
kim property modify instance_id key key_name key_name_key key_name_value
```

The *kim property modify* command incrementally builds the property instance by receiving property definition keys along with associated arguments. Each *key_name* is associated with a map containing one or more key-value pairs (in the form of *key_name_key-key_name_value* pairs). For example,

```
kim property modify 1 key species source-value Al
kim property modify 1 key mass    source-value 26.98154
kim property modify 1 key mass    source-unit  amu
```

where the special keyword "key" is followed by a *key_name* ("species" or "mass" in the above) and one or more key-value pairs. These key-value pairs may continue until either another "key" keyword is given or the end of the line is reached. Thus, the above could equivalently be written as

```
kim property modify 1 key species source-value Al       &
                      key mass    source-value 26.98154 &
                      key mass    source-unit  amu
```

As an example of modifying multiple key-value pairs belonging to the map of a single property key, the following command modifies the map of the "cohesive-potential-energy" property key to contain the key "source-unit" which is assigned a value of "eV" and the key "digits" which is assigned a value of 5,

```
kim property modify 2 key cohesive-potential-energy source-unit eV digits 5
```

**Note**

The relevant data types of the values in the map are handled automatically based on the specification of the key in the KIM Property Definition. In the example above, this means that the value "eV" will automatically be interpreted as a string while the value 5 will be interpreted as an integer.

The values contained in maps can either be scalars, as in all of the examples above, or arrays depending on which is stipulated in the corresponding Property Definition. For one-dimensional arrays, a single one-based index must be supplied that indicates which element of the array is to be modified. For multidimensional arrays, multiple indices must be given depending on the dimensionality of the array.

**Note**

All array indexing used by *kim property modify* is one-based, i.e. the indices are enumerated 1, 2, 3, ...

**Note**

The dimensionality of arrays are defined in the the corresponding Property Definition. The extent of each dimension of an array can either be a specific finite number or indefinite and determined at run time. If an array has a fixed extent, attempting to modify an out-of-range index will fail with an error message.

For example, the "species" property key of the cohesive-energy-relation-cubic-crystal property is a one-dimensional array that can contain any number of entries based on the number of atoms in the unit cell of a given cubic crystal. To assign an array containing the string "Al" four times to the "source-value" key of the "species" property key, we can do so by issuing:

```
kim property modify 2 key species source-value 1 Al
kim property modify 2 key species source-value 2 Al
kim property modify 2 key species source-value 3 Al
kim property modify 2 key species source-value 4 Al
```

**Note**

No declaration of the number of elements in this array was given; *kim property modify* will automatically handle memory management to allow an arbitrary number of elements to be added to the array.

**Note**

In the event that *kim property modify* is used to set the value of an array index without having set the values of all lesser indices, they will be assigned default values based on the data type associated with the key in the map:

| Data type | Default value |
|---|---|
| int | 0 |
| float | 0.0 |
| string | "" |
| file | "" |

For example, doing the following:

```
kim property create 2 cohesive-energy-relation-cubic-crystal
kim property modify 2 key species source-value 4 Al
```

will result in the "source-value" key in the map for the property key "species" being assigned the array ["", "", "", "Al"].

For convenience, the index argument provided may refer to an inclusive range of indices by specifying two integers separated by a colon (the first integer must be less than or equal to the second integer, and no whitespace should be included). Thus, the snippet above could equivalently be written:

```
kim property modify 2 key species source-value 1:4 Al Al Al Al
```

Calling this command with a non-positive index, e.g. `kim property modify 2 key species source-value 0 Al`, or an incorrect number of input arguments, e.g. `kim property modify 2 key species source-value 1:4 Al Al`, will result in an error.

As an example of modifying multidimensional arrays, consider the "basis-atoms" key in the cohesive-energy-relation-cubic-crystal property definition. This is a two-dimensional array containing the fractional coordinates of atoms in the unit cell of the cubic crystal. In the case of, e.g. a conventional fcc unit cell, the "source-value" key in the map associated with this key should be assigned the following value:

```
[[0.0, 0.0, 0.0],
 [0.5, 0.5, 0.0],
 [0.5, 0.0, 0.5],
 [0.0, 0.5, 0.5]]
```

While each of the twelve components could be set individually, we can instead set each row at a time using colon notation:

```
kim property modify 2 key basis-atom-coordinates source-value 1 1:3 0.0 0.0 0.0
kim property modify 2 key basis-atom-coordinates source-value 2 1:3 0.5 0.5 0.0
kim property modify 2 key basis-atom-coordinates source-value 3 1:3 0.5 0.0 0.5
kim property modify 2 key basis-atom-coordinates source-value 4 1:3 0.0 0.5 0.5
```

Where the first index given refers to a row and the second index refers to a column. We could, instead, choose to set each column at a time like so:

```
kim property modify 2 key basis-atom-coordinates source-value 1:4 1 0.0 0.5 0.5 0.0 &
                      key basis-atom-coordinates source-value 1:4 2 0.0 0.5 0.0 0.5 &
                      key basis-atom-coordinates source-value 1:4 3 0.0 0.0 0.5 0.5
```

**Note**

Multiple calls of *kim property modify* made for the same instance ID can be combined into a single invocation, meaning the following are both valid:

```
kim property modify 2 key basis-atom-coordinates source-value 1 1:3 0.0 0.0 0.0 &
                      key basis-atom-coordinates source-value 2 1:3 0.5 0.5 0.0 &
                      key basis-atom-coordinates source-value 3 1:3 0.5 0.0 0.5 &
                      key basis-atom-coordinates source-value 4 1:3 0.0 0.5 0.5
```

```
kim property modify 2 key short-name source-value 1 fcc                          &
                      key species source-value 1:4 Al Al Al Al                   &
                      key a source-value 1:5 3.9149 4.0000 4.032 4.0817 4.1602   &
                            source-unit angstrom                                 &
                            digits 5                                             &
                      key basis-atom-coordinates source-value 1 1:3 0.0 0.0 0.0 &
                      key basis-atom-coordinates source-value 2 1:3 0.5 0.5 0.0 &
                      key basis-atom-coordinates source-value 3 1:3 0.5 0.0 0.5 &
                      key basis-atom-coordinates source-value 4 1:3 0.0 0.5 0.5
```

**Note**

For multidimensional arrays, only one colon-separated range is allowed in the index listing. Therefore,

```
kim property modify 2 key basis-atom-coordinates 1 1:3 0.0 0.0 0.0
```

is valid but

```
kim property modify 2 key basis-atom-coordinates 1:2 1:3 0.0 0.0 0.0 0.0 0.0 0.0
```

is not.

**Note**

After one sets a value in a map with the *kim property modify* command, additional calls will overwrite the previous value.

**Remove**

```
kim property remove instance_id key key_name
```

The *kim property remove* command can be used to remove a property key from a property instance. For example,

```
kim property remove 2 key basis-atom-coordinates
```

**Destroy**

```
kim property destroy instance_id
```

The *kim property destroy* command deletes a previously created property instance ID. For example,

```
kim property destroy 2
```

> **Note**
>
> If this command is called with an instance ID that does not exist, no error is raised.

**Dump**

The *kim property dump* command can be used to write the content of all currently defined property instances to a file:

```
kim property dump file
```

For example,

```
kim property dump results.edn
```

> **Note**
>
> Issuing the *kim property dump* command clears all existing property instances from memory.

### 1.44.8 Citation of OpenKIM IMs

When publishing results obtained using OpenKIM IMs researchers are requested to cite the OpenKIM project *(Tadmor)*, KIM API *(Elliott)*, and the specific IM codes used in the simulations, in addition to the relevant scientific references for the IM. The citation format for an IM is displayed on its page on OpenKIM along with the corresponding BibTex file, and is automatically added to the LAMMPS citation reminder.

Citing the IM software (KIM infrastructure and specific PM or SM codes) used in the simulation gives credit to the researchers who developed them and enables open source efforts like OpenKIM to function.

### 1.44.9 Restrictions

The *kim* command is part of the KIM package. It is only enabled if LAMMPS is built with that package. A requirement for the KIM package, is the KIM API library that must be downloaded from the OpenKIM website and installed before LAMMPS is compiled. When installing LAMMPS from binary, the kim-api package is a dependency that is automatically downloaded and installed. The *kim query* command requires the *libcurl* library to be installed. The *kim property* command requires *Python* 3.6 or later and the *kim-property* python package to be installed. See the KIM section of the *Packages details* for details.

Furthermore, when using *kim* command to run KIM SMs, any packages required by the native potential being used or other commands or fixes that it invokes must be installed.

### 1.44.10 Related commands

*pair_style kim*

---

**(Tadmor)** Tadmor, Elliott, Sethna, Miller and Becker, JOM, 63, 17 (2011). doi: https://doi.org/10.1007/s11837-011-0102-6

**(Elliott)** Elliott, Tadmor and Bernstein, https://openkim.org/kim-api/ (2011) doi: https://doi.org/10.25950/FF8F563A

# 1.45 kspace_modify command

## 1.45.1 Syntax

```
kspace_modify keyword value ...
```

- one or more keyword/value pairs may be listed
- keyword = *collective* or *collective/self/copy* or *nonblocking* or *compute* or *cutoff/adjust* or *diff* or *disp/auto* or *fftbench* or *force/disp/kspace* or *force/disp/real* or *force* or *gewald/disp* or *gewald* or *kmax/ewald* or *mesh* or *minorder* or *mix/disp* or *order/disp* or *order* or *overlap* or *scafacos* or *slab* or *splittol* or *wire*

```
collective value = yes or no
collective/self/copy value = yes or no or onerank
nonblocking value = yes or no
compute value = yes or no
cutoff/adjust value = yes or no
diff value = ad or ik = 2 or 4 FFTs for PPPM in smoothed or non-smoothed mode
disp/auto value = yes or no
fftbench value = yes or no
force/disp/real value = accuracy (force units)
force/disp/kspace value = accuracy (force units)
force value = accuracy (force units)
gewald value = rinv (1/distance units)
  rinv = G-ewald parameter for Coulombics
gewald/disp value = rinv (1/distance units)
  rinv = G-ewald parameter for dispersion
kmax/ewald value = kx ky kz
  kx,ky,kz = number of Ewald sum kspace vectors in each dimension
mesh value = x y z
  x,y,z = grid size in each dimension for long-range Coulombics
mesh/disp value = x y z
  x,y,z = grid size in each dimension for 1/r^6 dispersion
minorder value = M
  M = min allowed extent of Gaussian when auto-adjusting to minimize grid␣
↪communication
mix/disp value = pair or geom or none
order value = N
  N = extent of Gaussian for PPPM or MSM mapping of charge to grid
order/disp value = N
  N = extent of Gaussian for PPPM mapping of dispersion term to grid
overlap = yes or no = whether the grid stencil for PPPM is allowed to overlap into␣
↪more than the nearest-neighbor processor
pressure/scalar value = yes or no
scafacos values = option value1 value2 ...
  option = tolerance
    value = energy or energy_rel or field or field_rel or potential or potential_rel
  option = fmm_tuning
    value = 0 or 1
slab value = volfactor or auto or nozforce
  volfactor = ratio of the total extended volume used in the
    2d approximation compared with the volume of the simulation domain
  auto chooses the extended z dimension from the current
    normalized force tolerance, lateral box dimensions, and G-ewald
  nozforce turns off kspace forces in the z direction
```

```
splittol value = tol
  tol = relative size of two eigenvalues (see discussion below)
wire value = volfactor (available with ELECTRODE package)
  volfactor = ratio of the total extended dimension used in the 1d
    approximation compared with the dimension of the simulation domain
```

### 1.45.2 Examples

```
kspace_modify mesh 24 24 30 order 6
kspace_modify slab 3.0
kspace_modify slab auto
kspace_modify scafacos tolerance energy
```

### 1.45.3 Description

Set parameters used by the kspace solvers defined by the *kspace_style* command. Not all parameters are relevant to all kspace styles.

---

The *collective* keyword applies only to PPPM. It is set to *no* by default, If this option is set to *yes*, LAMMPS will use MPI collective operations to remap data for 3d-FFT operations instead of the default point-to-point communication. This is faster on machines that have an efficient implementation of MPI collective operations *and* adequate hardware.

---

Added in version 4Jul2026.

The *collective/self/copy* keyword applies only to PPPM and only when *collective* is set to *yes*. It controls whether data remapped to the same MPI rank (the "self" contribution) is handled via a direct pack/unpack rather than being included in the `MPI_Alltoallv` collective. If set to *yes*, LAMMPS always handles the self contribution with a direct copy. If set to *no*, the self contribution is included in the `MPI_Alltoallv` along with data destined for other ranks. If set to *onerank*, the direct copy is used only when running on a single MPI rank, which avoids unnecessary collective overhead when all data maps to the same rank.

---

Added in version 10Dec2025.

The *nonblocking* keyword applies only to PPPM. It is set to *no* by default. If this option is set to *yes*, LAMMPS will use non-blocking point-to-point MPI operations to remap data for 3d-FFT operations instead of the default blocking point-to-point communication. This allows for better utilization of full network bandwidth by overlapping communication to multiple other ranks at the same time, as well as overlapping receiving/unpacking data and sending data.

---

The *compute* keyword allows Kspace computations to be turned off, even though a *kspace_style* is defined. This is not useful for running a real simulation, but can be useful for debugging purposes or for computing only partial forces that do not include the Kspace contribution. You can also do this by simply not defining a *kspace_style*, but a Kspace-compatible *pair_style* requires a kspace style to be defined. This keyword gives you that option.

---

The *cutoff/adjust* keyword applies only to MSM. If this option is turned on, the Coulombic cutoff will be automatically adjusted at the beginning of the run to give the desired estimated error. Other cutoffs such as LJ will not be affected. If the grid is not set using the *mesh* command, this command will also attempt to use the optimal grid that minimizes cost using an estimate given by *(Hardy)*. Note that this cost estimate is not exact, somewhat experimental, and still may not yield the optimal parameters.

---

The *diff* keyword specifies the differentiation scheme used by the PPPM method to compute forces on particles given electrostatic potentials on the PPPM mesh. The *ik* approach is the default for PPPM and is the original formulation used in *(Hockney)*. It performs differentiation in Kspace, and uses 3 FFTs to transfer each component of the computed fields back to real space for total of 4 FFTs per timestep.

The analytic differentiation *ad* approach uses only 1 FFT to transfer information back to real space for a total of 2 FFTs per timestep. It then performs analytic differentiation on the single quantity to generate the 3 components of the electric field at each grid point. This is sometimes referred to as "smoothed" PPPM. This approach requires a somewhat larger PPPM mesh to achieve the same accuracy as the *ik* method. Currently, only the *ik* method (default) can be used for a triclinic simulation cell with PPPM. The *ad* method is always used for MSM.

**Note**

Currently, not all PPPM styles support the *ad* option. Support for those PPPM variants will be added later.

---

The *disp/auto* option controls whether the pppm/disp is allowed to generate PPPM parameters automatically. If set to *no*, parameters have to be specified using the *gewald/disp*, *mesh/disp*, *force/disp/real* or *force/disp/kspace* keywords, or the code will stop with an error message. When this option is set to *yes*, the error message will not appear and the simulation will start. For a typical application, using the automatic parameter generation will provide simulations that are either inaccurate or slow. Using this option is thus not recommended. For guidelines on how to obtain good parameters, see the *long-range dispersion howto* discussion.

---

The *fftbench* keyword applies only to PPPM. It is off by default. If this option is turned on, LAMMPS will perform a short FFT benchmark computation and report its timings, and will thus finish some seconds later than it would if this option were off.

---

The *force/disp/real* and *force/disp/kspace* keywords set the force accuracy for the real and reciprocal space computations for the dispersion part of pppm/disp. As shown in *(Isele-Holder)*, optimal performance and accuracy in the results is obtained when these values are different.

---

The *force* keyword overrides the relative accuracy parameter set by the *kspace_style* command with an absolute accuracy. The accuracy determines the RMS error in per-atom forces calculated by the long-range solver and is thus specified in force units. A negative value for the accuracy setting means to use the relative accuracy parameter. The accuracy setting is used in conjunction with the pairwise cutoff to determine the number of K-space vectors for style *ewald*, the FFT grid size for style *pppm*, or the real space grid size for style *msm*.

---

The *gewald* keyword sets the value of the Ewald or PPPM G-ewald parameter for charge as *rinv* in reciprocal distance units. Without this setting, LAMMPS chooses the parameter automatically as a function of cutoff, precision, grid spacing, etc. This means it can vary from one simulation to the next which may not be desirable for matching a KSpace solver to a pre-tabulated pairwise potential. This setting can also be useful if Ewald or PPPM fails to choose a good grid spacing and G-ewald parameter automatically. If the value is set to 0.0, LAMMPS will choose the G-ewald parameter automatically. MSM does not use the *gewald* parameter.

---

The *gewald/disp* keyword sets the value of the Ewald or PPPM G-ewald parameter for dispersion as *rinv* in reciprocal distance units. It has the same meaning as the *gewald* setting for Coulombics.

---

The *kmax/ewald* keyword sets the number of kspace vectors in each dimension for kspace style *ewald*. The three values must be positive integers, or else (0,0,0), which unsets the option. When this option is not set, the Ewald sum scheme chooses its own kspace vectors, consistent with the user-specified accuracy and pairwise cutoff. In any case, if kspace style *ewald* is invoked, the values used are printed to the screen and the log file at the start of the run.

---

The *mesh* keyword sets the grid size for kspace style *pppm* or *msm*. In the case of PPPM, this is the FFT mesh, and each dimension must be factorizable into powers of 2, 3, and 5. In the case of MSM, this is the finest scale real-space mesh, and each dimension must be factorizable into powers of 2. When this option is not set, the PPPM or MSM solver chooses its own grid size, consistent with the user-specified accuracy and pairwise cutoff. Values for x,y,z of 0,0,0 unset the option.

---

The *mesh/disp* keyword sets the grid size for kspace style *pppm/disp*. This is the FFT mesh for long-range dispersion and ach dimension must be factorizable into powers of 2, 3, and 5. When this option is not set, the PPPM solver chooses its own grid size, consistent with the user-specified accuracy and pairwise cutoff. Values for x,y,z of 0,0,0 unset the option.

---

The *minorder* keyword allows LAMMPS to reduce the *order* setting if necessary to keep the communication of ghost grid point limited to exchanges between nearest-neighbor processors. See the discussion of the *overlap* keyword for details. If the *overlap* keyword is set to *yes*, which is the default, this is never needed. If it set to *no* and overlap occurs, then LAMMPS will reduce the order setting, one step at a time, until the ghost grid overlap only extends to nearest neighbor processors. The *minorder* keyword limits how small the *order* setting can become. The minimum allowed value for PPPM is 2, which is the default. If *minorder* is set to the same value as *order* then no reduction is allowed, and LAMMPS will generate an error if the grid communication is non-nearest-neighbor and *overlap* is set to *no*. The *minorder* keyword is not currently supported in MSM.

---

The *mix/disp* keyword selects the mixing rule for the dispersion coefficients. With *pair*, the dispersion coefficients of unlike types are computed as indicated with *pair_modify*. With *geom*, geometric mixing is enforced on the dispersion coefficients in the kspace coefficients. When using the arithmetic mixing rule, this will speed-up the simulations but introduces some error in the force computations, as shown in *(Wennberg)*. With *none*, it is assumed that no mixing rule is applicable. Splitting of the dispersion coefficients will be performed as described in *(Isele-Holder)*. The *none* splitting is only available with the PPPM dispersion solver; with *ewald/disp* the *none* value is rejected, while *pair* (the default) and *geom* are honored.

This splitting can be influenced with the *splittol* keywords. Only the eigenvalues that are larger than tol compared to the largest eigenvalues are included. Using this keywords the original matrix of dispersion coefficients is approximated. This leads to faster computations, but the accuracy in the reciprocal space computations of the dispersion part is decreased.

---

The *order* keyword determines how many grid spacings an atom's charge extends when it is mapped to the grid in kspace style *pppm* or *msm*. The default for this parameter is 5 for PPPM and 8 for MSM, which means each charge spans 5 or 8 grid cells in each dimension, respectively. For the LAMMPS implementation of MSM, the order can range from 4 to 10 and must be even. For PPPM, the minimum allowed setting is 2 and the maximum allowed setting is 7. The larger the value of this parameter, the smaller that LAMMPS will set the grid size, to achieve the requested accuracy. Conversely, the smaller the order value, the larger the grid size will be. Note that there is an inherent trade-off involved: a small grid will lower the cost of FFTs or MSM direct sum, but a larger order parameter will increase the cost of interpolating charge/fields to/from the grid.

The PPPM order parameter may be reset by LAMMPS when it sets up the FFT grid if the implied grid stencil extends beyond the grid cells owned by neighboring processors. Typically this will only occur when small problems are run on large numbers of processors. A warning will be generated indicating the order parameter is being reduced to allow LAMMPS to run the problem. Automatic adjustment of the order parameter is not supported in MSM.

---

The *order/disp* keyword determines how many grid spacings an atom's dispersion term extends when it is mapped to the grid in kspace style *pppm/disp*. It has the same meaning as the *order* setting for Coulombics.

---

The *overlap* keyword can be used in conjunction with the *minorder* keyword with the PPPM styles to adjust the amount of communication that occurs when values on the FFT grid are exchanged between processors. This communication is distinct from the communication inherent in the parallel FFTs themselves, and is required because processors interpolate charge and field values using grid point values owned by neighboring processors (i.e. ghost point communication). If the *overlap* keyword is set to *yes* then this communication is allowed to extend beyond nearest-neighbor processors, e.g. when using lots of processors on a small problem. If it is set to *no* then the communication will be limited to nearest-neighbor processors and the *order* setting will be reduced if necessary, as explained by the *minorder* keyword discussion. The *overlap* keyword is always set to *yes* in MSM.

---

The *pressure/scalar* keyword applies only to MSM. If this option is turned on, only the scalar pressure (i.e. (Pxx + Pyy + Pzz)/3.0) will be computed, which can be used, for example, to run an isotropic barostat. Computing the full pressure tensor with MSM is expensive, and this option provides a faster alternative. The scalar pressure is computed using a relationship between the Coulombic energy and pressure *(Hummer)* instead of using the virial equation. This option cannot be used to access individual components of the pressure tensor, to compute per-atom virial, or with suffix kspace/pair styles of MSM, like OMP or GPU.

---

The *scafacos* keyword is used for settings that are passed to the ScaFaCoS library when using *kspace_style scafacos*.

The *tolerance* option affects how the *accuracy* specified with the *kspace_style* command is interpreted by ScaFaCoS. The following values may be used:

- energy = absolute accuracy in total Coulombic energy
- energy_rel = relative accuracy in total Coulombic energy
- potential = absolute accuracy in total Coulombic potential
- potential_rel = relative accuracy in total Coulombic potential
- field = absolute accuracy in electric field
- field_rel = relative accuracy in electric field

The values with suffix _rel indicate the tolerance is a relative tolerance; the other values impose an absolute tolerance on the given quantity. Absolute tolerance in this case means, that for a given quantity q and a given absolute tolerance of t_a the result should be between q-t_a and q+t_a. For a relative tolerance t_r the relative error should not be greater than t_r, i.e. abs(1 - (result/q)) < t_r. As a consequence of this, the tolerance type should be checked, when performing computations with a high absolute field / energy. E.g. if the total energy in the system is 1000000.0 an absolute tolerance of 1e-3 would mean that the result has to be between 999999.999 and 1000000.001, which would be equivalent to a relative tolerance of 1e-9.

The energy and energy_rel values, set a tolerance based on the total Coulombic energy of the system. The potential and potential_rel set a tolerance based on the per-atom Coulombic energy. The field and field_rel tolerance types set a tolerance based on the electric field values computed by ScaFaCoS. Since per-atom forces are derived from the per-atom electric field, this effectively sets a tolerance on the forces, similar to other LAMMPS KSpace styles, as explained on the *kspace_style* doc page.

Note that not all ScaFaCoS solvers support all tolerance types. These are the allowed values for each method:

- fmm = energy and energy_rel
- p2nfft = field (1d-,2d-,3d-periodic systems) or potential (0d-periodic)
- p3m = field
- ewald = field
- direct = has no tolerance tuning

If the tolerance type is not changed, the default values for the tolerance type are the first values in the above list, e.g. energy is the default tolerance type for the fmm solver.

The *fmm_tuning* option is only relevant when using the FMM method. It activates (value=1) or deactivates (value=0) an internal tuning mechanism for the FMM solver. The tuning operation runs sequentially and can be very time-consuming. Usually it is not needed for systems with a homogeneous charge distribution. The default for this option is therefore *0*. The FMM internal tuning is performed once, when the solver is set up.

---

The *slab* keyword allows an Ewald or PPPM solver to be used for a systems that are periodic in x,y but non-periodic in z - a *boundary* setting of "boundary p p f". This is done by treating the system as if it were periodic in z, but inserting empty volume between atom slabs and removing dipole inter-slab interactions so that slab-slab interactions are effectively turned off. The volfactor value sets the ratio of the extended dimension in z divided by the actual dimension in z. It must be a value >= 1.0. A value of 1.0 (the default) means the slab approximation is not used.

The recommended value for volfactor is 3.0. A larger value is inefficient; a smaller value introduces unwanted slab-slab interactions. The use of fixed boundaries in z means that the user must prevent particle migration beyond the initial z-bounds, typically by providing a wall-style fix. The methodology behind the *slab* option is explained in the paper by *(Yeh)*. The *slab* option is also extended to non-neutral systems *(Ballenegger)*.

Added in version 4Jul2026.

As an alternative to specifying a fixed volfactor, the keyword *auto* can be used to determine the extended z dimension from the current normalized force tolerance, the lateral dimensions of the simulation cell, and the active Coulombic gewald parameter. This is useful for quasi-2D systems where the amount of extra vacuum required for the slab correction depends on both the requested tolerance and the box geometry. Here the tolerance entering the formula is made dimensionless by dividing the current absolute force accuracy by the force between two unit charges separated by 1 Angstrom in the active unit style. The automatic choice is not artificially clamped above 1.0, so it can in principle approach the original box size if the formula permits. The quasi-2D error estimate used for the automatic choice is discussed in *(Gao2025)* and *(Gan2025)*. For related error estimates and parameter-selection guidance in the more general dielectric-confined setting, see *(Gao2025)*. The *auto* setting is currently supported by *kspace_style ewald*, *pppm*, *pppm/cg*, *pppm/tip4p*, *pppm/stagger*, and the corresponding OpenMP/GPU/Intel variants that reuse the same slab-correction setup.

An alternative slab option can be invoked with the *nozforce* keyword in lieu of the volfactor. This turns off all kspace forces in the z direction. The *nozforce* option is not supported by MSM. For MSM, any combination of periodic, non-periodic, or shrink-wrapped boundaries can be set using *boundary* (the slab approximation is not needed). The *slab* keyword with a *volfactor* value (the Yeh-Berkowitz EW3DC correction) supports triclinic (non-orthogonal) simulation cells for *kspace_style ewald*, *ewald/disp*, *pppm*, *pppm/cg*, and *pppm/kk* (and the OpenMP variants that reuse the same slab-correction setup). Because the slab normal must be the Cartesian z axis, only an *xy* tilt is allowed: a triclinic slab box with a non-zero *xz* or *yz* tilt is rejected with an error. The z box length must be held fixed for the duration of the run. Triclinic cells are not yet supported with the *slab* keyword for the *pppm/tip4p*, *pppm/stagger*, *esp*, *pppm/gpu*, or *pppm/intel* styles, nor for the *slab nozforce* and *slab ew2d* options.

The slab correction has also been extended to point dipole interactions *(Klapp)* in *kspace_style ewald/disp*, *ewald/dipole*, and *pppm/dipole*. This dipole slab correction is currently limited to orthogonal (non-triclinic) simulation cells.

Changed in version 4Jul2026: The *slab* correction with a *volfactor* value now supports triclinic simulation cells for the *ewald*, *ewald/disp*, *pppm*, *pppm/cg*, and *pppm/kk* styles.

**Note**

If you wish to apply an electric field in the Z-direction, in conjunction with the *slab* keyword, you can do it either by adding explicit oppositely charged particles to the +/- Z surfaces, or by using the *fix efield* command.

---

The *force/disp/real* and *force/disp/kspace* keywords set the force accuracy for the real and reciprocal space computations for the dispersion part of pppm/disp. As shown in *(Isele-Holder)*, optimal performance and accuracy in the results is obtained when these values are different.

The *disp/auto* option controls whether the pppm/disp is allowed to generate PPPM parameters automatically. If set to *no*, parameters have to be specified using the *gewald/disp*, *mesh/disp*, *force/disp/real* or *force/disp/kspace* keywords, or the code will stop with an error message. When this option is set to *yes*, the error message will not appear and the simulation will start. For a typical application, using the automatic parameter generation will provide simulations that are either inaccurate or slow. Using this option is thus not recommended. For guidelines on how to obtain good parameters, see the *Howto dispersion* doc page.

---

### 1.45.4 Restrictions

The *collective* and *nonblocking* keywords cannot both be enabled at the same time. Whichever of the two keywords is enabled last will disable the other.

The *collective/self/copy* keyword has no effect unless *collective* is set to *yes*.

### 1.45.5 Related commands

*kspace_style*, *boundary*

### 1.45.6 Default

The option defaults are as follows:

- collective = no
- collective/self/copy = onerank
- nonblocking = no
- compute = yes
- cutoff/adjust = yes (MSM)
- diff = ik (PPPM)
- disp/auto = no
- fftbench = no (PPPM)
- force = -1.0
- force/disp/kspace = -1.0
- force/disp/real = -1.0
- gewald = gewald/disp = 0.0
- mesh = mesh/disp = 0 0 0
- minorder = 2

- mix/disp = pair
- order = 10 (MSM)
- order = order/disp = 5 (PPPM)
- order = order/disp = 7 (PPPM/intel)
- overlap = yes
- pressure/scalar = yes (MSM)
- slab = 1.0
- split = 0
- tol = 1.0e-6

For scafacos settings, the scafacos tolerance option depends on the method chosen, as documented above. The scafacos fmm_tuning default = 0.

---

**(Hockney)** Hockney and Eastwood, Computer Simulation Using Particles, Adam Hilger, NY (1989).

**(Yeh)** Yeh and Berkowitz, J Chem Phys, 111, 3155 (1999).

**(Ballenegger)** Ballenegger, Arnold, Cerda, J Chem Phys, 131, 094107 (2009).

**(Klapp)** Klapp, Schoen, J Chem Phys, 117, 8050 (2002).

**(Gao2025)** Gao, Zhou, Gan, Liang, J Chem Theory Comput, 21, 5890-5904 (2025). https://doi.org/10.1021/acs.jctc.5c00438

**(Gan2025)** Gan, Gao, Liang, Xu, SIAM J Sci Comput, 47, B846-B874 (2025). https://doi.org/10.1137/24M1655809

**(Hardy)** David Hardy thesis: Multilevel Summation for the Fast Evaluation of Forces for the Simulation of Biomolecules, University of Illinois at Urbana-Champaign, (2006).

**(Hummer)** Hummer, Gronbech-Jensen, Neumann, J Chem Phys, 109, 2791 (1998)

**(Isele-Holder)** Isele-Holder, Mitchell, Hammond, Kohlmeyer, Ismail, J Chem Theory Comput, 9, 5412 (2013).

**(Wennberg)** Wennberg, Murtola, Hess, Lindahl, J Chem Theory Comput, 9, 3527 (2013).

## 1.46 kspace_style command

### 1.46.1 Syntax

```
kspace_style style value
```

- style = *none* or *ewald* or ewald/gpu or *ewald/dipole* or *ewald/dipole/spin* or *ewald/disp* or *ewald/disp/dipole* or *ewald/omp* or *ewald/electrode* or *pppm* or *pppm/cg* or *pppm/disp* or *pppm/tip4p* or *pppm/stagger* or *pppm/disp/tip4p* or *pppm/gpu* or *pppm/intel* or *pppm/disp/intel* or *pppm/kk* or *pppm/omp* or *pppm/cg/omp* or *pppm/disp/tip4p/omp* or *pppm/tip4p/omp* or *pppm/dielectic* or *pppm/disp/dielectric* or *pppm/electrode* or *pppm/electrode/intel* or *pppm/rk* or *msm* or *msm/cg* or *msm/omp* or *msm/cg/omp* or *msm/dielectric* or *scafacos* or *zero*

```
none value = none
ewald value = accuracy
  accuracy = desired relative error in forces
ewald/gpu value = accuracy
  accuracy = desired relative error in forces
ewald/dipole value = accuracy
```

```
  accuracy = desired relative error in forces
ewald/dipole/spin value = accuracy
  accuracy = desired relative error in forces
ewald/disp value = accuracy
  accuracy = desired relative error in forces
ewald/disp/dipole value = accuracy
  accuracy = desired relative error in forces
ewald/omp value = accuracy
  accuracy = desired relative error in forces
ewald/electrode value = accuracy
  accuracy = desired relative error in forces
esp value1 value2 = splitting_accuracy spreading_accuracy
   splitting_accuracy = desired relative error in forces from kernel splitting part
   spreading_accuracy = desired relative error in forces from charge spreading part␣
↪(optional)
pppm value = accuracy
  accuracy = desired relative error in forces
pppm/cg values = accuracy (smallq)
  accuracy = desired relative error in forces
  smallq = cutoff for charges to be considered (optional) (charge units)
pppm/dipole value = accuracy
  accuracy = desired relative error in forces
pppm/dipole/spin value = accuracy
  accuracy = desired relative error in forces
pppm/disp value = accuracy
  accuracy = desired relative error in forces
pppm/tip4p value = accuracy
  accuracy = desired relative error in forces
pppm/disp/tip4p value = accuracy
  accuracy = desired relative error in forces
pppm/gpu value = accuracy
  accuracy = desired relative error in forces
pppm/intel value = accuracy
  accuracy = desired relative error in forces
pppm/disp/intel value = accuracy
  accuracy = desired relative error in forces
pppm/kk value = accuracy
  accuracy = desired relative error in forces
pppm/omp value = accuracy
  accuracy = desired relative error in forces
pppm/cg/omp values = accuracy (smallq)
  accuracy = desired relative error in forces
  smallq = cutoff for charges to be considered (optional) (charge units)
pppm/disp/omp value = accuracy
  accuracy = desired relative error in forces
pppm/tip4p/omp value = accuracy
  accuracy = desired relative error in forces
pppm/disp/tip4p/omp value = accuracy
  accuracy = desired relative error in forces
pppm/stagger value = accuracy
  accuracy = desired relative error in forces
pppm/dielectric value = accuracy
  accuracy = desired relative error in forces
pppm/disp/dielectric value = accuracy
```

```
    accuracy = desired relative error in forces
pppm/electrode value = accuracy
    accuracy = desired relative error in forces
pppm/electrode/intel value = accuracy
    accuracy = desired relative error in forces
pppm/rk value = accuracy
    accuracy = desired relative error in forces
msm value = accuracy
    accuracy = desired relative error in forces
msm/cg value = accuracy (smallq)
    accuracy = desired relative error in forces
    smallq = cutoff for charges to be considered (optional) (charge units)
msm/omp value = accuracy
    accuracy = desired relative error in forces
msm/cg/omp value = accuracy (smallq)
    accuracy = desired relative error in forces
    smallq = cutoff for charges to be considered (optional) (charge units)
msm/dielectric value = accuracy
    accuracy = desired relative error in forces
scafacos values = method accuracy
    method = fmm or p2nfft or p3m or ewald or direct
    accuracy = desired relative error in forces
zero value = accuracy
    accuracy = desired relative error in forces
```

### 1.46.2 Examples

```
kspace_style esp 1.0e-4
kspace_style pppm 1.0e-4
kspace_style pppm/cg 1.0e-5 1.0e-6
kspace_style msm 1.0e-4
kspace_style scafacos fmm 1.0e-4
kspace_style none
kspace_style zero 1.0e-6
```

Used in input scripts:

```
examples/peptide/in.peptide
```

### 1.46.3 Description

Define a long-range solver for LAMMPS to use each timestep to compute long-range Coulombic interactions or long-range $1/r^6$ interactions. Most of the long-range solvers perform their computation in K-space, hence the name of this command.

When such a solver is used in conjunction with an appropriate pair style, the cutoff for Coulombic or $1/r^N$ interactions is effectively infinite. If the Coulombic case, this means each charge in the system interacts with charges in an infinite array of periodic images of the simulation domain.

Note that using a long-range solver requires use of a matching *pair style* to perform consistent short-range pairwise calculations. This means that the name of the pair style contains a matching keyword to the name of the KSpace style, as in this table:

| Pair style | KSpace style |
|---|---|
| coul/long | ewald or pppm |
| coul/esp | esp |
| coul/msm | msm |
| lj/long or buck/long | disp (for dispersion) |
| tip4p/long | tip4p |
| dipole/long | dipole |

---

The *ewald* style performs a standard Ewald summation as described in any solid-state physics text.

Added in version 4Jul2026.

The *ewald/gpu* style is a GPU-accelerated version of the *ewald* style. Unlike *pppm/gpu*, it requires no FFTs and therefore avoids the bandwidth-intensive grid communication of the PPPM method, which makes it competitive with *pppm/gpu* for smaller system sizes despite the less favorable scaling of the Ewald method with system size. The reciprocal-space structure factors, per-atom forces, and per-atom energy and virial are computed on the device; only the small structure-factor vector is reduced across MPI ranks on the host. Non-orthogonal (triclinic) simulation boxes are not supported. For best accuracy the *mixed* or *double* precision builds of the GPU package are recommended; the *single* precision build is not recommended for the Ewald summation because of round-off in the structure-factor sums.

The *ewald/disp* style adds a long-range dispersion sum option for $1/r^6$ potentials and is useful for simulation of interfaces *(Veld)*. It also performs standard Coulombic Ewald summations, but in a more efficient manner than the *ewald* style. The $1/r^6$ capability means that Lennard-Jones or Buckingham potentials can be used without a cutoff, i.e. they become full long-range potentials.

The *ewald/disp/dipole* style can also be used with point-dipoles, see *(Toukmaji)*.

The *ewald/dipole* style adds long-range standard Ewald summations for dipole-dipole interactions, see *(Toukmaji)*.

The *ewald/dipole/spin* style adds long-range standard Ewald summations for magnetic dipole-dipole interactions between magnetic spins.

---

The *esp* style implements the Ewald Summation with Prolates (ESP) method *(Liang2025)*, a fast Ewald-summation approach that can reduce the cost of long-range electrostatics compared to PPPM and PME. Without any loss of accuracy, ESP alters the fast Ewald pipeline in two places. First, for kernel splitting it uses prolate spheroidal wave functions (PSWFs) instead of Gaussians, which, thanks to the optimal concentration of PSWFs among band-limited functions, significantly reduces the required Fourier grid. With everything else equal, the FFT length drops by about a factor of two per dimension at high accuracy ($\approx 8\times$ in 3D). The residual kernel also vanishes at the real-space cutoff, eliminating any need for an "energy shift." Second, for particle-mesh operations ESP employs PSWFs in place of the B-splines used by PPPM. For comparable accuracy without k-space upsampling, PSWFs require fewer neighboring grid points (e.g., $\approx$ 8 vs $\approx$ 12 for five-digit accuracy). In contrast, PPPM solvers typically sets the spreading/interpolation order to order:math:=*5*, which forces substantial Fourier-space upsampling. Consequently, at the same cutoff radius, it often needs a much larger FFT grid, whereas ESP achieves similar accuracy with far shorter transforms, yielding roughly a sixfold reduction in FFT length when other parameters are held fixed.

---

The *pppm* style invokes a particle-particle particle-mesh solver *(Hockney)* which maps atom charge to a 3d mesh, uses 3d FFTs to solve Poisson's equation on the mesh, then interpolates electric fields on the mesh points back to the atoms. It is closely related to the particle-mesh Ewald technique (PME) *(Darden)* used in AMBER and CHARMM. The cost of traditional Ewald summation scales as $N^{\frac{3}{2}}$ where $N$ is the number of atoms in the system. The PPPM solver scales as $N \log N$ due to the FFTs, so it is almost always a faster choice *(Pollock)*.

The *pppm/cg* style is identical to the *pppm* style except that it has an optimization for systems where most particles are uncharged. Similarly the *msm/cg* style implements the same optimization for *msm*. The optional *smallq* argument

defines the cutoff for the absolute charge value which determines whether a particle is considered charged or not. Its default value is 1.0e-5.

The *pppm/dipole* style invokes a particle-particle particle-mesh solver for dipole-dipole interactions, following the method of *(Cerda)*. When charges are present in the system, charge-charge and charge-dipole interactions are also computed.

The *pppm/dipole/spin* style invokes a particle-particle particle-mesh solver for magnetic dipole-dipole interactions between magnetic spins.

The *pppm/tip4p* style is identical to the *pppm* style except that it adds a charge at the massless fourth site in each TIP4P water molecule. It should be used with *pair styles* with a *tip4p/long* in their style name.

The *pppm/stagger* style performs calculations using two different meshes, one shifted slightly with respect to the other. This can reduce force aliasing errors and increase the accuracy of the method for a given mesh size. Or a coarser mesh can be used for the same target accuracy, which saves CPU time. However, there is a trade-off since FFTs on two meshes are now performed which increases the computation required. See *(Cerutti)*, *(Neelov)*, and *(Hockney)* for details of the method.

For high relative accuracy, using staggered PPPM allows the mesh size to be reduced by a factor of 2 in each dimension as compared to regular PPPM (for the same target accuracy). This can give up to a 4x speedup in the KSpace time (8x less mesh points, 2x more expensive). However, for low relative accuracy, the staggered PPPM mesh size may be essentially the same as for regular PPPM, which means the method will be up to 2x slower in the KSpace time (simply 2x more expensive). For more details and timings, see the *Speed tips* doc page.

**Note**

Using *pppm/stagger* may not give the same increase in the accuracy of energy and pressure as it does in forces, so some caution must be used if energy and/or pressure are quantities of interest, such as when using a barostat.

---

The *pppm/disp* and *pppm/disp/tip4p* styles add a mesh-based long-range dispersion sum option for 1/r^6 potentials *(Isele-Holder)*, similar to the *ewald/disp* style. The 1/r^6 capability means that Lennard-Jones or Buckingham potentials can be used without a cutoff, i.e. they become full long-range potentials.

For these styles, you will possibly want to adjust the default choice of parameters by using the *kspace_modify* command. This can be done by either choosing the Ewald and grid parameters, or by specifying separate accuracies for the real and kspace calculations. When not making any settings, the simulation will stop with an error message. Further information on the influence of the parameters and how to choose them is described in *(Isele-Holder)*, *(Isele-Holder2)* and the *Howto dispersion* doc page.

---

Added in version 4Jul2026.

The *pppm/rk* kspace style is a variant of *pppm* designed for a heterogeneous multicore pppm computation of long-range forces. The heterogeneity is in the sense that the MPI communicator is partitioned into two classes of MPI processes. Thus, the use of *pppm/rk* requires the use of the command line *-partition* option. E.g.,

```
mpirun -n 100 lmp -partition 96 4 -in in.somescript
```

The R-process class of processes primarily computes the short-range forces and atom-wise updates, the latter of which includes the accumulation of charge densities. In the above command-line example, there would be 96 R-processes and 4 K-processes. The K-process class is mainly responsible for solving the Poisson equations with 3d FFT. The number of K-processes divides the number of R-processes, and the total number of processes is furthermore partitioned orthogonally into inter-RK blocks (as seen also in the *VerletSplit* class). Each inter-RK block communicator has one representative K process, with the rest of the processes being R processes.

The *pppm/rk* kspace style must be paired with an analogous *rk* type run style, e.g., *verlet/split/rk*. An error is generated otherwise. Also, *pppm/rk* does not currently support group/group computation (*group_group_enable==0*).

This approach is based on the enhanced baseline decomposition of *(Dandurand)* and works cited within.

---

**Note**

All of the PPPM styles can be used with single-precision FFTs by using the compiler switch -DFFT_SINGLE for the FFT_INC setting in your low-level Makefile. This setting also changes some of the PPPM operations (e.g. mapping charge to mesh and interpolating electric fields to particles) to be performed in single precision. This option can speed-up long-range calculations, particularly in parallel or on GPUs. The use of the -DFFT_SINGLE flag is discussed on the *Build settings* doc page. MSM does not currently support the -DFFT_SINGLE compiler switch.

---

The *electrode* styles add methods that are required for the constant potential method implemented in *fix electrode/**. The styles *ewald/electrode*, *pppm/electrode* and *pppm/electrode/intel* are available. These styles do not support the *kspace_modify slab nozforce* command.

---

The *msm* style invokes a multi-level summation method MSM solver, *(Hardy)* or *(Hardy2)*, which maps atom charge to a 3d mesh, and uses a multi-level hierarchy of coarser and coarser meshes on which direct Coulomb solvers are done. This method does not use FFTs and scales as *N*. It may therefore be faster than the other K-space solvers for relatively large problems when running on large core counts. MSM can also be used for non-periodic boundary conditions and for mixed periodic and non-periodic boundaries.

MSM is most competitive versus Ewald and PPPM when only relatively low accuracy forces, about 1e-4 relative error or less accurate, are needed. Note that use of a larger Coulombic cutoff (i.e. 15 Angstroms instead of 10 Angstroms) provides better MSM accuracy for both the real space and grid computed forces.

Currently calculation of the full pressure tensor in MSM is expensive. Using the *kspace_modify pressure/scalar yes* command provides a less expensive way to compute the scalar pressure (Pxx + Pyy + Pzz)/3.0. The scalar pressure can be used, for example, to run an isotropic barostat. If the full pressure tensor is needed, then calculating the pressure at every timestep or using a fixed pressure simulation with MSM will cause the code to run slower.

---

The *scafacos* style is a wrapper on the ScaFaCoS Coulomb solver library which provides a variety of solver methods which can be used with LAMMPS. The paper by *(Sutman)* gives an overview of ScaFaCoS.

ScaFaCoS was developed by a consortium of German research facilities with a BMBF (German Ministry of Science and Education) funded project in 2009-2012. Participants of the consortium were the Universities of Bonn, Chemnitz, Stuttgart, and Wuppertal as well as the Forschungszentrum Juelich.

The library is available for download at "http://www.scafacos.de/" or can be cloned from the git-repository "https://github.com/scafacos/scafacos".

In order to use this KSpace style, you must download and build the ScaFaCoS library, then build LAMMPS with the SCAFACOS package installed package which links LAMMPS to the ScaFaCoS library. See details on *this page*.

**Note**

Unlike other KSpace solvers in LAMMPS, ScaFaCoS computes all Coulombic interactions, both short- and long-range. Thus you should NOT use a Coulombic pair style when using kspace_style scafacos. This also means the

total Coulombic energy (short- and long-range) will be tallied for *thermodynamic output* command as part of the *elong* keyword; the *ecoul* keyword will be zero.

**Note**

See the current restriction below about use of ScaFaCoS in LAMMPS with molecular charged systems or the TIP4P water model.

The specified *method* determines which ScaFaCoS algorithm is used. These are the ScaFaCoS methods currently available from LAMMPS:

- *fmm* = Fast Multi-Pole method
- *p2nfft* = FFT-based Coulomb solver
- *ewald* = Ewald summation
- *direct* = direct O(N^2) summation
- *p3m* = PPPM

We plan to support additional ScaFaCoS solvers from LAMMPS in the future. For an overview of the included solvers, refer to *(Sutmann)*

The specified *accuracy* is similar to the accuracy setting for other LAMMPS KSpace styles, but is passed to ScaFaCoS, which can interpret it in different ways for different methods it supports. Within the ScaFaCoS library the *accuracy* is treated as a tolerance level (either absolute or relative) for the chosen quantity, where the quantity can be either the Columic field values, the per-atom Columic energy or the total Columic energy. To select from these options, see the *kspace_modify scafacos accuracy* doc page.

The *kspace_modify scafacos* command also explains other ScaFaCoS options currently exposed to LAMMPS.

---

Added in version 12Jun2025.

The *zero* style does not do any calculations, but is compatible with all pair styles that require some version of a kspace style. The accuracy argument is required for some internal calculations but has no impact of forces or energy, since those will always be zero.

---

The specified *accuracy* determines the relative RMS error in per-atom forces calculated by the long-range solver. It is set as a dimensionless number, relative to the force that two unit point charges (e.g. 2 monovalent ions) exert on each other at a distance of 1 Angstrom. This reference value was chosen as representative of the magnitude of electrostatic forces in atomic systems. Thus an accuracy value of 1.0e-4 means that the RMS error will be a factor of 10000 smaller than the reference force.

The accuracy setting is used in conjunction with the pairwise cutoff to determine the number of K-space vectors for style *ewald* or the grid size for style *esp*, *pppm* or *msm*.

For style *esp*, one accuracy value requests the overall relative force accuracy and is used as the PSWF kernel-splitting target. LAMMPS applies a tighter internal default to the PSWF spreading/interpolation target so that the automatically selected FFT grid and stencil order meet the requested accuracy. A second value can be specified to set the spreading/interpolation target explicitly.

Note that styles *esp* and *pppm* only computes the grid size at the beginning of a simulation, so if the length or triclinic tilt of the simulation cell increases dramatically during the course of the simulation, the accuracy of the simulation may degrade. Likewise, if the *kspace_modify slab* option is used with shrink-wrap boundaries in the z-dimension, and the

box size changes dramatically in z. For example, for a triclinic system with all three tilt factors set to the maximum limit, the PPPM grid should be increased roughly by a factor of 1.5 in the y direction and 2.0 in the z direction as compared to the same system using a cubic orthogonal simulation cell. One way to handle this issue if you have a long simulation where the box size changes dramatically, is to break it into shorter simulations (multiple *run* commands). This works because the grid size is re-computed at the beginning of each run. Another way to ensure the described accuracy requirement is met is to run a short simulation at the maximum expected tilt or length, note the required grid size, and then use the *kspace_modify mesh* command to manually set the ESP/PPPM grid size to this value for the long run. The simulation then will be "too accurate" for some portion of the run.

RMS force errors in real space for *ewald* and *pppm* are estimated using equation 18 of *(Kolafa)*, which is also referenced as equation 9 of *(Petersen)*. RMS force errors in K-space for *ewald* are estimated using equation 11 of *(Petersen)*, which is similar to equation 32 of *(Kolafa)*. RMS force errors in K-space for *pppm* are estimated using equation 38 of *(Deserno)*. RMS force errors for *msm* are estimated using ideas from chapter 3 of *(Hardy)*, with equation 3.197 of particular note. When using *msm* with non-periodic boundary conditions, it is expected that the error estimation will be too pessimistic. RMS force errors for dipoles when using *ewald/disp* or *ewald/dipole* are estimated using equations 33 and 46 of *(Wang)*. The RMS force errors for *pppm/dipole* are estimated using the equations in *(Cerda)*.

See the *kspace_modify* command for additional options of the K-space solvers that can be set, including a *force* option for setting an absolute RMS error in forces, as opposed to a relative RMS error.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

**Note**

For the GPU package, the *pppm/gpu* style performs charge assignment and force interpolation calculations on the GPU. These processes are performed either in single or double precision, depending on whether the -DFFT_SINGLE setting was specified in your low-level Makefile, as discussed above. The FFTs themselves are still calculated on the CPU. If *pppm/gpu* is used with a GPU-enabled pair style, part of the PPPM calculation can be performed concurrently on the GPU while other calculations for non-bonded and bonded force calculation are performed on the CPU.

**Note**

For the KOKKOS package, the *pppm/kk* style performs charge assignment and force interpolation calculations, along with the FFTs themselves, on the GPU or (optionally) threaded on the CPU when using OpenMP and FFTW3. The specific FFT library is selected using the FFT_KOKKOS CMake parameter. See the *Build settings* doc page for how to select a 3rd-party FFT library.

---

### 1.46.4 Restrictions

Note that the long-range electrostatic solvers in LAMMPS assume conducting metal (tinfoil) boundary conditions for both charge and dipole interactions. Vacuum boundary conditions are not currently supported.

The *ewald/disp*, *ewald*, *esp*, *pppm*, *pppm/disp*, and *msm* styles support non-orthogonal (triclinic symmetry) simulation boxes. However, triclinic simulation cells may not yet be supported by all suffix versions of these styles.

Changed in version 4Jul2026.

Triclinic (non-orthogonal) box support was added to the *pppm/disp* style and its *pppm/disp/tip4p*, *pppm/disp/omp*, and *pppm/disp/tip4p/omp* variants. It requires *ik* differentiation (the default; the *kspace_modify diff ad* option is not supported together with a triclinic box). The *kspace_modify slab* correction is supported for a triclinic box only in the EW3DC *volfactor* form and only with an *xy* tilt (xz = yz = 0, so the slab normal remains the z axis); *slab nozforce* and *slab ew2d* are not available for a triclinic box. For the *tip4p* variants the slab correction is not (yet) available together with a triclinic box, and (as for all TIP4P styles) the communication cutoff must be large enough to reconstruct the M-site, see the *TIP4P howto*.

Most of the base kspace styles are part of the KSPACE package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The *msm/dielectric* and *pppm/dielectric* kspace styles are part of the DIELECTRIC package. They are only enabled if LAMMPS was built with that package **and** the KSPACE package. See the *Build package* page for more info.

For MSM, a simulation must be 3d and one can use any combination of periodic, non-periodic, but not shrink-wrapped boundaries (specified using the *boundary* command).

For Ewald, ESP and PPPM, a simulation must be 3d and periodic in all dimensions. The only exception is if the slab option is set with *kspace_modify*, in which case the xy dimensions must be periodic and the z dimension must be non-periodic.

The scafacos KSpace style will only be enabled if LAMMPS is built with the SCAFACOS package. See the *Build package* doc page for more info.

The use of ScaFaCos in LAMMPS does not yet support molecular charged systems where the short-range Coulombic interactions between atoms in the same bond/angle/dihedral are weighted by the *special_bonds* command. Likewise it does not support the "TIP4P water style" where a fictitious charge site is introduced in each water molecule. Finally, the methods *p3m* and *ewald* do not support computing the virial, so this contribution is not included.

### 1.46.5 Related commands

*kspace_modify*, *pair_style lj/cut/coul/esp*, *pair_style lj/cut/coul/long*, *pair_style lj/charmm/coul/long*, *pair_style lj/long/coul/long*, *pair_style buck/coul/long*

### 1.46.6 Default

```
kspace_style none
```

---

**(Dandurand)** Dandurand, Vandierendonck, de Supinski, 39th IEEE IPDPS, June 3-7, (2025).

**(Darden)** Darden, York, Pedersen, J Chem Phys, 98, 10089 (1993).

**(Deserno)** Deserno and Holm, J Chem Phys, 109, 7694 (1998).

**(Hockney)** Hockney and Eastwood, Computer Simulation Using Particles, Adam Hilger, NY (1989).

**(Kolafa)** Kolafa and Perram, Molecular Simulation, 9, 351 (1992).

**(Liang2025)** Liang, Lu, Barnett, Greengard, Jiang, arXiv:2505.09727 (2025).

**(Petersen)** Petersen, J Chem Phys, 103, 3668 (1995).

**(Wang)** Wang and Holm, J Chem Phys, 115, 6277 (2001).

**(Pollock)** Pollock and Glosli, Comp Phys Comm, 95, 93 (1996).

**(Cerutti)** Cerutti, Duke, Darden, Lybrand, Journal of Chemical Theory and Computation 5, 2322 (2009)

**(Neelov)** Neelov, Holm, J Chem Phys 132, 234103 (2010)

**(Veld)** In 't Veld, Ismail, Grest, J Chem Phys, 127, 144711 (2007).

**(Toukmaji)** Toukmaji, Sagui, Board, and Darden, J Chem Phys, 113, 10913 (2000).

**(Isele-Holder)** Isele-Holder, Mitchell, Ismail, J Chem Phys, 137, 174107 (2012).

**(Isele-Holder2)** Isele-Holder, Mitchell, Hammond, Kohlmeyer, Ismail, J Chem Theory Comput 9, 5412 (2013).

**(Hardy)** David Hardy thesis: Multilevel Summation for the Fast Evaluation of Forces for the Simulation of Biomolecules, University of Illinois at Urbana-Champaign, (2006).

**(Hardy2)** Hardy, Stone, Schulten, Parallel Computing, 35, 164-177 (2009).

**(Sutmann)** Sutmann, Arnold, Fahrenberger, et. al., Physical review / E 88(6), 063308 (2013)

**(Cerda)** Cerda, Ballenegger, Lenz, Holm, J Chem Phys 129, 234104 (2008)

**(Sutmann) G. Sutmann. ScaFaCoS - a Scalable library of Fast Coulomb Solvers for particle Systems.** In Bajaj, Zavattieri, Koslowski, Siegmund, Proceedings of the Society of Engineering Science 51st Annual Technical Meeting. 2014.

# 1.47 label command

## 1.47.1 Syntax

```
label ID
```

- ID = string used as label name

## 1.47.2 Examples

```
label xyz
label loop
```

## 1.47.3 Description

Label this line of the input script with the chosen ID. Unless a jump command was used previously, this does nothing. But if a *jump* command was used with a label argument to begin invoking this script file, then all commands in the script prior to this line will be ignored. I.e. execution of the script will begin at this line. This is useful for looping over a section of the input script as discussed in the *jump* command.

## 1.47.4 Restrictions

none

### 1.47.5 Related commands

*jump*, *next*

### 1.47.6 Default

none

## 1.48 labelmap command

### 1.48.1 Syntax

```
labelmap option args
```

- *option* = *atom* or *bond* or *angle* or *dihedral* or *improper* or *clear* or *write* or *check_labels*

```
clear = no args
write arg = filename
atom or bond or angle or dihedral or improper
  args = list of one or more numeric-type/type-label pairs
check_labels value = string
  string = string containing any of the following characters: 'b', 'a', 'd', or 'i'
```

### 1.48.2 Examples

```
labelmap atom 1 c1 2 hc 3 cp 4 nt
labelmap atom 3 carbon 4 'c3"' 5 "c1'" 6 "c#"
labelmap atom $(label2type(atom,carbon)) C  # change type label from 'carbon' to 'C'
labelmap clear
labelmap write mymap.include
labelmap bond 1 carbonyl 2 nitrile 3 """ c1'-c2" """
```

### 1.48.3 Description

Added in version 15Sep2022.

Define alphanumeric type labels to associate with one or more numeric atom, bond, angle, dihedral or improper types. A collection of type labels for all atom types, bond types, etc. is stored as a label map.

The label map can also be defined by the *read_data* command when it reads these sections in a data file: Atom Type Labels, Bond Type Labels, etc. See the *Howto type labels* doc page for a general discussion of how type labels can be used. See *(Gissinger)* for a discussion of the type label implementation in LAMMPS and its uses.

Valid type labels can contain any alphanumeric character, but must not start with a number, a '#', or a '*' character. They can contain other standard ASCII characters such as angular or square brackets '<' and '>' or '[' and ']', parenthesis '(' and ')', dash '-', underscore '_', plus '+' and equals '=' signs and more. They must not contain blanks or any other whitespace. Note that type labels must be put in single or double quotation marks if they contain the '#' character or if they contain a double (") or single quotation mark ('). If the label contains both a single and a double quotation mark, then triple quotation (""") must be used. When enclosing a type label with quotation marks, the LAMMPS input parser may require adding leading or trailing blanks around the type label so it can identify the enclosing quotation marks. Those blanks will be removed when defining the label.

A *labelmap* command can only modify the label map for one type-kind (atom types, bond types, etc). Any number of numeric-type/type-label pairs may follow. If a type label already exists for the same numeric type, it will be overwritten. Type labels must be unique; assigning the same type label to multiple numeric types within the same type-kind is not

allowed. When reading and writing data files, it is required that there is a label defined for *every* numeric type within a given type-kind in order to write out the type label section for that type-kind.

The *clear* option resets the label map and thus discards all previous settings.

The *write* option takes a filename as argument and writes the current label mappings to a file as a sequence of *labelmap* commands, so the file can be copied into a new LAMMPS input file or read in using the *include* command.

Added in version 30Mar2026.

The *check_labels* keyword provides a warning if the type label of a bond, angle, dihedral, or improper defined in the simulation is not consistent with the atom types of its constituent atoms. This consistency check is performed only once, when the simulation is initialized by the first *run* or *minimize* command after this *labelmap* command. The *check_labels* value is a single string that should contain one or more of the characters 'b', 'a', 'd', and 'i', which correspond to bonds, angles, dihedrals, and impropers, respectively. For example, the keyword/value pair 'check_labels badi' will check all the type labels of all higher-order interactions, while 'check_labels adi' will only check type labels for angles, dihedrals, and impropers. The *check_labels* keyword requires a specific *type label* format to infer the types of higher-order interactions. Bond, angle, dihedral, and improper type labels must contain their constituent atom types delimited by hyphens, e.g., 'c2-c2-c2-n' for a dihedral that contains three atoms of type 'c2' and one atom of 'n'. If the constituent atoms do not have these atom types in the proper order, a warning will be generated when using this *check_types* keyword. Interactions that have been disabled, e.g., via *fix_shake*, will not be checked.

---

### 1.48.4 Restrictions

This command must come after the simulation box is defined by a *read_data*, *read_restart*, or *create_box* command.

Label maps are considered experimental when using the KOKKOS package.

### 1.48.5 Related commands

*read_data*, *write_data*, *molecule*, *fix bond/react*

### 1.48.6 Default

none

---

**(Gissinger)** J. R. Gissinger, I. Nikiforov, Y. Afshar, B. Waters, M. Choi, D. S. Karls, A. Stukowski, W. Im, H. Heinz, A. Kohlmeyer, and E. B. Tadmor, J Phys Chem B, 128, 3282-3297 (2024).

## 1.49 lattice command

### 1.49.1 Syntax

```
lattice style scale keyword values ...
```

- style = *none* or *sc* or *bcc* or *fcc* or *hcp* or *diamond* or *sq* or *sq2* or *hex* or *custom*
- scale = scale factor between lattice and simulation box

  ```
  scale = reduced density rho* (for LJ units)
  scale = lattice constant in distance units (for all other units)
  ```

- zero or more keyword/value pairs may be appended
- keyword = *origin* or *orient* or *spacing* or *a1* or *a2* or *a3* or *basis* or *triclinic/general*

```
origin values = x y z
  x,y,z = fractions of a unit cell (0 <= x,y,z < 1)
orient values = dim i j k
  dim = x or y or z
  i,j,k = integer lattice directions
spacing values = dx dy dz
  dx,dy,dz = lattice spacings in the x,y,z box directions
a1,a2,a3 values = x y z
  x,y,z = primitive vector components that define unit cell
basis values = x y z
  x,y,z = fractional coords of a basis atom (0 <= x,y,z < 1)
triclinic/general values = no values
```

### 1.49.2 Examples

```
lattice fcc 3.52
lattice hex 0.85
lattice sq 0.8 origin 0.0 0.5 0.0 orient x 1 1 0 orient y -1 1 0
lattice custom 3.52 a1 1.0 0.0 0.0 a2 0.5 1.0 0.0 a3 0.0 0.0 0.5 &
                    basis 0.0 0.0 0.0 basis 0.5 0.5 0.5 triclinic/general
lattice none 2.0
```

### 1.49.3 Description

Define a lattice for use by other commands. In LAMMPS, a lattice is simply a set of points in space, determined by a unit cell with basis atoms, that is replicated infinitely in all dimensions. The arguments of the lattice command can be used to define a wide variety of crystallographic lattices.

A lattice is used by LAMMPS in two ways. First, the *create_atoms* command creates atoms on the lattice points inside the simulation box. Note that the *create_atoms* command allows different atom types to be assigned to different basis atoms of the lattice. Second, the lattice spacing in the x,y,z dimensions implied by the lattice, can be used by other commands as distance units (e.g. *create_box*, *region* and *velocity*), which are often convenient to use when the underlying problem geometry is atoms on a lattice.

The lattice style must be consistent with the dimension of the simulation - see the *dimension* command. Styles *sc* or *bcc* or *fcc* or *hcp* or *diamond* are for 3d problems. Styles *sq* or *sq2* or *hex* are for 2d problems. Style *custom* can be used for either 2d or 3d problems.

A lattice consists of a unit cell, a set of basis atoms within that cell, and a set of transformation parameters (scale, origin, orient) that map the unit cell into the simulation box. The vectors a1,a2,a3 are the edge vectors of the unit cell. This is the nomenclature for "primitive" vectors in solid-state crystallography, but in LAMMPS the unit cell they determine does not have to be a "primitive cell" of minimum volume.

Note that the lattice command can be used multiple times in an input script. Each time it is invoked, the lattice attributes are re-defined and are used for all subsequent commands (that use lattice attributes). For example, a sequence of lattice, *region*, and *create_atoms* commands can be repeated multiple times to build a poly-crystalline model with different geometric regions populated with atoms in different lattice orientations.

---

A lattice of style *none* does not define a unit cell and basis set, so it cannot be used with the *create_atoms* command. However it does define a lattice spacing via the specified scale parameter. As explained above the lattice spacings in x,y,z can be used by other commands as distance units. No additional keyword/value pairs can be specified for the *none* style. By default, a "lattice none 1.0" is defined, which means the lattice spacing is the same as one distance unit, as defined by the *units* command.

Lattices of style *sc*, *fcc*, *bcc*, and *diamond* are 3d lattices that define a cubic unit cell with edge length = 1.0. This means a1 = 1 0 0, a2 = 0 1 0, and a3 = 0 0 1. Style *hcp* has a1 = 1 0 0, a2 = 0 sqrt(3) 0, and a3 = 0 0 sqrt(8/3). The placement of the basis atoms within the unit cell are described in any solid-state physics text. A *sc* lattice has 1 basis atom at the lower-left-bottom corner of the cube. A *bcc* lattice has 2 basis atoms, one at the corner and one at the center of the cube. A *fcc* lattice has 4 basis atoms, one at the corner and 3 at the cube face centers. A *hcp* lattice has 4 basis atoms, two in the z = 0 plane and 2 in the z = 0.5 plane. A *diamond* lattice has 8 basis atoms.

Lattices of style *sq* and *sq2* are 2d lattices that define a square unit cell with edge length = 1.0. This means a1 = 1 0 0 and a2 = 0 1 0. A *sq* lattice has 1 basis atom at the lower-left corner of the square. A *sq2* lattice has 2 basis atoms, one at the corner and one at the center of the square. A *hex* style is also a 2d lattice, but the unit cell is rectangular, with a1 = 1 0 0 and a2 = 0 sqrt(3) 0. It has 2 basis atoms, one at the corner and one at the center of the rectangle.

A lattice of style *custom* allows you to specify a1, a2, a3, and a list of basis atoms to put in the unit cell. By default, a1 and a2 and a3 are 3 orthogonal unit vectors (edges of a unit cube). But you can specify them to be of any length and non-orthogonal to each other, so that they describe a tilted parallelepiped. Via the *basis* keyword you add atoms, one at a time, to the unit cell. Its arguments are fractional coordinates (0.0 <= x,y,z < 1.0). For 2d simulations, the fractional z coordinate for any basis atom must be 0.0.

The position vector x of a basis atom within the unit cell is a linear combination of the unit cell's 3 edge vectors, i.e. x = bx a1 + by a2 + bz a3, where bx,by,bz are the 3 values specified for the *basis* keyword.

---

This subsection discusses the arguments that determine how the idealized unit cell is transformed into a lattice of points within the simulation box.

The *scale* argument determines how the size of the unit cell will be scaled when mapping it into the simulation box. I.e. it determines a multiplicative factor to apply to the unit cell, to convert it to a lattice of the desired size and distance units in the simulation box. The meaning of the *scale* argument depends on the *units* being used in your simulation.

For all unit styles except *lj*, the scale argument is specified in the distance units defined by the unit style. For example, in *real* or *metal* units, if the unit cell is a unit cube with edge length 1.0, specifying scale = 3.52 would create a cubic lattice with a spacing of 3.52 Angstroms. In *cgs* units, the spacing would be 3.52 cm.

For unit style *lj*, the scale argument is the Lennard-Jones reduced density, typically written as rho*. LAMMPS converts this value into the multiplicative factor via the formula "factor^dim = rho/rho*", where rho = N/V with V = the volume of the lattice unit cell and N = the number of basis atoms in the unit cell (described below), and dim = 2 or 3 for the dimensionality of the simulation. Effectively, this means that if LJ particles of size sigma = 1.0 are used in the simulation, the lattice of particles will be at the desired reduced density.

The *origin* option specifies how the unit cell will be shifted or translated when mapping it into the simulation box. The x,y,z values are fractional values (0.0 <= x,y,z < 1.0) meaning shift the lattice by a fraction of the lattice spacing in each dimension. The meaning of "lattice spacing" is discussed below. For 2d simulations, the *origin* z value must be 0.0.

The *orient* option specifies how the unit cell will be rotated when mapping it into the simulation box. The *dim* argument is one of the 3 coordinate axes in the simulation box. The other 3 arguments are the crystallographic direction in the lattice that you want to orient along that axis, specified as integers. E.g. "orient x 2 1 0" means the x-axis in the simulation box will be the [210] lattice direction, and similarly for y and z. The 3 lattice directions you specify do not have to be unit vectors, but they must be mutually orthogonal and obey the right-hand rule, i.e. (X cross Y) points in the Z direction. For 2d simulations, the *orient* x and y vectors must define 0 for their 3rd component. Similarly the *orient* z vector must define 0 for its 1st and 2nd components.

**Note**

The preceding paragraph describing lattice directions is only valid for orthogonal cubic unit cells (or square in 2d). If you are using a *hcp* or *hex* lattice or the more general lattice style *custom* with non-orthogonal a1,a2,a3 vectors, then you should think of the 3 *orient* vectors as creating a 3x3 rotation matrix which is applied to a1,a2,a3 to rotate the original unit cell to a new orientation in the simulation box.

The *triclinic/general* option specifies that the defined lattice is for use with a general triclinic simulation box, as opposed to an orthogonal or restricted triclinic box. The *Howto triclinic* doc page explains all 3 kinds of simulation boxes LAMMPS supports.

If this option is specified, a *custom* lattice style must be used. The *a1*, *a2*, *a3* vectors should define the edge vectors of a single unit cell of the lattice with one or more basis atoms. They edge vectors can be arbitrary so long as they are non-zero, distinct, and not co-planar. In addition, they must define a right-handed system, such that (*a1* cross *a2*) points in the direction of *a3*. Note that a left-handed system can be converted to a right-handed system by simply swapping the order of any pair of the *a1*, *a2*, *a3* vectors. For 2d simulations, the *a3* vector must be specified as (0.0,0.0,1.0), which is its default value.

If this option is used, the *origin* and *orient* settings must have their default values. Namely (0.0,0.0,0.0) for the *origin* and (100), (010), (001) for the *orient* vectors.

The *create_box* command can be used to create a general triclinic box that replicates the *a1*, *a2*, *a3* unit cell vectors in each direction to create the 3 arbitrary edge vectors of the overall simulation box. It requires a lattice with the *triclinic/general* option.

Likewise, the *create_atoms* command can be used to add atoms (or molecules) to a general triclinic box which lie on the lattice points defined by *a1*, *a2*, *a3* and the unit cell basis atoms. To do this, it also requires a lattice with the *triclinic/general* option.

**Note**

LAMMPS allows specification of general triclinic lattices and simulation boxes as a convenience for users who may be converting data from solid-state crystallographic representations or from DFT codes for input to LAMMPS. However, as explained on the *Howto_triclinic* doc page, internally, LAMMPS only uses restricted triclinic simulation boxes. This means the box and per-atom information (e.g. coordinates, velocities) defined by the *create_box* and *create_atoms* commands are converted from general to restricted triclinic form when the two commands are invoked. It also means that any other commands which use lattice spacings from this command (e.g. the region command), will be operating on a restricted triclinic simulation box, even if the *triclinic/general* option was used to define the lattice. See the next section for details.

---

Several LAMMPS commands have the option to use distance units that are inferred from "lattice spacings" in the x,y,z box directions. E.g. the *region* command can create a block of size 10x20x20, where 10 means 10 lattice spacings in the x direction.

**Note**

Though they are called lattice spacings, all the commands that have a "units lattice" option, simply use the 3 values as scale factors on the distance units defined by the *units* command. Thus if you do not like the lattice spacings computed by LAMMPS (e.g. for a non-orthogonal or rotated unit cell), you can define the 3 values to be whatever you wish, via the *spacing* option.

If the *spacing* option is not specified, the lattice spacings are computed by LAMMPS in the following way. A unit cell of the lattice is mapped into the simulation box (scaled and rotated), so that it now has (perhaps) a modified size and orientation. The lattice spacing in X is defined as the difference between the min/max extent of the x coordinates of the 8 corner points of the modified unit cell (4 in 2d). Similarly, the Y and Z lattice spacings are defined as the difference in the min/max of the y and z coordinates.

**Note**

If the *triclinic/general* option is specified, the unit cell defined by *a1*, *a2*, *a3* edge vectors is first converted to a restricted triclinic orientation, which is a rotation operation. The min/max extent of the 8 corner points is then determined, as described in the preceding paragraph, to set the lattice spacings. As explained for the *triclinic/general* option above, this is because any use of the lattice spacings by other commands will be for a restricted triclinic simulation box, not a general triclinic box.

Note that if the unit cell is orthogonal with axis-aligned edges (no rotation via the *orient* keyword), then the lattice spacings in each dimension are simply the scale factor (described above) multiplied by the length of a1,a2,a3. Thus a *hex* style lattice with a scale factor of 3.0 Angstroms, would have a lattice spacing of 3.0 in x and 3*sqrt(3.0) in y.

**Note**

For non-orthogonal unit cells and/or when a rotation is applied via the *orient* keyword, then the lattice spacings computed by LAMMPS are typically less intuitive. In particular, in these cases, there is no guarantee that a particular lattice spacing is an integer multiple of the periodicity of the lattice in that direction. Thus, if you create an orthogonal periodic simulation box whose size in a dimension is a multiple of the lattice spacing, and then fill it with atoms via the *create_atoms* command, you will NOT necessarily create a periodic system. I.e. atoms may overlap incorrectly at the faces of the simulation box.

The *spacing* option sets the 3 lattice spacings directly. All must be non-zero (use 1.0 for dz in a 2d simulation). The specified values are multiplied by the multiplicative factor described above that is associated with the scale factor. Thus a spacing of 1.0 means one unit cell edge length independent of the scale factor. As mentioned above, this option can be useful if the spacings LAMMPS computes are inconvenient to use in subsequent commands, which can be the case for non-orthogonal or rotated lattices.

Note that whenever the lattice command is used, the values of the lattice spacings LAMMPS calculates are printed out. Thus their effect in commands that use the spacings should be decipherable.

---

Example commands for generating a Wurtzite crystal. The lattice constants approximate those of CdSe. The $\sqrt{3} \times 1$ orthorhombic supercell is used with the x, y, and z directions oriented along $[\bar{1}\bar{2}30]$, $[10\bar{1}0]$, and [0001], respectively.

```
variable a equal  4.34
variable b equal  $a*sqrt(3.0)
variable c equal  $a*sqrt(8.0/3.0)

variable third equal 1.0/3.0
variable five6 equal 5.0/6.0

lattice custom    1.0     &
        a1      $b        0.0     0.0     &
        a2      0.0       $a      0.0     &
        a3      0.0       0.0     $c      &
        basis   0.0       0.0     0.0     &
        basis   0.5       0.5     0.0     &
        basis   ${third}  0.0     0.5     &
        basis   ${five6}  0.5     0.5     &
        basis   0.0       0.0     0.625   &
        basis   0.5       0.5     0.625   &
        basis   ${third}  0.0     0.125   &
```

(continues on next page)

(continued from previous page)

```
        basis   ${five6} 0.5     0.125

region myreg block 0 1 0 1 0 1
create_box      2 myreg
create_atoms    1 box      &
        basis   5       2       &
        basis   6       2       &
        basis   7       2       &
        basis   8       2
```

### 1.49.4 Restrictions

The *a1,a2,a3,basis* keywords can only be used with style *custom*.

### 1.49.5 Related commands

*dimension*, *create_atoms*, *region*

### 1.49.6 Default

```
lattice none 1.0
```

For other lattice styles, the option defaults are origin = 0.0 0.0 0.0, orient = x 1 0 0, orient = y 0 1 0, orient = z 0 0 1, a1 = 1 0 0, a2 = 0 1 0, and a3 = 0 0 1.

## 1.50 log command

### 1.50.1 Syntax

```
log file keyword
```

- file = name of new logfile
- keyword = *append* if output should be appended to logfile (optional)

### 1.50.2 Examples

```
log log.equil
log log.equil append
```

### 1.50.3 Description

This command closes the current LAMMPS log file, opens a new file with the specified name, and begins logging information to it. If the specified file name is *none*, then no new log file is opened. If the optional keyword *append* is specified, then output will be appended to an existing log file, instead of overwriting it.

If multiple processor partitions are being used, the file name should be a variable, so that different processors do not attempt to write to the same log file.

The file "log.lammps" is the default log file for a LAMMPS run. The name of the initial log file can also be set by the *-log command-line switch*.

### 1.50.4 Restrictions

none

### 1.50.5 Related commands

none

### 1.50.6 Default

The default LAMMPS log file is named log.lammps

## 1.51 mass command

### 1.51.1 Syntax

```
mass I value
```

- I = atom type (see asterisk form below), or type label
- value = mass

### 1.51.2 Examples

```
mass 1 1.0
mass * 62.5
mass 2* 62.5

labelmap atom 1 C
mass C 12.01
```

### 1.51.3 Description

Set the mass for all atoms of one or more atom types. Per-type mass values can also be set in the *read_data* data file using the "Masses" keyword. See the *units* command for what mass units to use.

The I index can be specified in one of several ways. An explicit numeric value can be used, as in the first example above. Or I can be a type label, which is an alphanumeric string defined by the *labelmap* command or in a section of a data file read by the *read_data* command, and which converts internally to a numeric type. Or a wild-card asterisk can be used to set the mass for multiple atom types. This takes the form "*" or "*n" or "n*" or "m*n", where m and n are numbers. If N = the number of atom types, then an asterisk with no numeric values means all types from 1 to N. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to N (inclusive). A middle asterisk means all types from m to n (inclusive).

A line in a *data file* that follows the "Masses" keyword specifies mass using the same format as the arguments of the mass command in an input script, except that no wild-card asterisk can be used. For example, under the "Masses" section of a data file, the line that corresponds to the first example above would be listed as

```
1 1.0
```

Note that the mass command can only be used if the *atom style* requires per-type atom mass to be set. Currently, all but the *sphere* and *ellipsoid* and *peri* styles do. They require mass to be set for individual particles, not types. Per-atom masses are defined in the data file read by the *read_data* command, or set to default values by the *create_atoms* command. Per-atom masses can also be set to new values by the *set mass* or *set density* commands.

Also note that *pair_style eam* and *pair_style bop* commands define the masses of atom types in their respective potential files, in which case the mass command is normally not used.

If you define a *hybrid atom style* which includes one (or more) sub-styles which require per-type mass and one (or more) sub-styles which require per-atom mass, then you must define both. However, in this case the per-type mass will be ignored; only the per-atom mass will be used by LAMMPS.

### 1.51.4 Restrictions

This command must come after the simulation box is defined by a *read_data*, *read_restart*, or *create_box* command.

All masses must be defined before a simulation is run. They must also all be defined before a *velocity* or *fix shake* command is used.

The mass assigned to any type or atom must be > 0.0.

### 1.51.5 Related commands

none

### 1.51.6 Default

none

## 1.52 mdi command

### 1.52.1 Syntax

```
mdi option args
```

- option = *engine* or *plugin* or *connect* or *exit*

```
engine args = zero or more keyword/args pairs
  keywords = elements
    elements args = N_1 N_2 ... N_ntypes
      N_1,N_2,...N_ntypes = chemical symbol for each of ntypes LAMMPS atom types
plugin args = name keyword value keyword value ...
  name = name of plugin library (e.g., lammps means a liblammps.so library will be␣
↪loaded)
  keyword/value pairs in any order, some are required, some are optional
  keywords = mdi or infile or extra or command
    mdi value = args passed to MDI for driver to operate with plugins (required)
    infile value = filename the engine will read at start-up (optional)
    extra value = aditional command-line args to pass to engine library when loaded␣
↪(optional)
    command value = a LAMMPS input script command to execute (required)
connect args = none
exit args = none
```

### 1.52.2 Examples

```
mdi engine
mdi engine elements Al Cu
mdi plugin lammps mdi "-role ENGINE -name lammps -method LINK" &
          infile in.aimd.engine extra "-log log.aimd.engine.plugin" &
```

(continues on next page)

(continued from previous page)

```
            command "run 5"
mdi connect
mdi exit
```

### 1.52.3 Description

This command implements operations within LAMMPS to use the MDI Library for coupling to other codes in a client/server protocol.

See the Howto MDI doc page for a discussion of all the different ways 2 or more codes can interact via MDI.

The examples/mdi directory has examples which use LAMMPS in 4 different modes: as a driver using an engine as either a stand-alone code or as a plugin, and as an engine operating as either a stand-alone code or as a plugin. The README file in that directory shows how to launch and couple codes for all the 4 usage modes, and so they communicate via the MDI library using either MPI or sockets.

The scripts in that directory illustrate the use of all the options for this command.

The *engine* option enables LAMMPS to act as an MDI engine (server), responding to requests from an MDI driver (client) code.

The *plugin* option enables LAMMPS to act as an MDI driver (client), and load the MDI engine (server) code as a library plugin. In this case the MDI engine is a library plugin. An MDI engine can also be a stand-alone code, launched separately from LAMMPS, in which case the mdi plugin command is not used.

The *connect* and *exit* options are only used when LAMMPS is acting as an MDI driver. As explained below, these options are normally not needed, except for a specific kind of use case.

---

The *mdi engine* command is used to make LAMMPS operate as an MDI engine. It is typically used in an input script after LAMMPS has setup the system it is going to model consistent with what the driver code expects. Depending on when the driver code tells the LAMMPS engine to exit, other commands can be executed after this command, but typically it is used at the end of a LAMMPS input script.

To act as an MDI engine operating as an MD code (or surrogate QM code), this is the list of standard MDI commands issued by a driver code which LAMMPS currently recognizes. Using standard commands defined by the MDI library means that a driver code can work interchangeably with LAMMPS or other MD codes or with QM codes which support the MDI standard. See more details about these commands in the MDI library documentation

These commands are valid at the @DEFAULT node defined by MDI. Commands that start with ">" mean the driver is sending information to LAMMPS. Commands that start with "<" are requests by the driver for LAMMPS to send it information. Commands that start with an alphabetic letter perform actions. Commands that start with "@" are MDI "node" commands, which are described further below.

| Command name | Action |
|---|---|
| >CELL or <CELL | Send/request 3 simulation box edge vectors (9 values) |
| >CELL_DISPL or <CELL_DISPL | Send/request displacement of the simulation box from the origin (3 values) |
| >CHARGES or <CHARGES | Send/request charge on each atom (N values) |
| >COORDS or <COORDS | Send/request coordinates of each atom (3N values) |
| >ELEMENTS | Send elements (atomic numbers) for each atom (N values) |
| <ENERGY | Request total energy (potential + kinetic) of the system (1 value) |
| >FORCES or <FORCES | Send/request forces on each atom (3N values) |
| >+FORCES | Send forces to add to each atom (3N values) |
| <LABELS | Request string label of each atom (N values) |
| <MASSES | Request mass of each atom (N values) |
| MD | Perform an MD simulation for N timesteps (most recent >NSTEPS value) |
| OPTG | Perform an energy minimization to convergence (most recent >TOLERANCE values) |
| >NATOMS or <NATOMS | Sends/request number of atoms in the system (1 value) |
| >NSTEPS | Send number of timesteps for next MD dynamics run via MD command |
| <PE | Request potential energy of the system (1 value) |
| <STRESS | Request symmetric stress tensor (virial) of the system (9 values) |
| >TOLERANCE | Send 4 tolerance parameters for next MD minimization via OPTG command |
| >TYPES or <TYPES | Send/request the LAMMPS atom type for each atom (N values) |
| >VELOCITIES or <VELOCITIES | Send/request the velocity of each atom (3N values) |
| @INIT_MD or @INIT_OPTG | Driver tells LAMMPS to start single-step dynamics or minimization (see below) |
| EXIT | Driver tells LAMMPS to exit engine mode |

**Note**

The <ENERGY, <FORCES, <PE, and <STRESS commands trigger LAMMPS to compute atomic interactions for the current configuration of atoms and size/shape of the simulation box. I.e. LAMMPS invokes its pair, bond, angle, ..., kspace styles. If the driver is updating the atom coordinates and/or box incrementally (as in an MD simulation which the driver is managing), then the LAMMPS engine will do the same, and only occasionally trigger neighbor list builds. If the change in atom positions is large (since the previous >COORDS command), then LAMMPS will do a more expensive operation to migrate atoms to new processors as needed and re-neighbor. If the >NATOMS or >TYPES or >ELEMENTS commands have been sent (since the previous >COORDS command), then LAMMPS assumes the system is new and re-initializes an entirely new simulation.

**Note**

The >TYPES or >ELEMENTS commands are how the MDI driver tells the LAMMPS engine which LAMMPS atom type to assign to each atom. If both the MDI driver and the LAMMPS engine are initialized so that atom type values are consistent in both codes, then the >TYPES command can be used. If not, the optional *elements* keyword can be used to specify what element each LAMMPS atom type corresponds to. This is specified by the chemical symbol of the element, e.g. C or Al or Si. A symbol must be specified for each of the ntypes LAMMPS atom types. Each LAMMPS type must map to a unique element; two or more types cannot map to the same element. Ntypes is typically specified via the *create_box* command or in the data file read by the *read_data* command. Once this has been done, the MDI driver can send an >ELEMENTS command to the LAMMPS driver with the atomic number of each atom and the LAMMPS engine will be able to map it to a LAMMPS atom type.

The MD and OPTG commands perform an entire MD simulation or energy minimization (to convergence) with no

communication from the driver until the simulation is complete. By contrast, the @INIT_MD and @INIT_OPTG commands allow the driver to communicate with the engine at each timestep of a dynamics run or iteration of a minimization; see more info below.

The MD command performs a simulation using the most recent >NSTEPS value. The OPTG command performs a minimization using the 4 convergence parameters from the most recent >TOLERANCE command. The 4 parameters sent are those used by the *minimize* command in LAMMPS: etol, ftol, maxiter, and maxeval.

The mdi engine command also implements the following custom MDI commands which are LAMMPS-specific. These commands are also valid at the @DEFAULT node defined by MDI:

- – Command name
  - – Action
- – >NBYTES
  - – Send # of datums in a subsequent command (1 value)
- – >COMMAND
  - – Send a LAMMPS input script command as a string (Nbytes in length)
- – >COMMANDS
  - – Send multiple LAMMPS input script commands as a newline-separated string (Nbytes in length)
- – >INFILE
  - – Send filename of an input script to execute (filename Nbytes in length)
- – <KE
  - – Request kinetic energy of the system (1 value)

Note that other custom commands can easily be added if these are not sufficient to support what a user-written driver code needs. Code to support new commands can be added to the MDI package within LAMMPS, specifically to the src/MDI/mdi_engine.cpp file.

MDI also defines a standard mechanism for the driver to request that an MD engine (LAMMPS) perform a dynamics simulation one step at a time or an energy minimization one iteration at a time. This is so that the driver can (optionally) communicate with LAMMPS at intermediate points of the timestep or iteration by issuing MDI node commands which start with "@".

To tell LAMMPS to run dynamics in single-step mode, the driver sends as @INIT_MD command followed by the these commands. The driver can interact with LAMMPS at 3 node locations within each timestep: @COORDS, @FORCES, @ENDSTEP:

- – Command name
  - – Action
- – @COORDS
  - – Proceed to next @COORDS node = post-integrate location in LAMMPS timestep
- – @FORCES
  - – Proceed to next @FORCES node = post-force location in LAMMPS timestep
- – @ENDSTEP
  - – Proceed to next @ENDSTEP node = end-of-step location in LAMMPS timestep
- – @DEFAULT
  - – Exit MD simulation, return to @DEFAULT node

- – EXIT
  - – Driver tells LAMMPS to exit the MD simulation and engine mode

To tell LAMMPS to run an energy minimization in single-iteration mode. The driver can interact with LAMMPS at 2 node locations within each iteration of the minimizer: @COORDS, @FORCES:

- – Command name
  - – Action
- – @COORDS
  - – Proceed to next @COORDS node = min-pre-force location in LAMMPS min iteration
- – @FORCES
  - – Proceed to next @FORCES node = min-post-force location in LAMMPS min iteration
- – @DEFAULT
  - – Exit minimization, return to @DEFAULT node
- – EXIT
  - – Driver tells LAMMPS to exit the minimization and engine mode

While LAMMPS is at its @COORDS node, the following standard MDI commands are supported, as documented above: >COORDS or <COORDS, @COORDS, @FORCES, @ENDSTEP, @DEFAULT, EXIT.

While LAMMPS is at its @FORCES node, the following standard MDI commands are supported, as documented above: <COORDS, <ENERGY, >FORCES or >+FORCES or <FORCES, <KE, <PE, <STRESS, @COORDS, @FORCES, @ENDSTEP, @DEFAULT, EXIT.

While LAMMPS is at its @ENDSTEP node, the following standard MDI commands are supported, as documented above: <ENERGY, <FORCES, <KE, <PE, <STRESS, @COORDS, @FORCES, @ENDSTEP, @DEFAULT, EXIT.

---

The *mdi plugin* command is used to make LAMMPS operate as an MDI driver which loads an MDI engine as a plugin library. It is typically used in an input script after LAMMPS has setup the system it is going to model consistent with the engine code.

The *name* argument specifies which plugin library to load. A name like "lammps" is converted to a filename liblammps.so. The path for where this file is located is specified by the -plugin_path switch within the -mdi command-line switch, which is specified when LAMMPS is launched. See the examples/mdi/README files for examples of how this is done.

The *mdi* keyword is required and is used as the -mdi argument passed to the library when it is launched. The -role and -method settings are required. The -name setting can be anything you choose. MDI drivers and engines can query their names to verify they are values they expect.

The *infile* keyword is optional. It sets the name of an input script which the engine will open and process. MDI will pass it as a command-line argument to the library when it is launched. The file typically contains settings that an MD or QM code will use for its calculations.

The *extra* keyword is optional. It contains additional command-line arguments which MDI will pass to the library when it is launched.

The *command* keyword is required. It specifies a LAMMPS input script command (as a single argument in quotes if it is multiple words). Once the plugin library is launched, LAMMPS will execute this command. Other previously-defined commands in the input script, such as the *fix mdi/qm* command, should perform MDI communication with the engine, while the specified *command* executes. Note that if *command* is an *include* command, then it could specify a filename with multiple LAMMPS commands.

**Note**

When the *command* is complete, LAMMPS will send an MDI EXIT command to the plugin engine and the plugin will be removed. The "mdi plugin" command will then exit and the next command (if any) in the LAMMPS input script will be processed. A subsequent "mdi plugin" command could then load the same or a different MDI plugin if desired.

---

The *mdi connect* and *mdi exit* commands are only used when LAMMPS is operating as an MDI driver. And when other LAMMPS command(s) which send MDI commands and associated data to/from the MDI engine are not able to initiate and terminate the connection to the engine code.

The only current MDI driver command in LAMMPS is the *fix mdi/qm* command. If it is only used once in an input script then it can initiate and terminate the connection, but if it is being issued multiple times (e.g., in a loop that issues a *clear* command), then it cannot initiate or terminate the connection multiple times. Instead, the *mdi connect* and *mdi exit* commands should be used outside the loop to initiate or terminate the connection.

See the examples/mdi/in.series.driver script for an example of how this is done. The LOOP in that script is reading a series of data file configurations and passing them to an MDI engine (e.g., quantum code) for energy and force evaluation. A *clear* command inside the loop wipes out the current system so a new one can be defined. This operation also destroys all fixes. So the *fix mdi/qm* command is issued once per loop iteration. Note that it includes a "connect no" option which disables the initiate/terminate logic within that fix.

### 1.52.4 Restrictions

This command is part of the MDI package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

To use LAMMPS in conjunction with other MDI-enabled atomistic codes, the *units* command should be used to specify *real* or *metal* units. This will ensure the correct unit conversions between LAMMPS and MDI units, which the other codes will also perform in their preferred units.

LAMMPS can also be used as an MDI engine in other unit choices it supports (e.g., *lj*), but then no unit conversion is performed.

### 1.52.5 Related commands

*fix mdi/qm*

### 1.52.6 Default

None

## 1.53 min_modify command

### 1.53.1 Syntax

```
min_modify keyword values ...
```

- one or more keyword/value pairs may be listed

```
keyword = dmax or line or norm or alpha_damp or discrete_factor or integrator or␣
→abcfire or tmax
  dmax value = max
    max = maximum distance for line search to move (distance units)
```

```
line value = backtrack or quadratic or forcezero or spin_cubic or spin_none
  backtrack,quadratic,forcezero,spin_cubic,spin_none = style of linesearch to use
norm value = two or inf or max
  two = Euclidean two-norm (length of 3N vector)
  inf = max force component across all 3-vectors
  max = max force norm across all 3-vectors
alpha_damp value = damping
  damping = fictitious magnetic damping for spin minimization (adim)
discrete_factor value = factor
  factor = discretization factor for adaptive spin timestep (adim)
integrator value = eulerimplicit or verlet or leapfrog or eulerexplicit
  time integration scheme for fire minimization
abcfire value = yes or no (default no)
  yes = use ABC-FIRE variant of fire minimization style
  no  = use default FIRE variant of fire minimization style
tmax value = factor
  factor = maximum adaptive timestep for fire minimization (adim)
```

### 1.53.2 Examples

```
min_modify dmax 0.2
min_modify integrator verlet tmax 4
```

### 1.53.3 Description

This command sets parameters that affect the energy minimization algorithms selected by the *min_style* command. The various settings may affect the convergence rate and overall number of force evaluations required by a minimization, so users can experiment with these parameters to tune their minimizations.

The *cg* and *sd* minimization styles have an outer iteration and an inner iteration which is steps along a one-dimensional line search in a particular search direction. The *dmax* parameter is how far any atom can move in a single line search in any dimension (x, y, or z). For the *quickmin* and *fire* minimization styles, the *dmax* setting is how far any atom can move in a single iteration (timestep). Thus a value of 0.1 in real *units* means no atom will move further than 0.1 Angstroms in a single outer iteration. This prevents highly overlapped atoms from being moved long distances (e.g. through another atom) due to large forces.

The choice of line search algorithm for the *cg* and *sd* minimization styles can be selected via the *line* keyword. The default *quadratic* line search algorithm starts out using the robust backtracking method described below. However, once the system gets close to a local minimum and the linesearch steps get small, so that the energy is approximately quadratic in the step length, it uses the estimated location of zero gradient as the linesearch step, provided the energy change is downhill. This becomes more efficient than backtracking for highly-converged relaxations. The *forcezero* line search algorithm is similar to *quadratic*. It may be more efficient than *quadratic* on some systems.

The backtracking search is robust and should always find a local energy minimum. However, it will "converge" when it can no longer reduce the energy of the system. Individual atom forces may still be larger than desired at this point, because the energy change is measured as the difference of two large values (energy before and energy after) and that difference may be smaller than machine epsilon even if atoms could move in the gradient direction to reduce forces further.

The choice of a norm can be modified for the min styles *cg*, *sd*, *quickmin*, *fire*, *spin*, *spin/cg*, and *spin/lbfgs* using the *norm* keyword. The default *two* norm computes the 2-norm (Euclidean length) of the global force vector:

$$||\vec{F}||_2 = \sqrt{\vec{F}_1^2 + \cdots + \vec{F}_N^2}$$

The *max* norm computes the length of the 3-vector force for each atom (2-norm), and takes the maximum value of

those across all atoms

$$||\vec{F}||_{max} = \max\left(||\vec{F}_1||, \cdots, ||\vec{F}_N||\right)$$

The *inf* norm takes the maximum component across the forces of all atoms in the system:

$$||\vec{F}||_{inf} = \max\left(|F_1^1|, |F_1^2|, |F_1^3| \cdots, |F_N^1|, |F_N^2|, |F_N^3|\right)$$

For the min styles *spin*, *spin/cg* and *spin/lbfgs*, the force norm is replaced by the spin-torque norm.

Keywords *alpha_damp* and *discrete_factor* only make sense when a *min_spin* command is declared. Keyword *alpha_damp* defines an analog of a magnetic damping. It defines a relaxation rate toward an equilibrium for a given magnetic system. Keyword *discrete_factor* defines a discretization factor for the adaptive timestep used in the *spin* minimization. See *min_spin* for more information about those quantities.

The choice of a line search algorithm for the *spin/cg* and *spin/lbfgs* styles can be specified via the *line* keyword. The *spin_cubic* and *spin_none* keywords only make sense when one of those two minimization styles is declared. The *spin_cubic* performs the line search based on a cubic interpolation of the energy along the search direction. The *spin_none* keyword deactivates the line search procedure. The *spin_none* is a default value for *line* keyword for both *spin/lbfgs* and *spin/cg*. Convergence of *spin/lbfgs* can be more robust if *spin_cubic* line search is used.

The Newton *integrator* used for *fire* minimization can be selected to be either the symplectic Euler (*eulerimplicit*), velocity Verlet (*verlet*), Leapfrog (*leapfrog*) or non-symplectic forward Euler (*eulerexplicit* ). The keyword *tmax* defines the maximum value for the adaptive timestep during a *fire* minimization. It is a multiplication factor applied to the current *timestep* (not in time unit). For example, *tmax* = 4.0 with a *timestep* of 2fs, means that the maximum value the timestep can reach during a *fire* minimization is 4fs. Note that parameter defaults has been chosen to be reliable in most cases, but one should consider adjusting *timestep* and *tmax* to optimize the minimization for large or complex systems. Other parameters of the *fire* minimization can be tuned (*tmin*, *delaystep*, *dtgrow*, *dtshrink*, *alpha0*, and *alphashrink*). Please refer to the references describing the *min_style fire*. An additional stopping criteria *vdfmax* is used by *fire* in order to avoid unnecessary looping when it is reasonable to think the system will not be relaxed further. Note that in this case the system will NOT have reached your minimization criteria. This could happen when the system comes to be stuck in a local basin of the phase space. *vdfmax* is the maximum number of consecutive iterations with P(t) < 0.

Added in version 8Feb2023.

The *abcfire* keyword allows to activate the ABC-FIRE variant of the *fire* minimization algorithm. ABC-FIRE introduces an additional factor that modifies the bias and scaling of the velocities of the atoms during the mixing step *(Echeverri Restrepo)*. This can lead to faster convergence of the minimizer.

The *min_style fire* is an optimized implementation. It can behave similarly to the previous version by using the following set of parameters:

```
min_modify integrator eulerexplicit tmax 10.0 tmin 0.0 delaystep 5 &
           dtgrow 1.1 dtshrink 0.5 alpha0 0.1 alphashrink 0.99 &
           vdfmax 100000 halfstepback no initialdelay no
```

### 1.53.4 Restrictions

For magnetic GNEB calculations, only *spin_none* value for *line* keyword can be used when minimization styles *spin/cg* and *spin/lbfgs* are employed. See *neb/spin* for more explanation.

### 1.53.5 Related commands

*min_style*, *minimize*

### 1.53.6 Default

The option defaults are dmax = 0.1, line = quadratic and norm = two.

For the *spin*, *spin/cg* and *spin/lbfgs* styles, the option defaults are alpha_damp = 1.0, discrete_factor = 10.0, line = spin_none, and norm = euclidean.

For the *fire* style, the option defaults are integrator = eulerimplicit, tmax = 10.0, tmin = 0.02, delaystep = 20, dtgrow = 1.1, dtshrink = 0.5, alpha0 = 0.25, alphashrink = 0.99, vdfmax = 2000, halfstepback = yes and initialdelay = yes.

**(EcheverriRestrepo)** Echeverri Restrepo, Andric, Comput Mater Sci, 218, 111978 (2023).

## 1.54 min_style spin command

## 1.55 min_style spin/cg command

## 1.56 min_style spin/lbfgs command

### 1.56.1 Syntax

```
min_style spin
min_style spin/cg
min_style spin/lbfgs
```

### 1.56.2 Examples

```
min_style  spin/lbfgs
min_modify line spin_cubic discrete_factor 10.0
```

### 1.56.3 Description

Apply a minimization algorithm to use when a *minimize* command is performed.

Style *spin* defines a damped spin dynamics with an adaptive timestep, according to:

$$\frac{d\vec{s}_i}{dt} = \lambda\, \vec{s}_i \times \left(\vec{\omega}_i \times \vec{s}_i\right)$$

with $\lambda$ a damping coefficient (similar to a magnetic damping). $\lambda$ can be defined by setting the *alpha_damp* keyword with the *min_modify* command.

The minimization procedure solves this equation using an adaptive timestep. The value of this timestep is defined by the largest precession frequency that has to be solved in the system:

$$\Delta t_{\max} = \frac{2\pi}{\kappa\, |\vec{\omega}_{\max}|}$$

with $|\vec{\omega}_{\max}|$ the norm of the largest precession frequency in the system (across all processes, and across all replicas if a spin/neb calculation is performed).

$\kappa$ defines a discretization factor *discrete_factor* for the definition of this timestep. *discrete_factor* can be defined with the *min_modify* command.

Style *spin/cg* defines an orthogonal spin optimization (OSO) combined to a conjugate gradient (CG) algorithm. The *min_modify* command can be used to couple the *spin/cg* to a line search procedure, and to modify the discretization factor *discrete_factor*. By default, style *spin/cg* does not employ the line search procedure and uses the adaptive timestep technique in the same way as style *spin*.

Style *spin/lbfgs* defines an orthogonal spin optimization (OSO) combined to a limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) algorithm. By default, style *spin/lbfgs* does not employ line search procedure. If the line search procedure is not used then the discrete factor defines the maximum root mean squared rotation angle of spins by equation *pi/(5*Kappa)*. The default value for Kappa is 10. The *spin_cubic* line search option can improve the convergence of the *spin/lbfgs* algorithm.

The *min_modify* command can be used to activate the line search procedure, and to modify the discretization factor *discrete_factor*.

For more information about styles *spin/cg* and *spin/lbfgs*, see their implementation reported in *(Ivanov)*.

**Note**

All the *spin* styles replace the force tolerance by a torque tolerance. See *minimize* for more explanation.

**Note**

The *spin/cg* and *spin/lbfgs* styles can be used for magnetic NEB calculations only if the line search procedure is deactivated. See *neb/spin* for more explanation.

### 1.56.4 Restrictions

The *spin*, *spin/cg*, and *spin/lbfgps* styles are part of the SPIN package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This minimization procedure is only applied to spin degrees of freedom for a frozen lattice configuration.

### 1.56.5 Related commands

*min_style*, *minimize*, *min_modify*

### 1.56.6 Default

The option defaults are *alpha_damp* = 1.0, *discrete_factor* = 10.0, *line* = spin_none and *norm* = euclidean.

---

**(Ivanov)** Ivanov, Uzdin, Jonsson. arXiv preprint arXiv:1904.02669, (2019).

## 1.57 min_style cg command

Accelerator Variant: *cg/kk*

## 1.58 min_style hftn command

## 1.59 min_style sd command

## 1.60 min_style quickmin command

## 1.61 min_style fire command

Added in version 11Feb2026.

Accelerator Variant: *fire/kk*

## 1.62 *min_style spin* command

## 1.63 *min_style spin/cg* command

## 1.64 *min_style spin/lbfgs* command

### 1.64.1 Syntax

```
min_style style
```

- style = *cg* or *hftn* or *sd* or *quickmin* or *fire* or *spin* or *spin/cg* or *spin/lbfgs*

  *spin* is discussed briefly here and fully on *min_style spin* doc page
  *spin/cg* is discussed briefly here and fully on *min_style spin* doc page
  *spin/lbfgs* is discussed briefly here and fully on *min_style spin* doc page

### 1.64.2 Examples

```
min_style cg
min_style fire
min_style spin
```

### 1.64.3 Description

Choose a minimization algorithm to use when a *minimize* command is performed.

Style *cg* is the Polak-Ribiere version of the conjugate gradient (CG) algorithm. At each iteration the force gradient is combined with the previous iteration information to compute a new search direction perpendicular (conjugate) to the previous search direction. The PR variant affects how the direction is chosen and how the CG method is restarted when it ceases to make progress. The PR variant is thought to be the most effective CG choice for most problems.

Style *hftn* is a Hessian-free truncated Newton algorithm. At each iteration a quadratic model of the energy potential is solved by a conjugate gradient inner iteration. The Hessian (second derivatives) of the energy is not formed directly, but approximated in each conjugate search direction by a finite difference directional derivative. When close to an energy minimum, the algorithm behaves like a Newton method and exhibits a quadratic convergence rate to high accuracy. In most cases the behavior of *hftn* is similar to *cg*, but it offers an alternative if *cg* seems to perform poorly. This style is not affected by the *min_modify* command.

Style *sd* is a steepest descent algorithm. At each iteration, the search direction is set to the downhill direction corresponding to the force vector (negative gradient of energy). Typically, steepest descent will not converge as quickly as CG, but may be more robust in some situations.

Style *quickmin* is a damped dynamics method described in *(Sheppard)*, where the damping parameter is related to the projection of the velocity vector along the current force vector for each atom. The velocity of each atom is initialized to 0.0 by this style, at the beginning of a minimization.

Style *fire* is a damped dynamics method described in *(Bitzek)*, which is similar to *quickmin* but adds a variable timestep and alters the projection operation to maintain components of the velocity non-parallel to the current force vector. The velocity of each atom is initialized to 0.0 by this style, at the beginning of a minimization. This style correspond to an optimized version described in *(Guenole)* that include different time integration schemes and default parameters. The default parameters can be modified with the command *min_modify*.

Style *spin* is a damped spin dynamics with an adaptive timestep.

Style *spin/cg* uses an orthogonal spin optimization (OSO) combined to a conjugate gradient (CG) approach to minimize spin configurations.

Style *spin/lbfgs* uses an orthogonal spin optimization (OSO) combined to a limited-memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) approach to minimize spin configurations.

See the *min/spin* page for more information about the *spin*, *spin/cg* and *spin/lbfgs* styles.

Either the *quickmin* or the *fire* styles are useful in the context of nudged elastic band (NEB) calculations via the *neb* command.

Either the *spin*, *spin/cg*, or *spin/lbfgs* styles are useful in the context of magnetic geodesic nudged elastic band (GNEB) calculations via the *neb/spin* command.

**Note**

The damped dynamic minimizers use whatever timestep you have defined via the *timestep* command. Often they will converge more quickly if you use a timestep about 10x larger than you would normally use for dynamics simulations. For *fire*, the default timestep is recommended to be equal to the one you would normally use for dynamics simulations.

**Note**

The *quickmin*, *fire*, *hftn*, and *cg/kk* styles do not yet support the use of the *fix box/relax* command or minimizations involving the electron radius in *eFF* models.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 1.64.4 Restrictions

The *spin*, *spin/cg*, and *spin/lbfgps* styles are part of the SPIN package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 1.64.5 Related commands

*min_modify*, *minimize*, *neb*

### 1.64.6 Default

```
min_style cg
```

---

**(Sheppard)** Sheppard, Terrell, Henkelman, J Chem Phys, 128, 134106 (2008). See ref 1 in this paper for original reference to Qmin in Jonsson, Mills, Jacobsen.

**(Bitzek)** Bitzek, Koskinen, Gahler, Moseler, Gumbsch, Phys Rev Lett, 97, 170201 (2006).

**(Guenole)** Guenole, Noehring, Vaid, Houlle, Xie, Prakash, Bitzek, Comput Mater Sci, 175, 109584 (2020).

## 1.65 minimize command

Accelerator Variant: minimize/kk

### 1.65.1 Syntax

```
minimize etol ftol maxiter maxeval
```

- etol = stopping tolerance for energy (unitless)
- ftol = stopping tolerance for force (force units)
- maxiter = max iterations of minimizer
- maxeval = max number of force/energy evaluations

### 1.65.2 Examples

```
minimize 1.0e-4 1.0e-6 100 1000
minimize 0.0 1.0e-8 1000 100000
```

### 1.65.3 Description

Perform an energy minimization of the system, by iteratively adjusting atom coordinates. Iterations are terminated when one of the stopping criteria is satisfied. At that point the configuration will hopefully be in a local potential energy minimum. More precisely, the configuration should approximate a critical point for the objective function (see below), which may or may not be a local minimum.

The minimization algorithm used is set by the *min_style* command. Other options are set by the *min_modify* command. Minimize commands can be interspersed with *run* commands to alternate between relaxation and dynamics. The minimizers bound the distance atoms may move in one iteration, so that you can relax systems with highly overlapped atoms (large energies and forces) by pushing the atoms off of each other.

**Neighbor list update settings**

The distance that atoms can move during individual minimization steps can be quite large, especially at the beginning of a minimization. Thus *neighbor list settings* of *every* = *1* and *delay* = *0* are **required**. This may be combined with either *check* = *no* (always update the neighbor list) or *check* = *yes* (only update the neighbor list if at least one atom has moved more than half the *neighbor list skin* distance since the last reneighboring). Using *check* = *yes* is recommended since it avoids unneeded reneighboring steps when the system is closer to the minimum and thus atoms move only small distances. Using *check* = *no* may be required for debugging or when coupling LAMMPS with external codes that require a predictable sequence of neighbor list updates.

If the settings are **not** *every = 1* and *delay = 0*, LAMMPS will temporarily apply a *neigh_modify every 1 delay 0 check yes* setting during the minimization and restore the original setting at the end of the minimization. A corresponding message will be printed to the screen and log file, if this happens.

Alternate means of relaxing a system are to run dynamics with a small or *limited timestep*. Or dynamics can be run using *fix viscous* to impose a damping force that slowly drains all kinetic energy from the system. The *pair_style soft* potential can be used to un-overlap atoms while running dynamics.

Note that you can minimize some atoms in the system while holding the coordinates of other atoms fixed by applying *fix setforce 0.0 0.0 0.0* to the other atoms. See a more detailed discussion of *using fixes while minimizing below*.

The *minimization styles* *cg*, *sd*, and *hftn* involves an outer iteration loop which sets the search direction along which atom coordinates are changed. An inner iteration is then performed using a line search algorithm. The line search typically evaluates forces and energies several times to set new coordinates. Currently, a backtracking algorithm is used which may not be optimal in terms of the number of force evaluations performed, but appears to be more robust than previous line searches we have tried. The backtracking method is described in Nocedal and Wright's Numerical Optimization (Procedure 3.1 on p 41).

The *minimization styles* *quickmin* and *fire* perform damped dynamics using an Euler integration step. Thus they require a *timestep* be defined.

**Note**

The damped dynamic minimizer algorithms will use the timestep you have defined via the *timestep* command or its default value. Often they will converge more quickly if you use a timestep about 10x larger than you would normally use for regular molecular dynamics simulations.

---

In all cases, the objective function being minimized is the total potential energy of the system as a function of the N atom coordinates:

$$E(r_1, r_2, \ldots, r_N) = \sum_{i,j} E_{pair}(r_i, r_j) + \sum_{ij} E_{bond}(r_i, r_j) + \sum_{ijk} E_{angle}(r_i, r_j, r_k) + \sum_{ijkl} E_{dihedral}(r_i, r_j, r_k, r_l) + \sum_{ijkl} E_{improper}(r_i, r_j, r_k, r_l) + \sum_{i} E_{fix}(r_i)$$

where the first term is the sum of all non-bonded *pairwise interactions* including *long-range Coulombic interactions*, the second through fifth terms are *bond*, *angle*, *dihedral*, and *improper* interactions respectively, and the last term is energy due to *fixes* which can act as constraints or apply force to atoms, such as through interaction with a wall. See the discussion below about how fix commands affect minimization.

The starting point for the minimization is the current configuration of the atoms.

---

The minimization procedure stops if any of several criteria are met:

- the change in energy between outer iterations is less than *etol*
- the 2-norm (length) of the global force vector is less than the *ftol*
- the line search fails because the step distance backtracks to 0.0
- the number of outer iterations or timesteps exceeds *maxiter*
- the number of total force evaluations exceeds *maxeval*

**Note**

the *minimization style* *spin*, *spin/cg*, and *spin/lbfgs* replace the force tolerance *ftol* by a torque tolerance. The minimization procedure stops if the 2-norm (length) of the torque vector on atom (defined as the cross product between the atomic spin and its precession vectors omega) is less than *ftol*, or if any of the other criteria are met. Torque have the same units as the energy.

**Note**

You can also use the *fix halt* command to specify a general criterion for exiting a minimization, that is a calculation performed on the state of the current system, as defined by an *equal-style variable*.

For the first criterion, the specified energy tolerance *etol* is unitless; it is met when the energy change between successive iterations divided by the energy magnitude is less than or equal to the tolerance. For example, a setting of 1.0e-4 for *etol* means an energy tolerance of one part in 10^4. For the damped dynamics minimizers this check is not performed for a few steps after velocities are reset to 0, otherwise the minimizer would prematurely converge.

For the second criterion, the specified force tolerance *ftol* is in force units, since it is the length of the global force vector for all atoms, e.g. a vector of size 3N for N atoms. Since many of the components will be near zero after minimization, you can think of *ftol* as an upper bound on the final force on any component of any atom. For example, a setting of 1.0e-4 for *ftol* means no x, y, or z component of force on any atom will be larger than 1.0e-4 (in force units) after minimization.

Either or both of the *etol* and *ftol* values can be set to 0.0, in which case some other criterion will terminate the minimization.

During a minimization, the outer iteration count is treated as a timestep. Output is triggered by this timestep, e.g. thermodynamic output or dump and restart files.

Using the *thermo_style custom* command with the *fmax* or *fnorm* keywords can be useful for monitoring the progress of the minimization. Note that these outputs will be calculated only from forces on the atoms, and will not include any extra degrees of freedom, such as from the *fix box/relax* command.

Following minimization, a statistical summary is printed that lists which convergence criterion caused the minimizer to stop, as well as information about the energy, force, final line search, and iteration counts. An example is as follows:

```
Minimization stats:
  Stopping criterion = max iterations
  Energy initial, next-to-last, final =
       -0.626828169302     -2.82642039062     -2.82643549739
  Force two-norm initial, final = 2052.1 91.9642
  Force max component initial, final = 346.048 9.78056
  Final line search alpha, max atom move = 2.23899e-06 2.18986e-05
  Iterations, force evaluations = 2000 12724
```

The 3 energy values are for before and after the minimization and on the next-to-last iteration. This is what the *etol* parameter checks.

The two-norm force values are the length of the global force vector before and after minimization. This is what the *ftol* parameter checks.

The max-component force values are the absolute value of the largest component (x,y,z) in the global force vector, i.e. the infinity-norm of the force vector.

The alpha parameter for the line-search, when multiplied by the max force component (on the last iteration), gives the max distance any atom moved during the last iteration. Alpha will be 0.0 if the line search could not reduce the energy.

Even if alpha is non-zero, if the "max atom move" distance is tiny compared to typical atom coordinates, then it is possible the last iteration effectively caused no atom movement and thus the evaluated energy did not change and the minimizer terminated. Said another way, even with non-zero forces, it is possibly the effect of those forces to move atoms a distance less than machine precision, so that the energy cannot be further reduced.

The iterations and force evaluation values are what is checked by the *maxiter* and *maxeval* parameters.

---

**Note**

There are several force fields in LAMMPS which have discontinuities or other approximations which may prevent you from performing an energy minimization to tight tolerances. For example, you should use a *pair style* that goes to 0.0 at the cutoff distance when performing minimization (even if you later change it when running dynamics). If you do not do this, the total energy of the system will have discontinuities when the relative distance between any pair of atoms changes from cutoff *plus* epsilon to cutoff *minus* epsilon and the minimizer may thus behave poorly. Some of the many-body potentials use splines and other internal cutoffs that inherently have this problem. The *long-range Coulombic styles* (PPPM, Ewald) are approximate to within the user-specified tolerance, which means their energy and forces may not agree to a higher precision than the Kspace-specified tolerance. This agreement is further reduced when using tabulation to speed up the computation of the real-space part of the Coulomb interactions, which is enabled by default. In all these cases, the minimizer may give up and stop before finding a minimum to the specified energy or force tolerance.

Note that a cutoff Lennard-Jones potential (and others) can be shifted so that its energy is 0.0 at the cutoff via the *pair_modify* command. See the doc pages for individual *pair styles* for details. Note that most Coulombic potentials have a cutoff, unless versions with a long-range component are used (e.g. *pair_style lj/cut/coul/long*) or some other damping/smoothing schemes are used. The CHARMM potentials go to 0.0 at the cutoff (e.g. *pair_style lj/charmm/coul/charmm*), as do the GROMACS potentials (e.g. *pair_style lj/gromacs*).

If a soft potential (*pair_style soft*) is used the Astop value is used for the prefactor (no time dependence).

The *fix box/relax* command can be used to apply an external pressure to the simulation box and allow it to shrink/expand during the minimization.

Only a few other fixes (typically those that add forces) are invoked during minimization. See the doc pages for individual *fix* commands to see which ones are relevant. Current examples of fixes that can be used include:

- *fix addforce*
- *fix addtorque/group*
- *fix efield*
- *fix enforce2d*
- *fix indent*
- *fix lineforce*
- *fix planeforce*
- *fix setforce*
- *fix spring*
- *fix spring/self*
- *fix viscous*
- *fix wall*
- *fix wall/region*

**Note**

Some fixes which are invoked during minimization have an associated potential energy. For that energy to be included in the total potential energy of the system (the quantity being minimized), you MUST enable the *fix_modify energy* option for that fix. The doc pages for individual *fix* commands specify if this should be done.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 1.65.4 Restrictions

Changed in version 15Sep2022.

It used to be an error to use *fix shake* with minimization but this is no longer the case. While the SHAKE constraint equations cannot be applied during minimization, fix shake (and fix rattle) will substitute the constraints with stiff harmonic bonds. This allows some (small) changes in the geometry, but those are usually small enough so that when starting an MD run the constraints can be fulfilled during the first time step. The same effect can be achieved without fix shake by replacing the force constants for the bonds and/or angles that would otherwise be constrained by the SHAKE algorithm.

*Fix rigid* is not supported during minimization. It does not trigger an error to have it defined, but the energy minimization will not keep the defined body(s) rigid during the minimization. Note that if bonds, angles, etc internal to a rigid body have been turned off (e.g. via *neigh_modify exclude*), they will not contribute to the potential energy which is probably not what is desired. One can approximate a minimization with rigid bodies by performing simulated annealing.

Pair potentials that produce torque on a particle (e.g. *granular potentials* or the *GayBerne potential* for ellipsoidal particles) are not relaxed by a minimization. More specifically, radial relaxations are induced, but no rotations are induced by a minimization, so such a system will not fully relax.

### 1.65.5 Related commands

*min_modify*, *min_style*, *run_style*

### 1.65.6 Default

none

# 1.66 molecule command

## 1.66.1 Syntax

```
molecule ID file1 keyword values ... file2 keyword values ... fileN ...
```

- ID = user-assigned name for the molecule template
- file1,file2,... = names of files containing molecule descriptions
- zero or more keyword/value pairs may be appended after each file
- keyword = *offset* or *toff* or *boff* or *aoff* or *doff* or *ioff* or *scale* or *check_labels* or *auto*

```
offset values = Toff Boff Aoff Doff Ioff
  Toff = offset to add to atom types
  Boff = offset to add to bond types
  Aoff = offset to add to angle types
  Doff = offset to add to dihedral types
  Ioff = offset to add to improper types
toff value = Toff
  Toff = offset to add to atom types
boff value = Boff
  Boff = offset to add to bond types
aoff value = Aoff
  Aoff = offset to add to angle types
doff value = Doff
  Doff = offset to add to dihedral types
ioff value = Ioff
  Ioff = offset to add to improper types
scale value = sfactor
  sfactor = scale factor to apply to the size, mass, and dipole of the molecule
check_labels value = string
  string = string containing any of the following characters: 'b', 'a', 'd', or 'i'
auto value = string
  string = string containing any of the following characters:
   'a', 'd', or 'i', which will autogenerate angles, dihedrals, and
   impropers, respectively
```

## 1.66.2 Examples

```
molecule 1 mymol.txt
molecule water tip3p.json
molecule 1 co2.txt h2o.txt
molecule CO2 co2.txt boff 3 aoff 2
molecule 1 mymol.txt offset 6 9 18 23 14
molecule objects file.1 scale 1.5 file.1 scale 2.0 file.2 scale 1.3
molecule 1 mymol.txt auto ad
```

## 1.66.3 Description

Define a molecule template that can be used as part of other LAMMPS commands, typically to define a collection of particles as a bonded molecule or a rigid body. Commands that currently use molecule templates include:

- *fix deposit*

- *fix pour*
- *fix rigid/small*
- *fix shake*
- *fix gcmc*
- *fix bond/react*
- *create_atoms*
- *atom_style template*

Changed in version 4Jul2026.

It can also be used to define a collection of line segments (2d) or triangles (3d) which define an object's surface or a boundary condition for granular particles to interact with, via these commands:

- *fix surface/global*
- *fix surface/local*

See the *Howto granular surfaces* doc page for more details on these kinds of models.

The ID of a molecule template can only contain alphanumeric characters and underscores, same as other IDs in LAMMPS.

A single template can contain multiple molecules, listed one per file. Some of the commands listed above currently use only the first molecule in the template, and will issue a warning if the template contains multiple molecules. The *atom_style template* command allows multiple-molecule templates to define a system with more than one templated molecule.

The molecule file can be either in a *native* format or in JSON format. JSON format filenames **must** have the extension ".json". Files with any other name will be assumed to be in the "native" format. The details of the two formats are described below. When referencing multiple molecule files in a single *molecule* command, each of those files may be either format.

Each filename can be followed by optional keywords which are applied only to the molecule in the file as used in this template. This is to make it easy to use the same molecule file in different molecule templates or in different simulations. You can specify the same file multiple times with different optional keywords.

The *offset*, *toff*, *boff*, *aoff*, *doff*, *ioff* keywords add the specified offset values to the atom types, bond types, angle types, dihedral types, and/or improper types as they are read from the molecule file. E.g. if *toff* = 2, and the file uses atom types 1,2,3, then each created molecule will have atom types 3,4,5. For the *offset* keyword, all five offset values must be specified, but individual values will be ignored if the molecule template does not use that attribute (e.g. no bonds).

**Note**

Offsets are **ignored** on lines using type labels, as the type labels will determine the actual types directly depending on the current *labelmap* settings.

**Note**

For molecule files defining line segments or triangles, only the *toff* keyword is relevant; the other offset keywords are ignored

The *scale* keyword scales the size of the molecule. This can be useful for modeling polydisperse granular rigid bodies. The scale factor is applied to each of these properties in the molecule file, if they are defined: the individual particle coordinates (Coords or "coords" section), the individual mass of each particle (Masses or "masses" section), the individual diameters of each particle (Diameters or "diameters" section), the per-atom dipoles (Dipoles or "dipoles" section) the total mass of the molecule (header keyword = mass), the center-of-mass of the molecule (header keyword = com), and the moments of inertia of the molecule (header keyword = inertia). For line or tri molecules, the scale factor multiplies the line endpoints or triangle vertices.

Added in version 11Feb2026.

The *check_labels* keyword causes the molecule command to issue a warning if the type label of a bond, angle, dihedral, or improper defined in the molecule template is not consistent with the atom types of its constituent atoms. The *check_labels* value is a single string that should contain one or more of the characters 'b', 'a', 'd', and 'i', which correspond to bonds, angles, dihedrals, and impropers, respectively. For example, the keyword/value pair *check_labels badi* will check all the type labels of all bonded interactions, while *check_labels adi* will only check type labels for angles, dihedrals, and impropers but not for bonds. The *check_labels* keyword requires using a specific convention for *type label* for enabling commands in LAMMPS (like *fix bond/react*) to infer the types of bonded interactions. Bond, angle, dihedral, and improper type labels must contain their constituent atom types delimited by hyphens. That is, for a dihedral that formed by three atoms of type 'c2' and one atom of 'n', the dihedral type label must be 'c2-c2-c2-n' . If the constituent atoms do not have these atom types in the proper order, a warning will be generated when using the *check_labels* keyword. Certain symmetries are considered to account for equivalent ways of writing bonded interactions. Type labels for bonds, angles, and dihedrals are assumed to be equivalent to those written in reverse order. For example, an angle with type label 'c1-c2-n' is equivalent to 'n-c2-c1'. Symmetries for impropers are more complex and are described on the doc page for each improper style in the 'Symmetry convention' section.

**Note**

The molecule command can be used to define molecules with bonds, angles, dihedrals, impropers, and special bond lists of neighbors within a molecular topology, so that you can later add the molecules to your simulation, via one or more of the commands listed above. Since this topology-related information requires that suitable storage is reserved when LAMMPS creates the simulation box (e.g. when using the *create_box* command or the *read_data* command) suitable space has to be reserved at that step so you do not overflow those pre-allocated data structures when adding molecules later. Both the *create_box* command and the *read_data* command have "extra" options which ensure extra space is allocated for storing topology info for molecules that are added later. This feature is *not* available for the *read_restart command*, thus binary restart files need to be converted to data files first.

Added in version 30Mar2026.

The *auto* keyword allows the molecule command to generate new angles, dihedrals, and/or impropers, and assign their angle types, dihedral types, and/or improper types. New interactions are discovered by traversing the bond graph defined in the *Bonds* section, and new types are inferred using *type label*. Type labels for angle, dihedral, and improper types must already be defined (e.g., by the read_data command), to use the *auto* keyword, and each of the labels must be defined as a list with its constituent atom type labels separated by hyphens, as described for the *check_labels* keyword. The *auto* value is a single string that should contain one or more of the characters 'a', 'd', and 'i', which correspond to angles, dihedrals, and impropers, respectively. For example, the keyword/value pair *auto adi* will generate angle, dihedral, and improper information, while *auto di* will only generate information for dihedrals and impropers but not for angles. Angles are generated from all unique 1-2-3 paths through the bond graph. Dihedrals are generated from all unique 1-2-3-4 paths through the bond graph, as long as there are no duplicate atoms in the dihedral. Impropers are

generated from all atoms bonded to exactly three neighbors. The type assigned to each generated 2-, 3- and 4-body interaction is found by searching the list of type labels for a match, e.g., 'c1-c2-c3' or 'c3-c2-c1' in the case of a 3-atom angle. If a matching type cannot be found, LAMMPS will generate an error.

**Note**

This command requires *Special Bonds* data to exist, which are generated automatically by default.

### 1.66.4 Format of a native molecule file

The format of an "native" individual molecule file looks similar but is *different* from that of a data file read by the *read_data* commands. Here is a simple example for a TIP3P water molecule:

```
# Water molecule. TIP3P geometry
# header section:
3 atoms
2 bonds
1 angles

# body section:
Coords

1    0.00000  -0.06556   0.00000
2    0.75695   0.52032   0.00000
3   -0.75695   0.52032   0.00000

Types

1        1   # O
2        2   # H
3        2   # H

Charges

1       -0.834
2        0.417
3        0.417

Bonds

1   1      1      2
2   1      1      3

Angles

1   1      2      1      3
```

A molecule file has a header and a body. The header appears first. The first line of the header and thus of the molecule file is *always* skipped; it typically contains a description of the file or a comment from the software that created the file.

Then lines are read one line at a time. Lines can have a trailing comment starting with '#' that is ignored. There *must* be at least one blank between any valid content and the comment. If the line is blank (i.e. contains only white-space

after comments are deleted), it is skipped. If the line contains a header keyword, the corresponding value(s) is/are read from the line. A line that is *not* blank and does *not* contains a header keyword begins the body of the file.

The body of the file contains zero or more sections. The first line of a section has only a keyword. The next line is skipped. The remaining lines of the section contain values. The number of lines depends on the section keyword as described below. Zero or more blank lines can be used between sections. Sections can appear in any order, with a few exceptions as noted below.

Added in version 4Jul2026: New header keywords *lines* and *tris*

These are the recognized header keywords. Header lines can come in any order. The numeric value(s) are read from the beginning of the line. The keyword should appear at the end of the line. All these settings have default values, as explained below. A line need only appear if the value(s) are different than the default, except when defining a *body* particle, which requires setting the number of *atoms* to 1, and setting the *inertia* in a specific section (see below).

| Number(s) | Keyword | Meaning | Default Value |
|---|---|---|---|
| N | atoms | # of atoms N in molecule | keyword is *required* |
| Nb | bonds | # of bonds Nb in molecule | 0 |
| Na | angles | # of angles Na in molecule | 0 |
| Nd | dihedrals | # of dihedrals Nd in molecule | 0 |
| Ni | impropers | # of impropers Ni in molecule | 0 |
| Nf | fragments | # of fragments Nf in molecule | 0 |
| Ninteger Ndouble | body | # of integer and floating-point values in *body particle* | 0 0 |
| Mtotal | mass | total mass of molecule | computed |
| Xc Yc Zc | com | coordinates of center-of-mass of molecule | computed |
| Ixx Iyy Izz Ixy Ixz Iyz | inertia | 6 components of inertia tensor of molecule | computed |
| Nlines | lines | # of lines Nlines in molecule | 0 |
| Ntris | triangles | # of triangle Ntris in molecule | 0 |

A molecule file can contain only either the *atoms*, *lines*, or *triangles* keyword. All the other keywords can only be used for a molecule file containing the *atoms* keyword.

For *mass*, *com*, and *inertia*, the default is for LAMMPS to calculate this quantity itself if needed, assuming the molecules consist of a set of point particles or finite-size particles (with a non-zero diameter) that do **not** overlap. If finite-size particles in the molecule **do** overlap, LAMMPS will not account for the overlap effects when calculating any of these 3 quantities, so you should pre-compute them yourself and list the values in the file.

The mass and center-of-mass coordinates (Xc,Yc,Zc) are self-explanatory. The 6 moments of inertia (ixx,iyy,izz,ixy,ixz,iyz) should be the values consistent with the current orientation of the rigid body around its center of mass. The values are with respect to the simulation box XYZ axes, not with respect to the principal axes of the rigid body itself. LAMMPS performs the latter calculation internally.

These are the allowed section keywords for the body of the file.

- *Coords, Types, Molecules, Fragments, Charges, Diameters, Dipoles, Masses* = atom-property sections
- *Bonds, Angles, Dihedrals, Impropers* = molecular topology sections
- *Special Bond Counts, Special Bonds* = special neighbor info
- *Shake Flags, Shake Atoms, Shake Bond Types* = SHAKE info
- *Body Integers, Body Doubles* = body-property sections
- *Lines, Triangles* = corner points of lines or triangles

Similar to the corresponding keywords, a molecule file can only contain either a Coords, Lines, or Triangles section. All of the other sections can only be used in molecule files for atoms.

For the Types, Bonds, Angles, Dihedrals, and Impropers sections, each atom/bond/angle/etc type can be specified either as a number (numeric type) or as an alphanumeric type label. The latter is only allowed if type labels have been defined, either by the *labelmap* command or in data files read by the *read_data* command which have sections for Atom Type Labels, Bond Type Labels, Angle Type Labels, etc. See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used. When using type labels, any values specified as *offset* are ignored.

If a Bonds section is specified then the Special Bond Counts and Special Bonds sections can also be used, if desired, to explicitly list the 1-2, 1-3, 1-4 neighbors within the molecule topology (see details below). This is optional since if these sections are not included, LAMMPS will auto-generate this information. Note that LAMMPS uses this info to properly exclude or weight bonded pairwise interactions between bonded atoms. See the *special_bonds* command for more details. One reason to list the special bond info explicitly is for the *thermalized Drude oscillator model* which treats the bonds between nuclear cores and Drude electrons in a different manner.

**Note**

Whether a section is required depends on how the molecule template is used by other LAMMPS commands. For example, to add a molecule via the *fix deposit* command, the Coords and Types sections are required. To add a rigid body via the *fix pour* command, the Bonds (Angles, etc) sections are not required, since the molecule will be treated as a rigid body. Some sections are optional. For example, the *fix pour* command can be used to add "molecules" which are clusters of finite-size granular particles. If the Diameters section is not specified, each particle in the molecule will have a default diameter of 1.0. See the doc pages for LAMMPS commands that use molecule templates for more details.

Each section is listed below in alphabetic order. The format of each section is described including the number of lines it must contain and rules (if any) for whether it can appear in the data file. For per- atom sections, entries should be numbered from 1 to Natoms (where Natoms is the number of atoms in the template), indicating which atom (or bond, etc) the entry applies to. Per-atom sections need to include a setting for every atom, but the atoms can be listed in any order.

---

*Coords* section:

- one line per atom
- line syntax: ID x y z
- x,y,z = coordinate of atom

---

*Types* section:

- one line per atom
- line syntax: ID type
- type = atom type of atom (1-Natomtype, or type label)

---

*Molecules* section:

- one line per atom
- line syntax: ID molecule-ID
- molecule-ID = molecule ID of atom

---

*Fragments* section:

- one line per fragment
- line syntax: ID a b c d ...
- a,b,c,d,... = IDs of atoms in fragment

The ID of a fragment can only contain alphanumeric characters and underscores. The atom IDs should be values from 1 to Natoms, where Natoms = # of atoms in the molecule.

---

*Charges* section:

- one line per atom
- line syntax: ID q
- q = charge on atom

This section is only allowed for *atom styles* that support charge. If this section is not included, the default charge on each atom in the molecule is 0.0.

---

*Diameters* section:

- one line per atom
- line syntax: ID diam
- diam = diameter of atom

This section is only allowed for *atom styles* that support finite-size spherical particles, e.g. atom_style sphere. If not listed, the default diameter of each atom in the molecule is 1.0.

---

Added in version 7Feb2024.

*Dipoles* section:

- one line per atom
- line syntax: ID mux muy muz
- mux,muy,muz = x-, y-, and z-component of point dipole vector of atom

This section is only allowed for *atom styles* that support particles with point dipoles, e.g. atom_style dipole. If not listed, the default dipole component of each atom in the molecule is set to 0.0.

---

*Masses* section:

- one line per atom
- line syntax: ID mass
- mass = mass of atom

This section is only allowed for *atom styles* that support per-atom mass, as opposed to per-type mass. See the *mass* command for details. If this section is not included, the default mass for each atom is derived from its volume (see Diameters section) and a default density of 1.0, in *units* of mass/volume.

---

*Bonds* section:

- one line per bond
- line syntax: ID type atom1 atom2
- type = bond type (1-Nbondtype, or type label)
- atom1,atom2 = IDs of atoms in bond

The IDs for the two atoms in each bond should be values from 1 to Natoms, where Natoms = # of atoms in the molecule.

---

*Angles* section:

- one line per angle
- line syntax: ID type atom1 atom2 atom3
- type = angle type (1-Nangletype, or type label)
- atom1,atom2,atom3 = IDs of atoms in angle

The IDs for the three atoms in each angle should be values from 1 to Natoms, where Natoms = # of atoms in the molecule. The three atoms are ordered linearly within the angle. Thus the central atom (around which the angle is computed) is the atom2 in the list.

---

*Dihedrals* section:

- one line per dihedral
- line syntax: ID type atom1 atom2 atom3 atom4
- type = dihedral type (1-Ndihedraltype, or type label)
- atom1,atom2,atom3,atom4 = IDs of atoms in dihedral

The IDs for the four atoms in each dihedral should be values from 1 to Natoms, where Natoms = # of atoms in the molecule. The 4 atoms are ordered linearly within the dihedral.

---

*Impropers* section:

- one line per improper
- line syntax: ID type atom1 atom2 atom3 atom4
- type = improper type (1-Nimpropertype, or type label)
- atom1,atom2,atom3,atom4 = IDs of atoms in improper

The IDs for the four atoms in each improper should be values from 1 to Natoms, where Natoms = # of atoms in the molecule. The ordering of the 4 atoms determines the definition of the improper angle used in the formula for the defined *improper style*. See the doc pages for individual styles for details.

---

*Special Bond Counts* section:

- one line per atom
- line syntax: ID N1 N2 N3
- N1 = # of 1-2 bonds
- N2 = # of 1-3 bonds

- N3 = # of 1-4 bonds

N1, N2, N3 are the number of 1-2, 1-3, 1-4 neighbors respectively of this atom within the topology of the molecule. See the *special_bonds* page for more discussion of 1-2, 1-3, 1-4 neighbors. If this section appears, the Special Bonds section must also appear.

As explained above, LAMMPS will auto-generate this information if this section is not specified. If specified, this section will override what would be auto-generated.

---

*Special Bonds* section:

- one line per atom
- line syntax: ID a b c d ...
- a,b,c,d,... = IDs of atoms in N1+N2+N3 special bonds

A, b, c, d, etc are the IDs of the n1+n2+n3 atoms that are 1-2, 1-3, 1-4 neighbors of this atom. The IDs should be values from 1 to Natoms, where Natoms = # of atoms in the molecule. The first N1 values should be the 1-2 neighbors, the next N2 should be the 1-3 neighbors, the last N3 should be the 1-4 neighbors. No atom ID should appear more than once. See the *special_bonds* doc page for more discussion of 1-2, 1-3, 1-4 neighbors. If this section appears, the Special Bond Counts section must also appear.

As explained above, LAMMPS will auto-generate this information if this section is not specified. If specified, this section will override what would be auto-generated.

---

*Shake Flags* section:

- one line per atom
- line syntax: ID flag
- flag = 0,1,2,3,4

This section is only needed when molecules created using the template will be constrained by SHAKE via the "fix shake" command. The other two Shake sections must also appear in the file, following this one.

The meaning of the flag for each atom is as follows. See the *fix shake* page for a further description of SHAKE clusters.

- 0 = not part of a SHAKE cluster
- 1 = part of a SHAKE angle cluster (two bonds and the angle they form)
- 2 = part of a 2-atom SHAKE cluster with a single bond
- 3 = part of a 3-atom SHAKE cluster with two bonds
- 4 = part of a 4-atom SHAKE cluster with three bonds

---

*Shake Atoms* section:

- one line per atom
- line syntax: ID a b c d
- a,b,c,d = IDs of atoms in cluster

This section is only needed when molecules created using the template will be constrained by SHAKE via the "fix shake" command. The other two Shake sections must also appear in the file.

The a,b,c,d values are atom IDs (from 1 to Natoms) for all the atoms in the SHAKE cluster that this atom belongs to. The number of values that must appear is determined by the shake flag for the atom (see the Shake Flags section above). All atoms in a particular cluster should list their a,b,c,d values identically.

If flag = 0, no a,b,c,d values are listed on the line, just the (ignored) ID.

If flag = 1, a,b,c are listed, where a = ID of central atom in the angle, and b,c the other two atoms in the angle.

If flag = 2, a,b are listed, where a = ID of atom in bond with the lowest ID, and b = ID of atom in bond with the highest ID.

If flag = 3, a,b,c are listed, where a = ID of central atom, and b,c = IDs of other two atoms bonded to the central atom.

If flag = 4, a,b,c,d are listed, where a = ID of central atom, and b,c,d = IDs of other three atoms bonded to the central atom.

See the *fix shake* page for a further description of SHAKE clusters.

---

*Shake Bond Types* section:

- one line per atom
- line syntax: ID a b c
- a,b,c = bond types (or angle type) of bonds (or angle) in cluster

This section is only needed when molecules created using the template will be constrained by SHAKE via the "fix shake" command. The other two Shake sections must also appear in the file.

The a,b,c values are bond types for all bonds in the SHAKE cluster that this atom belongs to. Bond types may be either numbers (from 1 to Nbondtypes) or bond type labels as defined by the *labelmap* command or a "Bond Type Labels" section of a data file.

The number of values that must appear is determined by the shake flag for the atom (see the Shake Flags section above). All atoms in a particular cluster should list their a,b,c values identically.

If flag = 0, no a,b,c values are listed on the line, just the (ignored) ID.

If flag = 1, a,b,c are listed, where a = bondtype of the bond between the central atom and the first non-central atom (value b in the Shake Atoms section), b = bondtype of the bond between the central atom and the second non-central atom (value c in the Shake Atoms section), and c = the angle type (1 to Nangletypes, or angle type label) of the angle between the three atoms.

If flag = 2, only a is listed, where a = bondtype of the bond between the two atoms in the cluster.

If flag = 3, a,b are listed, where a = bondtype of the bond between the central atom and the first non-central atom (value b in the Shake Atoms section), and b = bondtype of the bond between the central atom and the second non-central atom (value c in the Shake Atoms section).

If flag = 4, a,b,c are listed, where a = bondtype of the bond between the central atom and the first non-central atom (value b in the Shake Atoms section), b = bondtype of the bond between the central atom and the second non-central atom (value c in the Shake Atoms section), and c = bondtype of the bond between the central atom and the third non-central atom (value d in the Shake Atoms section).

See the *fix shake* page for a further description of SHAKE clusters.

---

*Body Integers* section:

- one line

- line syntax: N E F
- N = number of sub-particles or number of vertices
- E,F = number of edges and faces

This section is only needed when the molecule is a body particle. the other Body section must also appear in the file.

The total number of values that must appear is determined by the body style, and must be equal to the Ninteger value given in the *body* header.

For *nparticle* and *rounded/polygon*, only the number of sub-particles or vertices N is required, and Ninteger should have a value of 1.

For *rounded/polyhedron*, the number of edges E and faces F is required, and Ninteger should have a value of 3.

See the *Howto body* page for a further description of the file format.

---

*Body Doubles* section:

- first line
- line syntax: Ixx Iyy Izz Ixy Ixz Iyz
- Ixx Iyy Izz Ixy Ixz Iyz = 6 components of inertia tensor of body particle
- one line per sub-particle or vertex
- line syntax: x y z
- x, y, z = coordinates of sub-particle or vertex
- one line per edge
- line syntax: N1 N2
- N1, N2 = vertex indices
- one line per face
- line syntax: N1 N2 N3 N4
- N1, N2, N3, N4 = vertex indices
- last line
- line syntax: diam
- diam = rounded diameter that surrounds each vertex

This section is only needed when the molecule is a body particle. the other Body section must also appear in the file.

The total number of values that must appear is determined by the body style, and must be equal to the Ndouble value given in the *body* header. The 6 moments of inertia and the 3N coordinates of the sub-particles or vertices are required for all body styles.

For *rounded/polygon*, the E = 6 + 3*N + 1 edges are automatically determined from the vertices.

For *rounded/polyhedron*, the 2E vertex indices for the end points of the edges and 4F vertex indices defining the faces are required.

See the *Howto body* page for a further description of the file format.

---

Added in version 4Jul2026.

*Lines* section:

- one line per line segment
- line syntax: ID molecule-ID type x1 y1 x2 y2
- molecule-ID = molecule-ID of the line segment
- type = type of the line segment
- x1,y1,x2,y2 = coords of two endpoints of line segment

Each line segment is assigned a molecule-ID and type, similar to an atom type. This allows a collection of lines to represent multiple 2d bodies, e.g. a collection of squares. Line-segments in each could have a different molecule-ID, or the left-facing edges could have a unique atom type.

The coords of two different line segments which are connected by a common point should list the exact same coordinates for the common point. This allows a command like *fix surface/local* to infer connectivity of the two line segments.

The ordering of the two points defines the direction of an outward normal for the line segment. This is defined by a right-hand rule. The outward normal N = (0,0,1) x (p2-p1), where p1 and p2 are the 2 points. In other words, a unit z-direction vector is crossed into the vector from p1 to p2 to determine the normal.

It depends on how the line segments are used by other commands in LAMMPS whether the normal direction matters or not.

---

Added in version 4Jul2026.

*Triangles* section:

- one line per triangle
- line syntax: ID molecule-ID type x1 y1 z1 x2 y2 z2 x3 y3 z3
- molecule-ID = molecule-ID assigned triangle
- type = type assigned to triangle
- x1,y1,z1,x2,y2,z2,x3,y3,z3 = coords of three corner points of triangle

Each triangle is assigned a molecule-ID and type, similar to an atom type. This allows a collection of triangles to represent multiple 3d objects, e.g. a curved surface on the left and right of the simulation box. Triangles in each could have a different molecule-ID, or the the triangles in the upper half could have a different type than those in the lower half.

The coords of two different triangles which share a common edge (2 points) or corner point (single point) should list the exact same coordinates for the common points. This allows a command like *fix surface/local* to infer connectivity of the two triangles.

The ordering of the three points defines the direction of the outward normal for the triangle. This is defined by a right-hand rule. The outward normal N = (p2-p1) x (p3-p1), where p1, p2, p3 are the 3 points. In other words, the edge from p1 to p2 is crossed into the edge from p1 to p3 to determine the normal.

It depends on how the triangles are used by other commands in LAMMPS whether the normal direction matters or not.

### 1.66.5 Format of a JSON molecule file

Added in version 12Jun2025.

The format of a JSON format individual molecule file must follow the JSON format, which evolved from the JavaScript programming language as a programming-language-neutral data interchange language. The JSON syntax is independent of its content, and thus the data in the file must follow suitable conventions to be correctly processed. LAMMPS provides a JSON schema file for JSON format molecule files in the *tools/json folder* to represent those conventions. Using the schema file any JSON format molecule files can be validated. Please note that the format requirement for JSON are very strict and the JSON reader in LAMMPS does not accept files with extensions like comments. Validating

a particular JSON format molecule file against this schema ensures that both, the JSON syntax requirement *and* the LAMMPS conventions for molecule template files are followed. LAMMPS should be able to read and parse any JSON file that passes the schema check. This is a formal check only and thus it **cannot** check whether the file contents are consistent or physically meaningful.

Here is a simple example for the same TIP3P water molecule from above in JSON format and also using *type labels* instead of numeric types:

```
{
    "application": "LAMMPS",
    "format": "molecule",
    "revision": 1,
    "title": "Water molecule. TIP3P geometry",
    "schema": "https://download.lammps.org/json/molecule-schema.json",
    "units": "real",
    "coords": {
        "format": ["atom-id", "x", "y", "z"],
        "data": [
            [1,  0.00000, -0.06556,  0.00000],
            [2,  0.75695,  0.52032,  0.00000],
            [3, -0.75695,  0.52032,  0.00000]
        ]
    },
    "types": {
        "format": ["atom-id", "type"],
        "data": [
            [1,  "OW"],
            [2,  "HO1"],
            [3,  "HO1"]
        ]
    },
    "charges": {
        "format": ["atom-id", "charge"],
        "data": [
            [1, -0.834],
            [2,  0.417],
            [3,  0.417]
        ]
    },
    "bonds": {
        "format": ["bond-type", "atom1", "atom2"],
        "data": [
            ["OW-HO1",  1,  2],
            ["OW-HO1",  1,  3]
        ]
    },
    "angles": {
        "format": ["angle-type", "atom1", "atom2", "atom3"],
        "data": [
            ["HO1-OW-HO1",  2,  1,  3]
        ]
    }
}
```

Unlike with the native molecule file format, there are no header or body sections, just a list of keywords with associated

data. JSON format data is read, parsed, and stored in an internal dictionary data structure in one step and thus the order of keywords is not relevant.

Data for keywords is either provided directly following the keyword or as a *data block*. A *data block* is a list that has to include two keywords, "format" and "data", where the former lists keywords of the properties that are stored in the columns of the "data" lists. The names and order of entries in the "format" list (and thus how the data is interpreted) are currently fixed.

Since the length of the various lists can be easily obtained from the internal data structure, several header keywords of the "native" molecule file are not needed. On the other hand, some additional keywords are required to identify the conventions applied to the generic JSON file format. The structure of the data itself mostly follows what is used for the "native" molecule file format.

| Keyword | Argument(s) | Required | Description |
|---|---|---|---|
| application | "LAMMPS | yes | indicates a LAMMPS JSON file; files from other applications may be accepted in the future |
| format | "molecule" | yes | indicates a molecule template file |
| revision | an integer | yes | currently 1, to facility backward compatibility on changes to the conventions |
| title | a string | no | information about the template which will echoed to the screen and log |
| schema | URL as string | no | location of a JSON schema file for validating the molecule file format |
| units | a string | no | indicates *units settings* for this molecule template |
| com | list with 3 doubles | no | overrides the auto-computed center-of-mass for the template |
| masstotal | double | no | overrides the auto-computed total mass for the template |
| inertia | list with 6 doubles | no | overrides the auto-computed moments of inertia |
| coords | a data block | no | contains atom positions with the format "atom-id", "x", "y", "z" (same as Coords) |
| types | a data block | yes | assigns atom types to atoms with the format "atom-id", "type" (same as Types) |
| molecu | a data block | no | assigns molecule-IDs to atoms with the format "atom-id", "molecule-id" (same as Molecules) |
| fragments | a data block | no | assigns atom-ids to fragment-IDs with the format "fragment-id", "atom-id-list" (same as Fragments) |
| charges | a data block | no | assigns charges to atoms with the format "atom-id", "charge" (same as Charges) |
| dipoles | a data block | no | assigns point dipoles to atoms with the format "atom-id", "mux", "muy", "muz" (same as Dipoles) |
| diameters | a data block | no | assigns diameters to atoms with the format "atom-id", "diameter" (same as Diameters) |
| masses | a data block | no | assigns per-atom masses to atoms with the format "atom-id", "mass" (same as Masses) |
| bonds | a data block | no | defines bonds in the molecule template with the format "bond-type", "atom1", "atom2" (same as Bonds without bond-ID) |
| angles | a data block | no | defines angles in the molecule template with the format "angle-type", "atom1", "atom2", "atom3" (same as Angles without angle-ID) |
| dihedrals | a data block | no | defines dihedrals in the molecule template with the format "dihedral-type", "atom1", "atom2", "atom3", "atom4" (same as Dihedrals without dihedral-ID) |
| impropers | a data block | no | defines impropers in the molecule template with the format "improper-type", "atom1", "atom2", "atom3", "atom4" (same as Impropers without improper-ID) |
| shake | 3 JSON objects | no | contains the sub-sections "flags", "atoms", "bonds" described below |
| special | 2 JSON objects | no | contains the sub-sections "counts" and "bonds" described below |
| body | 2 JSON objects | no | contains the "integers" and "doubles" sub-sections with arrays with the same data as Body Integers and Body Doubles, respectively |

The following table describes the sub-sections for the "special" entry from above:

| Sub-section | Argument(s) | Required | Description |
|---|---|---|---|
| counts | a data block | yes | contains the counts of 1-2, 1-3, and 1-4 special neighbors with the format "atom-id", "n12", "n13", "n14" (same as Special Bond Counts) |
| bonds | a data block | yes | contains the lists of special neighbors to atoms with the format "atom-id", "atom-id-list" (same as Special Bonds) |

The following table describes the sub-sections for the "shake" entry from above:

| Sub-section | Argument(s) | Required | Description |
|---|---|---|---|
| flags | a data block | yes | contains the counts shake flags for atoms with the format "atom-id", "flag" (same as Shake Flags) |
| atoms | a data block | yes | contains the lists of shake cluster atom-ids for atoms with the format "atom-id", "atom-id-list" (same as Shake Atoms) |
| bonds | a data block | yes | contains the lists of shake bond or angle types for atoms with the format "atom-id", "type-list" (same as Shake Bonds) |

The "special" and "shake" sections are usually not needed, since the data can be auto-generated as soon as the simulation box is defined. Below is an example for what would have to be *added* to the example JSON file above in case the molecule command needs to be issued earlier.

```
"special": {
    "counts": {
        "format": ["atom-id", "n12", "n13", "n14"],
        "data": [
            [1, 2, 0, 0],
            [2, 1, 1, 0],
            [3, 1, 1, 0]
        ]
    },
    "bonds": {
        "format": ["atom-id", "atom-id-list"],
        "data": [
            [1,  [2, 3]],
            [2,  [1, 3]],
            [3,  [1, 2]]
        ]
    }
},
"shake": {
    "flags": {
        "format": ["atom-id", "flag"],
        "data": [
            [1, 1],
            [2, 1],
            [3, 1]
        ]
    },
```

(continues on next page)

(continued from previous page)

```
    "atoms": {
        "format": ["atom-id", "atom-id-list"],
        "data": [
            [1,  [1, 2, 3]],
            [2,  [1, 2, 3]],
            [3,  [1, 2, 3]]
        ]
    },
    "types": {
        "format": ["atom-id", "type-list"],
        "data": [
            [1,  ["OW-HO1",  "OW-HO1", "HO1-OW-HO1"]],
            [2,  ["OW-HO1",  "OW-HO1", "HO1-OW-HO1"]],
            [3,  ["OW-HO1",  "OW-HO1", "HO1-OW-HO1"]]
        ]
    }
}
```

Below is a minimal example of a JSON format molecule template for a body particle for *pair style body/nparticle*. Molecule templates for body particles must contain only one atom:

```
{
    "application": "LAMMPS",
    "format": "molecule",
    "revision": 1,
    "title": "Square body for body/nparticles",
    "schema": "https://download.lammps.org/json/molecule-schema.json",
    "units": "real",
    "coords": {
        "format": ["atom-id", "x", "y", "z"],
        "data": [
            [1,  0.00000,  0.00000,  0.00000]
        ]
    },
    "types": {
        "format": ["atom-id", "type"],
        "data": [
            [1,  1]
        ]
    },
    "masses": {
        "format": ["atom-id", "mass"],
        "data": [
            [1,  1.0]
        ]
    },
    "body": {
        "integers": [4],
        "doubles": [
            1.0, 1.0, 4.0, 0.0, 0.0, 0.0,
            -0.70710678118654752440, -0.70710678118654752440, 0.0,
            -0.70710678118654752440,  0.70710678118654752440, 0.0,
```

(continues on next page)

(continued from previous page)

```
            0.7071067811865475244,  0.7071067811865475244, 0.0,
            0.7071067811865475244, -0.7071067811865475244, 0.0
        ]
    }
}
```

### 1.66.6 Restrictions

The *lines* and *tris* keywords and corresponding sections are currently not (yet) supported with molecule files in JSON format.

### 1.66.7 Related commands

*write_molecule*, *fix deposit*, *fix pour*, *fix gcmc*, *fix bond/react*, *create_atoms*

### 1.66.8 Default

The default keywords values are offset 0 0 0 0 0 and scale = 1.0.

## 1.67 neb command

### 1.67.1 Syntax

```
neb etol ftol N1 N2 Nevery file-style arg keyword values
```

- etol = stopping tolerance for energy (dimensionless)
- ftol = stopping tolerance for force (force units)
- N1 = max # of iterations (timesteps) to run initial NEB
- N2 = max # of iterations (timesteps) to run barrier-climbing NEB
- Nevery = print replica energies and reaction coordinates every this many timesteps
- file-style = *final* or *each* or *none*

```
final arg = filename
  filename = file with initial coords for final replica
    coords for intermediate replicas are linearly interpolated
    between first and last replica
each arg = filename
  filename = unique filename for each replica (except first)
    with its initial coords
none arg = no argument all replicas assumed to already have
    their initial coords
```

- zero or more keyword/value pairs may be appended
- keyword = *verbosity*

```
verbosity value = verbose or default or terse
  verbose = very detailed per-replica output
  default = some per-replica output
  terse = only global state output
```

### 1.67.2 Examples

```
neb 0.1 0.0 1000 500 50 final coords.final
neb 0.0 0.001 1000 500 50 each coords.initial.$i
neb 0.0 0.001 1000 500 50 none verbose
```

### 1.67.3 Description

Perform a nudged elastic band (NEB) calculation using multiple replicas of a system. Two or more replicas must be used; the first and last are the end points of the transition path.

NEB is a method for finding both the atomic configurations and height of the energy barrier associated with a transition state, e.g. for an atom to perform a diffusive hop from one energy basin to another in a coordinated fashion with its neighbors. The implementation in LAMMPS follows the discussion in these 4 papers: *(HenkelmanA)*, *(HenkelmanB)*, *(Nakano)* and *(Maras)*.

Each replica runs on a partition of one or more processors. Processor partitions are defined at run-time using the *-partition command-line switch*. Note that if you have MPI installed, you can run a multi-replica simulation with more replicas (partitions) than you have physical processors, e.g you can run a 10-replica simulation on just one or two processors. You will simply not get the performance speed-up you would see with one or more physical processors per replica. See the *Howto replica* doc page for further discussion.

**Note**

As explained below, a NEB calculation performs a damped dynamics minimization across all the replicas. The minimizer uses whatever timestep you have defined in your input script, via the *timestep* command. Often NEB will converge more quickly if you use a timestep about 10x larger than you would normally use for dynamics simulations.

When a NEB calculation is performed, it is assumed that each replica is running the same system, though LAMMPS does not check for this. I.e. the simulation domain, the number of atoms, the interaction potentials, and the starting configuration when the neb command is issued should be the same for every replica.

In a NEB calculation each replica is connected to other replicas by inter-replica nudging forces. These forces are imposed by the *fix neb* command, which must be used in conjunction with the neb command. The group used to define the fix neb command defines the NEB atoms which are the only ones that inter-replica springs are applied to. If the group does not include all atoms, then non-NEB atoms have no inter-replica springs and the forces they feel and their motion is computed in the usual way due only to other atoms within their replica. Conceptually, the non-NEB atoms provide a background force field for the NEB atoms. They can be allowed to move during the NEB minimization procedure (which will typically induce different coordinates for non-NEB atoms in different replicas), or held fixed using other LAMMPS commands such as *fix setforce*. Note that the *partition* command can be used to invoke a command on a subset of the replicas, e.g. if you wish to hold NEB or non-NEB atoms fixed in only the end-point replicas.

The initial atomic configuration for each of the replicas can be specified in different manners via the *file-style* setting, as discussed below. Only atoms whose initial coordinates should differ from the current configuration need be specified.

Conceptually, the initial and final configurations for the first replica should be states on either side of an energy barrier.

As explained below, the initial configurations of intermediate replicas can be atomic coordinates interpolated in a linear fashion between the first and last replicas. This is often adequate for simple transitions. For more complex transitions, it may lead to slow convergence or even bad results if the minimum energy path (MEP, see below) of states over the barrier cannot be correctly converged to from such an initial path. In this case, you will want to generate initial states for the intermediate replicas that are geometrically closer to the MEP and read them in.

For a *file-style* setting of *final*, a filename is specified which contains atomic coordinates for zero or more atoms, in the format described below. For each atom that appears in the file, the new coordinates are assigned to that atom in the final replica. Each intermediate replica also assigns a new position to that atom in an interpolated manner. This is done by using the current position of the atom as the starting point and the read-in position as the final point. The distance between them is calculated, and the new position is assigned to be a fraction of the distance. E.g. if there are 10 replicas, the second replica will assign a position that is 10% of the distance along a line between the starting and final point, and the 9th replica will assign a position that is 90% of the distance along the line. Note that for this procedure to produce consistent coordinates across all the replicas, the current coordinates need to be the same in all replicas. LAMMPS does not check for this, but invalid initial configurations will likely result if it is not the case.

**Note**

The "distance" between the starting and final point is calculated in a minimum-image sense for a periodic simulation box. This means that if the two positions are on opposite sides of a box (periodic in that dimension), the distance between them will be small, because the periodic image of one of the atoms is close to the other. Similarly, even if the assigned position resulting from the interpolation is outside the periodic box, the atom will be wrapped back into the box when the NEB calculation begins.

For a *file-style* setting of *each*, a filename is specified which is assumed to be unique to each replica. This can be done by using a variable in the filename, e.g.

```
variable i equal part
neb 0.0 0.001 1000 500 50 each coords.initial.$i
```

which in this case will substitute the partition ID (0 to N-1) for the variable I, which is also effectively the replica ID. See the *variable* command for other options, such as using world-, universe-, or uloop-style variables.

Each replica (except the first replica) will read its file, formatted as described below, and for any atom that appears in the file, assign the specified coordinates to its atom. The various files do not need to contain the same set of atoms.

For a *file-style* setting of *none*, no filename is specified. Each replica is assumed to already be in its initial configuration at the time the neb command is issued. This allows each replica to define its own configuration by reading a replica-specific data or restart or dump file, via the *read_data*, *read_restart*, or *read_dump* commands. The replica-specific names of these files can be specified as in the discussion above for the *each* file-style. Also see the section below for how a NEB calculation can produce restart files, so that a long calculation can be restarted if needed.

**Note**

None of the *file-style* settings change the initial configuration of any atom in the first replica. The first replica must thus be in the correct initial configuration at the time the neb command is issued.

---

A NEB calculation proceeds in two stages, each of which is a minimization procedure, performed via damped dynamics. To enable this, you must first define a damped dynamics *min_style*, such as *quickmin* or *fire*. The *cg*, *sd*, and *hftn* styles cannot be used, since they perform iterative line searches in their inner loop, which cannot be easily synchronized across multiple replicas.

The minimizer tolerances for energy and force are set by *etol* and *ftol*, the same as for the *minimize* command.

A non-zero *etol* means that the NEB calculation will terminate if the energy criterion is met by every replica. The energies being compared to *etol* do not include any contribution from the inter-replica nudging forces, since these are non-conservative. A non-zero *ftol* means that the NEB calculation will terminate if the force criterion is met by every replica. The forces being compared to *ftol* include the inter-replica nudging forces.

The maximum number of iterations in each stage is set by *N1* and *N2*. These are effectively timestep counts since each iteration of damped dynamics is like a single timestep in a dynamics *run*. During both stages, the potential energy of each replica and its normalized distance along the reaction path (reaction coordinate RD) will be printed to the screen and log file every *Nevery* timesteps. The RD is 0 and 1 for the first and last replica. For intermediate replicas, it is the cumulative distance (normalized by the total cumulative distance) between adjacent replicas, where "distance" is defined as the length of the 3N-vector of differences in atomic coordinates, where N is the number of NEB atoms involved in the transition. These outputs allow you to monitor NEB's progress in finding a good energy barrier. *N1* and *N2* must both be multiples of *Nevery*.

In the first stage of NEB, the set of replicas should converge toward a minimum energy path (MEP) of conformational states that transition over a barrier. The MEP for a transition is defined as a sequence of 3N-dimensional states, each of which has a potential energy gradient parallel to the MEP itself. The configuration of highest energy along a MEP corresponds to a saddle point. The replica states will also be roughly equally spaced along the MEP due to the inter-replica nudging force added by the *fix neb* command.

In the second stage of NEB, the replica with the highest energy is selected and the inter-replica forces on it are converted to a force that drives its atom coordinates to the top or saddle point of the barrier, via the barrier-climbing calculation described in *(HenkelmanB)*. As before, the other replicas rearrange themselves along the MEP so as to be roughly equally spaced.

When both stages are complete, if the NEB calculation was successful, the configurations of the replicas should be along (close to) the MEP and the replica with the highest energy should be an atomic configuration at (close to) the saddle point of the transition. The potential energies for the set of replicas represents the energy profile of the transition along the MEP.

---

A few other settings in your input script are required or advised to perform a NEB calculation. See the NOTE about the choice of timestep at the beginning of this doc page.

An atom map must be defined which it is not by default for *atom_style atomic* problems. The *atom_modify map* command can be used to do this.

The minimizers in LAMMPS operate on all atoms in your system, even non-NEB atoms, as defined above. To prevent non-NEB atoms from moving during the minimization, you should use the *fix setforce* command to set the force on each of those atoms to 0.0. This is not required, and may not even be desired in some cases, but if those atoms move too far (e.g. because the initial state of your system was not well-minimized), it can cause problems for the NEB procedure.

The damped dynamics *minimizers*, such as *quickmin* and *fire*), adjust the position and velocity of the atoms via an Euler integration step. Thus you must define an appropriate *timestep* to use with NEB. As mentioned above, NEB will often converge more quickly if you use a timestep about 10x larger than you would normally use for dynamics simulations.

---

Each file read by the neb command containing atomic coordinates used to initialize one or more replicas must be formatted as follows.

The file can be ASCII text or a compressed text file (detected by a *supported compression format suffix*). The file can contain initial blank lines or comment lines starting with "#" which are ignored. The first non-blank, non-comment line should list N = the number of lines to follow. The N successive lines contain the following information:

```
ID1 x1 y1 z1
ID2 x2 y2 z2
...
IDN xN yN zN
```

The fields are the atom ID, followed by the x,y,z coordinates. The lines can be listed in any order. Additional trailing information on the line is OK, such as a comment.

Note that for a typical NEB calculation you do not need to specify initial coordinates for very many atoms to produce differing starting and final replicas whose intermediate replicas will converge to the energy barrier. Typically only new coordinates for atoms geometrically near the barrier need be specified.

Also note there is no requirement that the atoms in the file correspond to the NEB atoms in the group defined by the *fix neb* command. Not every NEB atom need be in the file, and non-NEB atoms can be listed in the file.

---

Four kinds of output can be generated during a NEB calculation: energy barrier statistics, thermodynamic output by each replica, dump files, and restart files.

When running with multiple partitions (each of which is a replica in this case), the print-out to the screen and master log.lammps file contains a line of output, printed once every *Nevery* timesteps. The amount of information printed in this line can be selected with the *verbosity* keyword. Available options are *terse*, *default*, and *verbose*.

With the *terse* setting, it contains the timestep, the maximum force of a replica, the maximum force per atom (in any replica), potential gradients in the initial, final, and climbing replicas, the forward and backward energy barriers, the total reaction coordinate (RDT).

With the *default* setting, additionally the normalized reaction coordinate and potential energy of each replica are printed.

With the *verbose* setting, additional per-replica properties are printed: the "path angle" (pathangle), the angle between the 3N-length tangent vector and the 3N-length force vector at image *i* (angletangrad), the angle between the 3N-length energy gradient vector of replica *i* and that of replica *i*+1 (anglegrad), the norm of the energy gradient (gradV), the the two-norm of the 3N-length force vector (RepForce), and the maximum force component of any atom (MaxAtomForce).

The "maximum force per replica" is the two-norm of the 3N-length force vector for the atoms in each replica, maximized across replicas, which is what the *ftol* setting is checking against. In this case, N is all the atoms in each replica. The "maximum force per atom" is the maximum force component of any atom in any replica. The potential gradients are the two-norm of the 3N-length force vector solely due to the interaction potential i.e. without adding in inter-replica forces.

The "reaction coordinate" (RD) for each replica is the two-norm of the 3N-length vector of distances between its atoms and the preceding replica's atoms, added to the RD of the preceding replica. The RD of the first replica RD1 = 0.0; the RD of the final replica RDN = RDT, the total reaction coordinate. The normalized RDs are divided by RDT, so that they form a monotonically increasing sequence from zero to one. When computing RD, N only includes the atoms being operated on by the fix neb command.

The forward (reverse) energy barrier is the potential energy of the highest replica minus the energy of the first (last) replica.

The "path angle" (pathangle) for the replica i which is the angle between the 3N-length vectors $(R_{i-1}-R_i)$ and $(R_{i+1}-R_i)$ (where $R_i$ is the atomic coordinates of replica *i*). A "path angle" of 180 indicates that replicas *i*-1, *i* and *i*+1 are aligned. "angletangrad" is the angle between the 3N-length tangent vector and the 3N-length force vector at image *i*. The tangent vector is calculated as in *(HenkelmanA)* for all intermediate replicas and at R2 - R1 and RM - RM-1 for the first and last replica, respectively. "anglegrad" is the angle between the 3N-length energy gradient vector of replica *i* and that of replica *i*+1. It is not defined for the final replica and reads nan. gradV is the norm of the energy gradient of image *i* ($\nabla V$). ReplicaForce is the two-norm of the 3N-length force vector (including nudging forces) for replica *i*. MaxAtomForce is the maximum force component of any atom in replica *i*.

When a NEB calculation does not converge properly, the supplementary information can help understanding what is going wrong. For instance when the path angle becomes acute, the definition of tangent used in the NEB calculation is questionable and the NEB cannot may diverge *(Maras)*.

When running on multiple partitions, LAMMPS produces additional log files for each partition, e.g. log.lammps.0, log.lammps.1, etc. For a NEB calculation, these contain the thermodynamic output for each replica.

If *dump* commands in the input script define a filename that includes a *universe* or *uloop* style *variable*, then one dump file (per dump command) will be created for each replica. At the end of the NEB calculation, the final snapshot in each

file will contain the sequence of snapshots that transition the system over the energy barrier. Earlier snapshots will show the convergence of the replicas to the MEP.

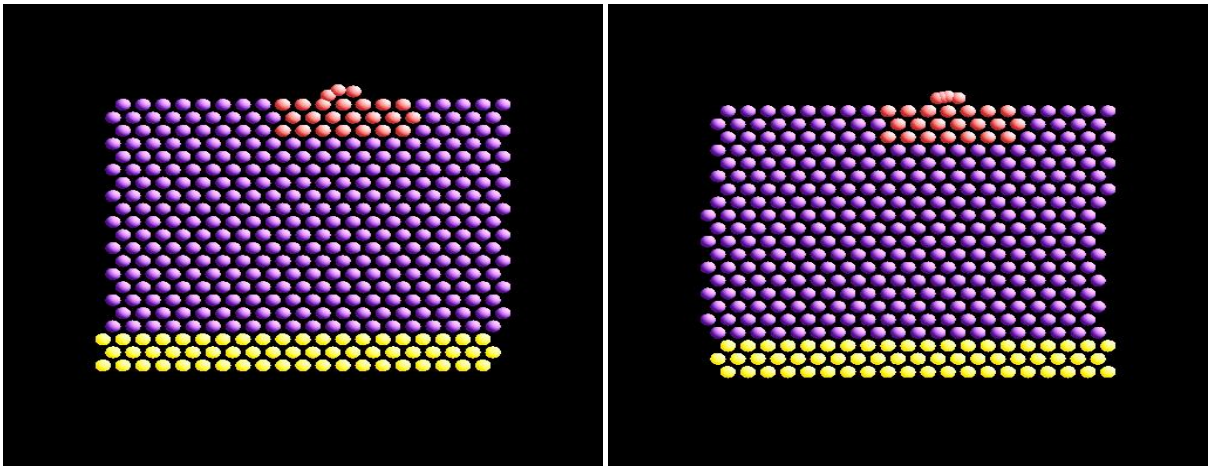
Likewise, *restart* filenames can be specified with a *universe* or *uloop* style *variable*, to generate restart files for each replica. These may be useful if the NEB calculation fails to converge properly to the MEP, and you wish to restart the calculation from an intermediate point with altered parameters.

There are 2 Python scripts provided in the tools/python directory, neb_combine.py and neb_final.py, which are useful in analyzing output from a NEB calculation. Assume a NEB simulation with M replicas, and the NEB atoms labeled with a specific atom type.

The neb_combine.py script extracts atom coords for the NEB atoms from all M dump files and creates a single dump file where each snapshot contains the NEB atoms from all the replicas and one copy of non-NEB atoms from the first replica (presumed to be identical in other replicas). This can be visualized/animated to see how the NEB atoms relax as the NEB calculation proceeds.

The neb_final.py script extracts the final snapshot from each of the M dump files to create a single dump file with M snapshots. This can be visualized to watch the system make its transition over the energy barrier.

To illustrate, here are images from the final snapshot produced by the neb_combine.py script run on the dump files produced by the two example input scripts in examples/neb.



## 1.67.4 Restrictions

This command can only be used if LAMMPS was built with the REPLICA package. See the *Build package* doc page for more info.

To read compressed files, you must compile LAMMPS with the `-DLAMMPS_GZIP` option. See the *Build settings* doc page for details.

## 1.67.5 Related commands

*prd*, *temper*, *fix langevin*, *fix viscous*, *fix neb*

## 1.67.6 Default

*verbosity* = *default*

**(HenkelmanA)** Henkelman and Jonsson, J Chem Phys, 113, 9978-9985 (2000).

**(HenkelmanB)** Henkelman, Uberuaga, Jonsson, J Chem Phys, 113, 9901-9904 (2000).

**(Nakano)** Nakano, Comp Phys Comm, 178, 280-289 (2008).

**(Maras)** Maras, Trushin, Stukowski, Ala-Nissila, Jonsson, Comp Phys Comm, 205, 13-21 (2016)

# 1.68 neb/spin command

## 1.68.1 Syntax

```
neb/spin etol ttol N1 N2 Nevery file-style arg keyword
```

- etol = stopping tolerance for energy (energy units)
- ttol = stopping tolerance for torque ( units)
- N1 = max # of iterations (timesteps) to run initial NEB
- N2 = max # of iterations (timesteps) to run barrier-climbing NEB
- Nevery = print replica energies and reaction coordinates every this many timesteps
- file-style = *final* or *each* or *none*

```
final arg = filename
  filename = file with initial coords for final replica
    coords for intermediate replicas are linearly interpolated
    between first and last replica
each arg = filename
  filename = unique filename for each replica (except first)
    with its initial coords
none arg = no argument all replicas assumed to already have
    their initial coords
```

- keyword = *verbose*

```
verbose = print supplemental information
```

## 1.68.2 Examples

```
neb/spin 0.1 0.0 1000 500 50 final coords.final
neb/spin 0.0 0.001 1000 500 50 each coords.initial.$i
neb/spin 0.0 0.001 1000 500 50 none verbose
```

## 1.68.3 Description

Perform a geodesic nudged elastic band (GNEB) calculation using multiple replicas of a system. Two or more replicas must be used; the first and last are the end points of the transition path.

GNEB is a method for finding both the spin configurations and height of the energy barrier associated with a transition state, e.g. spins to perform a collective rotation from one energy basin to another. The implementation in LAMMPS follows the discussion in the following paper: *(Bessarab)*.

Each replica runs on a partition of one or more processors. Processor partitions are defined at run-time using the *-partition command-line switch*. Note that if you have MPI installed, you can run a multi-replica simulation with more replicas (partitions) than you have physical processors, e.g you can run a 10-replica simulation on just one or two

processors. You will simply not get the performance speed-up you would see with one or more physical processors per replica. See the *Howto replica* doc page for further discussion.

**Note**

As explained below, a GNEB calculation performs a minimization across all the replicas. One of the *spin* style minimizers has to be defined in your input script.

When a GNEB calculation is performed, it is assumed that each replica is running the same system, though LAMMPS does not check for this. I.e. the simulation domain, the number of magnetic atoms, the interaction potentials, and the starting configuration when the neb command is issued should be the same for every replica.

In a GNEB calculation each replica is connected to other replicas by inter-replica nudging forces. These forces are imposed by the *fix neb/spin* command, which must be used in conjunction with the neb command. The group used to define the fix neb/spin command defines the GNEB magnetic atoms which are the only ones that inter-replica springs are applied to. If the group does not include all magnetic atoms, then non-GNEB magnetic atoms have no inter-replica springs and the torques they feel and their precession motion is computed in the usual way due only to other magnetic atoms within their replica. Conceptually, the non-GNEB atoms provide a background force field for the GNEB atoms. Their magnetic spins can be allowed to evolve during the GNEB minimization procedure.

The initial spin configuration for each of the replicas can be specified in different manners via the *file-style* setting, as discussed below. Only atomic spins whose initial coordinates should differ from the current configuration need to be specified.

Conceptually, the initial and final configurations for the first replica should be states on either side of an energy barrier.

As explained below, the initial configurations of intermediate replicas can be spin coordinates interpolated in a linear fashion between the first and last replicas. This is often adequate for simple transitions. For more complex transitions, it may lead to slow convergence or even bad results if the minimum energy path (MEP, see below) of states over the barrier cannot be correctly converged to from such an initial path. In this case, you will want to generate initial states for the intermediate replicas that are geometrically closer to the MEP and read them in.

---

For a *file-style* setting of *final*, a filename is specified which contains atomic and spin coordinates for zero or more atoms, in the format described below. For each atom that appears in the file, the new coordinates are assigned to that atom in the final replica. Each intermediate replica also assigns a new spin to that atom in an interpolated manner. This is done by using the current direction of the spin at the starting point and the read-in direction as the final point. The "angular distance" between them is calculated, and the new direction is assigned to be a fraction of the angular distance.

**Note**

The "angular distance" between the starting and final point is evaluated in the geodesic sense, as described in *(Bessarab)*.

**Note**

The angular interpolation between the starting and final point is achieved using Rodrigues formula:

$$\vec{m}_i^{\,v} = \vec{m}_i^{\,I}\cos(\omega_i^v) + (\vec{k}_i \times \vec{m}_i^{\,I})\sin(\omega_i^v) + (1.0 - \cos(\omega_i^v))\vec{k}_i(\vec{k}_i \cdot \vec{m}_i^{\,I})$$

where $\vec{m}_i^I$ is the initial spin configuration for spin i, $\omega_i^\nu$ is a rotation angle defined as:

$$\omega_i^\nu = (\nu - 1)\Delta\omega_i \text{ and } \Delta\omega_i = \frac{\omega_i}{Q - 1}$$

with $\nu$ the image number, Q the total number of images, and $\omega_i$ the total rotation between the initial and final spins. $\vec{k}_i$ defines a rotation axis such as:

$$\vec{k}_i = \frac{\vec{m}_i^I \times \vec{m}_i^F}{|\vec{m}_i^I \times \vec{m}_i^F|}$$

if the initial and final spins are not aligned. If the initial and final spins are aligned, then their cross product is null, and the expression above does not apply. If they point toward the same direction, the intermediate images conserve the same orientation. If the initial and final spins are aligned, but point toward opposite directions, an arbitrary rotation vector belonging to the plane perpendicular to initial and final spins is chosen. In this case, a warning message is displayed.

For a *file-style* setting of *each*, a filename is specified which is assumed to be unique to each replica. See the *neb* documentation page for more information about this option.

For a *file-style* setting of *none*, no filename is specified. Each replica is assumed to already be in its initial configuration at the time the neb command is issued. This allows each replica to define its own configuration by reading a replica-specific data or restart or dump file, via the *read_data*, *read_restart*, or *read_dump* commands. The replica-specific names of these files can be specified as in the discussion above for the *each* file-style. Also see the section below for how a NEB calculation can produce restart files, so that a long calculation can be restarted if needed.

**Note**

None of the *file-style* settings change the initial configuration of any atom in the first replica. The first replica must thus be in the correct initial configuration at the time the neb command is issued.

---

A NEB calculation proceeds in two stages, each of which is a minimization procedure. To enable this, you must first define a *min_style*, using either the *spin*, *spin/cg*, or *spin/lbfgs* style (see *min_spin* for more information). The other styles cannot be used, since they relax the lattice degrees of freedom instead of the spins.

The minimizer tolerances for energy and force are set by *etol* and *ttol*, the same as for the *minimize* command.

A non-zero *etol* means that the GNEB calculation will terminate if the energy criterion is met by every replica. The energies being compared to *etol* do not include any contribution from the inter-replica nudging forces, since these are non-conservative. A non-zero *ttol* means that the GNEB calculation will terminate if the torque criterion is met by every replica. The torques being compared to *ttol* include the inter-replica nudging forces.

The maximum number of iterations in each stage is set by *N1* and *N2*. These are effectively timestep counts since each iteration of damped dynamics is like a single timestep in a dynamics *run*. During both stages, the potential energy of each replica and its normalized distance along the reaction path (reaction coordinate RD) will be printed to the screen and log file every *Nevery* timesteps. The RD is 0 and 1 for the first and last replica. For intermediate replicas, it is the cumulative angular distance (normalized by the total cumulative angular distance) between adjacent replicas, where "distance" is defined as the length of the 3N-vector of the geodesic distances in spin coordinates, with N the number of GNEB spins involved (see equation (13) in *(Bessarab)*). These outputs allow you to monitor NEB's progress in finding a good energy barrier. *N1* and *N2* must both be multiples of *Nevery*.

In the first stage of GNEB, the set of replicas should converge toward a minimum energy path (MEP) of conformational states that transition over a barrier. The MEP for a transition is defined as a sequence of 3N-dimensional spin states, each of which has a potential energy gradient parallel to the MEP itself. The configuration of highest energy along a MEP corresponds to a saddle point. The replica states will also be roughly equally spaced along the MEP due to the inter-replica nudging force added by the *fix neb* command.

In the second stage of GNEB, the replica with the highest energy is selected and the inter-replica forces on it are converted to a force that drives its spin coordinates to the top or saddle point of the barrier, via the barrier-climbing calculation described in *(Bessarab)*. As before, the other replicas rearrange themselves along the MEP so as to be roughly equally spaced.

When both stages are complete, if the GNEB calculation was successful, the configurations of the replicas should be along (close to) the MEP and the replica with the highest energy should be a spin configuration at (close to) the saddle point of the transition. The potential energies for the set of replicas represents the energy profile of the transition along the MEP.

---

An atom map must be defined which it is not by default for *atom_style atomic* problems. The *atom_modify map* command can be used to do this.

An initial value can be defined for the timestep. Although, the *spin* minimization algorithm is an adaptive timestep methodology, so that this timestep is likely to evolve during the calculation.

The minimizers in LAMMPS operate on all spins in your system, even non-GNEB atoms, as defined above.

---

Each file read by the neb/spin command containing spin coordinates used to initialize one or more replicas must be formatted as follows.

The file can be ASCII text or a gzipped text file (detected by a .gz suffix). The file can contain initial blank lines or comment lines starting with "#" which are ignored. The first non-blank, non-comment line should list N = the number of lines to follow. The N successive lines contain the following information:

```
ID1 g1 x1 y1 z1 sx1 sy1 sz1
ID2 g2 x2 y2 z2 sx2 sy2 sz2
...
IDN gN yN zN sxN syN szN
```

The fields are the atom ID, the norm of the associated magnetic spin, followed by the *x,y,z* coordinates and the *sx,sy,sz* spin coordinates. The lines can be listed in any order. Additional trailing information on the line is OK, such as a comment.

Note that for a typical GNEB calculation you do not need to specify initial spin coordinates for very many atoms to produce differing starting and final replicas whose intermediate replicas will converge to the energy barrier. Typically only new spin coordinates for atoms geometrically near the barrier need be specified.

Also note there is no requirement that the atoms in the file correspond to the GNEB atoms in the group defined by the *fix neb* command. Not every GNEB atom need be in the file, and non-GNEB atoms can be listed in the file.

---

Four kinds of output can be generated during a GNEB calculation: energy barrier statistics, thermodynamic output by each replica, dump files, and restart files.

When running with multiple partitions (each of which is a replica in this case), the print-out to the screen and master log.lammps file contains a line of output, printed once every *Nevery* timesteps. It contains the timestep, the maximum torque per replica, the maximum torque per atom (in any replica), potential gradients in the initial, final, and climbing replicas, the forward and backward energy barriers, the total reaction coordinate (RDT), and the normalized reaction coordinate and potential energy of each replica.

The "maximum torque per replica" is the two-norm of the 3N-length vector given by the cross product of a spin by its precession vector omega, in each replica, maximized across replicas, which is what the *ttol* setting is checking against. In this case, N is all the atoms in each replica. The "maximum torque per atom" is the maximum torque component of any atom in any replica. The potential gradients are the two-norm of the 3N-length magnetic precession vector

solely due to the interaction potential i.e. without adding in inter-replica forces, and projected along the path tangent (as detailed in Appendix D of *(Bessarab)*).

The "reaction coordinate" (RD) for each replica is the two-norm of the 3N-length vector of geodesic distances between its spins and the preceding replica's spins (see equation (13) of *(Bessarab)*), added to the RD of the preceding replica. The RD of the first replica RD1 = 0.0; the RD of the final replica RDN = RDT, the total reaction coordinate. The normalized RDs are divided by RDT, so that they form a monotonically increasing sequence from zero to one. When computing RD, N only includes the spins being operated on by the fix neb/spin command.

The forward (reverse) energy barrier is the potential energy of the highest replica minus the energy of the first (last) replica.

Supplementary information for all replicas can be printed out to the screen and master log.lammps file by adding the *verbose* keyword. This information include the following. The "GradVidottan" are the projections of the potential gradient for the replica i on its tangent vector (as detailed in Appendix D of *(Bessarab)*). The "DNi" are the non normalized geodesic distances (see equation (13) of *(Bessarab)*), between a replica i and the next replica i+1. For the last replica, this distance is not defined and a "NAN" value is the corresponding output.

When a NEB calculation does not converge properly, the supplementary information can help understanding what is going wrong.

When running on multiple partitions, LAMMPS produces additional log files for each partition, e.g. log.lammps.0, log.lammps.1, etc. For a GNEB calculation, these contain the thermodynamic output for each replica.

If *dump* commands in the input script define a filename that includes a *universe* or *uloop* style *variable*, then one dump file (per dump command) will be created for each replica. At the end of the GNEB calculation, the final snapshot in each file will contain the sequence of snapshots that transition the system over the energy barrier. Earlier snapshots will show the convergence of the replicas to the MEP.

Likewise, *restart* filenames can be specified with a *universe* or *uloop* style *variable*, to generate restart files for each replica. These may be useful if the GNEB calculation fails to converge properly to the MEP, and you wish to restart the calculation from an intermediate point with altered parameters.

A c file script in provided in the tool/spin/interpolate_gneb directory, that interpolates the MEP given the information provided by the *verbose* output option (as detailed in Appendix D of *(Bessarab)*).

---

### 1.68.4 Restrictions

This command can only be used if LAMMPS was built with the SPIN package. See the *Build package* doc page for more info.

For magnetic GNEB calculations, only the *spin_none* value for the *line* keyword can be used when minimization styles *spin/cg* and *spin/lbfgs* are employed.

---

### 1.68.5 Related commands

*min/spin*, *fix neb/spin*

### 1.68.6 Default

none

---

**(Bessarab)** Bessarab, Uzdin, Jonsson, Comp Phys Comm, 196, 335-347 (2015).

# 1.69 neigh_modify command

## 1.69.1 Syntax

```
neigh_modify keyword values ...
```

- one or more keyword/value pairs may be listed

```
keyword = delay or every or check or once or cluster or include or exclude or page
  or one or binsize or collection/type or collection/interval
  delay value = N
    N = delay building neighbor lists until this many steps since last build
  every value = M
    M = consider building neighbor lists every this many steps
  check value = yes or no
    yes = only build if at least one atom has moved half the skin distance or more
    no = always build on 1st step where every and delay are conditions are satisfied
  once value = yes or no
    yes = only build neighbor list once at start of run and never rebuild
    no = rebuild neighbor list according to other settings
  cluster value = yes or no
    yes = check bond,angle,etc neighbor list for nearby clusters
    no = do not check bond,angle,etc neighbor list for nearby clusters
  include value = group-ID
    group-ID = only build pair neighbor lists for atoms in this group
  exclude values:
    type M N
      M,N = exclude if one atom in pair is type M, other is type N (M and N may be
  type labels)
    group group1-ID group2-ID
      group1-ID,group2-ID = exclude if one atom is in 1st group, other in 2nd
    molecule/intra group-ID
      group-ID = exclude if both atoms are in the same molecule and in group
    molecule/inter group-ID
      group-ID = exclude if both atoms are in different molecules and in group
    none
      delete all exclude settings
  page value = N
    N = number of pairs stored in a single neighbor page
  one value = N
    N = max number of neighbors of one atom
  binsize value = size
    size = bin size for neighbor list construction (distance units)
  collection/type values = N arg1 ... argN
    N = number of custom collections
    arg = N separate lists of types (see below)
  collection/interval values = N arg1 ... argN
    N = number of custom collections
    arg = N separate cutoffs for intervals (see below)
  bin/hash value = yes or no
    yes = use a hash table to store atoms in bins
    no = use a linked list to store atoms in bins
```

### 1.69.2 Examples

```
neigh_modify every 2 delay 10 check yes page 100000
neigh_modify exclude type 2 3
neigh_modify exclude group frozen frozen check no
neigh_modify exclude group residue1 chain3
neigh_modify exclude molecule/intra rigid
neigh_modify collection/type 2 1*2,5 3*4
neigh_modify collection/interval 2 1.0 10.0
```

### 1.69.3 Description

This command sets parameters that affect the building and use of pairwise neighbor lists. Depending on what pair interactions and other commands are defined, a simulation may require one or more neighbor lists.

The *every*, *delay*, *check*, and *once* options affect how often lists are built as a simulation runs. The *delay* setting means never build new lists until at least N steps after the previous build. The *every* setting means attempt to build lists every M steps (after the delay has passed). If the *check* setting is *no*, the lists are built on the first step that satisfies the *delay* and *every* settings. If the *check* setting is *yes*, then the *every* and *delay* settings determine when a build may possibly be performed, but an actual build only occurs if at least one atom has moved more than half the neighbor skin distance (specified in the *neighbor* command) since the last neighbor list build.

Regardless of the *every*, *delay*, *check*, and *once* settings, a neighbor list rebuild is always forced on any timestep when a *restart* file is written, and on timesteps when a fix requests reneighboring (e.g. fixes that create or delete atoms, such as *fix deposit* or *fix evaporate*). This ensures that all atoms are correctly assigned to their owning subdomains before the operation is performed. A consequence is that enabling *restart* output can change the total number of neighbor list builds reported at the end of a run (and thus, as noted below, subtly alter the trajectory) relative to an otherwise identical run without restart output.

**Impact of neighbor list settings**

The choice of neighbor list settings can have a significant impact on the (parallel) performance of LAMMPS and the correctness of the simulation results. Since building the neighbor lists is time consuming, doing it less frequently can speed up a calculation. If the lists are rebuilt too infrequently, however, interacting pairs may be missing and thus the resulting pairwise interactions incorrect. The optimal settings depend on many factors like the properties of the simulated system (density, geometry, topology, temperature, pressure), the force field parameters and settings, the size of the timestep, neighbor list skin distance and more. The default settings are chosen to be very conservative to guarantee correctness of the simulation. They depend on the *check* flag heuristics to reduce the number of neighbor list rebuilds at a minor expense for executing the check. Determining the correctness of a specific choice of neighbor list settings is complicated by the fact that a neighbor list rebuild changes the order in which pairwise interactions are computed and thus - due to the limitations of floating-point math - the trajectory.

If the *once* setting is yes, then the neighbor list is only built once at the beginning of each run, and never rebuilt, except for the forced rebuilds described above (when a restart file is written or a fix requests reneighboring). This setting should only be made if you are certain atoms will not move far enough that the neighbor list should be rebuilt, e.g. running a simulation of a cold crystal. Note that it is not that expensive to check if neighbor lists should be rebuilt.

When the rRESPA integrator is used (see the *run_style* command), the *every* and *delay* parameters refer to the longest (outermost) timestep.

The *cluster* option does a sanity test every time neighbor lists are built for bond, angle, dihedral, and improper interactions, to check that each set of 2, 3, or 4 atoms is a cluster of nearby atoms. It does this by computing the distance between pairs of atoms in the interaction and ensuring they are not further apart than half the periodic box length. If they are, an error is generated, since the interaction would be computed between far-away atoms instead of their nearby periodic images. The only way this should happen is if the pairwise cutoff is so short that atoms that are part of the

same interaction are not communicated as ghost atoms. This is an unusual model (e.g. no pair interactions at all) and the problem can be fixed by use of the *comm_modify cutoff* command. Note that to save time, the default *cluster* setting is *no*, so that this check is not performed.

The *include* option limits the building of pairwise neighbor lists to atoms in the specified group. This can be useful for models where a large portion of the simulation is particles that do not interact with other particles or with each other via pairwise interactions. The group specified with this option must also be specified via the *atom_modify first* command. Note that specifying "all" as the group-ID effectively turns off the *include* option.

The *exclude* option turns off pairwise interactions between certain pairs of atoms, by not including them in the neighbor list. These are sample scenarios where this is useful:

- In crack simulations, pairwise interactions can be shut off between 2 slabs of atoms to effectively create a crack.
- When a large collection of atoms is treated as frozen, interactions between those atoms can be turned off to save needless computation. E.g. Using the *fix setforce* command to freeze a wall or portion of a bio-molecule.
- When one or more rigid bodies are specified, interactions within each body can be turned off to save needless computation. See the *fix rigid* command for more details.

Changed in version 29Aug2024: Support for type labels was added.

The *exclude type* option turns off the pairwise interaction if one atom is of type M and the other of type N. M can equal N. The *exclude group* option turns off the interaction if one atom is in the first group and the other is the second. Group1-ID can equal group2-ID. The *exclude molecule/intra* option turns off the interaction if both atoms are in the specified group and in the same molecule, as determined by their molecule ID. The *exclude molecule/inter* turns off the interaction between pairs of atoms that have different molecule IDs and are both in the specified group.

Each of the exclude options can be specified multiple times. The *exclude type* option is the most efficient option to use; it requires only a single check, no matter how many times it has been specified. The other exclude options are more expensive if specified multiple times; they require one check for each time they have been specified.

Note that the exclude options only affect pairwise interactions; see the *delete_bonds* command for information on turning off bond interactions.

**Note**

Excluding pairwise interactions will not work correctly when also using a long-range solver via the *kspace_style* command. LAMMPS will give a warning to this effect. This is because the short-range pairwise interaction needs to subtract off a term from the total energy for pairs whose short-range interaction is excluded, to compensate for how the long-range solver treats the interaction. This is done correctly for pairwise interactions that are excluded (or weighted) via the *special_bonds* command. But it is not done for interactions that are excluded via these neigh_modify exclude options.

The *page* and *one* options affect how memory is allocated for the neighbor lists. For most simulations the default settings for these options are fine, but if a very large problem is being run or a very long cutoff is being used, these parameters can be tuned. The indices of neighboring atoms are stored in "pages", which are allocated one after another as they fill up. The size of each page is set by the *page* value. A new page is allocated when the next atom's neighbors could potentially overflow the list. This threshold is set by the *one* value which tells LAMMPS the maximum number of neighbor's one atom can have.

**Note**

LAMMPS can crash without an error message if the number of neighbors for a single particle is larger than the *page* setting, which means it is much, much larger than the *one* setting. This is because LAMMPS does not error check these limits for every pairwise interaction (too costly), but only after all the particle's neighbors have been found. This problem usually means something is very wrong with the way you have setup your problem (particle spacing,

cutoff length, neighbor skin distance, etc). If you really expect that many neighbors per particle, then boost the *one* and *page* settings accordingly.

The *binsize* option allows you to specify what size of bins will be used in neighbor list construction to sort and find neighboring atoms. By default, for *neighbor style bin*, LAMMPS uses bins that are 1/2 the size of the maximum pair cutoff. For *neighbor style multi*, the bins are 1/2 the size of the collection interaction cutoff. Typically these are good values for minimizing the time for neighbor list construction. This setting overrides the default. If you make it too big, there is little overhead due to looping over bins, but more atoms are checked. If you make it too small, the optimal number of atoms is checked, but bin overhead goes up. If you set the binsize to 0.0, LAMMPS will use the default binsize of 1/2 the cutoff.

The *collection/type* option allows you to define collections of atom types, used by the *multi* neighbor mode. By grouping atom types with similar physical size or interaction cutoff lengths, one may be able to improve performance by reducing overhead. You must first specify the number of collections N to be defined followed by N lists of types. Each list consists of a series of type ranges separated by commas. The range can be specified as a single numeric value, or a wildcard asterisk can be used to specify a range of values. This takes the form "*" or "*n" or "n*" or "m*n". For example, if M = the number of atom types, then an asterisk with no numeric values means all types from 1 to M. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to M (inclusive). A middle asterisk means all types from m to n (inclusive). Note that all atom types must be included in exactly one of the N collections.

The *collection/interval* option provides a similar capability. This command allows a user to define collections by specifying a series of cutoff intervals. LAMMPS will automatically sort atoms into these intervals based on their type-dependent cutoffs or their finite size. You must first specify the number of collections N to be defined followed by N values representing the upper cutoff of each interval. This command is particularly useful for granular pair styles where the interaction distance of particles depends on their radius and may not depend on their atom type.

Added in version 4Jul2026.

The *bin/hash* option toggles whether the atoms in a bin are stored in a linked list (default) or a hash table where each bin is a key that maps to a set of all atoms contained in that bin. This may be required when the simulation box is too large relative to the size of a neighbor bin (which will cause LAMMPS to error), but will also reduce memory usage and can improve performance in systems with lots of bins that do not contain atoms. For instance, in a binary system with a large disparity in atom sizes/cutoffs. Here, the total number of bins will be determined by the size of the smallest particle. However, almost all bins that lie within the span of the largest particle will be empty. The hash option is only available for the *multi* neighbor mode.

If this option is used to avoid errors regarding the number of bins used to construct neighbor lists, one may need to additionally increase the size of the bin used to sort atoms using the *atom_modify* command. If adjusted, it is recommended to keep the increase conservative such that bins are still reasonably small to preserve performance.

### 1.69.4 Restrictions

If the *delay* setting is non-zero, then it must be a multiple of the *every* setting.

The *molecule/intra* and *molecule/inter* exclusion options can only be used with atom styles that define molecule IDs.

The value of the *page* setting must be at least 10x larger than the *one* setting. This ensures neighbor pages are not mostly empty space.

The *exclude group* setting is currently not compatible with dynamic groups.

### 1.69.5 Related commands

*neighbor*, *delete_bonds*

### 1.69.6 Default

The option defaults are delay = 0, every = 1, check = yes, once = no, cluster = no, include = all (same as no include option defined), exclude = none, page = 100000, one = 2000, bin/hash no, and binsize = 0.0.

## 1.70 neighbor command

### 1.70.1 Syntax

```
neighbor skin style
```

- skin = extra distance beyond force cutoff (distance units)
- style = *bin* or *nsq* or *multi*

### 1.70.2 Examples

```
neighbor 0.3 bin
neighbor 2.0 nsq
```

### 1.70.3 Description

This command sets parameters that affect the building of pairwise neighbor lists. All atom pairs within a neighbor cutoff distance equal to the their force cutoff plus the *skin* distance are stored in the list. Typically, the larger the skin distance, the less often neighbor lists need to be built, but more pairs must be checked for possible force interactions every timestep. The default value for *skin* depends on the choice of units for the simulation; see the default values below.

The *skin* distance is also used to determine how often atoms migrate to new processors if the *check* option of the *neigh_modify* command is set to *yes*. Atoms are migrated (communicated) to new processors on the same timestep that neighbor lists are re-built.

The *style* value selects what algorithm is used to build the list. The *bin* style creates the list by binning which is an operation that scales linearly with N/P, the number of atoms per processor where N = total number of atoms and P = number of processors. It is almost always faster than the *nsq* style which scales as (N/P)^2. For unsolvated small molecules in a non-periodic box, the *nsq* choice can sometimes be faster. Either style should give the same answers.

The *multi* style is a modified binning algorithm that is useful for systems with a wide range of cutoff distances, e.g. due to different size particles. For granular pair styles, cutoffs are set to the sum of the maximum atomic radii for each atom type. For the *bin* style, the bin size is set to 1/2 of the largest cutoff distance between any pair of atom types and a single set of bins is defined to search over for all atom types. This can be inefficient if one pair of types has a very long cutoff, but other type pairs have a much shorter cutoff. The *multi* style uses different sized bins for collections of different sized particles, where "size" may mean the physical size of the particle or its cutoff distance for interacting with other particles. Different sets of bins are then used to construct the neighbor lists as as further described by Shire, Hanley, and Stratford *(Shire)* and Monti et al. *(Monti)*. This imposes some extra setup overhead, but the searches themselves may be much faster.

For instance in a dense binary system in d-dimensions with a ratio of the size of the largest to smallest collection bin $\lambda$, the computational costs of building a default neighbor list grows as $\lambda^{2d}$ while the costs for *multi* grows as $\lambda^{d}$, equivalent to the cost of force evaluations, as argued in Monti et al. *(Monti)*. In other words, the neighboring costs of *multi* are expected to scale the same as force calculations, such that its relative cost is independent of the particle size ratio. This is not the case for the default style which becomes substantially more expensive with increasing size ratios.

By default in *multi*, each atom type defines a separate collection of particles. For systems where two or more atom types have the same size (either physical size or cutoff distance), the definition of collections can be customized, which

can result in less overhead and faster performance. See the *neigh_modify* command for how to define custom collections. Whether the collection definition is customized or not, also see the *comm_modify mode multi* command for communication options that further improve performance in a manner consistent with neighbor style multi.

**Note**

If there are multiple sub-styles in a *hybrid/overlay pair style* that cover the same atom types, but have significantly different cutoffs, the *multi* style does not apply. Instead, the *pair_modify neigh/trim* setting applies (which is *yes* by default). Please check the neighbor list summary printed at the beginning of a calculation to verify that the desired set of neighbor list builds is performed.

The *neigh_modify* command has additional options that control how often neighbor lists are built and which pairs are stored in the list.

When a run is finished, counts of the number of neighbors stored in the pairwise list and the number of times neighbor lists were built are printed to the screen and log file. See the *Run output* page for details.

### 1.70.4 Restrictions

none

### 1.70.5 Related commands

*neigh_modify*, *units*, *comm_modify*

### 1.70.6 Default

0.3 bin for units = lj, skin = 0.3 sigma
2.0 bin for units = real or metal, skin = 2.0 Angstroms
0.001 bin for units = si, skin = 0.001 meters = 1.0 mm
0.1 bin for units = cgs, skin = 0.1 cm = 1.0 mm

---

**(Shire)** Shire, Hanley and Stratford, Comp. Part. Mech., (2020).

**(Monti)** Monti, Clemmer, Srivastava, Silbert, Grest, and Lechman, Phys. Rev. E, (2022).

## 1.71 newton command

### 1.71.1 Syntax

```
newton flag
newton flag1 flag2
```

- flag = *on* or *off* for both pairwise and bonded interactions
- flag1 = *on* or *off* for pairwise interactions
- flag2 = *on* or *off* for bonded interactions

### 1.71.2 Examples

```
newton off
newton on off
```

### 1.71.3 Description

This command turns Newton's third law *on* or *off* for pairwise and bonded interactions. For most problems, setting Newton's third law to *on* means a modest savings in computation at the cost of two times more communication. Whether this is faster depends on problem size, force cutoff lengths, a machine's compute/communication ratio, and how many processors are being used.

Setting the pairwise newton flag to *off* means that if two interacting atoms are on different processors, both processors compute their interaction and the resulting force information is not communicated. Similarly, for bonded interactions, newton *off* means that if a bond, angle, dihedral, or improper interaction contains atoms on 2 or more processors, the interaction is computed by each processor.

LAMMPS should produce the same answers for any newton flag settings, except for round-off issues.

With *run_style* *respa* and only bonded interactions (bond, angle, etc) computed in the innermost timestep, it may be faster to turn newton *off* for bonded interactions, to avoid extra communication in the innermost loop.

### 1.71.4 Restrictions

The newton bond setting cannot be changed after the simulation box is defined by a *read_data* or *create_box* command.

### 1.71.5 Related commands

*run_style* respa

### 1.71.6 Default

```
newton on
```

## 1.72 next command

### 1.72.1 Syntax

```
next variables
```

- variables = one or more variable names

### 1.72.2 Examples

```
next x
next a t x myTemp
```

### 1.72.3 Description

This command is used with variables defined by the *variable* command. It assigns the next value to the variable from the list of values defined for that variable by the *variable* command. Thus when that variable is subsequently substituted for in an input script command, the new value is used.

See the *variable* command for info on how to define and use different kinds of variables in LAMMPS input scripts. If a variable name is a single lower-case character from "a" to "z", it can be used in an input script command as $a or $z. If it is multiple letters, it can be used as ${myTemp}.

If multiple variables are used as arguments to the *next* command, then all must be of the same variable style: *index*, *loop*, *file*, *universe*, or *uloop*. An exception is that *universe*- and *uloop*-style variables can be mixed in the same *next* command.

All the variables specified with the next command are incremented by one value from their respective list of values. A *file*-style variable reads the next line from its associated file. An *atomfile*-style variable reads the next set of lines (one per atom) from its associated file. *String*- or *atom*- or *equal*- or *world*-style variables cannot be used with the next command, since they only store a single value.

When any of the variables in the next command has no more values, a flag is set that causes the input script to skip the next *jump* command encountered. This enables a loop containing a next command to exit. As explained in the *variable* command, the variable that has exhausted its values is also deleted. This allows it to be used and re-defined later in the input script. *File*-style and *atomfile*-style variables are exhausted when the end-of-file is reached.

When the next command is used with *index*- or *loop*-style variables, the next value is assigned to the variable for all processors. When the next command is used with *file*-style variables, the next line is read from its file and the string assigned to the variable. When the next command is used with *atomfile*-style variables, the next set of per-atom values is read from its file and assigned to the variable.

When the next command is used with *universe*- or *uloop*-style variables, all *universe*- or *uloop*-style variables must be listed in the next command. This is because of the manner in which the incrementing is done, using a single lock file for all variables. The next value (for each variable) is assigned to whichever processor partition executes the command first. All processors in the partition are assigned the same value(s). Running LAMMPS on multiple partitions of processors via the *-partition command-line switch*. *Universe*- and *uloop*-style variables are incremented using the files "tmp.lammps.variable" and "tmp.lammps.variable.lock" which you will see in your directory during and after such a LAMMPS run.

Here is an example of running a series of simulations using the next command with an *index*-style variable. If this input script is named in.polymer, 8 simulations would be run using data files from directories run1 through run8.

```
variable d index run1 run2 run3 run4 run5 run6 run7 run8
shell cd $d
read_data data.polymer
run 10000
shell cd ..
clear
next d
jump in.polymer
```

If the variable "d" were of style *universe*, and the same in.polymer input script were run on 3 partitions of processors, then the first 3 simulations would begin, one on each set of processors. Whichever partition finished first, it would assign variable "d" the fourth value and run another simulation, and so forth until all 8 simulations were finished.

Jump and next commands can also be nested to enable multi-level loops. For example, this script will run 15 simulations in a double loop.

```
variable i loop 3
  variable j loop 5
  clear
  ...
  read_data data.polymer.$i$j
  print Running simulation $i.$j
  run 10000
```

(continues on next page)

(continued from previous page)

```
  next j
  jump in.script
next i
jump in.script
```

Here is an example of a double loop which uses the *if* and *jump* commands to break out of the inner loop when a condition is met, then continues iterating through the outer loop.

```
label       loopa
variable    a loop 5
  label     loopb
  variable  b loop 5
  print     "A,B = $a,$b"
  run       10000
  if        $b > 2 then "jump in.script break"
  next      b
  jump      in.script loopb
label       break
variable    b delete

next        a
jump        in.script loopa
```

### 1.72.4 Restrictions

As described above.

### 1.72.5 Related commands

*jump*, *include*, *shell*, *variable*,

### 1.72.6 Default

none

## 1.73 package command


- *Syntax*
- *Examples*
- *Description*
  - *GPU package settings*
  - *INTEL package settings*
  - *KOKKOS package settings*
  - *OPENMP package settings*
- *Restrictions*
- *Related commands*

- *Defaults*

### 1.73.1 Syntax

```
package style args
```

- style = *gpu* or *intel* or *kokkos* or *omp*
- args = arguments specific to the style

```
gpu args = Ngpu keyword value ...
  Ngpu = # of GPUs per node
  zero or more keyword/value pairs may be appended
  keywords = neigh or newton or pair/only or binsize or split or gpuID or tpa or␣
↪blocksize or omp or platform or device_type or ocl_args
    neigh value = yes or no or hybrid
      yes = neighbor list build on GPU (default)
      no = neighbor list build on CPU
      hybrid = perform binning on the CPU but build neighbor list on the GPU
    newton = off or on
      off = set Newton pairwise flag off (default and required)
      on = set Newton pairwise flag on (currently not allowed)
    pair/only = off or on
      off = apply "gpu" suffix to all available styles in the GPU package (default)
      on = apply "gpu" suffix only pair styles
    binsize value = size
      size = bin size for neighbor list construction (distance units)
    split = fraction
      fraction = fraction of atoms assigned to GPU (default = 1.0)
    tpa value = Nlanes
      Nlanes = # of GPU vector lanes (CUDA threads) used per atom
    blocksize value = size
      size = thread block size for pair force computation
    omp value = Nthreads
      Nthreads = number of OpenMP threads to use on CPU (default = 0)
    platform value = id
      id = For OpenCL, platform ID for the GPU or accelerator
    gpuID values = id
      id = ID of first GPU to be used on each node
    device_type value = intelgpu or nvidiagpu or amdgpu or applegpu or generic or␣
↪custom,val1,val2,...
      val1,val2,... = custom OpenCL accelerator configuration parameters (see below␣
↪for details)
    ocl_args value = args
      args = List of additional OpenCL compiler arguments delimited by colons
intel args = Narg keyword value ...
  Narg = accepted for backward compatibility and ignored
  zero or more keyword/value pairs may be appended
  keywords = mode or omp or lrt or pppm_table
    mode value = single or mixed or double
      single = perform force calculations in single precision
      mixed = perform force calculations in mixed precision
      double = perform force calculations in double precision
```

```
    omp value = Nthreads
      Nthreads = number of OpenMP threads to use on CPU (default = 0)
    lrt value = yes or no
      yes = use additional thread dedicated for some PPPM calculations
      no = do not dedicate an extra thread for some PPPM calculations
    pppm_table value = yes or no
      yes = Precompute pppm values in table (doesn't change accuracy)
      no = Compute pppm values on the fly
kokkos args = keyword value ...
  zero or more keyword/value pairs may be appended
  keywords = neigh or neigh/qeq or neigh/thread or neigh/transpose or newton or␣
↪binsize or comm or comm/exchange or comm/forward or comm/pair/forward or comm/fix/
↪forward or comm/compute/forward or comm/reverse or comm/pair/reverse or comm/fix/
↪reverse or sort or atom/map or gpu/aware or pair/only
    neigh value = full or half
      full = full neighbor list
      half = half neighbor list built in thread-safe manner
    neigh/qeq value = full or half
      full = full neighbor list
      half = half neighbor list built in thread-safe manner
    neigh/thread value = off or on
      off = thread only over atoms
      on = thread over both atoms and neighbors
    neigh/transpose value = off or on
      off = use same memory layout for GPU neigh list build as pair style
      on = use transposed memory layout for GPU neigh list build
    newton = off or on
      off = set Newton pairwise and bonded flags off
      on = set Newton pairwise and bonded flags on
    binsize value = size
      size = bin size for neighbor list construction (distance units)
    comm value = no or host or device
      use value for comm/exchange and comm/forward and comm/pair/forward and comm/
↪fix/forward and comm/compute/forward and comm/reverse and comm/fix/reverse
    comm/exchange value = no or host or device
    comm/forward value = no or host or device
    comm/pair/forward value = no or device
    comm/fix/forward value = no or device
    comm/compute/forward value = no or device
    comm/reverse value = no or host or device
      no = perform communication pack/unpack in non-KOKKOS mode
      host = perform pack/unpack on host (e.g. with OpenMP threading)
      device = perform pack/unpack on device (e.g. on GPU)
    comm/pair/reverse value = no or device
      no = perform communication pack/unpack in non-KOKKOS mode
      device = perform pack/unpack on device (e.g. on GPU)
    comm/fix/reverse value = no or host or device
      no = perform communication pack/unpack in non-KOKKOS mode
      host = perform pack/unpack on host (e.g. with OpenMP threading)
      device = perform pack/unpack on device (e.g. on GPU)
    sort value = no or device
      no = perform atom sorting in non-KOKKOS mode
      device = perform atom sorting on device (e.g. on GPU)
    atom/map value = no or device
```

```
      no = build atom map in non-KOKKOS mode
      device = build atom map on device (e.g. on GPU)
    gpu/aware = off or on
      off = do not use GPU-aware MPI
      on = use GPU-aware MPI (default)
    pair/only = off or on
      off = use device acceleration (e.g. GPU) for all available styles in the␣
↪KOKKOS package (default)
      on  = use device acceleration only for pair styles (and host acceleration for␣
↪others)
    threads/per/atom args = Ntpa
      Ntpa = # of threads per atom for multiple GPU threads over the neighbor list␣
↪per atom
    pair/team/size args = Nteamsize
      Nteamsize = # of threads per block used for the pair compute kernel
    nbin/atoms/per/bin = Natomsperbin
      Natomsperbin = # of atoms per bin used for neighbor list builds
    *nbor/chunk/size = chunksize
      chunksize = # of iterations each thread will perform for the flat neighbor␣
↪build method
    *bond/chunk/size = blocksize
      chunksize = # of iterations each thread will perform for the bond force␣
↪computation
    *auto/tuning = nevery nsamples mode reltol
      nevery = # timesteps between auto-tuning adjustments (default = 0, no␣
↪auto-tuning)
      nsamples = # samples the tuner(s) collects for each parameter combination
      mode = how to pick a performance value from the samples collected, i.e.␣
↪maximum, average or median value
      reltol = relative tolerance for performance degradation that triggers␣
↪re-tuning of parameter values
  *omp args = Nthreads keyword value ...
    Nthreads = # of OpenMP threads to associate with each MPI process
    zero or more keyword/value pairs may be appended
    keywords = neigh
      neigh value = yes or no
        yes = threaded neighbor list build (default)
        no = non-threaded neighbor list build
```

### 1.73.2 Examples

```
package gpu 0
package gpu 1 split 0.75
package gpu 2 split -1.0
package gpu 0 omp 2 device_type intelgpu
package kokkos neigh half comm device
package omp 0 neigh no
package omp 4
package intel 1
package intel 2 omp 4 mode mixed
```

### 1.73.3 Description

This command invokes package-specific settings for the various accelerator packages available in LAMMPS. Currently the following packages use settings from this command: GPU, INTEL, KOKKOS, and OPENMP.

If this command is specified in an input script, it must be near the top of the script, before the simulation box has been defined. This is because it specifies settings that the accelerator packages use in their initialization, before a simulation is defined.

This command can also be specified from the command-line when launching LAMMPS, using the "-pk" *command-line switch*. The syntax is exactly the same as when used in an input script.

Note that all of the accelerator packages require the package command to be specified (except the OPT package), if the package is to be used in a simulation (LAMMPS can be built with an accelerator package without using it in a particular simulation). However, in all cases, a default version of the command is typically invoked by other accelerator settings.

The KOKKOS package requires a "-k on" *command-line switch* respectively, which invokes a "package kokkos" command with default settings.

For the GPU, INTEL, and OPENMP packages, if a "-sf gpu" or "-sf intel" or "-sf omp" *command-line switch* is used to auto-append accelerator suffixes to various styles in the input script, then those switches also invoke a "package gpu", "package intel", or "package omp" command with default settings.

**Note**

A package command for a particular style can be invoked multiple times when a simulation is setup, e.g. by the *-c on, -k on, -sf, and -pk command-line switches*, and by using this command in an input script. Each time it is used all of the style options are set, either to default values or to specified settings. I.e. settings from previous invocations do not persist across multiple invocations.

See the *Accelerator packages* page for more details about using the various accelerator packages for speeding up LAMMPS simulations.

#### GPU package settings

The *gpu* style invokes settings associated with the use of the *GPU package*.

The *Ngpu* argument sets the number of GPUs per node. If *Ngpu* is 0 and no other keywords are specified, GPU or accelerator devices are auto-selected. In this process, all platforms are searched for accelerator devices and GPUs are chosen if available. The device with the highest number of compute cores is selected. The number of devices is increased to be the number of matching accelerators with the same number of compute cores. If there are more devices than MPI tasks, the additional devices will be unused. The auto-selection of GPUs/ accelerator devices and platforms can be restricted by specifying a non-zero value for *Ngpu* and / or using the *gpuID*, *platform*, and *device_type* keywords as described below. If there are more MPI tasks (per node) than GPUs, multiple MPI tasks will share each GPU.

Optional keyword/value pairs can also be specified. Each has a default value as listed below.

Changed in version 10Dec2025: Updated description to the current state of the GPU package

The *neigh* keyword specifies where neighbor lists for pair style computation will be built. If *neigh* is *yes*, which is the default, neighbor list building is performed on the GPU. If *neigh* is *no*, neighbor list building is instead performed on the CPU. If *neigh* is *hybrid* the binning step of the neighbor list build is performed on the CPU and the list themselves on the GPU. GPU neighbor list building currently is not fully compatible with a triclinic box; if the behavior is significantly different from the CPU case, use the *neigh no* setting. GPU neighbor lists are not accessible for commands that are not GPU-enabled. When a non-GPU enabled command requires a neighbor list, it will be built on the CPU. In these cases, it can be more efficient to only use CPU neighbor list builds, particularly if the CPU neighbor list is perpetual, i.e. used in every step. If a GPU environment does not support building neighbor lists on the GPU, the default setting it will automatically change to *neigh no*.

The *newton* keyword sets the Newton flags for pairwise (not bonded) interactions to *off* or *on*, the same as the *newton* command allows. Currently, only an *off* value is allowed, since all the GPU package pair styles require this setting. This means more computation is done, but less communication. In the future a value of *on* may be allowed, so the *newton* keyword is included as an option for compatibility with the package command for other accelerator styles. Note that the newton setting for bonded interactions is not affected by this keyword.

The *pair/only* keyword can change how any "gpu" suffix is applied. By default a suffix is applied to all styles for which an accelerated variant is available. However, that is not always the most effective way to use an accelerator. With *pair/only* set to *on* the suffix will only by applied to supported pair styles, which tend to be the most effective in using an accelerator and their operation can be overlapped with all other computations on the CPU.

The *binsize* keyword sets the size of bins used to bin atoms in neighbor list builds performed on the GPU, if *neigh* = *yes* is set. If *binsize* is set to 0.0 (the default), then the binsize is set automatically using heuristics in the GPU package.

The *split* keyword can be used for load balancing force calculations between CPU and GPU cores in GPU-enabled pair styles. If 0 < *split* < 1.0, a fixed fraction of particles is offloaded to the GPU while force calculation for the other particles occurs simultaneously on the CPU. If *split* < 0.0, the optimal fraction (based on CPU and GPU timings) is calculated every 25 timesteps, i.e. dynamic load-balancing across the CPU and GPU is performed. If *split* = 1.0, all force calculations for GPU accelerated pair styles are performed on the GPU. In this case, other *hybrid* pair interactions, *bond*, *angle*, *dihedral*, *improper*, and *long-range* calculations can be performed on the CPU while the GPU is performing force calculations for the GPU-enabled pair style. If all CPU force computations complete before the GPU completes, LAMMPS will block until the GPU has finished before continuing the timestep.

As an example, if you have two GPUs per node and 8 CPU cores per node, and would like to run on 4 nodes (32 cores) with dynamic balancing of force calculation across CPU and GPU cores, you could specify

```
mpirun -np 32 -sf gpu -in in.script    # launch command
package gpu 2 split -1                 # input script command
```

In this case, all CPU cores and GPU devices on the nodes would be utilized. Each GPU device would be shared by 4 CPU cores. The CPU cores would perform force calculations for some fraction of the particles at the same time the GPUs performed force calculation for the other particles.

The *gpuID* keyword is used to specify the first ID for the GPU or other accelerator that LAMMPS will use. For example, if the ID is 1 and *Ngpu* is 3, GPUs 1-3 will be used. Device IDs should be determined from the output of nvc_get_devices, ocl_get_devices, or hip_get_devices as provided in the lib/gpu directory. When using OpenCL with accelerators that have main memory NUMA, the accelerators can be split into smaller virtual accelerators for more efficient use with MPI.

The *tpa* keyword sets the number of GPU vector lanes per atom used to perform force calculations. With a default value of 1, the number of lanes will be chosen based on the pair style, however, the value can be set explicitly with this keyword to fine-tune performance. For large cutoffs or with a small number of particles per GPU, increasing the value can improve performance. The number of lanes per atom must be a power of 2 and currently cannot be greater than the SIMD width for the GPU / accelerator. In the case it exceeds the SIMD width, it will automatically be decreased to meet the restriction.

The *blocksize* keyword allows you to tweak the number of threads used per thread block. This number should be a multiple of 32 (for GPUs) and its maximum depends on the specific GPU hardware. Typical choices are 64, 128, or 256. A larger block size increases occupancy of individual GPU cores, but reduces the total number of thread blocks, thus may lead to load imbalance. On modern hardware, the sensitivity to the blocksize is typically low.

The *Nthreads* value for the *omp* keyword sets the number of OpenMP threads allocated for each MPI task. This setting controls OpenMP parallelism only for routines run on the CPUs. For more details on setting the number of OpenMP threads, see the discussion of the *Nthreads* setting on this page for the "package omp" command. The meaning of *Nthreads* is exactly the same for the GPU, INTEL, and GPU packages.

The *platform* keyword is only used with OpenCL to specify the ID for an OpenCL platform. See the output from ocl_get_devices in the lib/gpu directory. In LAMMPS only one platform can be active at a time and by default (id=-1) the platform is auto-selected to find the GPU with the most compute cores. When *Ngpu* or other keywords are specified,

the auto-selection is appropriately restricted. For example, if *Ngpu* is 3, only platforms with at least 3 accelerators are considered. Similar restrictions can be enforced by the *gpuID* and *device_type* keywords.

The *device_type* keyword can be used for OpenCL to specify the type of GPU to use or specify a custom configuration for an accelerator. In most cases this selection will be automatic and there is no need to use the keyword. The *applegpu* type is not specific to a particular GPU vendor, but is separate due to the more restrictive Apple OpenCL implementation. For expert users, to specify a custom configuration, the *custom* keyword followed by the next parameters can be specified:

CONFIG_ID, SIMD_SIZE, MEM_THREADS, SHUFFLE_AVAIL, FAST_MATH, THREADS_PER_ATOM, THREADS_PER_CHARGE, THREADS_PER_THREE, BLOCK_PAIR, BLOCK_BIO_PAIR, BLOCK_ELLIPSE, PPPM_BLOCK_1D, BLOCK_NBOR_BUILD, BLOCK_CELL_2D, BLOCK_CELL_ID, MAX_SHARED_TYPES, MAX_BIO_SHARED_TYPES, PPPM_MAX_SPLINE, NBOR_PREFETCH.

CONFIG_ID can be 0. SHUFFLE_AVAIL in {0,1} indicates that inline-PTX (NVIDIA) or OpenCL extensions (Intel) should be used for horizontal vector operations. FAST_MATH in {0,1} indicates that OpenCL fast math optimizations are used during the build and hardware-accelerated transcendental functions are used when available. THREADS_PER_* give the default *tpa* values for ellipsoidal models, styles using charge, and any other styles. The BLOCK_* parameters specify the block sizes for various kernel calls and the MAX_*SHARED_* parameters are used to determine the amount of local shared memory to use for storing model parameters.

For OpenCL, the routines are compiled at runtime for the specified GPU or accelerator architecture. The *ocl_args* keyword can be used to specify additional flags for the runtime build.

### INTEL package settings

The *intel* style invokes settings associated with the use of the INTEL package.

Deprecated since version 4Jul2026.

Support for offloading to Intel(R) Xeon Phi(TM) co-processors was removed. The leading numeric argument (formerly the number of co-processors per node) is ignored.

Optional keyword/value pairs can also be specified. Each has a default value as listed below.

The *Nthreads* value for the *omp* keyword sets the number of OpenMP threads allocated for each MPI task. This setting controls OpenMP parallelism only for routines run on the CPUs. For more details on setting the number of OpenMP threads, see the discussion of the *Nthreads* setting on this page for the "package omp" command. The meaning of *Nthreads* is exactly the same for the GPU, INTEL, and GPU packages.

The *mode* keyword determines the precision mode to use for computing pair style forces on the CPU when using a INTEL supported *pair style*. It can take a value of *single*, *mixed* which is the default, or *double*. *Single* means single precision is used for the entire force calculation. *Mixed* means forces between a pair of atoms are computed in single precision, but accumulated and stored in double precision, including storage of forces, torques, energies, and virial quantities. *Double* means double precision is used for the entire force calculation.

The *lrt* keyword can be used to enable "Long Range Thread (LRT)" mode. It can take a value of *yes* to enable and *no* to disable. LRT mode generates an extra thread (in addition to any OpenMP threads specified with the OMP_NUM_THREADS environment variable or the *omp* keyword). The extra thread is dedicated for performing part of the *PPPM solver* computations and communications. This can improve parallel performance on processors supporting Simultaneous Multithreading (SMT) such as Hyper-Threading (HT) on Intel processors. In this mode, one additional thread is generated per MPI process. LAMMPS will generate a warning in the case that more threads are used than available in SMT hardware on a node. If the PPPM solver from the INTEL package is not used, then the LRT setting is ignored and no extra threads are generated. Enabling LRT will replace the *run_style* with the *verlet/lrt/intel* style that is identical to the default *verlet* style aside from supporting the LRT feature. This feature requires setting the pre-processor flag -DLMP_INTEL_USELRT in the makefile when compiling LAMMPS.

Added in version 15Jun2023.

The *pppm_table* keyword with the argument yes allows to use a pre-computed table to efficiently spread the charge to the PPPM grid. This feature is enabled by default but can be turned off using the keyword with the argument *no*.

### KOKKOS package settings

The *kokkos* style invokes settings associated with the use of the KOKKOS package.

All of the settings are optional keyword/value pairs. Each has a default value as listed below.

The *neigh* keyword determines how neighbor lists are built. A value of *half* uses a thread-safe variant of half-neighbor lists, the same as used by most pair styles in LAMMPS, which is the default when running on CPUs (i.e. the Kokkos CUDA back end is not enabled).

A value of *full* uses a full neighbor lists and is the default when running on GPUs. This performs twice as much computation as the *half* option, however that is often a win because it is thread-safe and does not require atomic operations in the calculation of pair forces. For that reason, *full* is the default setting for GPUs. However, when running on CPUs, a *half* neighbor list is the default because it are often faster, just as it is for non-accelerated pair styles. Similarly, the *neigh/qeq* keyword determines how neighbor lists are built for *fix qeq/reaxff/kk*.

If the *neigh/thread* keyword is set to *off*, then the KOKKOS package threads only over atoms. However, for small systems, this may not expose enough parallelism to keep a GPU busy. When this keyword is set to *on*, the KOKKOS package threads over both atoms and neighbors of atoms. When using *neigh/thread on*, the *newton pair* setting must be "off". Using *neigh/thread on* may be slower for large systems, so this this option is turned on by default only when running on one or more GPUs and there are 16k atoms or less owned by an MPI rank. Not all KOKKOS-enabled potentials support this keyword yet, and only thread over atoms. Many simple pairwise potentials such as Lennard-Jones do support threading over both atoms and neighbors.

If the *neigh/transpose* keyword is set to *off*, then the KOKKOS package will use the same memory layout for building the neighbor list on GPUs as used for the pair style. When this keyword is set to *on* it will use a different (transposed) memory layout to build the neighbor list on GPUs. This can be faster in some cases (e.g. ReaxFF HNS benchmark) but slower in others (e.g. Lennard Jones benchmark). The copy between different memory layouts is done out of place and therefore doubles the memory overhead of the neighbor list, which can be significant.

The *newton* keyword sets the Newton flags for pairwise and bonded interactions to *off* or *on*, the same as the *newton* command allows. The default for GPUs is *off* because this will almost always give better performance for the KOKKOS package. This means more computation is done, but less communication. However, when running on CPUs a value of *on* is the default since it can often be faster, just as it is for non-accelerated pair styles

The *binsize* keyword sets the size of bins used to bin atoms during neighbor list builds. The same value can be set by the *neigh_modify binsize* command. Making it an option in the package kokkos command allows it to be set from the command-line. The default value for CPUs is 0.0, which means the LAMMPS default will be used, which is bins = 1/2 the size of the pairwise cutoff + neighbor skin distance. This is fine when neighbor lists are built on the CPU. For GPU builds, a 2x larger binsize equal to the pairwise cutoff + neighbor skin is often faster, which is the default. Note that if you use a longer-than-usual pairwise cutoff, e.g. to allow for a smaller fraction of KSpace work with a *long-range Coulombic solver* because the GPU is faster at performing pairwise interactions, then this rule of thumb may give too large a binsize and the default should be overridden with a smaller value.

The *comm* and *comm/exchange* and *comm/forward* and *comm/pair/forward* and *comm/fix/forward* and *comm/compute/forward* and *comm/reverse* and *comm/pair/reverse* and *comm/fix/reverse* keywords determine whether the host or device performs the packing and unpacking of data when communicating per-atom data between processors. "Exchange" communication happens only on timesteps that neighbor lists are rebuilt. The data is only for atoms that migrate to new processors. "Forward" communication happens every timestep. "Reverse" communication happens every timestep if the *newton* option is on. The data is for atom coordinates and any other atom properties that needs to be updated for ghost atoms owned by each processor. "Comm/pair" controls additional communication in pair styles, such as pair_style EAM. "Comm/fix" controls additional communication in fixes, such as fix SHAKE. Similarly, "comm/compute" controls additional communication in computes.

The *comm* keyword is simply a short-cut to set the same value for all the comm keywords.

The value options for the keywords are *no* or *host* or *device*. A value of *no* means to use the standard non-KOKKOS method of packing/unpacking data for the communication. A value of *host* means to use the host, typically a multicore CPU, and perform the packing/unpacking in parallel with threads. A value of *device* means to use the device, typically a GPU, to perform the packing/unpacking operation.

For the *comm/pair/forward* or *comm/fix/forward* or *comm/compute/forward* or *comm/pair/reverse* keywords, if a value of *host* is used it will be automatically be changed to *no* since these keywords don't support *host* mode. The value of *no* will also always be used when running on the CPU, i.e. setting the value to *device* will have no effect if the pair/fix style is running on the CPU. For the *comm/fix/forward* or *comm/compute/forward* or *comm/pair/reverse* or *comm/fix/reverse* keywords, not all styles support *device* mode and in that case will run in *no* mode instead.

The optimal choice for these keywords depends on the input script and the hardware used. The *no* value is useful for verifying that the Kokkos-based *host* and *device* values are working correctly. It is the default when running on CPUs since it is usually the fastest.

When running on CPUs, the *host* and *device* values work identically. When using GPUs, the *device* value is the default since it will typically be optimal if all of your styles used in your input script are supported by the KOKKOS package. In this case data can stay on the GPU for many timesteps without being moved between the host and GPU, if you use the *device* value. If your script uses styles (e.g. fixes) which are not yet supported by the KOKKOS package, then data has to be moved between the host and device anyway, so it is typically faster to let the host handle communication, by using the *host* value. Using *host* instead of *no* will enable use of multiple threads to pack/unpack communicated data. When running small systems on a GPU, performing the exchange pack/unpack on the host CPU can give speedup since it reduces the number of CUDA kernel launches.

The *sort* keyword determines whether the host or device performs atom sorting, see the *atom_modify sort* command. The value options for the *sort* keyword are *no* or *device* similar to the *comm* keywords above. If a value of *host* is used it will be automatically be changed to *no* since the *sort* keyword does not support *host* mode. Not all fix styles with extra atom data support *device* mode and in that case a warning will be given and atom sorting will run in *no* mode instead.

Added in version 17Apr2024.

The *atom/map* keyword determines whether the host or device builds the atom_map, see the *atom_modify map* command. The value options for the *atom/map* keyword are identical to the *sort* keyword above.

The *gpu/aware* keyword chooses whether GPU-aware MPI will be used. When this keyword is set to *on*, buffers in GPU memory are passed directly through MPI send/receive calls. This reduces overhead of first copying the data to the host CPU. However GPU-aware MPI is not supported on all systems, which can lead to segmentation faults and would require using a value of *off*. If LAMMPS can safely detect that GPU-aware MPI is not available (currently only possible with OpenMPI v2.0.0 or later), then the *gpu/aware* keyword is automatically set to *off* by default. When the *gpu/aware* keyword is set to *off* while any of the *comm* keywords are set to *device*, the value for these *comm* keywords will be automatically changed to *no*. This setting has no effect if not running on GPUs or if using only one MPI rank. GPU-aware MPI is available for OpenMPI 1.8 (or later versions), Mvapich2 1.9 (or later) when the "MV2_USE_CUDA" environment variable is set to "1", CrayMPI, and IBM Spectrum MPI when the "-gpu" flag is used.

The *pair/only* keyword can change how the KOKKOS suffix "kk" is applied when using an accelerator device. By default device acceleration is always used for all available styles. With *pair/only* set to *on* the suffix setting will choose device acceleration only for pair styles and run all other force computations on the host CPU. The *comm* flags, along with the *sort* and *atom/map* keywords will also automatically be changed to *no*. This can result in better performance for certain configurations and system sizes.

The following parameters allow users to tune the overall performance depending on the simulated systems. If not explicitly specified, their values will be set internally by the KOKKOS package.

The *threads/per/atom* keyword sets the number of GPU vector lanes per atom used to perform force calculations. This keyword is only applicable when *neigh/thread* is set to *on*. For large cutoffs or with a small number of particles per GPU, increasing the value can improve performance. The number of lanes per atom must be a power of 2 and currently cannot be greater than the SIMD width for the GPU / accelerator. In the case it exceeds the SIMD width, it will automatically be decreased to meet the restriction.

The *pair/team/size* keyword sets the number of threads per block for the pair force compute kernel. This keyword is only applicable when *neigh/thread* is set to *on*. The default value of this parameter is determined based on the GPU architecture at runtime.

The *nbin/atoms/per/bin* keyword sets the number of atoms per bin used for the neighbor list builds on the GPU, which then determines the number of GPU threads per bin. The default value of this parameter is 16.

The *nbor/chunk/size* keyword sets the number of iterations that a work item is scheduled for the neighbor list builds on the GPU using the flat method (i.e., each thread finds the neighbor list of an atom). If not specified, the default value of this parameter is determined based on the GPU architecture at runtime.

The *bond/chunk/size* keyword sets the number of iterations that a work item is scheduled for the bond force kernel on the GPU. The default value of this parameter is determined based on the GPU architecture at runtime.

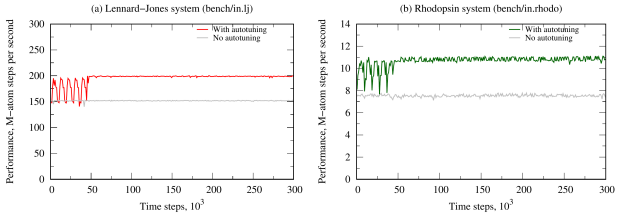
Added in version 4Jul2026.

The *auto/tuning* keyword enables the auto-tuning feature of the KOKKOS package when using GPUs. The following KOKKOS styles currently support auto-tuning:

- *bond/class2/kk*
- *bond/fene/kk*
- *bond/harmonic/kk*
- *lj/cut/kk*
- *lj/cut/coul/long/kk*
- *lj/charmm/coul/long/kk*
- *lj/cut/dipole/cut/kk*
- *dpd/kk*
- *sw/kk*
- *tersoff/kk*
- *zbl/kk*

These styles serve as templates for incorporating auto-tuners into other KOKKOS styles in the future. To avoid duplicated codes, one possibility is to refactor the tuners so that KOKKOS styles can request and enable them through a consistent API.

It is recommended that auto-tuning is enabled only when the simulated system is in steady state. The performance gain due to auto-tuning typically varies with the KOKKOS styles and the simulated system. For the benchmark systems *bench/in.lj* and *bench/in.rhodo*, the speedup vs without auto-tuning could be 30% and 50%, respectively (figure below).




When enabled, the tuner of the KOKKOS styles in use will scan through the possible values of kernel launch parameters, such as *pair/team/size threads/per/atom* for pair styles, *bond/chunk/size* for bond styles, and *nbin/atoms/per/bin* for neighbor builds. When the scanning completes, the tuner stores the best overall performance (in terms of the number of time steps per second) with the corresponding kernel launch parameter combination.

The tuner then uses the optimal parameter combination to launch the kernels on the GPU and monitors the simulation performance periodically. If the performance repeatedly drops below a certain relative tolerance from the last stored optimal value, the tuner may attempt to find the new optimal parameter combination by rerunning the scanning process.

There could be multiple tuners active in a run, for instance, when *bench/in.rhodo* is run on the GPUs, both *lj/charmm/coul/long/kk* and *bond/harmonic/kk* have their tuners activated. During the run, each tuner writes the current parameter combination and measured performance to a file named *tuning-[style-name].log*.

The following parameters are needed for the auto-tuning process.

**nevery:** controls the interval used to estimate the overall performance for a combination of these two parameters. *nevery* needs to be large enough to have a stable estimate of the performance, to achieve a sufficiently large number of kernel calls, while small enough to reduce the time required for scanning over all the combinations. *nevery* = 100 is usually a reasonable value.

**nsamples:** indicates the number of samples the tuners will collect for each parameter combination. *nsamples* = 5 is usually a reasonable value.

**mode:** determines how the performance of a collection of samples is *determined: max* means the maximum value, *ave* means the arithmetic *average value, and *median* means the median value.

**reltol:** sets the relative tolerance for performance degradation compared to the last optimal performance, which may trigger a re-scan of the parameter space. Setting *reltol* to be equal or greater than 1.0 will disable re-scanning.

For example, suppose that *nevery* = 100, *nsamples* = 5, *mode* = median and *reltol* = 0.2. For a typical pair style, the tuner scans over 90 parameter combinations (*pair/team/size*, *threads/per/atom*): 15 values of *pair/team/size* starting from 64 to 512 incremental by 32, and 6 values of *threads/per/atom*. For each parameter combination, the tuner monitors and saves the performance after 100 steps. When a scan over 90 combinations completes (9000 steps in this example), the best performance value is stored as a sample. After *nsamples* = 5 samples are collected, the median value is stored and the optimal parameter combination (*pair/team/size*, *threads/per/atom*) will be used for the next kernel launches. The figures above illustrate the simulation performance over time with auto-tuning using this setting for *bench/in.lj* (left panel) and *bench/in.rhodo* (right panel). The plateau regions correspond to the regime where the kernels are launched with the corresponding optimal parameter combinations.

Suppose that the last stored best performance is 1000 time steps per second. After the optimal parameter combination is locked in, the tuner keeps monitoring the performance every *nevery* = 100 time steps. If the performance drops below (1 - *reltol*) * 1000 = 800 time steps per second by *nsamples* = 5 times (not necessarily consecutively) the tuner will trigger a re-scanning over the parameter combinations to find a new optimal parameter combination.

If *nevery* is 0, auto-tuning is effectively disabled. The 3 parameters *nsamples*, *mode* and *reltol* still need to be specified but have no effect to the run.

One can also use the output in the log files *tuning-[style-name].log* as hints to select a combination of *pair/team/size and *threads/per/atom* to specify at command line arguments without enabling auto-tuning.

**Note**

Do not use *auto/tuning* in tandem with *pair/team/size* and/or *threads/per/atom* in the same command line because the latter override the scanning process of the tuners.

### OPENMP package settings

The *omp* style invokes settings associated with the use of the OPENMP package.

The *Nthreads* argument sets the number of OpenMP threads allocated for each MPI task. For example, if your system has nodes with dual quad-core processors, it has a total of 8 cores per node. You could use two MPI tasks per node (e.g. using the -ppn option of the mpirun command in MPICH or -npernode in OpenMPI), and set *Nthreads* = 4. This

would use all 8 cores on each node. Note that the product of MPI tasks * threads/task should not exceed the physical number of cores (on a node), otherwise performance will suffer.

Setting *Nthreads* = 0 instructs LAMMPS to use whatever value is the default for the given OpenMP environment. This is usually determined via the *OMP_NUM_THREADS* environment variable or the compiler runtime. Note that in most cases the default for OpenMP capable compilers is to use one thread for each available CPU core when *OMP_NUM_THREADS* is not explicitly set, which can lead to poor performance.

Here are examples of how to set the environment variable when launching LAMMPS:

```
env OMP_NUM_THREADS=4 lmp_machine -sf omp -in in.script
env OMP_NUM_THREADS=2 mpirun -np 2 lmp_machine -sf omp -in in.script
mpirun -x OMP_NUM_THREADS=2 -np 2 lmp_machine -sf omp -in in.script
```

or you can set it permanently in your shell's start-up script. All three of these examples use a total of 4 CPU cores.

Note that different MPI implementations have different ways of passing the OMP_NUM_THREADS environment variable to all MPI processes. The second example line above is for MPICH; the third example line with -x is for OpenMPI. Check your MPI documentation for additional details.

What combination of threads and MPI tasks gives the best performance is difficult to predict and can depend on many components of your input. Not all features of LAMMPS support OpenMP threading via the OPENMP package and the parallel efficiency can be very different, too.

**Note**

If you build LAMMPS with the GPU, INTEL, and / or OPENMP packages, be aware these packages all allow setting of the *Nthreads* value via their package commands, but there is only a single global *Nthreads* value used by OpenMP. Thus if multiple package commands are invoked, you should ensure the values are consistent. If they are not, the last one invoked will take precedence, for all packages. Also note that if the *-sf hybrid intel omp command-line switch* is used, it invokes a "package intel" command, followed by a "package omp" command, both with a setting of *Nthreads* = 0. Likewise for a hybrid suffix for gpu and omp. Note that KOKKOS also supports setting the number of OpenMP threads from the command-line using the "-k on" *command-line switch*. The default for KOKKOS is 1 thread per MPI task, so any other number of threads should be explicitly set using the "-k on" command-line switch (and this setting should be consistent with settings from any other packages used).

Optional keyword/value pairs can also be specified. Each has a default value as listed below.

The *neigh* keyword specifies whether neighbor list building will be multi-threaded in addition to force calculations. If *neigh* is set to *no* then neighbor list calculation is performed only by MPI tasks with no OpenMP threading. If *mode* is *yes* (the default), a multi-threaded neighbor list build is used. Using *neigh* = *yes* is almost always faster and should produce identical neighbor lists at the expense of using more memory. Specifically, neighbor list pages are allocated for all threads at the same time and each thread works within its own pages.

---

### 1.73.4 Restrictions

This command cannot be used after the simulation box is defined by a *read_data* or *create_box* command.

The *gpu* style of this command can only be invoked if LAMMPS was built with the GPU package. See the *Build package* doc page for more info.

The *intel* style of this command can only be invoked if LAMMPS was built with the INTEL package. See the *Build package* page for more info.

The *kokkos* style of this command can only be invoked if LAMMPS was built with the KOKKOS package. See the *Build package* doc page for more info.

The *omp* style of this command can only be invoked if LAMMPS was built with the OPENMP package. See the *Build package* doc page for more info.

### 1.73.5 Related commands

*suffix*, *-pk command-line switch*

### 1.73.6 Defaults

For the GPU package, the default parameters and settings are:

```
Ngpu = 0, neigh = yes, newton = off, binsize = 0.0, split = 1.0, gpuID = 0 to Ngpu-1,␣
↪tpa = 1, omp = 0, platform=-1.
```

These settings are made automatically if the "-sf gpu" *command-line switch* is used. If it is not used, you must invoke the package gpu command in your input script or via the "-pk gpu" *command-line switch*.

For the INTEL package, the default parameters and settings are:

```
omp = 0, mode = mixed, lrt = no, pppm_table = yes
```

These settings are made automatically if the "-sf intel" *command-line switch* is used. If it is not used, you must invoke the package intel command in your input script or via the "-pk intel" *command-line switch*.

For the KOKKOS package when using GPUs, the option defaults are:

```
neigh = full, neigh/qeq = full, newton = off, binsize = 2x LAMMPS default value, comm =␣
↪device, sort = device, atom/map = device, neigh/transpose = off, gpu/aware = on, auto/
↪tuning = disabled
```

For GPUs, option neigh/thread = on when there are 16k atoms or less on an MPI rank, otherwise it is "off". When LAMMPS can safely detect that GPU-aware MPI is not available, the default value of gpu/aware becomes "off".

For the KOKKOS package when using CPUs, the option defaults are:

```
neigh = half, neigh/qeq = half, newton = on, binsize = 0.0, comm = no, sort = no, atom/
↪map = no
```

These settings are made automatically by the required "-k on" *command-line switch*. You can change them by using the package kokkos command in your input script or via the *-pk kokkos command-line switch*.

For the OMP package, the defaults are

```
Nthreads = 0, neigh = yes
```

These settings are made automatically if the "-sf omp" *command-line switch* is used. If it is not used, you must invoke the package omp command in your input script or via the "-pk omp" *command-line switch*.

## 1.74 pair_coeff command

### 1.74.1 Syntax

```
pair_coeff I J args
```

- I,J = numeric atom types (see asterisk form below), or type labels
- args = coefficients for one or more pairs of atom types

### 1.74.2 Examples

```
pair_coeff 1 2 1.0 1.0 2.5
pair_coeff 2 * 1.0 1.0
pair_coeff 3* 1*2 1.0 1.0 2.5
pair_coeff * * 1.0 1.0
pair_coeff * * nialhjea 1 1 2
pair_coeff * 3 morse.table ENTRY1
pair_coeff 1 2 lj/cut 1.0 1.0 2.5 # (for pair_style hybrid)

labelmap atom 1 C
labelmap atom 2 H
pair_coeff C H 1.0 1.0 2.5
```

### 1.74.3 Description

Specify the pairwise force field coefficients for one or more pairs of atom types. The number and meaning of the coefficients depends on the pair style. Pair coefficients can also be set in the data file read by the *read_data* command or in a restart file.

I and J can be specified in one of several ways. Explicit numeric values can be used for each, as in the first example above. Or, one or both of the types in the I,J pair can be a type label, which is an alphanumeric string defined by the *labelmap* command or in a section of a data file read by the *read_data* command, and which converts internally to a numeric type. Internally, LAMMPS will set coefficients for the symmetric J,I interaction to the same values as the I,J interaction.

For numeric values only, a wildcard asterisk can be used in place of or in conjunction with the I,J arguments to set the coefficients for multiple pairs of atom types. This takes the form "*" or "*n" or "n*" or "m*n". If $N$ is the number of atom types, then an asterisk with no numeric values means all types from 1 to $N$. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to $N$ (inclusive). A middle asterisk means all types from m to n (inclusive). For the asterisk syntax, only type pairs with I <= J are considered; if asterisks imply type pairs where J < I, they are ignored. Again internally, LAMMPS will set the coefficients for the symmetric J,I interactions to the same values as the I <= J interactions.

Note that a pair_coeff command can override a previous setting for the same I,J pair. For example, these commands set the coeffs for all I,J pairs, then overwrite the coeffs for just the I,J = 2,3 pair:

```
pair_coeff * * 1.0 1.0 2.5
pair_coeff 2 3 2.0 1.0 1.12
```

A line in a data file that specifies pair coefficients uses the exact same format as the arguments of the pair_coeff command in an input script, with the exception of the I,J type arguments. In each line of the "Pair Coeffs" section of a data file, only a single type I is specified, which sets the coefficients for type I interacting with type I. This is because the section has exactly $N$ lines, where $N$ is the number of atom types. For this reason, the wild-card asterisk should also not be used as part of the I argument. Thus in a data file, the line corresponding to the first example above would be listed as

```
2 1.0 1.0 2.5
```

For many potentials, if coefficients for type pairs with I != J are not set explicitly by a pair_coeff command, the values are inferred from the I,I and J,J settings by mixing rules; see the *pair_modify* command for a discussion. Details on this option as it pertains to individual potentials are described on the page for the potential.

Many pair styles, typically for many-body potentials, use tabulated potential files as input, when specifying the pair_coeff command. Potential files provided with LAMMPS are in the potentials directory of the distribution. For some potentials, such as EAM, other archives of suitable files can be found on the Web. They can be used with LAMMPS so long as they are in the format LAMMPS expects, as discussed on the individual doc pages. The first line

of potential files may contain metadata with upper case tags followed their value. These may be parsed and used by LAMMPS. Currently supported are the "DATE:" tag and the `UNITS:` tag. For pair styles that have been programmed to support the metadata, the value of the "DATE:" tag is printed to the screen and logfile so that the version of a potential file can be later identified. The `UNITS:` tag indicates the *units* setting required for this particular potential file. If the potential file was created for a different sets of units, LAMMPS will terminate with an error. If the potential file does not contain the tag, no check will be made and it is the responsibility of the user to determine that the unit style is correct.

In some select cases and for specific combinations of unit styles, LAMMPS is capable of automatically converting potential parameters from a file. In those cases, a warning message signaling that an automatic conversion has happened is printed to the screen.

When a pair_coeff command using a potential file is specified, LAMMPS looks for the potential file in 2 places. First it looks in the location specified. E.g. if the file is specified as "niu3.eam", it is looked for in the current working directory. If it is specified as "../potentials/niu3.eam", then it is looked for in the potentials directory, assuming it is a sister directory of the current working directory. If the file is not found, it is then looked for in one of the directories specified by the `LAMMPS_POTENTIALS` environment variable. Thus if this is set to the potentials directory in the LAMMPS distribution, then you can use those files from anywhere on your system, without copying them into your working directory. Environment variables are set in different ways for different shells. Here are example settings for

csh, tcsh:

```
setenv LAMMPS_POTENTIALS /path/to/lammps/potentials
```

bash:

```
export LAMMPS_POTENTIALS=/path/to/lammps/potentials
```

Windows:

```
set LAMMPS_POTENTIALS="C:\\Path to LAMMPS\\Potentials"
```

The `LAMMPS_POTENTIALS` environment variable may contain paths to multiple folders, if they are separated by ";" on Windows and ":" on all other operating systems, just like the `PATH` and similar environment variables.

---

The alphabetic list of pair styles defined in LAMMPS is given on the *pair_style* doc page. They are also listed in more compact form on the *Commands pair* doc page.

Click on the style to display the formula it computes and its coefficients as specified by the associated pair_coeff command.

---

### 1.74.4 Restrictions

This command must come after the simulation box is defined by a *read_data*, *read_restart*, or *create_box* command.

### 1.74.5 Related commands

*pair_style*, *pair_modify*, *read_data*, *read_restart*, *pair_write*

### 1.74.6 Default

none

# 1.75 pair_modify command

## 1.75.1 Syntax

```
pair_modify keyword values ...
```

- one or more keyword/value pairs may be listed
- keyword = *pair* or *shift* or *mix* or *table* or *table/disp* or *tabinner* or *tabinner/disp* or *tail* or *compute* or *nofdotr* or *special* or *compute/tally* or *neigh/trim*

```
pair value = sub-style N
  sub-style = sub-style of pair hybrid
  N = which instance of sub-style (1 to M), only specify if sub-style is used␣
↪multiple times
mix value = geometric or arithmetic or sixthpower
shift value = yes or no
table value = N
  2^N = # of values in table
table/disp value = N
  2^N = # of values in table
tabinner value = cutoff
  cutoff = inner cutoff at which to begin table (distance units)
tabinner/disp value = cutoff
  cutoff = inner cutoff at which to begin table (distance units)
tail value = yes or no
compute value = yes or no
nofdotr value = none
special values = which wt1 wt2 wt3
   which = lj/coul or lj or coul
   w1,w2,w3 = 1-2, 1-3, 1-4 weights from 0.0 to 1.0 inclusive
compute/tally value = yes or no
neigh/trim value = yes or no
```

## 1.75.2 Examples

```
pair_modify shift yes mix geometric
pair_modify tail yes
pair_modify table 12
pair_modify pair lj/cut compute no
pair_modify pair tersoff compute/tally no
pair_modify pair lj/cut/coul/long 1 special lj/coul 0.0 0.0 0.0
pair_modify pair lj/cut/coul/long special lj 0.0 0.0 0.5 special coul 0.0 0.0 0.8333333
```

## 1.75.3 Description

Modify the parameters of the currently defined pair style. If the pair style is *hybrid or hybrid/overlay*, then the specified parameters are by default modified for all the hybrid sub-styles.

**Note**

The behavior for hybrid pair styles can be changed by using the *pair* keyword, which allows selection of a specific sub-style to apply all remaining keywords to. The *special* and *compute/tally* keywords can **only** be used in

conjunction with the *pair* keyword. See further details about these 3 keywords below.

The *mix* keyword affects pair coefficients for interactions between atoms of type I and J, when I != J and the coefficients are not explicitly set in the input script. Note that coefficients for I = J must be set explicitly, either in the input script via the *pair_coeff* command or in the "Pair Coeffs" or "PairIJ Coeffs" sections of the *data file*. For some pair styles it is not necessary to specify coefficients when I != J, since a "mixing" rule will create them from the I,I and J,J settings. The pair_modify *mix* value determines what formulas are used to compute the mixed coefficients. In each case, the cutoff distance is mixed the same way as sigma.

Note that not all pair styles support mixing and some mix options are not available for certain pair styles. Also, there are additional restrictions when using *pair style hybrid or hybrid/overlay*. See the page for individual pair styles for those restrictions. Note also that the *pair_coeff* command also can be used to directly set coefficients for a specific I != J pairing, in which case no mixing is performed. If possible, LAMMPS will print an informational message about how many of the mixed pair coefficients were generated and which mixing rule was applied.

- mix *geometric*

$$\varepsilon_{ij} = \sqrt{\varepsilon_i \varepsilon_j}$$
$$\sigma_{ij} = \sqrt{\sigma_i \sigma_j}$$

- mix *arithmetic*

$$\varepsilon_{ij} = \sqrt{\varepsilon_i \varepsilon_j}$$
$$\sigma_{ij} = \frac{1}{2}(\sigma_i + \sigma_j)$$

- mix *sixthpower*

$$\varepsilon_{ij} = \frac{2\sqrt{\varepsilon_i \varepsilon_j}\sigma_i^3 \sigma_j^3}{\sigma_i^6 + \sigma_j^6}$$
$$\sigma_{ij} = \left(\frac{1}{2}(\sigma_i^6 + \sigma_j^6)\right)^{\frac{1}{6}}$$

The *shift* keyword determines whether a Lennard-Jones potential is shifted at its cutoff to 0.0. If so, this adds an energy term to each pairwise interaction which will be included in the thermodynamic output, but does not affect pair forces or atom trajectories. See the doc page for individual pair styles to see which ones support this option.

The *table* and *table/disp* keywords apply to pair styles with a long-range Coulombic term or long-range dispersion term respectively; see the page for individual styles to see which potentials support these options. If N is non-zero, a table of length 2^N is pre-computed for forces and energies, which can shrink their computational cost by up to a factor of 2. The table is indexed via a bit-mapping technique *(Wolff)* and a linear interpolation is performed between adjacent table values. In our experiments with different table styles (lookup, linear, spline), this method typically gave the best performance in terms of speed and accuracy.

The choice of table length is a tradeoff in accuracy versus speed. A larger N yields more accurate force computations, but requires more memory which can slow down the computation due to cache misses. A reasonable value of N is between 8 and 16. The default value of 12 (table of length 4096) gives approximately the same accuracy as the no-table (N = 0) option. For N = 0, forces and energies are computed directly, using a polynomial fit for the needed erfc() function evaluation, which is what earlier versions of LAMMPS did. Values greater than 16 typically slow down the simulation and will not improve accuracy; values from 1 to 8 give unreliable results.

The *tabinner* and *tabinner/disp* keywords set an inner cutoff above which the pairwise computation is done by table lookup (if tables are invoked), for the corresponding Coulombic and dispersion tables discussed with the *table* and *table/disp* keywords. The smaller the cutoff is set, the less accurate the table becomes (for a given number of table

values), which can require use of larger tables. The default cutoff value is sqrt(2.0) distance units which means nearly all pairwise interactions are computed via table lookup for simulations with "real" units, but some close pairs may be computed directly (non-table) for simulations with "lj" units.

When the *tail* keyword is set to *yes*, certain pair styles will add a long-range VanderWaals tail "correction" to the energy and pressure. These corrections are bookkeeping terms which do not affect dynamics, unless a constant-pressure simulation is being performed. See the page for individual styles to see which support this option. These corrections are included in the calculation and printing of thermodynamic quantities (see the *thermo_style* command). Their effect will also be included in constant NPT or NPH simulations where the pressure influences the simulation box dimensions (e.g. the *fix npt* and *fix nph* commands). The formulas used for the long-range corrections come from equation 5 of *(Sun)*.

**Note**

The tail correction terms are computed at the beginning of each run, using the current atom counts of each atom type. If atoms are deleted (or lost) or created during a simulation, e.g. via the *fix gcmc* command, the correction factors are not re-computed. If you expect the counts to change dramatically, you can break a run into a series of shorter runs so that the correction factors are re-computed more frequently.

Several additional assumptions are inherent in using tail corrections, including the following:

- The simulated system is a 3d bulk homogeneous liquid. This option should not be used for systems that are non-liquid, 2d, have a slab geometry (only 2d periodic), or inhomogeneous.
- G(r), the radial distribution function (rdf), is unity beyond the cutoff, so a fairly large cutoff should be used (i.e. 2.5 sigma for an LJ fluid), and it is probably a good idea to verify this assumption by checking the rdf. The rdf is not exactly unity beyond the cutoff for each pair of interaction types, so the tail correction is necessarily an approximation.

  The tail corrections are computed at the beginning of each simulation run. If the number of atoms changes during the run, e.g. due to atoms leaving the simulation domain, or use of the *fix gcmc* command, then the corrections are not updated to reflect the changed atom count. If this is a large effect in your simulation, you should break the long run into several short runs, so that the correction factors are re-computed multiple times.
- Thermophysical properties obtained from calculations with this option enabled will not be thermodynamically consistent with the truncated force-field that was used. In other words, atoms do not feel any LJ pair interactions beyond the cutoff, but the energy and pressure reported by the simulation include an estimated contribution from those interactions.

The *compute* keyword allows pairwise computations to be turned off, even though a *pair_style* is defined. This is not useful for running a real simulation, but can be useful for debugging purposes or for performing a *rerun* simulation, when you only wish to compute partial forces that do not include the pairwise contribution.

Two examples are as follows. First, this option allows you to perform a simulation with *pair_style hybrid* with only a subset of the hybrid sub-styles enabled. Second, this option allows you to perform a simulation with only long-range interactions but no short-range pairwise interactions. Doing this by simply not defining a pair style will not work, because the *kspace_style* command requires a Kspace-compatible pair style be defined.

The *nofdotr* keyword allows to disable an optimization that computes the global stress tensor from the total forces and atom positions rather than from summing forces between individual pairs of atoms.

---

The *pair* keyword can only be used with the *hybrid and hybrid/overlay* pair styles. If used, it must appear first in the list of keywords.

Its meaning is that all the following parameters will only be modified for the specified sub-style. If the sub-style is defined multiple times, then an additional numeric argument *N* must also be specified, which is a number from 1 to M

where M is the number of times the sub-style was listed in the *pair_style hybrid* command. The extra number indicates which instance of the sub-style the remaining keywords will be applied to.

The *special* and *compute/tally* keywords can **only** be used in conjunction with the *pair* keyword and they must directly follow it. I.e. any other keyword, must appear after *pair*, *special*, and *compute/tally*.

The *special* keyword overrides the global *special_bonds* 1-2, 1-3, 1-4 exclusion settings (weights) for the sub-style selected by the *pair* keyword.

Similar to the *special_bonds* command, it takes 4 arguments. The *which* argument can be *lj* to change only the non-Coulomb weights (e.g. Lennard-Jones or Buckingham), *coul* to change only the Coulombic settings, or *lj/coul* to change both to the same values. The *wt1,wt2,wt3* values are numeric weights from 0.0 to 1.0 inclusive, for the 1-2, 1-3, and 1-4 bond topology neighbors, respectively. The *special* keyword can be used multiple times, e.g. to set the *lj* and *coul* settings to different values.

**Note**

The *special* keyword is not compatible with pair styles from the GPU or the INTEL package and attempting to use it will cause an error.

**Note**

Weights of exactly 0.0 or 1.0 in the *special_bonds* command have implications on the neighbor list construction, which means that they cannot be overridden by using the *special* keyword. One workaround for this restriction is to use the *special_bonds* command with weights like 1.0e-10 or 0.999999999 instead of 0.0 or 1.0, respectively, which enables to reset each them to any value between 0.0 and 1.0 inclusively. Otherwise you can set **all** global weights to an arbitrary number between 0.0 or 1.0, like 0.5, and then you have to override **all** *special* settings for **all** sub-styles which use the 1-2, 1-3, and 1-4 exclusion weights in their force/energy computation.

The *compute/tally* keyword disables or enables registering *compute */tally* computes for the sub-style specified by the *pair* keyword. Use *no* to disable, or *yes* to enable.

**Note**

The "pair_modify pair compute/tally" command must be issued **before** the corresponding compute style is defined.

Added in version 3Aug2022.

The *neigh/trim* keyword controls whether an explicit cutoff is set for each neighbor list request issued by individual pair sub-styles when using *pair hybrid/overlay*. When this keyword is set to *no*, then the cutoff of each pair sub-style neighbor list will be set equal to the largest cutoff, even if a shorter cutoff is specified for a particular sub-style. If possible the neighbor list will be copied directly from another list. When this keyword is set to *yes* then the cutoff of the neighbor list will be explicitly set to the value requested by the pair sub-style, and if possible the list will be created by trimming neighbors from another list with a longer cutoff, otherwise a new neighbor list will be created with the specified cutoff. The *yes* option can be faster when there are multiple pair styles with different cutoffs since the number of pair-wise distance checks between neighbors is reduced (but the time required to build the neighbor lists is increased). The *no* option could be faster when two or more neighbor lists have similar (but not exactly the same) cutoffs.

**Note**

The "pair_modify neigh/trim" command *only* applies when there are multiple pair sub-styles for the same atoms

with different cutoffs, i.e. when using pair style hybrid/overlay. If you have different cutoffs for different pairs for atoms type, the *neighbor style multi* should be used to create optimized neighbor lists.

---

### 1.75.4 Restrictions

You cannot use *shift* yes with *tail* yes, since those are conflicting options. You cannot use *tail* yes with 2d simulations. You cannot use *special* with pair styles from the GPU or INTEL package.

### 1.75.5 Related commands

*pair_style*, *pair_style hybrid*, *pair_coeff*, *thermo_style*, *compute */tally*, *neighbor multi*

### 1.75.6 Default

The option defaults are mix = geometric, shift = no, table = 12, tabinner = sqrt(2.0), tail = no, compute = yes, and neigh/trim yes.

Note that some pair styles perform mixing, but only a certain style of mixing. See the doc pages for individual pair styles for details.

---

**(Wolff)** Wolff and Rudd, Comp Phys Comm, 120, 200-32 (1999).

**(Sun)** Sun, J Phys Chem B, 102, 7338-7364 (1998).

## 1.76 pair_style command

### 1.76.1 Syntax

```
pair_style style args
```

- style = one of the styles from the list below
- args = arguments used by a particular style

### 1.76.2 Examples

```
pair_style lj/cut 2.5
pair_style eam/alloy
pair_style hybrid lj/charmm/coul/long 10.0 eam
pair_style table linear 1000
pair_style none
```

### 1.76.3 Description

Set the formula(s) LAMMPS uses to compute pairwise interactions. In LAMMPS, pair potentials are defined between pairs of atoms that are within a cutoff distance and the set of active interactions typically changes over time. See the *bond_style* command to define potentials between pairs of bonded atoms, which typically remain in place for the duration of a simulation.

In LAMMPS, pairwise force fields encompass a variety of interactions, some of which include many-body effects, e.g. EAM, Stillinger-Weber, Tersoff, REBO potentials. They are still classified as "pairwise" potentials because the set of

interacting atoms changes with time (unlike molecular bonds) and thus a neighbor list is used to find nearby interacting atoms.

Hybrid models where specified pairs of atom types interact via different pair potentials can be setup using the *hybrid* pair style.

The coefficients associated with a pair style are typically set for each pair of atom types, and are specified by the *pair_coeff* command or read from a file by the *read_data* or *read_restart* commands.

The *pair_modify* command sets options for mixing of type I-J interaction coefficients and adding energy offsets or tail corrections to Lennard-Jones potentials. Details on these options as they pertain to individual potentials are described on the doc page for the potential. Likewise, info on whether the potential information is stored in a *restart file* is listed on the potential doc page.

In the formulas listed for each pair style, $E$ is the energy of a pairwise interaction between two atoms separated by a distance $r$. The force between the atoms is the negative derivative of this expression.

If the pair_style command has a cutoff argument, it sets global cutoffs for all pairs of atom types. The distance(s) can be smaller or larger than the dimensions of the simulation box.

In many cases, the global cutoff value can be overridden for a specific pair of atom types by the *pair_coeff* command.

If a new pair_style command is specified with a new style, all previous *pair_coeff* and *pair_modify* command settings are erased; those commands must be re-specified if necessary.

If a new pair_style command is specified with the same style, then only the global settings in that command are reset. Any previous doc:*pair_coeff* <*pair_coeff*> and *pair_modify* command settings are preserved. The only exception is that if the global cutoff in the pair_style command is changed, it will override the corresponding cutoff in any of the previous *pair_modify* commands.

Two pair styles which do not follow this rule are the pair_style *table* and *hybrid* commands. A new pair_style command for these styles will wipe out all previously specified *pair_coeff* and *pair_modify* settings, including for the sub-styles of the *hybrid* command.

---

Here is an alphabetic list of pair styles defined in LAMMPS. They are also listed in more compact form on the *Commands pair* doc page.

Click on the style to display the formula it computes, any additional arguments specified in the pair_style command, and coefficients specified by the associated *pair_coeff* command.

There are also additional accelerated pair styles included in the LAMMPS distribution for faster performance on CPUs, GPUs, and KNLs. The individual style names on the *Commands pair* doc page are followed by one or more of (g,i,k,o,t) to indicate which accelerated styles exist.

- *none* - turn off pairwise interactions
- *hybrid* - multiple styles of pairwise interactions
- *hybrid/molecular* - different pair styles for intra- and inter-molecular interactions
- *hybrid/overlay* - multiple styles of superposed pairwise interactions
- *hybrid/scaled* - multiple styles of scaled superposed pairwise interactions
- *zero* - neighbor list but no interactions
- *adp* - angular dependent potential (ADP) of Mishin
- *agni* - AGNI machine-learning potential
- *aip/water/2dm* - anisotropic interfacial potential for water in 2d geometries
- *airebo* - AIREBO potential of Stuart

- *airebo/bc* - AIREBO with bond-centric P_CC (Hur and Stuart)
- *airebo/morse* - AIREBO with Morse instead of LJ
- *amoeba* -
- *atm* - Axilrod-Teller-Muto potential
- *beck* - Beck potential
- *body/nparticle* - interactions between body particles
- *body/rounded/polygon* - granular-style 2d polygon potential
- *body/rounded/polyhedron* - granular-style 3d polyhedron potential
- *bondval* - Bond-Valence potential
- *bondval/vec* - Bond-Valence vector potential
- *bop* - BOP potential of Pettifor
- *born* - Born-Mayer-Huggins potential
- *born/coul/dsf* - Born with damped-shifted-force model
- *born/coul/dsf/cs* - Born with damped-shifted-force and core/shell model
- *born/coul/long* - Born with long-range Coulomb
- *born/coul/long/cs* - Born with long-range Coulomb and core/shell
- *born/coul/msm* - Born with long-range MSM Coulomb
- *born/coul/wolf* - Born with Wolf potential for Coulomb
- *born/coul/wolf/cs* - Born with Wolf potential for Coulomb and core/shell model
- *born/gauss* - Born-Mayer / Gaussian potential
- *bpm/spring* - repulsive harmonic force with damping
- *brownian* - Brownian potential for Fast Lubrication Dynamics
- *brownian/poly* - Brownian potential for Fast Lubrication Dynamics with polydispersity
- *buck* - Buckingham potential
- *buck/coul/cut* - Buckingham with cutoff Coulomb
- *buck/coul/long* - Buckingham with long-range Coulomb
- *buck/coul/long/cs* - Buckingham with long-range Coulomb and core/shell
- *buck/coul/msm* - Buckingham with long-range MSM Coulomb
- *buck/long/coul/long* - long-range Buckingham with long-range Coulomb
- *buck/mdf* - Buckingham with a taper function
- *buck6d/coul/gauss/dsf* - dispersion-damped Buckingham with damped-shift-force model
- *buck6d/coul/gauss/long* - dispersion-damped Buckingham with long-range Coulomb
- *colloid* - integrated colloidal potential
- *comb* - charge-optimized many-body (COMB) potential
- *comb3* - charge-optimized many-body (COMB3) potential
- *cosine/squared* - Cooke-Kremer-Deserno membrane model potential

- *coul/ctip* - Charge Transfer Interatomic (Coulomb) Potential
- *coul/cut* - cutoff Coulomb potential
- *coul/cut/dielectric* -
- *coul/cut/global* - cutoff Coulomb potential
- *coul/cut/soft* - Coulomb potential with a soft core
- *coul/cut/soft/gapsys* - Coulomb potential with a soft core with continuous force and first derivative of force
- *coul/debye* - cutoff Coulomb potential with Debye screening
- *coul/diel* - Coulomb potential with dielectric permittivity
- *coul/dsf* - Coulomb with damped-shifted-force model
- *coul/esp* - long-range Coulomb potential for ESP kspace solver
- *coul/exclude* - subtract Coulomb potential for excluded pairs
- *coul/long* - long-range Coulomb potential
- *coul/long/cs* - long-range Coulomb potential and core/shell
- *coul/long/dielectric* -
- *coul/long/soft* - long-range Coulomb potential with a soft core
- *coul/msm* - long-range MSM Coulomb
- *coul/slater/cut* - smeared out Coulomb
- *coul/slater/long* - long-range smeared out Coulomb
- *coul/shield* - Coulomb for boron nitride for use with *ilp/graphene/hbn* potential
- *coul/streitz* - Coulomb via Streitz/Mintmire Slater orbitals
- *coul/tt* - damped charge-dipole Coulomb for Drude dipoles
- *coul/wolf* - Coulomb via Wolf potential
- *coul/wolf/cs* - Coulomb via Wolf potential with core/shell adjustments
- *dispersion/d3* - Dispersion correction for potentials derived from DFT functionals
- *dpd* - dissipative particle dynamics (DPD)
- *dpd/coul/slater/long* - dissipative particle dynamics (DPD) with electrostatic interactions
- *dpd/ext* - generalized force field for DPD
- *dpd/ext/tstat* - pairwise DPD thermostatting with generalized force field
- *dpd/fdt* - DPD for constant temperature and pressure
- *dpd/fdt/energy* - DPD for constant energy and enthalpy
- *dpd/tstat* - pairwise DPD thermostatting
- *dsmc* - Direct Simulation Monte Carlo (DSMC)
- *e3b* - Explicit-three body (E3B) water model
- *drip* - Dihedral-angle-corrected registry-dependent interlayer potential (DRIP)
- *eam* - embedded atom method (EAM)
- *eam/alloy* - alloy EAM

- *eam/cd* - concentration-dependent EAM
- *eam/cd/old* - older two-site model for concentration-dependent EAM
- *eam/fs* - Finnis-Sinclair EAM
- *eam/fs/apip* - *adaptive precision* version of FS EAM, used as fast potential
- *eam/he* - Finnis-Sinclair EAM modified for Helium in metals
- *eam/apip* - *adaptive-precision* version of EAM, used as fast potential
- *edip* - three-body EDIP potential
- *edip/multi* - multi-element EDIP potential
- *edpd* - eDPD particle interactions
- *eff/cut* - electron force field with a cutoff
- *eim* - embedded ion method (EIM)
- *exp6/rx* - reactive DPD potential
- *extep* - extended Tersoff potential
- *gauss* - Gaussian potential
- *gauss/cut* - generalized Gaussian potential
- *gayberne* - Gay-Berne ellipsoidal potential
- *granular* - Generalized granular potential
- *granular/superellipsoid* - Generalized granular potential for superellipsoids
- *gran/hertz/history* - granular potential with Hertzian interactions
- *gran/hooke* - granular potential without history effects
- *gran/hooke/history* - granular potential with history effects
- *gw* - Gao-Weber potential
- *gw/zbl* - Gao-Weber potential with a repulsive ZBL core
- *harmonic/cut* - repulsive-only harmonic potential
- *hbond/dreiding/lj* - DREIDING hydrogen bonding LJ potential
- *hbond/dreiding/lj/angleoffset* - DREIDING hydrogen bonding LJ potential with offset for hbond angle
- *hbond/dreiding/morse* - DREIDING hydrogen bonding Morse potential
- *hbond/dreiding/morse/angleoffset* - DREIDING hydrogen bonding Morse potential with offset for hbond angle
- *hdnnp* - High-dimensional neural network potential
- *hippo* -
- *ilp/graphene/hbn* - registry-dependent interlayer potential (ILP)
- *ilp/tmd* - interlayer potential (ILP) potential for transition metal dichalcogenides (TMD)
- *kim* - interface to potentials provided by KIM project
- *kolmogorov/crespi/full* - Kolmogorov-Crespi (KC) potential with no simplifications
- *kolmogorov/crespi/z* - Kolmogorov-Crespi (KC) potential with normals along z-axis

- *lambda/input/apip* - constant as input for the precision calculation of an *adaptive-precision interatomic potential (APIP)*
- *lambda/input/csp/apip* - CSP as input for the precision calculation of an *adaptive-precision interatomic potential (APIP)*
- *lambda/zone/apip* - transition zone of an *adaptive-precision interatomic potential*
- *lcbop* - long-range bond-order potential (LCBOP)
- *ldd* - Local density and gradient of density interactions (LDD)
- *lebedeva/z* - Lebedeva interlayer potential for graphene with normals along z-axis
- *lennard/mdf* - LJ potential in A/B form with a taper function
- *lepton* - pair potential from evaluating a string
- *lepton/coul* - pair potential from evaluating a string with support for charges
- *lepton/sphere* - pair potential from evaluating a string with support for radii
- *line/lj* - LJ potential between line segments
- *list* - potential between pairs of atoms explicitly listed in an input file
- *lj/charmm/coul/charmm* - CHARMM potential with cutoff Coulomb
- *lj/charmm/coul/charmm/implicit* - CHARMM for implicit solvent
- *lj/charmm/coul/long* - CHARMM with long-range Coulomb
- *lj/charmm/coul/long/soft* - CHARMM with long-range Coulomb and a soft core
- *lj/charmm/coul/msm* - CHARMM with long-range MSM Coulomb
- *lj/charmmfsw/coul/charmmfsh* - CHARMM with force switching and shifting
- *lj/charmmfsw/coul/long* - CHARMM with force switching and long-rnage Coulomb
- *lj/class2* - COMPASS (class 2) force field without Coulomb
- *lj/class2/coul/cut* - COMPASS with cutoff Coulomb
- *lj/class2/coul/cut/soft* - COMPASS with cutoff Coulomb with a soft core
- *lj/class2/coul/long* - COMPASS with long-range Coulomb
- *lj/class2/coul/long/cs* - COMPASS with long-range Coulomb with core/shell adjustments
- *lj/class2/coul/long/soft* - COMPASS with long-range Coulomb with a soft core
- *lj/class2/soft* - COMPASS (class 2) force field with no Coulomb with a soft core
- *lj/cubic* - LJ with cubic after inflection point
- *lj/cut* - cutoff Lennard-Jones potential without Coulomb
- *lj/cut/coul/cut* - LJ with cutoff Coulomb
- *lj/cut/coul/cut/dielectric* -
- *lj/cut/coul/cut/soft* - LJ with cutoff Coulomb with a soft core
- *lj/cut/coul/debye* - LJ with Debye screening added to Coulomb
- *lj/cut/coul/debye/dielectric* -
- *lj/cut/coul/dsf* - LJ with Coulomb via damped shifted forces
- *lj/cut/coul/esp* - LJ with long-range Coulomb for ESP kspace solver

- *lj/cut/coul/long* - LJ with long-range Coulomb
- *lj/cut/coul/long/cs* - LJ with long-range Coulomb with core/shell adjustments
- *lj/cut/coul/long/dielectric* -
- *lj/cut/coul/long/soft* - LJ with long-range Coulomb with a soft core
- *lj/cut/coul/msm* - LJ with long-range MSM Coulomb
- *lj/cut/coul/msm/dielectric* -
- *lj/cut/coul/wolf* - LJ with Coulomb via Wolf potential
- *lj/cut/dipole/cut* - point dipoles with cutoff
- *lj/cut/dipole/long* - point dipoles with long-range Ewald
- *lj/cut/soft* - LJ with a soft core
- *lj/cut/soft/gapsys* - LJ potential with a soft core with continuous force and first derivative of force
- *lj/cut/sphere* - LJ where per-atom radius is used as LJ sigma
- *lj/cut/thole/long* - LJ with Coulomb with thole damping
- *lj/cut/tip4p/cut* - LJ with cutoff Coulomb for TIP4P water
- *lj/cut/tip4p/long* - LJ with long-range Coulomb for TIP4P water
- *lj/cut/tip4p/long/soft* - LJ with cutoff Coulomb for TIP4P water with a soft core
- *lj/expand* - Lennard-Jones for variable size particles
- *lj/expand/coul/long* - Lennard-Jones for variable size particles with long-range Coulomb
- *lj/expand/sphere* - Variable size LJ where per-atom radius is used as delta (size)
- *lj/gromacs* - GROMACS-style Lennard-Jones potential
- *lj/gromacs/coul/gromacs* - GROMACS-style LJ and Coulomb potential
- *lj/long/coul/long* - long-range LJ and long-range Coulomb
- *lj/long/coul/long/dielectric* -
- *lj/long/dipole/long* - long-range LJ and long-range point dipoles
- *lj/long/tip4p/long* - long-range LJ and long-range Coulomb for TIP4P water
- *lj/mdf* - LJ potential with a taper function
- *lj/pirani* - Improved LJ potential
- *lj/relres* - LJ using multiscale Relative Resolution (RelRes) methodology *(Chaimovich)*.
- *lj/spica* - LJ for SPICA coarse-graining
- *lj/spica/coul/long* - LJ for SPICA coarse-graining with long-range Coulomb
- *lj/spica/coul/msm* - LJ for SPICA coarse-graining with long-range Coulomb via MSM
- *lj/sf/dipole/sf* - LJ with dipole interaction with shifted forces
- *lj/smooth* - smoothed Lennard-Jones potential
- *lj/smooth/linear* - linear smoothed LJ potential
- *lj/switch3/coulgauss/long* - smoothed LJ vdW potential with Gaussian electrostatics
- *lj96/cut* - Lennard-Jones 9/6 potential

- *local/density* - Generalized basic local density potential
- *lubricate* - Hydrodynamic lubrication forces
- *lubricate/poly* - Hydrodynamic lubrication forces with polydispersity
- *lubricateU* - Hydrodynamic lubrication forces for Fast Lubrication Dynamics
- *lubricateU/poly* - Hydrodynamic lubrication forces for Fast Lubrication with polydispersity
- *mbx* - Many-Body eXpansion (MBX) potential
- *mdpd* - mDPD particle interactions
- *mdpd/rhosum* - mDPD particle interactions for mass density
- *meam* - Modified embedded atom method (MEAM)
- *meam/ms* - Multi-state modified embedded atom method (MS-MEAM)
- *meam/spline* - Splined version of MEAM
- *meam/sw/spline* - Splined version of MEAM with a Stillinger-Weber term
- *mesocnt* - Mesoscopic vdW potential for (carbon) nanotubes
- *mesocnt/viscous* - Mesoscopic vdW potential for (carbon) nanotubes with friction
- *mgpt* - Simplified model generalized pseudopotential theory (MGPT) potential
- *mie/cut* - Mie potential
- *mliap* - Multiple styles of machine-learning potential
- *mm3/switch3/coulgauss/long* - Smoothed MM3 vdW potential with Gaussian electrostatics
- *momb* - Many-Body Metal-Organic (MOMB) force field
- *morse* - Morse potential
- *morse/smooth/linear* - Linear smoothed Morse potential
- *morse/soft* - Morse potential with a soft core
- *multi/lucy* - DPD potential with density-dependent force
- *multi/lucy/rx* - reactive DPD potential with density-dependent force
- *nb3b/harmonic* - Non-bonded 3-body harmonic potential
- *nb3b/screened* - Non-bonded 3-body screened harmonic potential
- *nm/cut* - N-M potential
- *nm/cut/coul/cut* - N-M potential with cutoff Coulomb
- *nm/cut/coul/long* - N-M potential with long-range Coulomb
- *nm/cut/split* - Split 12-6 Lennard-Jones and N-M potential
- *oxdna/coaxstk* -
- *oxdna/excv* -
- *oxdna/hbond* -
- *oxdna/stk* -
- *oxdna/xstk* -
- *oxdna2/coaxstk* -

- *oxdna2/dh* -
- *oxdna2/excv* -
- *oxdna2/hbond* -
- *oxdna2/stk* -
- *oxdna2/xstk* -
- *oxdna3/coaxstk* -
- *oxdna3/dh* -
- *oxdna3/excv* -
- *oxdna3/hbond* -
- *oxdna3/stk* -
- *oxdna3/xstk* -
- *oxrna2/coaxstk* -
- *oxrna2/dh* -
- *oxrna2/excv* -
- *oxrna2/hbond* -
- *oxrna2/stk* -
- *oxrna2/xstk* -
- *pace* - Atomic Cluster Expansion (ACE) machine-learning potential
- *pace/extrapolation* - Atomic Cluster Expansion (ACE) machine-learning potential with extrapolation grades
- *pace/apip* - *adaptive-precision* version of ACE, used as precise potential
- *pace/fast/apip* - *adaptive-precision* version of ACE, used as fast potential
- *pace/precise/apip* - *adaptive-precision* version of ACE, used as precise potential
- *pedone* - Pedone (PMMCS) potential (non-Coulomb part)
- *pod* - Proper orthogonal decomposition (POD) machine-learning potential
- *peri/eps* - Peridynamic EPS potential
- *peri/lps* - Peridynamic LPS potential
- *peri/pmb* - Peridynamic PMB potential
- *peri/ves* - Peridynamic VES potential
- *polymorphic* - Polymorphic 3-body potential
- *python* -
- *quip* -
- *rann* -
- *reaxff* - ReaxFF potential
- *rebo* - Second generation REBO potential of Brenner
- *rebomos* - REBOMoS potential for MoS2
- *rheo* - fluid interactions in RHEO package

- *rheo/solid* - solid interactions in RHEO package
- *resquared* - Everaers RE-Squared ellipsoidal potential
- *runner* - high-dimensional neural network potentials ML-RUNNER package
- *saip/metal* - Interlayer potential for hetero-junctions formed with hexagonal 2D materials and metal surfaces
- *saip/metal/tmd* - Interlayer potential for transition-metal dichalcogenide / metal interfaces
- *sdpd/taitwater/isothermal* - Smoothed dissipative particle dynamics for water at isothermal conditions
- *smatb* - Second Moment Approximation to the Tight Binding
- *smatb/single* - Second Moment Approximation to the Tight Binding for single-element systems
- *smd/hertz* -
- *smd/tlsph* -
- *smd/tri_surface* -
- *smd/ulsph* -
- *smtbq* -
- *snap* - SNAP machine-learning potential
- *soft* - Soft (cosine) potential
- *sph/heatconduction* -
- *sph/idealgas* -
- *sph/lj* -
- *sph/rhosum* -
- *sph/taitwater* -
- *sph/taitwater/morris* -
- *spin/dipole/cut* -
- *spin/dipole/long* -
- *spin/dmi* -
- *spin/exchange* -
- *spin/exchange/biquadratic* -
- *spin/magelec* -
- *spin/neel* -
- *srp* -
- *srp/react* -
- *surf/granular* - line/triangle interactions with granular spheres
- *sw* - Stillinger-Weber 3-body potential
- *sw/angle/table* - Stillinger-Weber potential with tabulated angular term
- *sw/mod* - modified Stillinger-Weber 3-body potential
- *table* - tabulated pair potential
- *table/rx* -

- *tdpd* - tDPD particle interactions
- *tersoff* - Tersoff 3-body potential
- *tersoff/mod* - modified Tersoff 3-body potential
- *tersoff/mod/c* -
- *tersoff/table* -
- *tersoff/zbl* - Tersoff/ZBL 3-body potential
- *thole* - Coulomb interactions with thole damping
- *threebody/table* - generic tabulated three-body potential
- *tip4p/cut* - Coulomb for TIP4P water w/out LJ
- *tip4p/long* - long-range Coulomb for TIP4P water w/out LJ
- *tip4p/long/soft* -
- *tracker* - monitor information about pairwise interactions
- *tri/lj* - LJ potential between triangles
- *ufm* -
- *uf3* - UF3 machine-learning potential
- *vashishta* - Vashishta 2-body and 3-body potential
- *vashishta/table* -
- *wf/cut* - Wang-Frenkel Potential for short-ranged interactions
- *ylz* - Yuan-Li-Zhang Potential for anisotropic interactions
- *yukawa* - Yukawa potential
- *yukawa/colloid* - screened Yukawa potential for finite-size particles
- *zbl* - Ziegler-Biersack-Littmark potential

---

### 1.76.4 Restrictions

This command must be used before any coefficients are set by the *pair_coeff*, *read_data*, or *read_restart* commands.

Some pair styles are part of specific packages. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info. The doc pages for individual pair potentials tell if it is part of a package.

### 1.76.5 Related commands

*pair_coeff*, *read_data*, *pair_modify*, *kspace_style*, *dielectric*, *pair_write*

### 1.76.6 Default

```
pair_style none
```

# 1.77 pair_write command

## 1.77.1 Syntax

```
pair_write itype jtype N style inner outer file keyword Qi Qj
```

- itype,jtype = 2 atom types (numeric or type label)
- N = # of values
- style = *r* or *rsq* or *bitmap*
- inner,outer = inner and outer cutoff (distance units)
- file = name of file to write values to
- keyword = section name in file for this set of tabulated values
- Qi,Qj = 2 atom charges (charge units) (optional)

## 1.77.2 Examples

```
pair_write 1 3 500 r 1.0 10.0 table.txt LJ
pair_write 1 1 1000 rsq 2.0 8.0 table.txt Yukawa_1_1 -0.5 0.5

labelmap atom 1 C 2 H
pair_write C H 500 r 1.0 10.0 table.txt LJ
```

## 1.77.3 Description

Write energy and force values to a file as a function of distance for the currently defined pair potential. This is useful for plotting the potential function or otherwise debugging its values. If the file already exists, the table of values is appended to the end of the file to allow multiple tables of energy and force to be included in one file. In case a new file is created, the first line will be a comment containing a "DATE:" and "UNITS:" tag with the current date and the current *units* setting as argument. For subsequent invocations of the pair_write command, the current units setting is compared against the entry in the file, if present, and pair_write will refuse to add a table if the units are not the same.

The energy and force values are computed at distances from inner to outer for 2 interacting atoms of type *itype* and *jtype*, using the appropriate *pair_coeff* coefficients. If the style is *r*, then N distances are used, evenly spaced in r; if the style is *rsq*, N distances are used, evenly spaced in r^2.

For example, for N = 7, style = *r*, inner = 1.0, and outer = 4.0, values are computed at r = 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0.

If the style is *bitmap*, then 2^N values are written to the file in a format and order consistent with how they are read in by the *pair_coeff* command for pair style *table*. For reasonable accuracy in a bitmapped table, choose N >= 12, an *inner* value that is smaller than the distance of closest approach of 2 atoms, and an *outer* value <= cutoff of the potential.

If the pair potential is computed between charged atoms, the charges of the pair of interacting atoms can optionally be specified. If not specified, values of Qi = Qj = 1.0 are used.

The file is written in the format used as input for the *pair_style table* option with *keyword* as the section name. Each line written to the file lists an index number (1-N), a distance (in distance units), an energy (in energy units), and a force (in force units).

### 1.77.4 Restrictions

All force field coefficients for pair and other kinds of interactions must be set before this command can be invoked.

Due to how the pairwise force is computed, an inner value > 0.0 must be specified even if the potential has a finite value at r = 0.0.

The *pair_write* command can only be used for pairwise additive interactions for which a *Pair::single()* function can be and has been implemented. This excludes for example manybody potentials or TIP4P coulomb styles.

### 1.77.5 Related commands

*pair_style table*, *pair_style*, *pair_coeff*

### 1.77.6 Default

none

## 1.78 partition command

### 1.78.1 Syntax

```
partition style N command ...
```

- style = *yes* or *no*
- N = partition number (see asterisk form below)
- command = any LAMMPS command

### 1.78.2 Examples

```
partition yes 1 processors 4 10 6
partition no 5 print "Active partition"
partition yes *5 fix all nve
partition yes 6* fix all nvt temp 1.0 1.0 0.1
```

### 1.78.3 Description

This command invokes the specified command on a subset of the partitions of MPI processes you have defined via the *-partition command-line switch*.

Normally, every input script command in your script is invoked by every partition. This behavior can be modified by defining world- or universe-style *variables* that have different values for each partition. This mechanism can be used to cause your script to jump to different input script files on different partitions, if such a variable is used in a *jump* command.

The "partition" command is another mechanism for having as input script operate differently on different partitions. It is basically a prefix on any LAMMPS command. The command will only be invoked on the partition(s) specified by the *style* and *N* arguments.

If the *style* is *yes*, the command will be invoked on any partition which matches the *N* argument. If the *style* is *no* the command will be invoked on all the partitions which do not match the *Np* argument.

Partitions are numbered from 1 to *Np*, where *Np* is the number of partitions specified by the *-partition command-line switch*.

*N* can be specified in one of two ways. An explicit numeric value can be used, as in the first example above. Or a wild-card asterisk can be used to span a range of partition numbers. This takes the form "*" or "*n" or "n*" or "m*n". An asterisk with no numeric values means all partitions from 1 to *Np*. A leading asterisk means all partitions from 1 to n (inclusive). A trailing asterisk means all partitions from n to *Np* (inclusive). A middle asterisk means all partitions from m to n (inclusive).

This command can be useful for the "run_style verlet/split" command which imposed requirements on how the *processors* command lays out a 3d grid of MPI processes in each of 2 partitions.

### 1.78.4 Restrictions

none

### 1.78.5 Related commands

*run_style verlet/split*

### 1.78.6 Default

none

## 1.79 plugin command

### 1.79.1 Syntax

```
plugin command args
```

- command = *load* or *unload* or *list* or *clear* or *restore*
- args = list of arguments for a particular plugin command

```
load file = load plugin(s) from shared object in file
unload style name = unload plugin name of style style
    style = pair or bond or angle or dihedral or improper or kspace or compute or␣
↪fix or region or command or run or min
list = print a list of currently loaded plugins
clear = unload all currently loaded plugins
restore = restore all loaded plugins
```

### 1.79.2 Examples

```
plugin load morse2plugin.so
plugin unload pair morse2/omp
plugin unload command hello
plugin list
plugin clear
plugin restore
```

### 1.79.3 Description

The plugin command allows to load (and unload) additional styles and commands into a LAMMPS binary from so-called dynamic shared object (DSO) files. This enables to add new functionality to an existing LAMMPS binary without having to recompile and link the entire executable.

**Plugins are a global, per-executable property**

Unlike most settings in LAMMPS, plugins are a per-executable global property. Loading a plugin means that it is not only available for the current LAMMPS instance but for all *future* LAMMPS instances.

After a *clear* command, all currently loaded plugins will be restored and do not need to be loaded again.

When using the library interface or the Python or Fortran module to create multiple concurrent LAMMPS instances, all plugins should be loaded by the first created LAMMPS instance as all future instances will inherit them. To import plugins that were loaded by a different LAMMPS instance, use the *restore* command.

The *load* command will load and initialize all plugins contained in the plugin DSO with the given filename. A message with information about the plugin style and name and more will be printed. Individual DSO files may contain multiple plugins. If a plugin is already loaded it will be skipped. More details about how to write and compile the plugin DSO is given in programmer's guide part of the manual under *Writing plugins*.

The *unload* command will remove the given style or the given name from the list of available styles. If the plugin style is currently in use, that style instance will be deleted and replaced by the default setting for that style.

The *list* command will print a list of the loaded plugins and their styles and names.

The *clear* command will unload all currently loaded plugins.

Added in version 12Jun2025.

The *restore* command will restore all currently loaded plugins. This allows to "import" plugins into a different LAMMPS instance.

**Automatic loading of plugins**

Added in version 4May2022.

When the environment variable `LAMMPS_PLUGIN_PATH` is set, then LAMMPS will search the directory (or directories) listed in this path for files with names that end in `plugin.so` (e.g. `helloplugin.so`) and will try to load the contained plugins automatically at start-up.

### 1.79.4 Restrictions

The *plugin* command is part of the PLUGIN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

If plugins access functions or classes from a package, LAMMPS must have been compiled with that package included.

Plugins are dependent on the LAMMPS binary interface (ABI) and particularly the MPI library used. So they are not guaranteed to work when the plugin was compiled with a different MPI library or different compilation settings or a different LAMMPS version. There are no checks, so if there is a mismatch the plugin object will either not load or data corruption and crashes may happen.

### 1.79.5 Related commands

none

### 1.79.6 Default

none

# 1.80 prd command

## 1.80.1 Syntax

```
prd N t_event n_dephase t_dephase t_correlate compute-ID seed keyword value ...
```

- N = # of timesteps to run (not including dephasing/quenching)
- t_event = timestep interval between event checks
- n_dephase = number of velocity randomizations to perform in each dephase run
- t_dephase = number of timesteps to run dynamics after each velocity randomization during dephase
- t_correlate = number of timesteps within which 2 consecutive events are considered to be correlated
- compute-ID = ID of the compute used for event detection
- random_seed = random # seed (positive integer)
- zero or more keyword/value pairs may be appended
- keyword = *min* or *temp* or *vel* or *time*

```
min values = etol ftol maxiter maxeval
  etol = stopping tolerance for energy, used in quenching
  ftol = stopping tolerance for force, used in quenching
  maxiter = max iterations of minimize, used in quenching
  maxeval = max number of force/energy evaluations, used in quenching
temp value = Tdephase
  Tdephase = target temperature for velocity randomization, used in dephasing
vel values = loop dist
  loop = all or local or geom, used in dephasing
  dist = uniform or gaussian, used in dephasing
time value = steps or clock
  steps = simulation runs for N timesteps on each replica (default)
  clock = simulation runs for N timesteps across all replicas
```

## 1.80.2 Examples

```
prd 5000 100 10 10 100 1 54982
prd 5000 100 10 10 100 1 54982 min 0.1 0.1 100 200
```

## 1.80.3 Description

Run a parallel replica dynamics (PRD) simulation using multiple replicas of a system. One or more replicas can be used. The total number of steps *N* to run can be interpreted in one of two ways; see discussion of the *time* keyword below.

PRD is described in *(Voter1998)* by Art Voter. Similar to global or local hyperdynamics (HD), PRD is a method for performing accelerated dynamics that is suitable for infrequent-event systems that obey first-order kinetics. A good overview of accelerated dynamics methods (AMD) for such systems in given in this review paper *(Voter2002)* from Art's group. To quote from the paper: "The dynamical evolution is characterized by vibrational excursions within a potential basin, punctuated by occasional transitions between basins. The transition probability is characterized by p(t) = k*exp(-kt) where k is the rate constant."

Both PRD and HD produce a time-accurate trajectory that effectively extends the timescale over which a system can be simulated, but they do it differently. PRD creates Nr replicas of the system and runs dynamics on each independently with a normal unbiased potential until an event occurs in one of the replicas. The time between events is reduced by a

factor of Nr replicas. HD uses a single replica of the system and accelerates time by biasing the interaction potential in a manner such that each timestep is effectively longer. For both methods, per CPU second, more physical time elapses and more events occur. See the *hyper* page for more info about HD.

In PRD, each replica runs on a partition of one or more processors. Processor partitions are defined at run-time using the *-partition command-line switch*. Note that if you have MPI installed, you can run a multi-replica simulation with more replicas (partitions) than you have physical processors, e.g you can run a 10-replica simulation on one or two processors. However for PRD, this makes little sense, since running a replica on virtual instead of physical processors,offers no effective parallel speed-up in searching for infrequent events. See the *Howto replica* doc page for further discussion.

When a PRD simulation is performed, it is assumed that each replica is running the same model, though LAMMPS does not check for this. I.e. the simulation domain, the number of atoms, the interaction potentials, etc should be the same for every replica.

A PRD run has several stages, which are repeated each time an "event" occurs in one of the replicas, as explained below. The logic for a PRD run is as follows:

```
while (time remains):
  dephase for n_dephase*t_dephase steps
  until (event occurs on some replica):
    run dynamics for t_event steps
    quench
    check for uncorrelated event on any replica
  until (no correlated event occurs):
    run dynamics for t_correlate steps
    quench
    check for correlated event on this replica
  event replica shares state with all replicas
```

Before this loop begins, the state of the system on replica 0 is shared with all replicas, so that all replicas begin from the same initial state. The first potential energy basin is identified by quenching (an energy minimization, see below) the initial state and storing the resulting coordinates for reference.

In the first stage, dephasing is performed by each replica independently to eliminate correlations between replicas. This is done by choosing a random set of velocities, based on the *random_seed* that is specified, and running *t_dephase* timesteps of dynamics. This is repeated *n_dephase* times. At each of the *n_dephase* stages, if an event occurs during the *t_dephase* steps of dynamics for a particular replica, the replica repeats the stage until no event occurs.

If the *temp* keyword is not specified, the target temperature for velocity randomization for each replica is the current temperature of that replica. Otherwise, it is the specified *Tdephase* temperature. The style of velocity randomization is controlled using the keyword *vel* with arguments that have the same meaning as their counterparts in the *velocity* command.

In the second stage, each replica runs dynamics continuously, stopping every *t_event* steps to check if a transition event has occurred. This check is performed by quenching the system and comparing the resulting atom coordinates to the coordinates from the previous basin. The first time through the PRD loop, the "previous basin" is the set of quenched coordinates from the initial state of the system.

A quench is an energy minimization and is performed by whichever algorithm has been defined by the *min_style* command. Minimization parameters may be set via the *min_modify* command and by the *min* keyword of the PRD command. The latter are the settings that would be used with the *minimize* command. Note that typically, you do not need to perform a highly-converged minimization to detect a transition event, though you may need to in order to prevent a set of atoms in the system from relaxing to a saddle point.

The event check is performed by a compute with the specified *compute-ID*. Currently there is only one compute that works with the PRD command, which is the *compute event/displace* command. Other event-checking computes may be added. *Compute event/displace* checks whether any atom in the compute group has moved further than a specified threshold distance. If so, an "event" has occurred.

In the third stage, the replica on which the event occurred (event replica) continues to run dynamics to search for correlated events. This is done by running dynamics for *t_correlate* steps, quenching every *t_event* steps, and checking if another event has occurred.

The first time no correlated event occurs, the final state of the event replica is shared with all replicas, the new basin reference coordinates are updated with the quenched state, and the outer loop begins again. While the replica event is searching for correlated events, all the other replicas also run dynamics and event checking with the same schedule, but the final states are always overwritten by the state of the event replica.

The outer loop of the pseudocode above continues until *N* steps of dynamics have been performed. Note that *N* only includes the dynamics of stages 2 and 3, not the steps taken during dephasing or the minimization iterations of quenching. The specified *N* is interpreted in one of two ways, depending on the *time* keyword. If the *time* value is *steps*, which is the default, then each replica runs for *N* timesteps. If the *time* value is *clock*, then the simulation runs until *N* aggregate timesteps across all replicas have elapsed. This aggregate time is the "clock" time defined below, which typically advances nearly M times faster than the timestepping on a single replica, where M is the number of replicas.

---

Four kinds of output can be generated during a PRD run: event statistics, thermodynamic output by each replica, dump files, and restart files.

When running with multiple partitions (each of which is a replica in this case), the print-out to the screen and master log.lammps file is limited to event statistics. Note that if a PRD run is performed on only a single replica then the event statistics will be intermixed with the usual thermodynamic output discussed below.

The quantities printed each time an event occurs are the timestep, CPU time, clock, event number, a correlation flag, the number of coincident events, and the replica number of the chosen event.

The timestep is the usual LAMMPS timestep, except that time does not advance during dephasing or quenches, but only during dynamics. Note that are two kinds of dynamics in the PRD loop listed above that contribute to this timestepping. The first is when all replicas are performing independent dynamics, waiting for an event to occur. The second is when correlated events are being searched for, but only one replica is running dynamics.

The CPU time is the total elapsed time on each processor, since the start of the PRD run.

The clock is the same as the timestep except that it advances by M steps per timestep during the first kind of dynamics when the M replicas are running independently. The clock advances by only 1 step per timestep during the second kind of dynamics, when only a single replica is checking for a correlated event. Thus "clock" time represents the aggregate time (in steps) that has effectively elapsed during a PRD simulation on M replicas. If most of the PRD run is spent in the second stage of the loop above, searching for infrequent events, then the clock will advance nearly M times faster than it would if a single replica was running. Note the clock time between successive events should be drawn from p(t).

The event number is a counter that increments with each event, whether it is uncorrelated or correlated.

The correlation flag will be 0 when an uncorrelated event occurs during the second stage of the loop listed above, i.e. when all replicas are running independently. The correlation flag will be 1 when a correlated event occurs during the third stage of the loop listed above, i.e. when only one replica is running dynamics.

When more than one replica detects an event at the end of the same event check (every *t_event* steps) during the second stage, then one of them is chosen at random. The number of coincident events is the number of replicas that detected an event. Normally, this value should be 1. If it is often greater than 1, then either the number of replicas is too large, or *t_event* is too large.

The replica number is the ID of the replica (from 0 to M-1) in which the event occurred.

---

When running on multiple partitions, LAMMPS produces additional log files for each partition, e.g. log.lammps.0, log.lammps.1, etc. For the PRD command, these contain the thermodynamic output for each replica. You will see short runs and minimizations corresponding to the dynamics and quench operations of the loop listed above. The timestep will be reset appropriately depending on whether the operation advances time or not.

After the PRD command completes, timing statistics for the PRD run are printed in each replica's log file, giving a breakdown of how much CPU time was spent in each stage (dephasing, dynamics, quenching, etc).

---

Any *dump files* defined in the input script, will be written to during a PRD run at timesteps corresponding to both uncorrelated and correlated events. This means the requested dump frequency in the *dump* command is ignored. There will be one dump file (per dump command) created for all partitions.

The atom coordinates of the dump snapshot are those of the minimum energy configuration resulting from quenching following a transition event. The timesteps written into the dump files correspond to the timestep at which the event occurred and NOT the clock. A dump snapshot corresponding to the initial minimum state used for event detection is written to the dump file at the beginning of each PRD run.

---

If the *restart* command is used, a single restart file for all the partitions is generated, which allows a PRD run to be continued by a new input script in the usual manner.

The restart file is generated at the end of the loop listed above. If no correlated events are found, this means it contains a snapshot of the system at time T + *t_correlate*, where T is the time at which the uncorrelated event occurred. If correlated events were found, then it contains a snapshot of the system at time T + *t_correlate*, where T is the time of the last correlated event.

The restart frequency specified in the *restart* command is interpreted differently when performing a PRD run. It does not mean the timestep interval between restart files. Instead it means an event interval for uncorrelated events. Thus a frequency of 1 means write a restart file every time an uncorrelated event occurs. A frequency of 10 means write a restart file every 10th uncorrelated event.

When an input script reads a restart file from a previous PRD run, the new script can be run on a different number of replicas or processors. However, it is assumed that *t_correlate* in the new PRD command is the same as it was previously. If not, the calculation of the "clock" value for the first event in the new run will be slightly off.

---

### 1.80.4 Restrictions

This command can only be used if LAMMPS was built with the REPLICA package. See the *Build package* doc page for more info.

The *N* and *t_correlate* settings must be integer multiples of *t_event*.

Runs restarted from restart file written during a PRD run will not produce identical results due to changes in the random numbers used for dephasing.

This command cannot be used when any fixes are defined that keep track of elapsed time to perform time-dependent operations. Examples include the "ave" fixes such as *fix ave/chunk*. Also *fix dt/reset* and *fix deposit*.

### 1.80.5 Related commands

*compute event/displace*, *min_modify*, *min_style*, *run_style*, *minimize*, *velocity*, *temper*, *neb*, *tad*, *hyper*

### 1.80.6 Default

The option defaults are min = 0.1 0.1 40 50, no temp setting, vel = geom gaussian, and time = steps.

---

**(Voter1998)** Voter, Phys Rev B, 57, 13985 (1998).

**(Voter2002)** Voter, Montalenti, Germann, Annual Review of Materials Research 32, 321 (2002).

# 1.81 print command

## 1.81.1 Syntax

```
print string keyword value
```

- string = text string to print, which may contain variables
- zero or more keyword/value pairs may be appended
- keyword = *file* or *append* or *screen* or *universe*

  *file* value = filename
  *append* value = filename
  *screen* value = *yes* or *no*
  *universe* value = *yes* or *no*

## 1.81.2 Examples

```
print "Done with equilibration" file info.dat
print Vol=$v append info.dat screen no
print "The system volume is now $v"
print 'The system volume is now $v'
print "NEB calculation 1 complete" screen no universe yes
print """
System volume = $v
System temperature = $t
"""
```

## 1.81.3 Description

Print a text string to the screen and logfile. The text string must be a single argument, so if it is one line but more than one word, it should be enclosed in single or double quotes. To generate multiple lines of output, the string can be enclosed in triple quotes, as in the last example above. If the text string contains variables, they will be evaluated and their current values printed.

Added in version 15Jun2023: support for vector style variables

See the *variable* command for a description of *equal* and *vector* style variables which are typically the most useful ones to use with the print command. Equal- and vector-style variables can calculate formulas involving mathematical operations, atom properties, group properties, thermodynamic properties, global values calculated by a *compute* or *fix*, or references to other *variables*. Vector-style variables are printed in a bracketed, comma-separated format, e.g. [1,2,3,4] or [12.5,2,4.6,10.1].

**Note**

As discussed on the *Commands parse* doc page, the text string can use "immediate" variables, specified as $(formula) with parenthesis, where the numeric formula has the same syntax as equal-style variables described on the *variable* doc page. This is a convenient way to evaluate a formula immediately without using the variable command to define a named variable and then use that variable in the text string. The formula can include a trailing colon and format string which determines the precision with which the numeric value is output. This is also explained on the *Commands parse* doc page.

If you want the print command to be executed multiple times (with changing variable values), there are 3 options. First, consider using the *fix print* command, which will print a string periodically during a simulation. Second, the print

command can be used as an argument to the *every* option of the *run* command. Third, the print command could appear in a section of the input script that is looped over (see the *jump* and *next* commands).

If the *file* or *append* keyword is used, a filename is specified to which the output will be written. If *file* is used, then the filename is overwritten if it already exists. If *append* is used, then the filename is appended to if it already exists, or created if it does not exist.

If the *screen* keyword is used, output to the screen and logfile can be turned on or off as desired.

If the *universe* keyword is used, output to the global screen and logfile can be turned on or off as desired. In multi-partition calculations, the *screen* option and the corresponding output only apply to the screen and logfile of the individual partition.

### 1.81.4 Restrictions

none

### 1.81.5 Related commands

*fix print*, *variable*

### 1.81.6 Default

The option defaults are no file output, screen = yes, and universe = no.

## 1.82 processors command

### 1.82.1 Syntax

```
processors Px Py Pz keyword args ...
```

- Px,Py,Pz = # of MPI processes in each dimension of 3d grid overlaying the simulation domain
- zero or more keyword/arg pairs may be appended
- keyword = *grid* or *map* or *part* or *file*

```
grid arg = gstyle params ...
  gstyle = onelevel or twolevel or numa or custom
    onelevel params = none
    twolevel params = Nc Cx Cy Cz
      Nc = number of cores per node
      Cx,Cy,Cz = # of cores in each dimension of 3d sub-grid assigned to each node
    numa params = none
    custom params = infile
      infile = file containing grid layout
numa_nodes arg = Nn
      Nn = number of numa domains per node
map arg = cart or cart/reorder or xyz or xzy or yxz or yzx or zxy or zyx
   cart = use MPI_Cart_xxx() methods to map MPI processes to 3d grid with reorder =␣
↪0
   cart/reorder = use MPI_Cart_xxx() methods to map MPI processes to 3d grid with␣
↪reorder = 1
   xyz,xzy,yxz,yzx,zxy,zyx = map MPI processes to 3d grid in IJK ordering
part args = Psend Precv cstyle
  Psend = partition # (1 to Np) which will send its processor layout
  Precv = partition # (1 to Np) which will recv the processor layout
```

```
    cstyle = multiple
      multiple = Psend grid will be multiple of Precv grid in each dimension
  file arg = outfile
    outfile = name of file to write 3d grid of MPI process ranks to
```

### 1.82.2 Examples

```
processors * * 5
processors 2 4 4
processors * * 8 map xyz
processors * * * grid numa
processors * * * grid twolevel 4 * * 1
processors 4 8 16 grid custom myfile
processors * * * part 1 2 multiple
```

### 1.82.3 Description

Specify how MPI processes are mapped as a regular 3d grid to the global simulation box. The mapping involves 2 steps. First if there are $P$ MPI processes, it means choosing a factorization $P = Px$ by $Py$ by $Pz$ so that there are $Px$ MPI processes in the x dimension, and similarly for the y and z dimensions. Second, the $P$ MPI processes are mapped to the regular 3d grid. The arguments to this command control each of these 2 steps.

The $Px$, $Py$, $Pz$ parameters affect the factorization. Any of the 3 parameters can be specified with an asterisk "*", which means LAMMPS will choose the number of MPI processes in that dimension of the grid. It will do this based on the size and shape of the global simulation box so as to minimize the surface-to-volume ratio of each MPI rank's subdomain.

Choosing explicit values for $Px$ or $Py$ or $Pz$ can be used to override the default manner in which LAMMPS will create the regular 3d grid of MPI processes, if it is known to be sub-optimal for a particular problem. E.g. a problem where the extent of atoms will change dramatically in a particular dimension over the course of the simulation.

The product of $Px$, $Py$, $Pz$ must equal $P$, the total # of MPI processes LAMMPS is running on. For a *2d simulation*, $Pz$ must equal 1.

Note that if you run on a prime number of MPI processes $P$, then a grid such as 1 x $P$ x 1 will be required, which may incur extra communication costs due to the high surface area of each processor's subdomain.

Also note that if multiple partitions are being used then $P$ is the number of MPI processes in this partition; see the *-partition command-line switch* page for details. Also note that you can prefix the *processors* command with the *partition* command to easily specify different $Px$, $Py$, $Pz$ values for different partitions.

You can use the *partition* command to specify different processor grids for different partitions, e.g.

```
partition yes 1 processors 4 4 4
partition yes 2 processors 2 3 2
```

**Note**

This command only affects the initial regular 3d grid created when the simulation box is first specified via a *create_box* or *read_data* or *read_restart* command. Or if the simulation box is re-created via the *replicate* command. The same regular grid is initially created, regardless of which *comm_style* command is in effect.

If load-balancing is never invoked via the *balance* or *fix balance* commands, then the initial regular grid will persist for all simulations. If balancing is performed, some of the methods invoked by those commands retain the logical topology of the initial 3d grid, and the mapping of MPI processes to the grid specified by the *processors* command. However the

grid spacings in different dimensions may change, so that MPI processes would own subdomains of different sizes. If the *comm_style tiled* command is used, methods invoked by the balancing commands may discard the 3d grid of MPI processes and tile the simulation domain with subdomains of different sizes and shapes which no longer have a logical 3d connectivity. If that occurs, all the information specified by the *processors* command is ignored.

---

The *grid* keyword affects the factorization of *P* into *Px*, *Py*, *Pz* and it can also affect how the *P* MPI ranks are mapped to the 3d grid of MPI processes.

The *onelevel* style creates a 3d grid that is compatible with the *Px*, *Py*, *Pz* settings, and which minimizes the surface-to-volume ratio of each processor's subdomain, as described above. The mapping of MPI processes to the grid is determined by the *map* keyword setting.

The *twolevel* style can be used on machines with multicore nodes to minimize off-node communication. It ensures that contiguous subsections of the 3d grid are assigned to all the cores of a node. For example if *Nc* is 4, then 2x2x1 or 2x1x2 or 1x2x2 subsections of the 3d grid will correspond to the cores of each node. This affects both the factorization and mapping steps.

The *Cx*, *Cy*, *Cz* settings are similar to the *Px*, *Py*, *Pz* settings, only their product should equal *Nc*. Any of the 3 parameters can be specified with an asterisk "*", which means LAMMPS will choose the number of cores in that dimension of the node's sub-grid. As with *Px*, *Py*, *Pz*, it will do this based on the size and shape of the global simulation box so as to minimize the surface-to-volume ratio of each processor's subdomain.

**Note**

For the *twolevel* style to work correctly, it assumes the ranks of the MPI processes that LAMMPS is running on are ordered by core and then by node. E.g. if you are running on 2 quad-core nodes, for a total of 8 MPI processes, then it assumes ranks 0, 1, 2, and 3 are on node 1, and ranks 4, 5, 6, and 7 are on node 2. This is the default rank ordering for most MPI implementations, but some MPI libraries provide options for customizing this ordering, e.g. via environment variable settings.

The *numa* style operates similar to the *twolevel* keyword except that it auto-detects which cores are running on which nodes. It will also subdivide the cores into NUMA domains. Currently, the number of NUMA domains is not auto-detected and must be specified using the *numa_nodes* keyword. Otherwise, the default value is used. The *numa* style uses a different algorithm than the *twolevel* keyword for doing the two-level factorization of the simulation box into a 3d processor grid to minimize off-node communication and communication across NUMA domains. It does its own MPI-based mapping of nodes and cores to the regular 3d grid. Thus it may produce a different layout of the MPI ranks than the *twolevel* options.

The *numa* style will give an error if the number of MPI processes is not divisible by the number of cores used per node, or any of the *Px* or *Py* or *Pz* values is greater than 1.

**Note**

Unlike the *twolevel* style, the *numa* style does not require any particular ordering of MPI ranks in order to work correctly. This is because it auto-detects which processes are running on which nodes. However, it assumes that the lowest ranks are in the first NUMA domain, and so forth. MPI rank orderings that do not preserve this property might result in more intra-node communication between CPUs.

The *custom* style uses the file *infile* to define both the 3d factorization and the mapping of processors to the grid.

The file should have the following format. Any number of initial blank or comment lines (starting with a "#" character) can be present. The first non-blank, non-comment line should have 3 values:

```
Px Py Pz
```

These must be compatible with the total number of processors and the *Px*, *Py*, *Pz* settings of the processors command.

This line should be immediately followed by P = *Px* * *Py* * *Pz* lines of the form:

```
ID I J K
```

where ID is an MPI process rank (from 0 to P-1) and *I*, *J*, *K* are the MPI rank's location in the 3d grid. *I* must be a number from 1 to *Px* (inclusive) and similarly for *J* and *Py* or *K* and *Pz*. The *P* lines can be listed in any order, but no MPI rank should appear more than once.

---

The *numa_nodes* keyword is used to specify the number of NUMA domains per node. It is currently only used by the *numa* style for two-level factorization to reduce the amount of MPI communications between CPUs. A good setting for this will typically be equal to the number of CPU sockets per node.

---

The *map* keyword affects how the *P* MPI process ranks (from 0 to P-1) are mapped to the 3d grid of MPI processes. It is only used by the *onelevel* and *twolevel* grid settings.

The *cart* style uses the family of MPI Cartesian functions to perform the mapping, namely MPI_Cart_create(), MPI_Cart_get(), MPI_Cart_shift(), and MPI_Cart_rank(). It invokes the MPI_Cart_create() function with its reorder flag = 0, so that MPI is not free to reorder the processors.

The *cart/reorder* style does the same thing as the *cart* style except it sets the reorder flag to 1, so that MPI can reorder processors if it desires.

The *xyz*, *xzy*, *yxz*, *yzx*, *zxy*, and *zyx* styles are all similar. If the style is IJK, then it maps the *P* processors to the grid so that the MPI ranks in the *I* direction varies fastest, the MPI ranks in the *J* direction varies next fastest, and the MPI rank in the *K* direction varies slowest. For example, if you select style *xyz* and you have a 2x2x2 grid of 8 processors, the assignments of the 8 octants of the simulation domain will be:

```
proc 0 = lo x, lo y, lo z octant
proc 1 = hi x, lo y, lo z octant
proc 2 = lo x, hi y, lo z octant
proc 3 = hi x, hi y, lo z octant
proc 4 = lo x, lo y, hi z octant
proc 5 = hi x, lo y, hi z octant
proc 6 = lo x, hi y, hi z octant
proc 7 = hi x, hi y, hi z octant
```

Note that, in principle, an MPI implementation on a particular machine should be aware of both the machine's network topology and the specific subset of processors and nodes that were assigned to your simulation. Thus its `MPI_Cart_xxx()` calls can optimize the assignment of MPI processes to the 3d grid to minimize communication costs. In practice, however, few if any MPI implementations actually do this. So it is likely that the *cart* and *cart/reorder* styles simply give the same result as one of the *IJK* styles.

Also note, that for the *twolevel* grid style, the *map* setting is used to first map the nodes to the 3d grid, then again to the cores within each node. For the latter step, the *cart* and *cart/reorder* styles are not supported, so an *xyz* style is used in their place.

---

The *part* keyword affects the factorization of *P* into *Px*, *Py*, *Pz*.

It can be useful when running in multi-partition mode, e.g. with the *run_style verlet/split* command. It specifies a dependency between a sending partition *Psend* and a receiving partition *Precv* which is enforced when each is setting

up their own mapping of their processors to the simulation box. Each of *Psend* and *Precv* must be integers from 1 to *Np*, where *Np* is the number of partitions you have defined via the *-partition command-line switch*.

A "dependency" means that the sending partition will create its regular 3d grid as *Px* by *Py* by *Pz* and after it has done this, it will send the *Px*, *Py*, *Pz* values to the receiving partition. The receiving partition will wait to receive these values before creating its own regular 3d grid and will use the sender's *Px*, *Py*, *Pz* values as a constraint. The nature of the constraint is determined by the *cstyle* argument.

For a *cstyle* of *multiple*, each dimension of the sender's processor grid is required to be an integer multiple of the corresponding dimension in the receiver's processor grid. This is a requirement of the *run_style verlet/split* command.

For example, assume the sending partition creates a 4x6x10 grid = 240 processor grid. If the receiving partition is running on 80 processors, it could create a 4x2x10 grid, but it will not create a 2x4x10 grid, since in the y-dimension, 6 is not an integer multiple of 4.

**Note**

If you use the *partition* command to invoke different "processors" commands on different partitions, and you also use the *part* keyword, then you must ensure that both the sending and receiving partitions invoke the "processors" command that connects the 2 partitions via the *part* keyword. LAMMPS cannot easily check for this, but your simulation will likely hang in its setup phase if this error has been made.

---

The *file* keyword writes the mapping of the factorization of *P* MPI processes and their mapping to the 3d grid to the specified file *outfile*. This is useful to check that you assigned physical processors in the manner you desired, which can be tricky to figure out, especially when running on multiple partitions or on, a multicore machine or when the MPI ranks were reordered by use of the *-reorder command-line switch* or due to use of MPI-specific launch options such as a config file.

If you have multiple partitions you should ensure that each one writes to a different file, e.g. using a *world-style variable* for the filename. The file has a self-explanatory header, followed by one-line per processor in this format:

world-ID universe-ID original-ID: I J K: name

The IDs are the MPI ranks in this simulation (the world), the universe (of multiple simulations), and the original MPI communicator used to instantiate LAMMPS, respectively. The world and universe IDs will only be different if you are running on more than one partition; see the *-partition command-line switch*. The universe and original IDs will only be different if you used the *-reorder command-line switch* to reorder the MPI processes differently than their rank in the original communicator LAMMPS was instantiated with.

*I*, *J*, *K* are the indices of the processor in the regular 3d grid, each from 1 to *Nd*, where *Nd* is the number of MPI processes in that dimension of the grid.

The *name* is what is returned by a call to MPI_Get_processor_name() and should represent an identifier relevant to the physical processors in your machine. Note that depending on the MPI implementation, multiple cores can have the same *name*.

---

### 1.82.4 Restrictions

This command cannot be used *after* the simulation box is defined by a *read_data* or *create_box* command. It can be used *before* a restart file is read to change the 3d processor grid from what is specified in the restart file.

The *grid numa* keyword only currently works with the *map cart* option.

The *part* keyword (for the receiving partition) only works with the *grid onelevel* or *grid twolevel* options.

### 1.82.5 Related commands

*partition*, *-reorder command-line switch*

### 1.82.6 Default

The option defaults are *Px Py Pz* = * * *, grid = onelevel, map = cart, and numa_nodes = 2.

## 1.83 python command

### 1.83.1 Syntax

```
python mode keyword args ...
```

- mode = *source* or *name* of Python function

  if mode is *source*:

  ```
  keyword = here or name of a Python file
    here arg = inline
      inline = one or more lines of Python code which will be executed immediately
               must be a single argument, typically enclosed between triple quotes
    Python file = name of a file with Python code which will be executed immediately
  ```

- if *mode* is *name* of a Python function:

  ```
  one or more keywords with/without arguments must be appended
  keyword = invoke or input or return or format or length or file or here or exists
    invoke arg = logreturn (optional)
      invoke the previously-defined Python function
      if logreturn is specified, print the return value of the invoked function to␣
  ↪the screen and logfile
    input args = N i1 i2 ... iN
      N = # of inputs to function
      i1,...,iN = value, SELF, or LAMMPS variable name
        value = integer number, floating point number, or string
        SELF = reference to LAMMPS itself which can then be accessed by Python␣
  ↪function
        variable = v_name, where name = name of a LAMMPS variable, e.g. v_abc
        internal variable = iv_name, where name = name of a LAMMPS internal-style␣
  ↪variable, e.g. iv_xyz
    return arg = varReturn
      varReturn = v_name  = LAMMPS variable name which the return value of the Python␣
  ↪function will be assigned to
    format arg = fstring with M characters
      M = N if no return value, where N = # of inputs
      M = N+1 if there is a return value
      fstring = each character (i,f,s,p) corresponds (in order) to an input or return␣
  ↪value
        'i' = integer, 'f' = floating point, 's' = string, 'p' = SELF
    length arg = Nlen
      Nlen = max length of string returned from Python function
    file arg = filename
      filename = file of Python code, which defines the Python function
    here arg = inline
      inline = one or more lines of Python code which defines the Python function
  ```

```
                must be a single argument, typically enclosed between triple quotes
      exists arg = none = Python code has been loaded by previous python command
```

### 1.83.2 Examples

```
python pForce input 2 v_x 20.0 return v_f format fff file force.py
python pForce invoke logreturn

python factorial input 1 myN return v_fac format ii here """
def factorial(n):
  if n == 1: return n
  return n * factorial(n-1)
 """

python loop input 1 SELF return v_value format pf here """
def loop(lmpptr,N,cut0):
  from lammps import lammps
  lmp = lammps(ptr=lmpptr)

  # loop N times, increasing cutoff each time

  for i in range(N):
    cut = cut0 + i*0.1
    lmp.set_variable("cut",cut)               # set a variable in LAMMPS
    lmp.command("pair_style lj/cut ${cut}")   # LAMMPS commands
    lmp.command("pair_coeff * * 1.0 1.0")
    lmp.command("run 100")
"""

python source funcdef.py

python source here "from lammps import lammps"
```

### 1.83.3 Description

The *python* command interfaces LAMMPS with an embedded Python interpreter and enables executing arbitrary python code in that interpreter. This can be done immediately, by using *mode* = *source*. Or execution can be deferred, by registering a Python function for later execution, by using *mode* = *name* of a Python function.

Later execution can be triggered in one of two ways. One is to use the python command again with its *invoke* keyword. The other is to trigger the evaluation of a python-style, equal-style, vector-style, or atom-style variable. A python-style variable invokes its associated Python function; its return value becomes the value of the python-style variable. Equal-, vector-, and atom-style variables can use a Python function wrapper in their formulas which encodes the python-style variable name, and specifies arguments (which themselves can be numeric formulas) to pass to the Python function associated with the python-style variable.

As explained on the *variable* doc page, the definition of a python-style variable associates a Python function name with the variable. Its specification must match the *mode* argument of the *python* command for the Python function name. For example these two commands would be consistent:

```
variable foo python myMultiply
python myMultiply return v_foo format f file funcs.py
```

The two commands can appear in either order in the input script so long as both are specified before the Python function is invoked for the first time.

Note that python-style, equal-style, vector-style, and atom-style variables can be used in many different ways within LAMMPS. They can be evaluated directly in an input script, effectively replacing the variable with its value. Or they can be passed to various commands as arguments, so that the variable is evaluated multiple times during a simulation run. See the *variable* command doc page for more details on variable styles which enable Python function evaluation.

The Python code for a Python function can be included directly in the input script or in a separate Python file. The function can be standard Python code or it can make "callbacks" to LAMMPS through its library interface to query or set internal values within LAMMPS. This is a powerful mechanism for performing complex operations in a LAMMPS input script that are not possible with the simple input script and variable syntax which LAMMPS defines. Thus your input script can operate more like a true programming language.

Use of this command requires building LAMMPS with the PYTHON package which links to the Python library so that the Python interpreter is embedded in LAMMPS. More details about this process are given below.

A broader overview of how Python can be used with LAMMPS is given in the *Use Python with LAMMPS* section of the documentation. There is also an `examples/python` directory which illustrates use of the python command.

---

The first argument to the *python* command is the *mode* setting, which is either *source* or the *name* of a Python function.

Changed in version 22Dec2022.

If *source* is used, it is followed by either the *here* keyword or a file name containing Python code. The *here* keyword is followed by a single *inline* argument which is a string containing one or more python commands. The string can either be on the same line as the *python* command, enclosed in quotes, or it can be multiple lines enclosed in triple quotes.

In either case, the in-line code or the file contents are passed to the python interpreter and executed immediately. The code will be loaded into and run in the "main" module of the Python interpreter. This allows running arbitrary Python code at any time while processing the LAMMPS input file. This can be used to pre-load Python modules, initialize global variables, define functions or classes, or perform operations using the Python programming language. The Python code will be executed in parallel on all the MPI processes being used to run LAMMPS. Note that no arguments can be passed to the executed Python code.

If the *mode* setting is the *name* of a Python function, then it will be registered with LAMMPS for future execution (or can already be defined, see the *exists* keyword). One or more keywords must follow the *mode* function name. One of the keywords must be *invoke*, *file*, *here*, or *exists*, which specifies what Python code to load into the Python interpreter. Note that only one of those 4 keywords is allowed since their operations are mutually exclusive.

---

If the *invoke* keyword is used, no other keywords can be used. A previous *python* command must have registered the Python function referenced by this command, which can then be invoked multiple times in an input script via the *invoke* keyword. Each invocation passes current values for arguments to the Python function. A return value of the Python function will be ignored unless the Python function is linked to a *python style variable* with the *return* keyword. This return value can be logged to the screen and logfile by adding the optional *logreturn* argument to the *invoke* keyword. In that case a message with the name of the python command and the return value is printed. Note that return values of python functions are otherwise *only* accessible when the function is invoked indirectly by evaluating its associated *python style variable*, as described below.

The *file* keyword gives the name of a file containing Python code, which should end with a ".py" suffix. The code will be immediately loaded into and run in the "main" module of the Python interpreter. The Python code will be executed in parallel on all MPI processes. Note that Python code which contains a function definition does NOT "execute" the function when it is run; it simply defines the function so that it can be invoked later.

The *here* keyword does the same thing, except that the Python code follows as a single argument to the *here* keyword. This can be done using triple quotes as delimiters, as in the examples above and below. This allows Python code to be listed verbatim in your input script, with proper indentation, blank lines, and comments, as desired. See the *Commands parse* doc page, for an explanation of how triple quotes can be used as part of input script syntax.

The *exists* keyword takes no argument. It simply means that Python code containing the needed Python function has already been loaded into the LAMMPS Python interpreter, for example by previous *python source* command or in a file that was loaded previously with the *file* keyword. This allows use of a single file of Python code which contains multiple functions, any of which can be used in the same (or different) input scripts (see below).

Note that the Python code that is loaded and run by the *file* or *here* keyword must contain a function with the specified function *name*. To operate properly when the function is later invoked, the code for the function must match the *input* and *return* and *format* keywords specified by the python command. Otherwise Python will generate an error.

---

The other keywords which can be used with the *python* command are *input*, *return*, *format*, and *length*.

The *input* keyword defines how many arguments *N* the Python function expects. If it takes no arguments, then the *input* keyword should not be used. Each argument can be specified directly as a value, e.g. '6' or '3.14159' or 'abc' (a string of characters). The type of each argument is specified by the *format* keyword as explained below, so that Python will know how to interpret the value. If the word SELF is used for an argument it has a special meaning. A pointer is passed to the Python function which it can convert into a reference to LAMMPS itself using the *LAMMPS Python module*. This enables the function to call back to LAMMPS through its library interface as explained below. This allows the Python function to query or set values internal to LAMMPS which can affect the subsequent execution of the input script.

A LAMMPS variable can also be used as an *input* argument, specified as v_name, where "name" is the name of the variable defined in the input script. Any style of LAMMPS variable returning a scalar or a string can be used, as defined by the *variable* command. The style of variable must be consistent with the *format* keyword specification for the type of data that is passed to Python. Each time the Python function is invoked, the LAMMPS variable is evaluated and its value is passed as an argument to the Python function. Note that a python-style variable can be used as an argument, which means that the a Python function can use arguments which invoke other Python functions.

A LAMMPS internal-style variable can also be used as an *input* argument, specified as iv_name, where "name" is the name of the internal-style variable. The internal-style variable does not have to be defined in the input script (though it can be); if it is not defined, this command creates an *internal-style variable* with the specified name.

An internal-style variable must be used when an equal-style, vector-style, or atom-style variable triggers the invocation of the Python function defined by this command, by including a Python function wrapper with arguments in its formula. Each of the arguments must be specified as an internal-style variable via the *input* keyword.

In brief, the syntax for a Python function wrapper in a variable formula is `py_varname(arg1,arg2,...argN)`, where "varname" is the name of a python-style variable associated with a Python function defined by this command. One or more arguments to the function wrapper can themselves be sub-formulas which the variable command will evaluate and pass as arguments to the Python function. This is done by assigning the numeric result for each argument to an internal-style variable; thus the *input* keyword must specify the arguments as internal-style variables and their format (see below) as "f" for floating point. This is because LAMMPS variable formulas are calculated with floating point arithmetic (any integer values are converted to floating point). Note that the Python function can also have additional inputs, also specified by the *input* keyword, which are NOT arguments in the Python function wrapper. See the example below for the `mixedargs` Python function.

See the *variable* command doc page for full details on formula syntax including for Python function wrappers. Examples using Python function wrappers are shown below. Note that as explained above with python-style variables, Python function wrappers can be nested; a sub-formula for an argument can contain its own Python function wrapper which invokes another Python function.

The *return* keyword is only needed if the Python function returns a value. The specified *varReturn* is of the form v_name, where "name" is the name of a python-style LAMMPS variable, defined by the *variable* command. The Python function can return a numeric or string value, as specified by the *format* keyword. This return value is *only* accessible when its associated python-style variable is evaluated. When the *invoke* keyword is used, the return value of the python function is ignored unless the optional *logreturn* argument is specified.

The *format* keyword must be used if the *input* or *return* keywords are used. It defines an *fstring* with M characters, where M = sum of number of inputs and outputs. The order of characters corresponds to the N inputs, followed by

the return value (if it exists). Each character must be one of the following: "i" for integer, "f" for floating point, "s" for string, or "p" for SELF. Each character defines the type of the corresponding input or output value of the Python function and affects the type conversion that is performed internally as data is passed back and forth between LAMMPS and Python. Note that it is permissible to use a *python-style variable* in a LAMMPS command that allows for an equal-style variable as an argument, but only if the output of the Python function is flagged as a numeric value ("i" or "f") via the *format* keyword.

If the *return* keyword is used and the *format* keyword specifies the output as a string, then the default maximum length of that string is 63 characters (64-1 for the string terminator). If you want to return a longer string, the *length* keyword can be specified with its *Nlen* value set to a larger number. LAMMPS will then allocate Nlen+1 space to include the string terminator. If the Python function generates a string longer than the default 63 or the specified *Nlen*, it will be truncated.

---

This section describes how Python code can be written to work with LAMMPS.

Whether you load Python code from a file or directly from your input script, via the *file* and *here* keywords, the code can be identical. It must be indented properly as Python requires. It can contain comments or blank lines. If the code is in your input script, it cannot however contain triple-quoted Python strings, since that will conflict with the triple-quote parsing that the LAMMPS input script performs.

All the Python code you specify via one or more python commands is loaded into the Python "main" module, i.e. `__name__ == '__main__'`. The code can define global variables, define global functions, define classes or execute statements that are outside of function definitions. It can contain multiple functions, only one of which matches the *func* setting in the python command. This means you can use the *file* keyword once to load several functions, and the *exists* keyword thereafter in subsequent python commands to register the other functions that were previously loaded with LAMMPS.

A Python function you define (or more generally, the code you load) can import other Python modules or classes, it can make calls to other system functions or functions you define, and it can access or modify global variables (in the "main" module) which will persist between successive function calls. The latter can be useful, for example, to prevent a function from being invoke multiple times per timestep by different commands in a LAMMPS input script that access the returned python-style variable associated with the function. For example, consider this function loaded with two global variables defined outside the function:

```
nsteplast = -1
nvaluelast = 0

def expensive(nstep):
  global nsteplast,nvaluelast
  if nstep == nsteplast: return nvaluelast
  nsteplast = nstep
  # perform complicated calculation
  nvalue = ...
  nvaluelast = nvalue
  return nvalue
```

The variable 'nsteplast' stores the previous timestep the function was invoked (passed as an argument to the function). The variable 'nvaluelast' stores the return value computed on the last function invocation. If the function is invoked again on the same timestep, the previous value is simply returned, without re-computing it. The "global" statement inside the Python function allows it to overwrite the global variables from within the local context of the function.

Also note that if you load Python code multiple times (via multiple python commands), you can overwrite previously loaded variables and functions if you are not careful. E.g. if the code above were loaded twice, the global variables would be re-initialized, which might not be what you want. Likewise, if a function with the same name exists in two chunks of Python code you load, the function loaded second will override the function loaded first.

It's important to realize that if you are running LAMMPS in parallel, each MPI task will load the Python interpreter and execute a local copy of the Python function(s) you define. There is no connection between the Python interpreters running on different processors. This implies three important things.

First, if you put a print or other statement creating output to the screen in your Python function, you will see P copies of the output, when running on P processors. If the prints occur at (nearly) the same time, the P copies of the output may be mixed together.

It is possible to avoid this issue, by passing the pointer to the current LAMMPS class instance to the Python function via the {input} SELF argument described above. The Python function can then use the Python interface to the LAMMPS library interface, and determine the MPI rank of the current process. The Python code can then ensure output will only appear on MPI rank 0. The following LAMMPS input demonstrates how this could be done. The text 'Hello, LAMPS!' should be printed only once, even when running LAMMPS in parallel.

```
python python_hello input 1 SELF format p here """
def python_hello(handle):
    from lammps import lammps
    lmp = lammps(ptr=handle)
    me = lmp.extract_setting('world_rank')
    if me == 0:
        print('Hello, LAMMPS!')
"""

python python_hello invoke
```

Second, if your Python code loads Python modules that are not pre-loaded by the Python library, then it will load the module from disk. This may be a bottleneck if 1000s of processors try to load a module at the same time. On some large supercomputers, loading of modules from disk by Python may be disabled. In this case you would need to pre-build a Python library that has the required modules pre-loaded and link LAMMPS with that library.

Third, if your Python code calls back to LAMMPS (discussed in the next section) and causes LAMMPS to perform an MPI operation requires global communication (e.g. via MPI_Allreduce), such as computing the global temperature of the system, then you must ensure all your Python functions (running independently on different processors) call back to LAMMPS. Otherwise the code may hang.

---

As mentioned above, a Python function can "call back" to LAMMPS through its library interface, if the SELF input is used to pass Python a pointer to LAMMPS. The mechanism for doing this is as follows:

```
def foo(handle,...):
  from lammps import lammps
  lmp = lammps(ptr=handle)
  lmp.command('print "Hello from inside Python"')
  ...
```

The function definition must include a variable ('handle' in this case) which corresponds to SELF in the *python* command. The first line of the function imports the *"lammps" Python module*. The second line creates a Python object `lmp` which wraps the instance of LAMMPS that called the function. The 'ptr=handle' argument is what makes that happen. The third line invokes the command() function in the LAMMPS library interface. It takes a single string argument which is a LAMMPS input script command for LAMMPS to execute, the same as if it appeared in your input script. In this case, LAMMPS should output

```
Hello from inside Python
```

to the screen and log file. Note that since the LAMMPS print command itself takes a string in quotes as its argument, the Python string must be delimited with a different style of quotes.

The *Use Python with LAMMPS* page describes the syntax for how Python wraps the various functions included in the LAMMPS library interface.

A more interesting example is in the `examples/python/in.python` script which loads and runs the following function from `examples/python/funcs.py`:

```
def loop(N,cut0,thresh,lmpptr):
  print("LOOP ARGS", N, cut0, thresh, lmpptr)
  from lammps import lammps
  lmp = lammps(ptr=lmpptr)
  natoms = lmp.get_natoms()

  for i in range(N):
    cut = cut0 + i*0.1

    lmp.set_variable("cut",cut)                 # set a variable in LAMMPS
    lmp.command("pair_style lj/cut ${cut}")     # LAMMPS command
    #lmp.command("pair_style lj/cut %d" % cut)  # alternate form of LAMMPS command

    lmp.command("pair_coeff * * 1.0 1.0")       # ditto
    lmp.command("run 10")                       # ditto
    pe = lmp.extract_compute("thermo_pe",0,0)   # extract total PE from LAMMPS
    print("PE", pe/natoms, thresh)
    if pe/natoms < thresh: return
```

with these input script commands:

```
python          loop input 4 10 1.0 -4.0 SELF format iffp file funcs.py
python          loop invoke
```

This has the effect of looping over a series of 10 short runs (10 timesteps each) where the pair style cutoff is increased from a value of 1.0 in distance units, in increments of 0.1. The looping stops when the per-atom potential energy falls below a threshold of -4.0 in energy units. More generally, Python can be used to implement a loop with complex logic, much more so than can be created using the LAMMPS *jump* and *if* commands.

Several LAMMPS library functions are called from the loop function. Get_natoms() returns the number of atoms in the simulation, so that it can be used to normalize the potential energy that is returned by extract_compute() for the "thermo_pe" compute that is defined by default for LAMMPS thermodynamic output. Set_variable() sets the value of a string variable defined in LAMMPS. This library function is a useful way for a Python function to return multiple values to LAMMPS, more than the single value that can be passed back via a return statement. This cutoff value in the "cut" variable is then substituted (by LAMMPS) in the pair_style command that is executed next. Alternatively, the "alternate form of LAMMPS command" line could be used in place of the 2 preceding lines, to have Python insert the value into the LAMMPS command string.

**Note**

When using the callback mechanism just described, recognize that there are some operations you should not attempt because LAMMPS cannot execute them correctly. If the Python function is invoked between runs in the LAMMPS input script, then it should be OK to invoke any LAMMPS input script command via the library interface command() or file() functions, so long as the command would work if it were executed in the LAMMPS input script directly at the same point.

---

As noted above, a Python function can be invoked during a run, whenever an associated python-style variable it is

assigned to is evaluated.

If the variable is an input argument to another LAMMPS command (e.g. *fix setforce*), then the Python function will be invoked inside the class for that command, possibly in one of its methods that is invoked in the middle of a timestep. You cannot execute arbitrary input script commands from the Python function (again, via the command() or file() functions) at that point in the run and expect it to work. Other library functions such as those that invoke computes or other variables may have hidden side effects as well. In these cases, LAMMPS has no simple way to check that something illogical is being attempted.

The same constraints apply to Python functions called during a simulation run at each time step using the *fix python/invoke* command.

---

As noted above, a Python function can also be invoked within the formula for an equal-style, vector-style, or atom-style variable. This means the Python function will be invoked whenever that variable is invoked. In the case of a vector-style variable, the Python function can be invoked once per element of the global vector. In the case of an atom-style variable, the Python function can be invoked once per atom.

Here are three simple examples using equal-, vector-, and atom-style variables to trigger execution of a Python function:

```
variable        foo python truncate
python          truncate return v_foo input 1 iv_arg format fi here """
def truncate(x):
  return int(x)
"""
variable        ptrunc equal py_foo(press)
print           "TRUNCATED pressure = ${ptrunc}"
```

The Python `truncate` function simply converts a floating-point value to an integer value. When the LAMMPS print command evaluates the equal-style `ptrunc` variable, the current thermodynamic pressure is passed to the Python function. The truncated value is output to the screen and logfile by the print command. Note that the *input* keyword for the *python* command, specifies an internal-style variable named "arg" as iv_arg which is required to invoke the Python function from a Python function wrapper.

The last 2 lines can be replaced by these to define a vector-style variable which invokes the same Python `truncate` function:

```
compute         ke all temp
variable        ke vector c_ke
variable        ketrunc vector py_foo(v_ke)
thermo_style    custom step temp epair v_ketrunc[*6]
```

The vector-style variable `ketrunc` invokes the Python `truncate` function on each of the 6 components of the global kinetic energy tensor calculated by the *compute ke* command. The 6 truncated values will be printed with thermo output to the screen and log file.

Or the last 2 lines of the equal-style variable example can be replaced by these to define atom-style variables which invoke the same Python `truncate` function:

```
variable        xtrunc atom py_foo(x)
variable        ytrunc atom py_foo(y)
variable        ztrunc atom py_foo(z)
dump            1 all custom 100 tmp.dump id x y z v_xtrunc v_ytrunc v_ztrunc
```

When the dump command invokes the 3 atom-style variables, their arguments x,y,z to the Python function wrapper are the current per-atom coordinates of each atom. The Python `truncate` function is thus invoked 3 times for each atom, and the truncated coordinate values for each atom are written to the dump file.

Note that when using a Python function wrapper in a variable, arguments can be passed to the Python function either from the variable formula or by *input* keyword to the *python command*. For example, consider these (made up) commands:

```
variable        foo python mixedargs
python          mixedargs return v_foo input 6 7.5 v_myValue iv_arg1 iv_argy iv_argz v_
↪flag &
                format fffffsf here """
def mixedargs(a,b,x,y,z,flag):
  ...
  return result
"""
variable        flag string optionABC
variable        myValue equal "2.0*temp*c_pe"
compute         pe all pe
compute         peatom all pe/atom
variable        field atom py_foo(x+3.0,sqrt(y),(z-zlo)*c_peatom)
```

They define a Python `mixedargs` function with 6 arguments. Three of them are internal-style variables, which the variable formula calculates as numeric values for each atom and passes to the function. In this example, these arguments are themselves small formulas containing the x,y,z coordinates of each atom as well as a per-atom compute (c_peratom) and thermodynamic keyword (zlo).

The other three arguments `(7.5,v_myValue,v_flag)` are defined by the *python* command. The first and last are constant values ("7.5" and the `optionABC` string). The second argument (`myValue`) is the result of an equal-style variable formula which accesses the system temperature and potential energy.

The "result" returned by each invocation of the Python `mixedargs` function becomes the per-atom value in the atom-style "field" variable, which could be output to a dump file or used elsewhere in the input script.

---

If you run Python code directly on your workstation, either interactively or by using Python to launch a Python script stored in a file, and your code has an error, you will typically see informative error messages. That is not the case when you run Python code from LAMMPS using an embedded Python interpreter. The code will typically fail silently. LAMMPS will catch some errors but cannot tell you where in the Python code the problem occurred. For example, if the Python code cannot be loaded and run because it has syntax or other logic errors, you may get an error from Python pointing to the offending line, or you may get one of these generic errors from LAMMPS:

```
Could not process Python file
Could not process Python string
```

When the Python function is invoked, if it does not return properly, you will typically get this generic error from LAMMPS:

```
Python function evaluation failed
```

Here are three suggestions for debugging your Python code while running it under LAMMPS.

First, don't run it under LAMMPS, at least to start with! Debug it using plain Python. Load and invoke your function, pass it arguments, check return values, etc.

Second, add Python print statements to the function to check how far it gets and intermediate values it calculates. See the discussion above about printing from Python when running in parallel.

Third, use Python exception handling. For example, say this statement in your Python function is failing, because you have not initialized the variable foo:

```
foo += 1
```

If you put one (or more) statements inside a "try" statement, like this:

```
import exceptions
print("Inside simple function")
try:
  foo += 1      # one or more statements here
except Exception as e:
  print("FOO error:", e)
```

then you will get this message printed to the screen:

```
FOO error: local variable 'foo' referenced before assignment
```

If there is no error in the try statements, then nothing is printed. Either way the function continues on (unless you put a return or sys.exit() in the except clause).

---

### 1.83.4 Restrictions

This command is part of the PYTHON package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

If you use Python code which calls back to LAMMPS, via the SELF input argument explained above, there is an extra step required when building LAMMPS. LAMMPS must also be built as a shared library and your Python function must be able to load the *"lammps" Python module* that wraps the LAMMPS library interface.

These are the same steps required to use Python by itself to wrap LAMMPS. Details on these steps are explained on the *Python* doc page. Note that it is important that the stand-alone LAMMPS executable and the LAMMPS shared library be consistent (built from the same source code files) in order for this to work. If the two have been built at different times using different source files, problems may occur.

Another limitation of calling back to Python from the LAMMPS module using the *python* command in a LAMMPS input is that both, the Python interpreter and LAMMPS, must be linked to the same Python runtime as a shared library. If the Python interpreter is linked to Python statically (which seems to happen with Conda) then loading the shared LAMMPS library will create a second python "main" module that hides the one from the Python interpreter and all previous defined function and global variables will become invisible.

### 1.83.5 Related commands

*shell*, *variable*, *fix python/invoke*

### 1.83.6 Default

none

## 1.84 quit command

### 1.84.1 Syntax

```
quit status
```

status = numerical exit status (optional)

### 1.84.2 Examples

```
quit
if "$n > 10000" then "quit 1"
```

### 1.84.3 Description

This command causes LAMMPS to exit, after shutting down all output cleanly.

It can be used as a debug statement in an input script, to terminate the script at some intermediate point.

It can also be used as an invoked command inside the "then" or "else" portion of an *if* command.

The optional status argument is an integer which signals the return status to a program calling LAMMPS. A return status of 0 usually indicates success. A status != 0 is failure, where the specified value can be used to distinguish the kind of error, e.g. where in the input script the quit was invoked. If not specified, a status of 0 is returned.

### 1.84.4 Restrictions

none

### 1.84.5 Related commands

*if*

### 1.84.6 Default

none

## 1.85 read_data command

### 1.85.1 Syntax

```
read_data file keyword args ...
```

- file = name of data file to read in
- zero or more keyword/arg pairs may be appended
- keyword = *add* or *offset* or *shift* or *extra/atom/types* or *extra/bond/types* or *extra/angle/types* or *extra/dihedral/types* or *extra/improper/types* or *extra/bond/per/atom* or *extra/angle/per/atom* or *extra/dihedral/per/atom* or *extra/improper/per/atom* or *extra/special/per/atom* or *group* or *nocoeff* or *fix*

```
add arg = append or IDoffset or IDoffset MOLoffset or merge
  append = add new atoms with atom IDs appended to current IDs
  IDoffset = add new atoms with atom IDs having IDoffset added
  MOLoffset = add new atoms with molecule IDs having MOLoffset added (only when␣
↪molecule IDs are enabled)
  merge = add new atoms with their atom IDs (and molecule IDs) unchanged
offset args = toff boff aoff doff ioff
  toff = offset to add to atom types
  boff = offset to add to bond types
  aoff = offset to add to angle types
  doff = offset to add to dihedral types
  ioff = offset to add to improper types
shift args = Sx Sy Sz
```

```
  Sx,Sy,Sz = distance to shift atoms when adding to system (distance units)
extra/atom/types arg = # of extra atom types
extra/bond/types arg = # of extra bond types
extra/angle/types arg = # of extra angle types
extra/dihedral/types arg = # of extra dihedral types
extra/improper/types arg = # of extra improper types
extra/bond/per/atom arg = leave space for this many new bonds per atom
extra/angle/per/atom arg = leave space for this many new angles per atom
extra/dihedral/per/atom arg = leave space for this many new dihedrals per atom
extra/improper/per/atom arg = leave space for this many new impropers per atom
extra/special/per/atom arg = leave space for extra 1-2,1-3,1-4 interactions per atom
group args = groupID
  groupID = add atoms in data file to this group
nocoeff = ignore force field parameters
fix args = fix-ID header-string section-string
  fix-ID = ID of fix to process header lines and sections of data file
  header-string = header lines containing this string will be passed to fix
  section-string = section names with this string will be passed to fix
```

### 1.85.2 Examples

```
read_data data.lj
read_data ../run7/data.polymer.gz
read_data data.protein fix mycmap crossterm CMAP
read_data data.water add append offset 3 1 1 1 1 shift 0.0 0.0 50.0
read_data data.water add merge group solvent
```

### 1.85.3 Description

Read in a data file containing information LAMMPS needs to run a simulation. The file can be ASCII text or a compressed text file (detected by its suffix) if LAMMPS has been compiled with support for *compression commands*.

This is one of 3 ways to specify the simulation box: see the *create_box* and *read_restart* and commands for alternative methods. It is also one of 3 ways to specify initial atom coordinates: see the *create_atoms* and *read_restart* and commands for alternative methods. Also see the explanation of the *-restart command-line switch* which can convert a restart file to a data file.

This command can be used multiple times to add new atoms and their properties to an existing system by using the *add*, *offset*, and *shift* keywords. However, it is important to understand that several system parameters, like the number of types of different kinds and per atom settings are **locked in** after the first *read_data* command, which means that no type ID (including its offset) may have a larger value when processing additional data files than what is set by the first data file and the corresponding *read_data* command options. See more details on this situation below, which includes the use case for the *extra* keywords.

The *group* keyword adds all the atoms in the data file to the specified group-ID. The group will be created if it does not already exist. This is useful if you are reading multiple data files and wish to put sets of atoms into different groups so they can be operated on later. E.g. a group of added atoms can be moved to new positions via the *displace_atoms* command. Note that atoms read from the data file are also always added to the "all" group. The *group* command discusses atom groups, as used in LAMMPS.

The *nocoeff* keyword tells read_data to ignore force field parameters. The various Coeff sections are still read and have to have the correct number of lines, but they are not applied. This also allows to read a data file without having any pair, bond, angle, dihedral or improper styles defined, or to read a data file for a different force field.

The use of the *fix* keyword is discussed below.

### 1.85.4 Reading multiple data files

The read_data command can be used multiple times with the same or different data files to build up a complex system from components contained in individual data files. For example one data file could contain fluid in a confined domain; a second could contain wall atoms, and the second file could be read a third time to create a wall on the other side of the fluid. The third set of atoms could be rotated to an opposing direction using the *displace_atoms* command, after the third read_data command is used.

The *add*, *offset*, *shift*, *extra*, and *group* keywords are useful in this context.

If a simulation box does not yet exist, the *add* keyword cannot be used; the read_data command is being used for the first time. If a simulation box does exist, due to using the *create_box* command, or a previous read_data command, then the *add* keyword must be used.

**Note**

If the first read_data command defined an orthogonal or restricted triclinic or general triclinic simulation box (see the sub-section below on header keywords), then subsequent data files must define the same kind of simulation box. For orthogonal boxes, the new box can be a different size; see the next Note. For a restricted triclinic box, the 3 new tilt factors ("xy xz yz" keyword) must have the same values as in the original data file. For a general triclinic box, the new avec, bvec, cvec, and "abc origin" keywords must have the same values in the original data file. files. Also the *shift* keyword cannot be used in subsequent read_data commands for a general triclinic box.

**Note**

For orthogonal boxes, the simulation box size in the new data file will be merged with the existing simulation box to create a large enough box in each dimension to contain both the existing and new atoms. Each box dimension never shrinks due to this merge operation, it only stays the same or grows. Care must be used if you are growing the existing simulation box in a periodic dimension. If there are existing atoms with bonds that straddle that periodic boundary, then the atoms may become far apart if the box size grows. This will separate the atoms in the bond, which can lead to "lost" bond atoms or bad dynamics.

The three choices for the *add* argument affect how the atom IDs and molecule IDs of atoms in the data file are treated.

If *append* is specified, atoms in the data file are added to the current system, with their atom IDs reset so that an atom-ID = M in the data file becomes atom-ID = N+M, where N is the largest atom ID in the current system. This rule is applied to all occurrences of atom IDs in the data file, e.g. in the Velocity or Bonds section. This is also done for molecule IDs, if the atom style does support molecule IDs or they are enabled via fix property/atom.

If *IDoffset* is specified, then *IDoffset* is a numeric value is given, e.g. 1000, so that an atom-ID = M in the data file becomes atom-ID = 1000+M. For systems with enabled molecule IDs, another numerical argument *MOLoffset* is required representing the equivalent offset for molecule IDs.

If *merge* is specified, the data file atoms are added to the current system without changing their IDs. They are assumed to merge (without duplication) with the currently defined atoms. It is up to you to ensure there are no multiply defined atom IDs, as LAMMPS only performs an incomplete check that this is the case by ensuring the resulting max atom-ID >= the number of atoms. For molecule IDs, there is no check done at all.

The *offset* and *shift* keywords can only be used if the *add* keyword is also specified.

The *offset* keyword adds the specified offset values to the atom types, bond types, angle types, dihedral types, and improper types as they are read from the data file. E.g. if *toff* = 2, and the file uses atom types 1,2,3, then the added atoms will have atom types 3,4,5. These offsets apply to all occurrences of types in the data file, e.g. for the Atoms or Masses or Pair Coeffs or Bond Coeffs sections. This makes it easy to use atoms and molecules and their attributes from a data file in different simulations, where you want their types (atom, bond, angle, etc) to be different depending

on what other types already exist. All five offset values must be specified, but individual values will be ignored if the data file does not use that attribute (e.g. no bonds).

**Note**

Offsets are **ignored** on lines using type labels, as the type labels will determine the actual types directly depending on the current *labelmap* settings.

The *shift* keyword can be used to specify an (Sx, Sy, Sz) displacement applied to the coordinates of each atom. Sz must be 0.0 for a 2d simulation. This is a mechanism for adding structured collections of atoms at different locations within the simulation box, to build up a complex geometry. It is up to you to ensure atoms do not end up overlapping unphysically which would lead to bad dynamics. Note that the *displace_atoms* command can be used to move a subset of atoms after they have been read from a data file. Likewise, the *delete_atoms* command can be used to remove overlapping atoms. Note that the shift values (Sx, Sy, Sz) are also added to the simulation box information (xlo, xhi, ylo, yhi, zlo, zhi) in the data file to shift its boundaries. E.g. xlo_new = xlo + Sx, xhi_new = xhi + Sx.

The *extra* keywords can only be used the first time the read_data command is used. They are useful if you intend to add new atom, bond, angle, etc types later with additional read_data commands. This is because the maximum number of allowed atom, bond, angle, etc types is set by LAMMPS when the system is first initialized. If you do not use the *extra* keywords, then the number of these types will be limited to what appears in the first data file you read. For example, if the first data file is a solid substrate of Si, it will likely specify a single atom type. If you read a second data file with a different material (water molecules) that sit on top of the substrate, you will want to use different atom types for those atoms. You can only do this if you set the *extra/atom/types* keyword to a sufficiently large value when reading the substrate data file. Note that use of the *extra* keywords also allows each data file to contain sections like Masses or Pair Coeffs or Bond Coeffs which are sized appropriately for the number of types in that data file. If the *offset* keyword is used appropriately when each data file is read, the values in those sections will be stored correctly in the larger data structures allocated by the use of the *extra* keywords. E.g. the substrate file can list mass and pair coefficients for type 1 silicon atoms. The water file can list mass and pair coefficients for type 1 and type 2 hydrogen and oxygen atoms. Use of the *extra* and *offset* keywords will store those mass and pair coefficient values appropriately in data structures that allow for 3 atom types (Si, H, O). Of course, you would still need to specify coefficients for H/Si and O/Si interactions in your input script to have a complete pairwise interaction model.

An alternative to using the *extra* keywords with the read_data command, is to use the *create_box* command to initialize the simulation box and all the various type limits you need via its *extra* keywords. Then use the read_data command one or more times to populate the system with atoms, bonds, angles, etc, using the *offset* keyword if desired to alter types used in the various data files you read.

---

### 1.85.5 Format of a data file

The structure of the data file is important, though many settings and sections are optional or can come in any order. See the examples directory for sample data files for different problems.

The file will be read line by line, but there is a limit of 254 characters per line and characters beyond that limit will be ignored.

A data file has a header and a body. The header appears first. The first line of the header and thus of the data file is *always* skipped; it typically contains a description of the file or a comment from the software that created the file.

Then lines are read one line at a time. Lines can have a trailing comment starting with '#' that is ignored. There *must* be at least one blank between any valid content and the comment. If a line is blank (i.e. contains only white-space after comments are deleted), it is skipped. If the line contains a header keyword, the corresponding value(s) is/are read from the line. A line that is *not* blank and does *not* contain a header keyword begins the body of the file.

The body of the file contains zero or more sections. The first line of a section has only a keyword. This line can have a trailing comment starting with '#' that is either ignored or can be used to check for a style match, as described below.

There must be a blank between the keyword and any comment. The *next* line is *always* skipped. The remaining lines of the section contain values. The number of lines depends on the section keyword as described below. Zero or more blank lines can be used *between* sections. Sections can appear in any order, with a few exceptions as noted below.

The keyword *fix* can be used one or more times. Each usage specifies a fix that will be used to process a specific portion of the data file. Any header line containing *header-string* and any section that is an exact match to *section-string* will be passed to the specified fix. See the *fix property/atom* command for an example of a fix that operates in this manner. The doc page for the fix defines the syntax of the header line(s) and section that it reads from the data file. Note that the *header-string* can be specified as NULL, in which case no header lines are passed to the fix. This means the fix can infer the length of its Section from standard header settings, such as the number of atoms. Also the *section-string* may be specified as NULL, and in that case the fix ID is used as section name.

The formatting of individual lines in the data file (indentation, spacing between words and numbers) is not important except that header and section keywords (e.g. atoms, xlo xhi, Masses, Bond Coeffs) must be capitalized as shown and cannot have extra white-space between their words - e.g. two spaces or a tab between the 2 words in "xlo xhi" or the 2 words in "Bond Coeffs", is not valid.

---

### 1.85.6 Format of the header of a data file

These are the recognized header keywords. Header lines can come in any order. Each keyword takes a single value unless noted in this list. The value(s) are read from the beginning of the line. Thus the keyword *atoms* should be in a line like "1000 atoms"; the keyword *ylo yhi* should be in a line like "-10.0 10.0 ylo yhi"; the keyword *xy xz yz* should be in a line like "0.0 5.0 6.0 xy xz yz".

All these settings have a default value of 0, except for the simulation box size settings; their defaults are explained below. A keyword line need only appear if its value is different than the default.

- *atoms* = # of atoms in system
- *bonds* = # of bonds in system
- *angles* = # of angles in system
- *dihedrals* = # of dihedrals in system
- *impropers* = # of impropers in system
- *atom types* = # of atom types in system
- *bond types* = # of bond types in system
- *angle types* = # of angle types in system
- *dihedral types* = # of dihedral types in system
- *improper types* = # of improper types in system
- *extra bond per atom* = leave space for this many new bonds per atom (deprecated, use extra/bond/per/atom keyword)
- *extra angle per atom* = leave space for this many new angles per atom (deprecated, use extra/angle/per/atom keyword)
- *extra dihedral per atom* = leave space for this many new dihedrals per atom (deprecated, use extra/dihedral/per/atom keyword)
- *extra improper per atom* = leave space for this many new impropers per atom (deprecated, use extra/improper/per/atom keyword)
- *extra special per atom* = leave space for this many new special bonds per atom (deprecated, use extra/special/per/atom keyword)

- *ellipsoids* = # of ellipsoids in system
- *lines* = # of line segments in system
- *triangles* = # of triangles in system
- *bodies* = # of bodies in system
- *xlo xhi* = simulation box boundaries in x dimension (2 values)
- *ylo yhi* = simulation box boundaries in y dimension (2 values)
- *zlo zhi* = simulation box boundaries in z dimension (2 values)
- *xy xz yz* = simulation box tilt factors for triclinic system (3 values)
- *avec* = first edge vector of a general triclinic simulation box (3 values)
- *bvec* = second edge vector of a general triclinic simulation box (3 values)
- *cvec* = third edge vector of a general triclinic simulation box (3 values)
- *abc origin* = origin of a general triclinic simulation box (3 values)

---

### 1.85.7 Header specification of the simulation box size and shape

The last 8 keywords in the list of header keywords are for simulation boxes of 3 kinds which LAMMPS supports:

- orthogonal box = faces are perpendicular to the xyz coordinate axes
- restricted triclinic box = a parallelepiped defined by 3 edge vectors oriented in a constrained manner
- general triclinic box = a parallelepiped defined by 3 arbitrary edge vectors

For restricted and general triclinic boxes, see the *Howto_triclinic* doc page for a fuller description than is given here.

The units of the values for all 8 keywords in in distance units; see the *units* command for details.

For all 3 kinds of simulation boxes, the system may be periodic or non-periodic in any dimension; see the *boundary* command for details.

When the simulation box is created by the read_data command, it is also partitioned into a regular 3d grid of subdomains, one per processor, based on the number of processors being used and the settings of the *processors* command. For each kind of simulation box the subdomains have the same shape as the simulation box, i.e. smaller orthogonal bricks for orthogonal boxes, smaller parallelepipeds for triclinic boxes. The partitioning can later be changed by the *balance* or *fix balance* commands.

For an orthogonal box, only the *xlo xhi*, *ylo yhi*, *zlo zhi* keywords are used. They define the extent of the simulation box in each dimension so that the resulting edge vectors of an orthogonal box are:

- **A** = (xhi-xlo,0,0)
- **B** = (0,yhi-ylo,0)
- **C** = (0,0,zhi-zlo)

The origin (lower left corner) of the orthogonal box is at (xlo,ylo,zlo). The default values for these 3 keywords are -0.5 and 0.5 for each lo/hi pair. For a 2d simulation, the zlo and zhi values must straddle zero. The default zlo/zhi values do this, so that keyword is not needed in 2d.

For a restricted triclinic box, the *xy xz yz* keyword is used in addition to the *xlo xhi*, *ylo yhi*, *zlo zhi* keywords. The three *xy,xz,yz* values can be 0.0 or positive or negative, and are called "tilt factors" because they are the amount of displacement applied to edges of faces of an orthogonal box to transform it into a restricted triclinic parallelepiped.

The *Howto_triclinic* doc page discusses the tilt factors in detail and explains that the resulting edge vectors of a restricted triclinic box are:

- **A** = (xhi-xlo,0,0)
- **B** = (xy,yhi-ylo,0)
- **C** = (xz,yz,zhi-zlo)

This restricted form of edge vectors requires that **A** be in the direction of the x-axis, **B** be in the xy plane with its y-component in the +y direction, and **C** have its z-component in the +z direction. The origin (lower left corner) of the restricted triclinic box is at (xlo,ylo,zlo).

For a 2d simulation, the zlo and zhi values must straddle zero. The default zlo/zhi values do this, so that keyword is not needed in 2d. The xz and yz values must also be zero in 2d. The shape of the 2d restricted triclinic simulation box is effectively a parallelogram.

**Note**

When a restricted triclinic box is used, the simulation domain should normally be periodic in any dimensions that tilt is applied to, which is given by the second dimension of the tilt factor (e.g. y for xy tilt). This is so that pairs of atoms interacting across that boundary will have one of them shifted by the tilt factor. Periodicity is set by the *boundary* command which also describes the shifting by the tilt factor. For example, if the xy tilt factor is non-zero, then the y dimension should be periodic. Similarly, the z dimension should be periodic if xz or yz is non-zero. LAMMPS does not require this periodicity, but you may lose atoms if this is not the case.

**Note**

Normally, the specified tilt factors (xy,xz,yz) should not skew the simulation box by more than half the distance of the corresponding parallel box length for computational efficiency. For example, if $x_{\rm lo} = 2$ and $x_{\rm hi} = 12$, then the $x$ box length is 10 and the $xy$ tilt factor should be between $-5$ and 5. LAMMPS will issue a warning if this is not the case. See the last sub-section of the *Howto_triclinic* doc page for more details.

**Note**

If a simulation box is initially orthogonal, but will tilt during a simulation, e.g. via the *fix deform* command, then the box should be defined as restricted triclinic with all 3 tilt factors = 0.0. Alternatively, the *change box* command can be used to convert an orthogonal box to a restricted triclinic box.

For a general triclinic box, the *avec*, *bvec*, *cvec*, and *abc origin* keywords are used. The *xlo xhi*, *ylo yhi*, *zlo zhi*, and *xy xz yz* keywords are NOT used. The first 3 keywords define the 3 edge vectors **A**, **B**, **C** of the general triclinic box. They can be arbitrary vectors so long as they are distinct, non-zero, and not co-planar. They must also define a right-handed system such that (**A** x **B**) points in the direction of **C**. Note that a left-handed system can be converted to a right-handed system by simply swapping the order of any pair of the **A**, **B**, **C** vectors. The origin of the box (origin of the 3 edge vectors) is set by the *abc origin* keyword.

The default values for these 4 keywords are as follows:

- avec = (1,0,0)
- bvec = (0,1,0)
- cvec = (0,0,1)
- abc origin = (0,0,0) for 3d, (0,0,-0.5) for 2d

For 2d simulations, *cvec* = (0,0,1) is required, and the 3rd value of *abc origin* must be -0.5. These are the default values, so the *cvec* keyword is not needed in 2d.

**Note**

LAMMPS allows specification of general triclinic simulation boxes as a convenience for users who may be converting data from solid-state crystallographic representations or from DFT codes for input to LAMMPS. However, as explained on the *Howto_triclinic* doc page, internally, LAMMPS only uses restricted triclinic simulation boxes. This means the box and per-atom information (e.g. coordinates, velocities) in the data file are converted (rotated) from general to restricted triclinic form when the file is read. Other sections of the data file must also list their per-atom data appropriately if vector quantities are specified. This requirement is explained below for the relevant sections. The *Howto_triclinic* doc page also discusses other LAMMPS commands which can input/output general triclinic representations of the simulation box and per-atom data.

The following explanations apply to all 3 kinds of simulation boxes: orthogonal, restricted triclinic, and general triclinic.

If the system is periodic (in a dimension), then atom coordinates can be outside the bounds (in that dimension); they will be remapped (in a periodic sense) back inside the box. For triclinic boxes, periodicity in x,y,z refers to the faces of the parallelepiped defined by the **A**,**B**,**C** edge vectors of the simulation box. See the *boundary* command doc page for a fuller discussion.

Note that if the *add* option is being used to add atoms to a simulation box that already exists, this periodic remapping will be performed using simulation box bounds that are the union of the existing box and the box boundaries in the new data file.

If the system is non-periodic (in a dimension), then an image flag for that direction has no meaning, since there cannot be periodic images without periodicity and the data file is therefore - technically speaking - invalid. This situation would happen when a data file was written with periodic boundaries and then read back for non-periodic boundaries. Accepting a non-zero image flag can lead to unexpected results for any operations and computations in LAMMPS that internally use unwrapped coordinates (for example computing the center of mass of a group of atoms). Thus all non-zero image flags for non-periodic dimensions will be be reset to zero on reading the data file and LAMMPS will print a warning message, if that happens. This is equivalent to wrapping atoms individually back into the principal unit cell in that direction. This operation is equivalent to the behavior of the *change_box command* when used to change periodicity.

If those atoms with non-zero image flags are involved in bonded interactions, this reset can lead to undesired changes, when the image flag values differ between the atoms, i.e. the bonded interaction straddles domain boundaries. For example a bond can become stretched across the unit cell if one of its atoms is wrapped to one side of the cell and the second atom to the other. In those cases the data file needs to be pre-processed externally to become valid again. This can be done by first unwrapping coordinates and then wrapping entire molecules instead of individual atoms back into the principal simulation cell and finally expanding the cell dimensions in the non-periodic direction as needed, so that the image flag would be zero.

**Note**

If the system is non-periodic (in a dimension), then all atoms in the data file must have coordinates (in that dimension) that are "greater than or equal to" the lo value and "less than or equal to" the hi value. If the non-periodic dimension is of style "fixed" (see the *boundary* command), then the atom coords must be strictly "less than" the hi value, due to the way LAMMPS assign atoms to processors. Note that you should not make the lo/hi values radically smaller/larger than the extent of the atoms. For example, if atoms extend from 0 to 50, you should not specify the box bounds as -10000 and 10000 unless you also use the *processors command*. This is because LAMMPS uses the specified box size to layout the 3d grid of processors. A huge (mostly empty) box will be sub-optimal for performance when using "fixed" boundary conditions (see the *boundary* command). When using "shrink-wrap" boundary conditions (see the *boundary* command), a huge (mostly empty) box may cause a parallel simulation to

lose atoms when LAMMPS shrink-wraps the box around the atoms. The read_data command will generate an error in this case.

---

### 1.85.8 Meaning of other header keywords

The "extra bond per atom" setting (angle, dihedral, improper) is only needed if new bonds (angles, dihedrals, impropers) will be added to the system when a simulation runs, e.g. by using the *fix bond/create* command. Using this header flag is deprecated; please use the *extra/bond/per/atom* keyword (and correspondingly for angles, dihedrals and impropers) in the read_data command instead. Either will pre-allocate space in LAMMPS data structures for storing the new bonds (angles, dihedrals, impropers).

The "extra special per atom" setting is typically only needed if new bonds/angles/etc will be added to the system, e.g. by using the *fix bond/create* command. Or if entire new molecules will be added to the system, e.g. by using the *fix deposit* or *fix pour* commands, which will have more special 1-2,1-3,1-4 neighbors than any other molecules defined in the data file. Using this header flag is deprecated; please use the *extra/special/per/atom* keyword instead. Using this setting will pre-allocate space in the LAMMPS data structures for storing these neighbors. See the *special_bonds* and *molecule* doc pages for more discussion of 1-2,1-3,1-4 neighbors.

**Note**

All of the "extra" settings are only applied in the first data file read and when no simulation box has yet been created; as soon as the simulation box is created (and read_data implies that), these settings are *locked* and cannot be changed anymore. Please see the description of the *add* keyword above for reading multiple data files. If they appear in later data files, they are ignored.

The "ellipsoids" and "lines" and "triangles" and "bodies" settings are only used with *atom_style ellipsoid or line or tri or body* and specify how many of the atoms are finite-size ellipsoids or lines or triangles or bodies; the remainder are point particles. See the discussion of ellipsoidflag and the *Ellipsoids* section below. See the discussion of lineflag and the *Lines* section below. See the discussion of triangleflag and the *Triangles* section below. See the discussion of bodyflag and the *Bodies* section below.

**Note**

For *atom_style template*, the molecular topology (bonds,angles,etc) is contained in the molecule templates read-in by the *molecule* command. This means you cannot set the *bonds*, *angles*, etc header keywords in the data file, nor can you define *Bonds*, *Angles*, etc sections as discussed below. You can set the *bond types*, *angle types*, etc header keywords, though it is not necessary. If specified, they must match the maximum values defined in any of the template molecules.

---

### 1.85.9 Format of the body of a data file

These are the section keywords for the body of the file.

- *Atoms, Velocities, Masses, Ellipsoids, Lines, Triangles, Bodies* = atom-property sections
- *Bonds, Angles, Dihedrals, Impropers* = molecular topology sections
- *Atom Type Labels, Bond Type Labels, Angle Type Labels, Dihedral Type Labels, Improper Type Labels* = type label maps

- *Pair Coeffs, PairIJ Coeffs, Bond Coeffs, Angle Coeffs, Dihedral Coeffs, Improper Coeffs* = force field sections
- *BondBond Coeffs, BondAngle Coeffs, MiddleBondTorsion Coeffs, EndBondTorsion Coeffs, AngleTorsion Coeffs, AngleAngleTorsion Coeffs, BondBond13 Coeffs, AngleAngle Coeffs* = class 2 force field sections

These keywords will check an appended comment for a match with the currently defined style:

- *Atoms, Pair Coeffs, PairIJ Coeffs, Bond Coeffs, Angle Coeffs, Dihedral Coeffs, Improper Coeffs*

For example, these lines:

```
Atoms # sphere
Pair Coeffs # lj/cut
```

will check if the currently-defined *atom_style* is *sphere*, and the current *pair_style* is *lj/cut*. If not, LAMMPS will issue a warning to indicate that the data file section likely does not contain the correct number or type of parameters expected for the currently-defined style.

Each section is listed below in alphabetic order. The format of each section is described including the number of lines it must contain and rules (if any) for where it can appear in the data file.

Any individual line in the various sections can have a trailing comment starting with "#" for annotation purposes. There must be at least one blank between valid content and the comment. E.g. in the Atoms section:

```
10 1 17 -1.0 10.0 5.0 6.0   # salt ion
```

---

*Angle Coeffs* section:

- one line per angle type
- line syntax: ID coeffs

  ```
  ID = angle type (1-N)
  coeffs = list of coeffs
  ```

- example:

  ```
  6 70 108.5 0 0
  ```

The number and meaning of the coefficients are specific to the defined angle style. See the *angle_style* and *angle_coeff* commands for details. Coefficients can also be set via the *angle_coeff* command in the input script.

---

*Angle Type Labels* section:

- one line per angle type
- line syntax: ID label

  ```
  ID = angle type (1-N)
  label = alphanumeric type label
  ```

Define alphanumeric type labels for each numeric angle type. These can be used in the Angles section in place of a numeric type, but only if the this section appears before the Angles section.

See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

---

*AngleAngle Coeffs* section:

- one line per improper type
- line syntax: ID coeffs

```
ID = improper type (1-N)
coeffs = list of coeffs (see improper_coeff)
```

---

*AngleAngleTorsion Coeffs* section:

- one line per dihedral type
- line syntax: ID coeffs

```
ID = dihedral type (1-N)
coeffs = list of coeffs (see dihedral_coeff)
```

---

*Angles* section:

- one line per angle
- line syntax: ID type atom1 atom2 atom3

```
ID = number of angle (1-Nangles)
type = angle type (1-Nangletype, or type label)
atom1,atom2,atom3 = IDs of 1st,2nd,3rd atom in angle
```

example:

```
2 2 17 29 430
```

The three atoms are ordered linearly within the angle. Thus the central atom (around which the angle is computed) is the atom2 in the list. E.g. H,O,H for a water molecule. The *Angles* section must appear after the *Atoms* section.

All values in this section must be integers (1, not 1.0). However, the type can be a numeric value or an alphanumeric label. The latter is only allowed if the type label has been defined by the *labelmap* command or an Angle Type Labels section earlier in the data file. See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

---

*AngleTorsion Coeffs* section:

- one line per dihedral type
- line syntax: ID coeffs

```
ID = dihedral type (1-N)
coeffs = list of coeffs (see dihedral_coeff)
```

---

*Atom Type Labels* section:

- one line per atom type
- line syntax: ID label

```
ID = numeric atom type (1-N)
label = alphanumeric type label
```

Define alphanumeric type labels for each numeric atom type. These can be used in the Atoms section in place of a numeric type, but only if the Atom Type Labels section appears before the Atoms section.

See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

---

*Atoms* section:

- one line per atom
- line syntax: depends on atom style

An *Atoms* section must appear in the data file if natoms > 0 in the header section. The atoms can be listed in any order. These are the line formats for each *atom style* in LAMMPS. As discussed below, each line can optionally have 3 flags (nx,ny,nz) appended to it, which indicate which image of a periodic simulation box the atom is in. These may be important to include for some kinds of analysis.

**Note**

For orthogonal and restricted and general triclinic simulation boxes, the atom coordinates (x,y,z) listed in this section should be inside the corresponding simulation box. For restricted triclinic boxes that means the parallelepiped defined by the *xlo xhi*, *ylo yhi*, *zlo zhi*, and *xy xz yz*, keywords. For general triclinic boxes that means the parallelepiped defined by the 3 edge vectors and origin specified by the *avec*, *bvec*, *cvec*, and *abc origin* header keywords. See the discussion in the header section above about how atom coordinates outside the simulation box are (or are not) remapped to be inside the box.

| | |
|---|---|
| angle | atom-ID molecule-ID atom-type x y z |
| atomic | atom-ID atom-type x y z |
| body | atom-ID atom-type bodyflag mass x y z |
| bond | atom-ID molecule-ID atom-type x y z |
| bpm/sphere | atom-ID molecule-ID atom-type diameter density x y z |
| charge | atom-ID atom-type q x y z |
| dielectric | atom-ID atom-type q x y z mux muy muz area ed em epsilon curvature |
| dipole | atom-ID atom-type q x y z mux muy muz |
| dpd | atom-ID atom-type theta x y z |
| edpd | atom-ID atom-type edpd_temp edpd_cv x y z |
| electron | atom-ID atom-type q espin eradius x y z |
| ellipsoid | atom-ID atom-type ellipsoidflag density x y z |
| full | atom-ID molecule-ID atom-type q x y z |
| line | atom-ID molecule-ID atom-type lineflag density x y z |
| mdpd | atom-ID atom-type rho x y z |
| molecular | atom-ID molecule-ID atom-type x y z |
| peri | atom-ID atom-type volume density x y z |
| rheo | atom-ID atom-type status rho x y z |
| rheo/thermal | atom-ID atom-type status rho energy x y z |
| smd | atom-ID atom-type molecule volume mass kradius cradius x0 y0 z0 x y z |
| sph | atom-ID atom-type rho esph cv x y z |
| sphere | atom-ID atom-type diameter density x y z |
| spin | atom-ID atom-type x y z spx spy spz sp |
| tdpd | atom-ID atom-type x y z cc1 cc2 ... ccNspecies |
| template | atom-ID atom-type molecule-ID template-index template-atom x y z |
| tri | atom-ID molecule-ID atom-type triangleflag density x y z |
| hybrid | atom-ID atom-type x y z sub-style1 sub-style2 ... |

The per-atom values have these meanings and units, listed alphabetically:

- atom-ID = integer ID of atom
- atom-type = type of atom (1-Ntype, or type label)
- bodyflag = 1 for body particles, 0 for point particles
- ccN = chemical concentration for tDPD particles for each species (mole/volume units)
- cradius = contact radius for SMD particles (distance units)
- cv = heat capacity (need units) for SPH particles
- density = density of particle (mass/distance^3 or mass/distance^2 or mass/distance units, depending on dimensionality of particle)
- diameter = diameter of spherical atom (distance units)
- edpd_temp = temperature for eDPD particles (temperature units)
- edpd_cv = volumetric heat capacity for eDPD particles (energy/temperature/volume units)
- ellipsoidflag = 1 for ellipsoidal particles, 0 for point particles
- eradius = electron radius (or fixed-core radius)
- esph = energy (need units) for SPH particles
- espin = electron spin (+1/-1), 0 = nuclei, 2 = fixed-core, 3 = pseudo-cores (i.e. ECP)
- kradius = kernel radius for SMD particles (distance units)
- lineflag = 1 for line segment particles, 0 for point or spherical particles
- mass = mass of particle (mass units)
- molecule-ID = integer ID of molecule the atom belongs to
- mux,muy,muz = components of dipole moment of atom (dipole units) (see general triclinic note below)
- q = charge on atom (charge units)
- rho = density (need units) for SPH particles
- sp = magnitude of magnetic spin of atom (Bohr magnetons)
- spx,spy,spz = components of magnetic spin of atom (unit vector) (see general triclinic note below)
- template-atom = which atom within a template molecule the atom is
- template-index = which molecule within the molecule template the atom is part of
- theta = internal temperature of a DPD particle
- triangleflag = 1 for triangular particles, 0 for point or spherical particles
- volume = volume of Peridynamic particle (distance^3 units)
- x,y,z = coordinates of atom (distance units)
- x0,y0,z0 = original (strain-free) coordinates of atom (distance units) (see general triclinic note below)

The units for these quantities depend on the unit style; see the *units* command for details.

For 2d simulations, the atom coordinate z must be specified as 0.0. If the data file is created by another program, then z values for a 2d simulation can be within epsilon of 0.0, and LAMMPS will force them to zero.

**Note**

If the data file defines a general triclinic box, then the following per-atom values in the list above are per-atom vectors which imply an orientation: (mux,muy,muz) and (spx,spy,spz). This means they should be specified consistent with the general triclinic box and its orientation relative to the standard x,y,z coordinate axes. For example a dipole moment vector which will be in the +x direction once LAMMPS converts from a general to restricted triclinic box, should be specified in the data file in the direction of the **A** edge vector. Likewise the (x0,y0,z0) per-atom strain-free coordinates should be inside the general triclinic simulation box as explained in the note above. See the *Howto triclinic* doc page for more details.

The atom-ID is used to identify the atom throughout the simulation and in dump files. Normally, it is a unique value from 1 to Natoms for each atom. Unique values larger than Natoms can be used, but they will cause extra memory to be allocated on each processor, if an atom map array is used, but not if an atom map hash is used; see the *atom_modify* command for details. If an atom map is not used (e.g. an atomic system with no bonds), and you don't care if unique atom IDs appear in dump files, then the atom-IDs can all be set to 0.

The atom-type can be a numeric value or an alphanumeric label. The latter is only allowed if the type label has been defined by the *labelmap* command or an Atom Type Labels section earlier in the data file. See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

The molecule ID is a second identifier attached to an atom. Normally, it is a number from 1 to N, identifying which molecule the atom belongs to. It can be 0 if it is a non-bonded atom or if you don't care to keep track of molecule assignments.

The diameter specifies the size of a finite-size spherical particle. It can be set to 0.0, which means that atom is a point particle.

The ellipsoidflag, lineflag, triangleflag, and bodyflag determine whether the particle is a finite-size ellipsoid or line or triangle or body of finite size, or whether the particle is a point particle. Additional attributes must be defined for each ellipsoid, line, triangle, or body in the corresponding *Ellipsoids*, *Lines*, *Triangles*, or *Bodies* section.

The *template-index* and *template-atom* are only defined used by *atom_style template*. In this case the *molecule* command is used to define a molecule template which contains one or more molecules (as separate files). If an atom belongs to one of those molecules, its *template-index* and *template-atom* are both set to positive integers; if not the values are both 0. The *template-index* is which molecule (1 to Nmols) the atom belongs to. The *template-atom* is which atom (1 to Natoms) within the molecule the atom is.

Some pair styles and fixes and computes that operate on finite-size particles allow for a mixture of finite-size and point particles. See the doc pages of individual commands for details.

For finite-size particles, the density is used in conjunction with the particle volume to set the mass of each particle as mass = density * volume. In this context, volume can be a 3d quantity (for spheres or ellipsoids), a 2d quantity (for triangles), or a 1d quantity (for line segments). If the volume is 0.0, meaning a point particle, then the density value is used as the mass. One exception is for the body atom style, in which case the mass of each particle (body or point particle) is specified explicitly. This is because the volume of the body is unknown.

Note that for 2d simulations of spheres, this command will treat them as spheres when converting density to mass. However, they can also be modeled as 2d discs (circles) if the *set density/disc* command is used to reset their mass after the read_data command is used. A *disc* keyword can also be used with time integration fixes, such as *fix nve/sphere* and *fix nvt/sphere* to time integrate their motion as 2d discs (not 3d spheres), by changing their moment of inertia.

For atom_style hybrid, following the 5 initial values (ID,type,x,y,z), specific values for each sub-style must be listed. The order of the sub-styles is the same as they were listed in the *atom_style* command. The specific values for each sub-style are those that are not the 5 standard ones (ID,type,x,y,z). For example, for the "charge" sub-style, a "q" value would appear. For the "full" sub-style, a "molecule-ID" and "q" would appear. These are listed in the same order they appear as listed above. Thus if

```
atom_style hybrid charge sphere
```

were used in the input script, each atom line would have these fields:

```
atom-ID atom-type x y z q diameter density
```

Note that if a non-standard value is defined by multiple sub-styles, it only appears once in the atom line. E.g. the atom line for atom_style hybrid dipole full would list "q" only once, with the dipole sub-style fields; "q" does not appear with the full sub-style fields.

```
atom-ID atom-type x y z q mux muy myz molecule-ID
```

Atom lines specify the (x,y,z) coordinates of atoms. These can be inside or outside the simulation box. When the data file is read, LAMMPS wraps coordinates outside the box back into the box for dimensions that are periodic. As discussed above, if an atom is outside the box in a non-periodic dimension, it will be lost.

LAMMPS always stores atom coordinates as values which are inside the simulation box. It also stores 3 flags which indicate which image of the simulation box (in each dimension) the atom would be in if its coordinates were unwrapped across periodic boundaries. An image flag of 0 means the atom is still inside the box when unwrapped. A value of 2 means add 2 box lengths to get the unwrapped coordinate. A value of -1 means subtract 1 box length to get the unwrapped coordinate. LAMMPS updates these flags as atoms cross periodic boundaries during the simulation. The *dump* command can output atom coordinates in wrapped or unwrapped form, as well as the 3 image flags.

In the data file, atom lines (all lines or none of them) can optionally list 3 trailing integer values (nx,ny,nz), which are used to initialize the atom's image flags. If nx,ny,nz values are not listed in the data file, LAMMPS initializes them to 0. Note that the image flags are immediately updated if an atom's coordinates need to wrapped back into the simulation box.

It is only important to set image flags correctly in a data file if a simulation model relies on unwrapped coordinates for some calculation; otherwise they can be left unspecified. Examples of LAMMPS commands that use unwrapped coordinates internally are as follows:

- Atoms in a rigid body (see *fix rigid*, *fix rigid/small*) must have consistent image flags, so that when the atoms are unwrapped, they are near each other, i.e. as a single body.
- If the *replicate* command is used to generate a larger system, image flags must be consistent for bonded atoms when the bond crosses a periodic boundary. I.e. the values of the image flags should be different by 1 (in the appropriate dimension) for the two atoms in such a bond.
- If you plan to *dump* image flags and perform post-analysis that will unwrap atom coordinates, it may be important that a continued run (restarted from a data file) begins with image flags that are consistent with the previous run.

**Note**

If your system is an infinite periodic crystal with bonds then it is impossible to have fully consistent image flags. This is because some bonds will cross periodic boundaries and connect two atoms with the same image flag.

Atom velocities and other atom quantities not defined above are set to 0.0 when the *Atoms* section is read. Velocities can be set later by a *Velocities* section in the data file or by a *velocity* or *set* command in the input script.

---

*Bodies* section:

- one or more lines per body
- first line syntax: atom-ID Ninteger Ndouble

```
Ninteger = # of integer quantities for this particle
Ndouble = # of floating-point quantities for this particle
```

- 0 or more integer lines with total of Ninteger values
- 0 or more double lines with total of Ndouble values
- example:

```
12 3 6
2 3 2
1.0 2.0 3.0 1.0 2.0 4.0
```

- example:

```
12 0 14
1.0 2.0 3.0 1.0 2.0 4.0 1.0
2.0 3.0 1.0 2.0 4.0 4.0 2.0
```

The *Bodies* section must appear if *atom_style body* is used and any atoms listed in the *Atoms* section have a bodyflag = 1. The number of bodies should be specified in the header section via the "bodies" keyword.

Each body can have a variable number of integer and/or floating-point values. The number and meaning of the values is defined by the body style, as described in the *Howto body* doc page. The body style is given as an argument to the *atom_style body* command.

The Ninteger and Ndouble values determine how many integer and floating-point values are specified for this particle. Ninteger and Ndouble can be as large as needed and can be different for every body. Integer values are then listed next on subsequent lines. Lines are read one at a time until Ninteger values are read. Floating-point values follow on subsequent lines, Again lines are read one at a time until Ndouble values are read. Note that if there are no values of a particular type, no lines appear for that type.

The *Bodies* section must appear after the *Atoms* section.

---

*Bond Coeffs* section:

- one line per bond type
- line syntax: ID coeffs

```
ID = bond type (1-N)
coeffs = list of coeffs
```

- example:

```
4 250 1.49
```

The number and meaning of the coefficients are specific to the defined bond style. See the *bond_style* and *bond_coeff* commands for details. Coefficients can also be set via the *bond_coeff* command in the input script.

---

*Bond Type Labels* section:

- one line per bond type
- line syntax: ID label

```
ID = bond type (1-N)
label = alphanumeric type label
```

Define alphanumeric type labels for each numeric bond type. These can be used in the Bonds section in place of a numeric type, but only if the this section appears before the Angles section.

See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

---

*BondAngle Coeffs* section:

- one line per angle type
- line syntax: ID coeffs

```
ID = angle type (1-N)
coeffs = list of coeffs (see class 2 section of angle_coeff)
```

---

*BondBond Coeffs* section:

- one line per angle type
- line syntax: ID coeffs

```
ID = angle type (1-N)
coeffs = list of coeffs (see class 2 section of angle_coeff)
```

---

*BondBond13 Coeffs* section:

- one line per dihedral type
- line syntax: ID coeffs

```
ID = dihedral type (1-N)
coeffs = list of coeffs (see class 2 section of dihedral_coeff)
```

---

*Bonds* section:

- one line per bond
- line syntax: ID type atom1 atom2

```
ID = bond number (1-Nbonds)
type = bond type (1-Nbondtype, or type label)
atom1,atom2 = IDs of 1st,2nd atom in bond
```

- example:

```
12 3 17 29
```

The *Bonds* section must appear after the *Atoms* section.

All values in this section must be integers (1, not 1.0). However, the type can be a numeric value or an alphanumeric label. The latter is only allowed if the type label has been defined by the *labelmap* command or a Bond Type Labels section earlier in the data file. See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

---

*Dihedral Coeffs* section:

- one line per dihedral type
- line syntax: ID coeffs

```
ID = dihedral type (1-N)
coeffs = list of coeffs
```

- example:

```
3 0.6 1 0 1
```

The number and meaning of the coefficients are specific to the defined dihedral style. See the *dihedral_style* and *dihedral_coeff* commands for details. Coefficients can also be set via the *dihedral_coeff* command in the input script.

---

*Dihedral Type Labels* section:

- one line per dihedral type
- line syntax: ID label

```
ID = dihedral type (1-N)
label = alphanumeric type label
```

Define alphanumeric type labels for each numeric dihedral type. These can be used in the Dihedrals section in place of a numeric type, but only if the this section appears before the Dihedrals section.

See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

---

*Dihedrals* section:

- one line per dihedral
- line syntax: ID type atom1 atom2 atom3 atom4

```
ID = number of dihedral (1-Ndihedrals)
type = dihedral type (1-Ndihedraltype, or type label)
atom1,atom2,atom3,atom4 = IDs of 1st,2nd,3rd,4th atom in dihedral
```

- example:

```
12 4 17 29 30 21
```

The 4 atoms are ordered linearly within the dihedral. The *Dihedrals* section must appear after the *Atoms* section.

All values in this section must be integers (1, not 1.0). However, the type can be a numeric value or an alphanumeric label. The latter is only allowed if the type label has been defined by the *labelmap* command or a Dihedral Type Labels section earlier in the data file. See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

---

*Ellipsoids* section:

- one line per ellipsoid
- line syntax: atom-ID shapex shapey shapez quatw quati quatj quatk
- line syntax (*superellipsoids*): atom-ID shapex shapey shapez quatw quati quatj quatk block1 block2

```
atom-ID = ID of atom which is an ellipsoid
shapex,shapey,shapez = 3 diameters of ellipsoid (distance units)
quatw,quati,quatj,quatk = quaternion components for orientation of atom
block1,block2 = 2 blockiness parameters (for superellipsoids only)
```

- examples:

```
12 1 2 1 1 0 0 0
12 1 2 1 1 0 0 0 2 2
```

The *Ellipsoids* section must appear if *atom_style ellipsoid* is used and any atoms are listed in the *Atoms* section with an ellipsoidflag = 1. The number of ellipsoids should be specified in the header section via the "ellipsoids" keyword.

The 3 shape values specify the 3 diameters or aspect ratios of a finite-size ellipsoidal particle, when it is oriented along the 3 coordinate axes. They must all be non-zero values.

The values *quatw*, *quati*, *quatj*, and *quatk* set the orientation of the atom as a quaternion (4-vector). Note that the shape attributes specify the aspect ratios of an ellipsoidal particle, which is oriented by default with its x-axis along the simulation box's x-axis, and similarly for y and z. If this body is rotated (via the right-hand rule) by an angle theta around a unit vector (a,b,c), then the quaternion that represents its new orientation is given by (cos(theta/2), a*sin(theta/2), b*sin(theta/2), c*sin(theta/2)). These 4 components are quatw, quati, quatj, and quatk as specified above. LAMMPS normalizes each atom's quaternion in case (a,b,c) is not specified as a unit vector.

Added in version 30Mar2026.

The blockiness values *block1*, *block2* generalize the geometry to a super ellipsoid for use in granular simulations. Sections through the center and parallel to the z-axis are superellipses with squareness *block1* and sections in the x-y plane are superellipses with squareness *block2*. These parameters are optional and default to a value of 2, recovering ellipsoid geometry. When specified, both values must be greater than or equal to 2.

If the data file defines a general triclinic box, then the quaternion for each ellipsoid should be specified for its orientation relative to the standard x,y,z coordinate axes. When the system is converted to a restricted triclinic box, the ellipsoid quaternions will be altered to reflect the new orientation of the ellipsoid.

The *Ellipsoids* section must appear after the *Atoms* section.

---

*EndBondTorsion Coeffs* section:

- one line per dihedral type
- line syntax: ID coeffs

```
ID = dihedral type (1-N)
coeffs = list of coeffs (see class 2 section of dihedral_coeff)
```

---

*Improper Coeffs* section:

- one line per improper type
- line syntax: ID coeffs

```
ID = improper type (1-N)
coeffs = list of coeffs
```

- example:

```
2 20 0.0548311
```

The number and meaning of the coefficients are specific to the defined improper style. See the *improper_style* and *improper_coeff* commands for details. Coefficients can also be set via the *improper_coeff* command in the input script.

---

*Improper Type Labels* section:

- one line per improper type
- line syntax: ID label

```
ID = improper type (1-N)
label = alphanumeric type label
```

Define alphanumeric type labels for each numeric improper type. These can be used in the Impropers section in place of a numeric type, but only if the this section appears before the Impropers section.

See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

---

*Impropers* section:

- one line per improper
- line syntax: ID type atom1 atom2 atom3 atom4

```
ID = number of improper (1-Nimpropers)
type = improper type (1-Nimpropertype, or type label)
atom1,atom2,atom3,atom4 = IDs of 1st,2nd,3rd,4th atom in improper
```

- example:

```
12 3 17 29 13 100
```

The ordering of the 4 atoms determines the definition of the improper angle used in the formula for each *improper style*. See the doc pages for individual styles for details.

The *Impropers* section must appear after the *Atoms* section.

All values in this section must be integers (1, not 1.0). However, the type can be a numeric value or an alphanumeric label. The latter is only allowed if the type label has been defined by the *labelmap* command or a Improper Type Labels section earlier in the data file. See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

---

*Lines* section:

- one line per line segment
- line syntax: atom-ID x1 y1 x2 y2

```
atom-ID = ID of atom which is a line segment
x1,y1 = 1st end point
x2,y2 = 2nd end point
```

- example:

```
12 1.0 0.0 2.0 0.0
```

The *Lines* section must appear if *atom_style line* is used and any atoms are listed in the *Atoms* section with a lineflag = 1. The number of lines should be specified in the header section via the "lines" keyword.

The 2 end points are the end points of the line segment. They should be values close to the center point of the line segment specified in the Atoms section of the data file, even if individual end points are outside the simulation box.

The ordering of the 2 points should be such that using a right-hand rule to cross the line segment with a unit vector in the +z direction, gives an "outward" normal vector perpendicular to the line segment. I.e. normal = (c2-c1) x (0,0,1). This orientation may be important for defining some interactions.

If the data file defines a general triclinic box, then the x1,y1 and x2,y2 values for each line segment should be specified for its orientation relative to the standard x,y,z coordinate axes. When the system is converted to a restricted triclinic box, the x1,y1,x2,y2 values will be altered to reflect the new orientation of the line segment.

The *Lines* section must appear after the *Atoms* section.

---

*Masses* section:

- one line per atom type
- line syntax: ID mass

```
ID = atom type (1-N or atom type label)
mass = mass value
```

- example:

```
3 1.01
```

This defines the mass of each atom type. This can also be set via the *mass* command in the input script. This section cannot be used for atom styles that define a mass for individual atoms - e.g. *atom_style sphere*.

Using type labels instead of atom type numbers is only allowed if the type label has been defined by the *labelmap* command or a Atom Type Labels section earlier in the data file. See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

---

*MiddleBondTorsion Coeffs* section:

- one line per dihedral type
- line syntax: ID coeffs

```
ID = dihedral type (1-N)
coeffs = list of coeffs (see class 2 section of dihedral_coeff)
```

---

*Pair Coeffs* section:

- one line per atom type
- line syntax: ID coeffs

```
ID = atom type (1-N)
coeffs = list of coeffs
```

- example:

```
3 0.022 2.35197 0.022 2.35197
```

The number and meaning of the coefficients are specific to the defined pair style. See the *pair_style* and *pair_coeff* commands for details. Since pair coefficients for types I != J are not specified, these will be generated automatically by the pair style's mixing rule. See the individual pair_style doc pages and the *pair_modify mix* command for details. Pair coefficients can also be set via the *pair_coeff* command in the input script.

---

*PairIJ Coeffs* section:

- one line per pair of atom types for all I,J with I <= J
- line syntax: ID1 ID2 coeffs

```
ID1 = atom type I = 1-N
ID2 = atom type J = I-N, with I <= J
coeffs = list of coeffs
```

- examples:

```
3 3 0.022 2.35197 0.022 2.35197
3 5 0.022 2.35197 0.022 2.35197
```

This section must have N*(N+1)/2 lines where N = # of atom types. The number and meaning of the coefficients are specific to the defined pair style. See the *pair_style* and *pair_coeff* commands for details. Since pair coefficients for types I != J are all specified, these values will turn off the default mixing rule defined by the pair style. See the individual pair_style doc pages and the *pair_modify mix* command for details. Pair coefficients can also be set via the *pair_coeff* command in the input script.

---

*Triangles* section:

- one line per triangle
- line syntax: atom-ID x1 y1 z1 x2 y2 z2 x3 y3 z3

```
atom-ID = ID of atom which is a line segment
x1,y1,z1 = 1st corner point
x2,y2,z2 = 2nd corner point
x3,y3,z3 = 3rd corner point
```

- example:

```
12 0.0 0.0 0.0 2.0 0.0 1.0 0.0 2.0 1.0
```

The *Triangles* section must appear if *atom_style tri* is used and any atoms are listed in the *Atoms* section with a triangleflag = 1. The number of lines should be specified in the header section via the "triangles" keyword.

The 3 corner points are the corner points of the triangle. They should be values close to the center point of the triangle specified in the Atoms section of the data file, even if individual corner points are outside the simulation box.

The ordering of the 3 points should be such that using a right-hand rule to go from point1 to point2 to point3 gives an "outward" normal vector to the face of the triangle. I.e. normal = (c2-c1) x (c3-c1). This orientation may be important for defining some interactions.

If the data file defines a general triclinic box, then the x1,y1,z1 and x2,y2,z2 and x3,y3,z3 values for each triangle should be specified for its orientation relative to the standard x,y,z coordinate axes. When the system is converted

to a restricted triclinic box, the x1,y1,z1,x2,y2,z2,x3,y3,z3 values will be altered to reflect the new orientation of the triangle.

The *Triangles* section must appear after the *Atoms* section.

---

*Velocities* section:

- one line per atom
- line syntax: depends on atom style

| | |
|---|---|
| all styles except those listed | atom-ID vx vy vz |
| electron | atom-ID vx vy vz ervel |
| ellipsoid | atom-ID vx vy vz lx ly lz |
| sphere | atom-ID vx vy vz wx wy wz |
| hybrid | atom-ID vx vy vz sub-style1 sub-style2 ... |

where the keywords have these meanings:

```
vx,vy,vz = translational velocity of atom
lx,ly,lz = angular momentum of aspherical atom
wx,wy,wz = angular velocity of spherical atom
ervel = electron radial velocity (0 for fixed-core)
```

The velocity lines can appear in any order. This section can only be used after an *Atoms* section. This is because the *Atoms* section must have assigned a unique atom ID to each atom so that velocities can be assigned to them.

Vx, vy, vz, and ervel are in *units* of velocity. Lx, ly, lz are in units of angular momentum (distance-velocity-mass). Wx, Wy, Wz are in units of angular velocity (radians/time).

If the data file defines a general triclinic box, then each of the 3 vectors (translational velocity, angular momentum, angular velocity) should be specified for the rotated coordinate axes of the general triclinic box. See the *Howto triclinic* doc page for more details.

For atom_style hybrid, following the 4 initial values (ID,vx,vy,vz), specific values for each sub-style must be listed. The order of the sub-styles is the same as they were listed in the *atom_style* command. The sub-style specific values are those that are not the 5 standard ones (ID,vx,vy,vz). For example, for the "sphere" sub-style, "wx", "wy", "wz" values would appear. These are listed in the same order they appear as listed above. Thus if

```
atom_style hybrid electron sphere
```

were used in the input script, each velocity line would have these fields:

```
atom-ID vx vy vz ervel wx wy wz
```

Translational velocities can also be (re)set by the *velocity* command in the input script.

---

### 1.85.10 Restrictions

To read compressed data files, you must compile LAMMPS with the `-DLAMMPS_GZIP` option. See the *Build settings* doc page for details.

Label maps are currently not supported when using the KOKKOS package.

### 1.85.11 Related commands

*read_dump*, *read_restart*, *create_atoms*, *write_data*, *labelmap*

### 1.85.12 Default

The default for all the *extra* keywords is 0.

## 1.86 read_dump command

### 1.86.1 Syntax

```
read_dump file Nstep field1 field2 ... keyword values ...
```

- file = name of dump file to read
- Nstep = snapshot timestep to read from file
- one or more fields may be appended

```
field = x or y or z or vx or vy or vz or q or mol or ix or iy or iz or fx or fy or␣
↪fz or apip_lambda
  x,y,z = atom coordinates
  vx,vy,vz = velocity components
  q = charge
  mol = molecule ID
  ix,iy,iz = image flags in each dimension
  fx,fy,fz = force components
  apip_lambda = switching parameter of an adaptive-precision interatomic potential
```

- zero or more keyword/value pairs may be appended
- keyword = *nfile* or *box* or *timestep* or *replace* or *purge* or *trim* or *add* or *label* or *scaled* or *wrapped* or *format*

```
nfile value = Nfiles = how many parallel dump files exist
box value = yes or no = replace simulation box with dump box
timestep value = yes or no = reset simulation timestep with dump timestep
replace value = yes or no = overwrite atoms with dump atoms
purge value = yes or no = delete all atoms before adding dump atoms
trim value = yes or no = trim atoms not in dump snapshot
add value = yes or keep or no = add new dump atoms to system
label value = field column
  field = one of the listed fields or id or type
  column = label on corresponding column in dump file
scaled value = yes or no = coords in dump file are scaled/unscaled
wrapped value = yes or no = coords in dump file are wrapped/unwrapped
format values = format of dump file, must be last keyword if used
  native = native LAMMPS dump file
  xyz = XYZ file
  adios [timeout value] = dump file written by the dump adios command
    timeout = specify waiting time for the arrival of the timestep when running␣
↪concurrently.
              The value is a float number and is interpreted in seconds.
  molfile style path = VMD molfile plugin interface
    style = dcd or xyz or others supported by molfile plugins
    path = optional path for location of molfile plugins
```

### 1.86.2 Examples

```
read_dump dump.file 5000 x y z
read_dump dump.xyz 5 x y z box no format xyz
read_dump dump.xyz 10 x y z box no format molfile xyz "../plugins"
read_dump dump.dcd 0 x y z box yes format molfile dcd
read_dump dump.file 1000 x y z vx vy vz box yes format molfile lammpstrj /usr/local/lib/
↪vmd/plugins/LINUXAMD64/plugins/molfile
read_dump dump.file 5000 x y vx vy trim yes
read_dump dump.file 5000 x y vx vy add yes box no timestep no
read_dump ../run7/dump.file.gz 10000 x y z box yes
read_dump dump.xyz 10 x y z box no format molfile xyz ../plugins
read_dump dump.dcd 0 x y z format molfile dcd
read_dump dump.file 1000 x y z vx vy vz format molfile lammpstrj /usr/local/lib/vmd/
↪plugins/LINUXAMD64/plugins/molfile
read_dump dump.bp 5000 x y z vx vy vz format adios
read_dump dump.bp 5000 x y z vx vy vz format adios timeout 60.0
```

### 1.86.3 Description

Read atom information from a dump file to overwrite the current atom coordinates, and optionally the atom velocities and image flags, the simulation timestep, and the simulation box dimensions. This is useful for restarting a run from a particular snapshot in a dump file. See the *read_restart* and *read_data* commands for alternative methods to do this. Also see the *rerun* command for a means of reading multiple snapshots from a dump file.

Note that a simulation box must already be defined before using the read_dump command. This can be done by the *create_box*, *read_data*, or *read_restart* commands. The read_dump command can reset the simulation box dimensions, as explained below.

Also note that reading per-atom information from a dump snapshot is limited to the atom coordinates, velocities and image flags, as explained below. Other atom properties, which may be necessary to run a valid simulation, such as atom charge, or bond topology information for a molecular system, are not read from (or may not even be contained in) dump files. Thus this auxiliary information should be defined in the usual way, e.g. in a data file read in by a *read_data* command, before using the read_dump command, or by the *set* command, after the dump snapshot is read.

---

If the dump filename specified as *file* ends with ".gz", the dump file is read in gzipped format.

You can read dump files that were written (in parallel) to multiple files via the "%" wild-card character in the dump file name. If any specified dump file name contains a "%", they must all contain it. See the *dump* command for details. The "%" wild-card character is only supported by the *native* format for dump files, described next.

If reading parallel dump files, you must also use the *nfile* keyword to tell LAMMPS how many parallel files exist, via its specified *Nfiles* value.

The format of the dump file is selected through the *format* keyword. If specified, it must be the last keyword used, since all remaining arguments are passed on to the dump reader. The *native* format is for native LAMMPS dump files, written with a *dump atom* or *dump custom* command. The *xyz* format is for generic XYZ formatted dump files (see details below). These formats take no additional values.

The *molfile* format supports reading data through using the VMD molfile plugin interface. This dump reader format is only available, if the MOLFILE package has been installed when compiling LAMMPS.

The *molfile* format takes one or two additional values. The *style* value determines the file format to be used and can be any format that the molfile plugins support, such as DCD or XYZ. Note that DCD dump files can be written by LAMMPS via the *dump dcd* command. The *path* value specifies a list of directories which LAMMPS will search for the molfile plugins appropriate to the specified *style*. The syntax of the *path* value is like other search paths: it can

contain multiple directories separated by a colon (or semicolon on windows). The *path* keyword is optional and defaults to ".", i.e. the current directory.

The *adios* format supports reading data that was written by the *dump adios* command. The entire dump is read in parallel across all the processes, dividing the atoms evenly among the processes. The number of writers that has written the dump file does not matter. Using the adios style for dump and read_dump is a convenient way to dump all atoms from *N* writers and read it back by *M* readers. If one is running two LAMMPS instances concurrently where one dumps data and the other is reading it with the rerun command, the timeout option can be specified to wait on the reader side for the arrival of the requested step.

Support for other dump format readers may be added in the future.

---

Global information is first read from the dump file, namely timestep and box information.

The dump file is scanned for a snapshot with a timestamp that matches the specified *Nstep*. This means the LAMMPS timestep the dump file snapshot was written on for the *native* or *adios* formats.

The list of timestamps available in an adios .bp file is stored in the variable *ntimestep*:

```
console

$ bpls dump.bp -d ntimestep
  uint64_t  ntimestep  5*scalar
    (0)    0 50 100 150 200
```

Note that the *xyz* and *molfile* formats do not store the timestep. For these formats, timesteps are numbered logically, in a sequential manner, starting from 0. Thus to access the 10th snapshot in an *xyz* or *mofile* formatted dump file, use *Nstep* = 9.

The dimensions of the simulation box for the selected snapshot are also read; see the *box* keyword discussion below. For the *native* format, an error is generated if the snapshot is for a triclinic box and the current simulation box is orthogonal or vice versa. A warning will be generated if the snapshot box boundary conditions (periodic, shrink-wrapped, etc) do not match the current simulation boundary conditions, but the boundary condition information in the snapshot is otherwise ignored. See the "boundary" command for more details. The *adios* reader does the same as the *native* format reader.

For the *xyz* format, no information about the box is available, so you must set the *box* flag to *no*. See details below.

For the *molfile* format, reading simulation box information is typically supported, but the location of the simulation box origin is lost and no explicit information about periodicity or orthogonal/triclinic box shape is available. The MOLFILE package makes a best effort to guess based on heuristics, but this may not always work perfectly.

---

Per-atom information from the dump file snapshot is then read from the dump file snapshot. This corresponds to the specified *fields* listed in the read_dump command. It is an error to specify a z-dimension field, namely *z*, *vz*, or *iz*, for a 2d simulation.

For dump files in *native* format, each column of per-atom data has a text label listed in the file. A matching label for each field must appear, e.g. the label "vy" for the field *vy*. For the *x*, *y*, *z* fields any of the following labels are considered a match:

```
x, xs, xu, xsu for field x
y, ys, yu, ysu for field y
z, zs, zu, zsu for field z
```

The meaning of xs (scaled), xu (unwrapped), and xsu (scaled and unwrapped) is explained on the *dump* command doc page. These labels are searched for in the list of column labels in the dump file, in order, until a match is found.

The dump file must also contain atom IDs, with a column label of "id".

If the *add* keyword is specified with a value of *yes* or *keep*, as discussed below, the dump file must contain atom types, with a column label of "type".

If a column label you want to read from the dump file is not a match to a specified field, the *label* keyword can be used to specify the specific column label from the dump file to associate with that field. An example is if a time-averaged coordinate is written to the dump file via the *fix ave/atom* command. The column will then have a label corresponding to the fix-ID rather than "x" or "xs". The *label* keyword can also be used to specify new column labels for fields *id* and *type*.

For dump files in *xyz* format, only the *type*, *x*, *y*, and *z* fields are supported. There are many variants of the XYZ file format. LAMMPS will read the number of atoms from the first line of each frame, ignore the second (title) line, and then read one line for each atom in the format:

```
<label> <x coordinate>  <y coordinate> <z coordinate>
```

If the atom label is a numeric integer (like with XYZ files created by created with default settings by *dump style xyz*), that number will be used as the atom type. If the atom label is a string, then a type map must be created using the *labelmap command*. This map needs to associate each (numeric) atom type with a string label. The numeric atom type is stored internally.

The xyz format dump file does not store atom IDs, so these are assigned consecutively to the atoms as they appear in the dump file, starting from 1. Thus you should ensure that the order of atoms is consistent from snapshot to snapshot in the XYZ dump file. See the *dump_modify sort* command if the XYZ dump file was written by LAMMPS.

For dump files in *molfile* format, the *x*, *y*, *z*, *vx*, *vy*, and *vz* fields can be specified. However, not all molfile formats store velocities, or their respective plugins may not support reading of velocities. The molfile dump files do not store atom IDs, so these are assigned consecutively to the atoms as they appear in the dump file, starting from 1. Thus you should ensure that the order of atoms are consistent from snapshot to snapshot in the molfile dump file. See the *dump_modify sort* command if the dump file was written by LAMMPS.

The *adios* format supports all fields that the *native* format supports except for the *q* charge field. The list of fields stored in an adios .bp file is recorded in the attributes *columns* (array of short strings) and *columnstr* (space-separated single string).

```
console

$ bpls -la dump.bp column*
  string    columns            attr   = {"id", "type", "x", "y", "z", "vx", "vy", "vz"}
  string    columnstr          attr   = "id type x y z vx vy vz "
```

---

Information from the dump file snapshot is used to overwrite or replace properties of the current system. There are various options for how this is done, determined by the specified fields and optional keywords.

Changed in version 3Aug2022.

The timestep of the snapshot becomes the current timestep for the simulation unless the *timestep* keyword is specified with a *no* value (default setting is *yes*). See the *reset_timestep* command if you wish to change this to a different value after the dump snapshot is read.

If the *box* keyword is specified with a *yes* value, then the current simulation box dimensions are replaced by the dump snapshot box dimensions. If the *box* keyword is specified with a *no* value, the current simulation box is unchanged.

If the *purge* keyword is specified with a *yes* value, then all current atoms in the system are deleted before any of the operations invoked by the *replace*, *trim*, or *add* keywords take place.

If the *replace* keyword is specified with a *yes* value, then atoms with IDs that are in both the current system and the dump snapshot have their properties overwritten by field values. If the *replace* keyword is specified with a *no* value, atoms with IDs that are in both the current system and the dump snapshot are not modified.

If the *trim* keyword is specified with a *yes* value, then atoms with IDs that are in the current system but not in the dump snapshot are deleted. These atoms are unaffected if the *trim* keyword is specified with a *no* value.

If the *add* keyword is specified with a *no* value (default), then dump file atoms with IDs that are not in the current system are not added to the system. They are simply ignored.

If a *yes* value is specified, the atoms with new IDs are added to the system but their atom IDs are not preserved. Instead, after all the atoms are added, new IDs are assigned to them in the same manner as is described for the *create_atoms* command. Basically the largest existing atom ID in the system is identified, and all the added atoms are assigned IDs that consecutively follow the largest ID.

If a *keep* value is specified, the atoms with new IDs are added to the system and their atom IDs are preserved. This may lead to non-contiguous IDs for the combined system.

Note that atoms added via the *add* keyword will only have the attributes read from the dump file due to the *field* arguments. For example, if *x* or *y* or *z* or *q* is not specified as a field, a value of 0.0 is used for added atoms. Added atoms must have an atom type, so this value must appear in the dump file.

Any other attributes (e.g. charge or particle diameter for spherical particles) will be set to default values, the same as if the *create_atoms* command were used.

---

Atom coordinates read from the dump file are first converted into unscaled coordinates, relative to the box dimensions of the snapshot. These coordinates are then be assigned to an existing or new atom in the current simulation. The coordinates will then be remapped to the simulation box, whether it is the original box or the dump snapshot box. If periodic boundary conditions apply, this means the atom will be remapped back into the simulation box if necessary. If shrink-wrap boundary conditions apply, the new coordinates may change the simulation box dimensions. If fixed boundary conditions apply, the atom will be lost if it is outside the simulation box.

For *native* format dump files, the 3 xyz image flags for an atom in the dump file are set to the corresponding values appearing in the dump file if the *ix*, *iy*, *iz* fields are specified. If not specified, the image flags for replaced atoms are not changed and image flags for new atoms are set to default values. If coordinates read from the dump file are in unwrapped format (e.g. *xu*) then the image flags for read-in atoms are also set to default values. The remapping procedure described in the previous paragraph will then change images flags for all atoms (old and new) if periodic boundary conditions are applied to remap an atom back into the simulation box.

**Note**

If you get a warning about inconsistent image flags after reading in a dump snapshot, it means one or more pairs of bonded atoms now have inconsistent image flags. As discussed on the *Errors common* page this may or may not cause problems for subsequent simulations. One way this can happen is if you read image flag fields from the dump file but do not also use the dump file box parameters.

LAMMPS knows how to compute unscaled and remapped coordinates for the snapshot column labels discussed above, e.g. *x*, *xs*, *xu*, *xsu*. If another column label is assigned to the *x* or *y* or *z* field via the *label* keyword, e.g. for coordinates output by the *fix ave/atom* command, then LAMMPS needs to know whether the coordinate information in the dump file is scaled and/or wrapped. This can be set via the *scaled* and *wrapped* keywords. Note that the value of the *scaled* and *wrapped* keywords is ignored for fields *x* or *y* or *z* if the *label* keyword is not used to assign a column label to that field.

The scaled/unscaled and wrapped/unwrapped setting must be identical for any of the *x*, *y*, *z* fields that are specified. Thus you cannot read *xs* and *yu* from the dump file. Also, if the dump file coordinates are scaled and the simulation

box is triclinic, then all 3 of the *x*, *y*, *z* fields must be specified, since they are all needed to generate absolute, unscaled coordinates.

---

### 1.86.4 Restrictions

To read gzipped dump files, you must compile LAMMPS with the -DLAMMPS_GZIP option. See the *Build settings* doc page for details.

The *molfile* dump file formats are part of the MOLFILE package. They are only enabled if LAMMPS was built with that packages. See the *Build package* page for more info.

To write and read adios .bp files, you must compile LAMMPS with the *ADIOS* package.

### 1.86.5 Related commands

*dump*, *dump molfile*, *dump adios*, *read_data*, *read_restart*, *rerun*

### 1.86.6 Default

The option defaults are box = yes, timestep = yes, replace = yes, purge = no, trim = no, add = no, scaled = no, wrapped = yes, and format = native.

## 1.87 read_restart command

### 1.87.1 Syntax

```
read_restart file
```

- file = name of binary restart file to read in

### 1.87.2 Examples

```
read_restart save.10000
read_restart restart.*
```

### 1.87.3 Description

Read in a previously saved system configuration from a restart file. This allows continuation of a previous run. Details about what information is stored (and not stored) in a restart file is given below. Basically this operation will re-create the simulation box with all its atoms and their attributes as well as some related global settings, at the point in time it was written to the restart file by a previous simulation. The simulation box will be partitioned into a regular 3d grid of rectangular bricks, one per processor, based on the number of processors in the current simulation and the settings of the *processors* command. The partitioning can later be changed by the *balance* or *fix balance* commands.

Deprecated since version 23Jun2022.

Atom coordinates that are found to be outside the simulation box when reading the restart will be remapped back into the box and their image flags updated accordingly. This previously required specifying the *remap* option, but that is no longer required.

Restart files are saved in binary format to enable exact restarts, meaning that the trajectories of a restarted run will precisely match those produced by the original run had it continued on.

Some information about a restart file can be gathered directly from the command-line when using LAMMPS with the *-restart2info* command-line flag. On Unix-like operating systems (like Linux or macOS), one can also *configure the "file" command-line program* to display basic information about a restart file

The binary restart file format was not designed with backward, forward, or cross-platform compatibility in mind, so the files are only expected to be read correctly by the same LAMMPS executable on the same platform. Changes to the architecture, compilation settings, or LAMMPS version can render a restart file unreadable or it may read the data incorrectly. If you want a more portable format, you can use the data file format as created by the *write_data* command. Binary restart files can also be converted into a data file from the command-line by the LAMMPS executable that wrote them using the *-restart2data* command-line flag.

Several things can prevent exact restarts due to round-off effects, in which case the trajectories in the 2 runs will slowly diverge. These include running on a different number of processors or changing certain settings such as those set by the *newton* or *processors* commands. LAMMPS will issue a warning in these cases.

Certain fixes will not restart exactly, though they should provide statistically similar results. These include *fix shake* and *fix langevin*.

Certain pair styles will not restart exactly, though they should provide statistically similar results. This is because the forces they compute depend on atom velocities, which are used at half-step values every timestep when forces are computed. When a run restarts, forces are initially evaluated with a full-step velocity, which is different than if the run had continued. These pair styles include *granular pair styles*, *pair dpd*, and *pair lubricate*.

If a restarted run is immediately different than the run which produced the restart file, it could be a LAMMPS bug, so consider *reporting it* if you think the behavior is a bug.

Because restart files are binary, they may not be portable to other machines. In this case, you can use the *-restart command-line switch* to convert a restart file to a data file.

Similar to how restart files are written (see the *write_restart* and *restart* commands), the restart filename can contain two wild-card characters. If a "*" appears in the filename, the directory is searched for all filenames that match the pattern where "*" is replaced with a timestep value. The file with the largest timestep value is read in. Thus, this effectively means, read the latest restart file. It's useful if you want your script to continue a run from where it left off. See the *run* command and its "upto" option for how to specify the run command so it does not need to be changed either.

If a "%" character appears in the restart filename, LAMMPS expects a set of multiple files to exist. The *restart* and *write_restart* commands explain how such sets are created. Read_restart will first read a filename where "%" is replaced by "base". This file tells LAMMPS how many processors created the set and how many files are in it. Read_restart then reads the additional files. For example, if the restart file was specified as save.% when it was written, then read_restart reads the files save.base, save.0, save.1, ... save.P-1, where P is the number of processors that created the restart file.

Note that P could be the total number of processors in the previous simulation, or some subset of those processors, if the *fileper* or *nfile* options were used when the restart file was written; see the *restart* and *write_restart* commands for details. The processors in the current LAMMPS simulation share the work of reading these files; each reads a roughly equal subset of the files. The number of processors which created the set can be different the number of processors in the current LAMMPS simulation. This can be a fast mode of input on parallel machines that support parallel I/O.

---

Here is the list of information included in a restart file, which means these quantities do not need to be re-specified in the input script that reads the restart file, though you can redefine many of these settings after the restart file is read.

- *units*
- *newton bond* (see discussion of newton command below)
- *atom style* and *atom_modify* settings id, map, sort
- *comm style* and *comm_modify* settings mode, cutoff, vel
- *timestep size* and *timestep number*

- simulation box size and shape and *boundary* settings
- atom *group* definitions
- per-type atom settings such as *mass*
- per-atom attributes including their group assignments and molecular topology attributes (bonds, angles, etc)
- force field styles (*pair*, *bond*, *angle*, etc)
- force field coefficients (*pair*, *bond*, *angle*, etc) in some cases (see below)
- *pair_modify* settings, except the compute option
- *special_bonds* settings

Here is a list of information not stored in a restart file, which means you must re-issue these commands in your input script, after reading the restart file.

- *newton pair* (see discussion of newton command below)
- *fix* commands (see below)
- *compute* commands (see below)
- *variable* commands
- *region* commands
- *neighbor list* criteria including *neigh_modify* settings
- *kspace_style* and *kspace_modify* settings
- info for *thermodynamic*, *dump*, or *restart* output

The *newton* command has two settings, one for pairwise interactions, the other for bonded. Both settings are stored in the restart file. For the bond setting, the value in the file will overwrite the current value (at the time the read_restart command is issued) and warn if the two values are not the same and the current value is not the default. For the pair setting, the value in the file will not overwrite the current value (so that you can override the previous run's value), but a warning is issued if the two values are not the same and the current value is not the default.

Note that some force field styles (pair, bond, angle, etc) do not store their coefficient info in restart files. Typically these are many-body or tabulated potentials which read their parameters from separate files. In these cases you will need to re-specify the *pair_coeff*, *bond_coeff*, etc commands in your restart input script. The doc pages for individual force field styles mention if this is the case. This is also true of *pair_style hybrid* (bond hybrid, angle hybrid, etc) commands; they do not store coefficient info.

As indicated in the above list, the *fixes* used for a simulation are not stored in the restart file. This means the new input script should specify all fixes it will use. However, note that some fixes store an internal "state" which is written to the restart file. This allows the fix to continue on with its calculations in a restarted simulation. To re-enable such a fix, the fix command in the new input script must be of the same style and use the same fix-ID as was used in the input script that wrote the restart file.

If a match is found, LAMMPS prints a message indicating that the fix is being re-enabled. If no match is found before the first run or minimization is performed by the new script, the "state" information for the saved fix is discarded. At the time the discard occurs, LAMMPS will also print a list of fixes for which the information is being discarded. See the doc pages for individual fixes for info on which ones can be restarted in this manner. Note that fixes which are created internally by other LAMMPS commands (computes, fixes, etc) will have style names which are all-capitalized, and IDs which are generated internally.

Likewise, the *computes* used for a simulation are not stored in the restart file. This means the new input script should specify all computes it will use. However, some computes create a fix internally to store "state" information that persists from timestep to timestep. An example is the *compute msd* command which uses a fix to store a reference coordinate for each atom, so that a displacement can be calculated at any later time. If the compute command in the new input script

uses the same compute-ID and group-ID as was used in the input script that wrote the restart file, then it will create the same fix in the restarted run. This means the re-created fix will be re-enabled with the stored state information as described in the previous paragraph, so that the compute can continue its calculations in a consistent manner.

**Note**

There are a handful of commands which can be used before or between runs which may require a system initialization. Examples include the "balance", "displace_atoms", "delete_atoms", "set" (some options), and "velocity" (some options) commands. This is because they can migrate atoms to new processors. Thus they will also discard unused "state" information from fixes. You will know the discard has occurred because a list of discarded fixes will be printed to the screen and log file, as explained above. This means that if you wish to retain that info in a restarted run, you must re-specify the relevant fixes and computes (which create fixes) before those commands are used.

Some pair styles, like the *granular pair styles*, also use a fix to store "state" information that persists from timestep to timestep. In the case of granular potentials, it is contact information between pairs of touching particles. This info will also be re-enabled in the restart script, assuming you re-use the same granular pair style.

LAMMPS allows bond interactions (angle, etc) to be turned off or deleted in various ways, which can affect how their info is stored in a restart file.

If bonds (angles, etc) have been turned off by the *fix shake* or *delete_bonds* command, their info will be written to a restart file as if they are turned on. This means they will need to be turned off again in a new run after the restart file is read.

Bonds that are broken (e.g. by a bond-breaking potential) are written to the restart file as broken bonds with a type of 0. Thus these bonds will still be broken when the restart file is read.

Bonds that have been broken by the *fix bond/break* command have disappeared from the system. No information about these bonds is written to the restart file.

---

### 1.87.4 Restrictions

none

### 1.87.5 Related commands

*read_data*, *read_dump*, *write_restart*, *restart*

### 1.87.6 Default

none

## 1.88 region command

Accelerator Variants: *block/kk*, *sphere/kk*

### 1.88.1 Syntax

```
region ID style args keyword arg ...
```

- ID = user-assigned name for the region
- style = *delete* or *block* or *cone* or *cylinder* or *ellipsoid* or *plane* or *prism* or *sphere* or *union* or *intersect*

```
delete = no args
block args = xlo xhi ylo yhi zlo zhi
  xlo,xhi,ylo,yhi,zlo,zhi = bounds of block in all dimensions (distance units)
    xlo,xhi,ylo,yhi,zlo,zhi can be a variable (see below)
cone args = dim c1 c2 radlo radhi lo hi
  dim = x or y or z = axis of cone
  c1,c2 = coords of cone axis in other 2 dimensions (distance units)
  radlo,radhi = cone radii at lo and hi end (distance units)
  lo,hi = bounds of cone in dim (distance units)
    c1,c2,radlo,radhi,lo,hi can be a variable (see below)
cylinder args = dim c1 c2 radius lo hi
  dim = x or y or z = axis of cylinder
  c1,c2 = coords of cylinder axis in other 2 dimensions (distance units)
  radius = cylinder radius (distance units)
    c1,c2, and radius can be a variable (see below)
  lo,hi = bounds of cylinder in dim (distance units)
ellipsoid args = x y z a b c
  x,y,z = center of ellipsoid (distance units)
  a,b,c = half the length of the principal axes of the ellipsoid (distance units)
    x,y,z,a,b and c can be a variable (see below)
plane args = px py pz nx ny nz
  px,py,pz = point on the plane (distance units)
  nx,ny,nz = direction normal to plane (distance units)
    px,py,pz,nx,ny,nz can be a variable (see below)
prism args = xlo xhi ylo yhi zlo zhi xy xz yz
  xlo,xhi,ylo,yhi,zlo,zhi = bounds of untilted prism (distance units)
  xy = distance to tilt y in x direction (distance units)
  xz = distance to tilt z in x direction (distance units)
  yz = distance to tilt z in y direction (distance units)
    xlo,xhi,ylo,yhi,zlo,zhi,xy,xz,yz can be a variable (see below)
sphere args = x y z radius
  x,y,z = center of sphere (distance units)
  radius = radius of sphere (distance units)
    x,y,z, and radius can be a variable (see below)
union args = N reg-ID1 reg-ID2 ...
  N = # of regions to follow, must be 2 or greater
  reg-ID1,reg-ID2, ... = IDs of regions to join together
intersect args = N reg-ID1 reg-ID2 ...
  N = # of regions to follow, must be 2 or greater
  reg-ID1,reg-ID2, ... = IDs of regions to intersect
```

- zero or more keyword/arg pairs may be appended
- keyword = *side* or *units* or *move* or *rotate* or *open*

```
side value = in or out
  in = the region is inside the specified geometry
  out = the region is outside the specified geometry
units value = lattice or box
  lattice = the geometry is defined in lattice units
  box = the geometry is defined in simulation box units
move args = v_x v_y v_z
  v_x,v_y,v_z = equal-style variables for x,y,z displacement of region over time␣
↪(distance units)
rotate args = v_theta Px Py Pz Rx Ry Rz
```

```
    v_theta = equal-style variable for rotaton of region over time (in radians)
    Px,Py,Pz = origin for axis of rotation (distance units)
    Rx,Ry,Rz = axis of rotation vector
  open value = integer from 1-6 corresponding to face index (see below)
```

- accelerated styles (with same args) = *block/kk, sphere/kk*

## 1.88.2 Examples

```
region 1 block -3.0 5.0 INF 10.0 INF INF
region 2 sphere 0.0 0.0 0.0 5 side out
region void cylinder y 2 3 5 -5.0 EDGE units box
region 1 prism 0 10 0 10 0 10 2 0 0
region outside union 4 side1 side2 side3 side4
region 2 sphere 0.0 0.0 0.0 5 side out move v_left v_up NULL
region openbox block 0 10 0 10 0 10 open 5 open 6 units box
region funnel cone z 10 10 2 5 0 10 open 1 units box
```

## 1.88.3 Description

This command defines a geometric region of space. Various other commands use regions. For example, the region can be filled with atoms via the *create_atoms* command. Or a bounding box around the region, can be used to define the simulation box via the *create_box* command. Or the atoms in the region can be identified as a group via the *group* command, or deleted via the *delete_atoms* command. Or the surface of the region can be used as a boundary wall via the *fix wall/region* command.

Commands which use regions typically test whether an atom's position is contained in the region or not. For this purpose, coordinates exactly on the region boundary are considered to be interior to the region. This means, for example, for a spherical region, an atom on the sphere surface would be part of the region if the sphere were defined with the *side in* keyword, but would not be part of the region if it were defined using the *side out* keyword. See more details on the *side* keyword below.

Normally, regions in LAMMPS are "static", meaning their geometric extent does not change with time unless their geometry is defined using references to *equal style variables* where the value changes over time. See below for more details on "variable shapes". If the *move* or *rotate* keyword is used, as described below, the region becomes "dynamic", meaning its location or orientation can change with time. This may be useful, for example, when thermostatting a region, via the compute temp/region command, or when the fix wall/region command uses a region surface as a bounding wall on particle motion, i.e. a rotating container.

The *delete* style removes the named region. Since there is little overhead to defining extra regions, there is normally no need to do this, unless you are defining and discarding large numbers of regions in your input script.

The lo/hi values for *block* or *cone* or *cylinder* or *prism* styles can be specified as EDGE or INF. EDGE means they extend all the way to the global simulation box boundary. Note that this is the current box boundary; if the box changes size during a simulation, the region does not. INF means a large negative or positive number (1.0e20), so it should encompass the simulation box even if it changes size. If a region is defined before the simulation box has been created (via *create_box* or *read_data* or *read_restart* commands), then an EDGE or INF parameter cannot be used. For a *prism* region, a non-zero tilt factor in any pair of dimensions cannot be used if both the lo/hi values in either of those dimensions are INF. E.g. if the xy tilt is non-zero, then xlo and xhi cannot both be INF, nor can ylo and yhi.

**Note**

Regions in LAMMPS do not get wrapped across periodic boundaries, as specified by the *boundary* command. For example, a spherical region that is defined so that it overlaps a periodic boundary is not treated as 2 half-spheres, one on either side of the simulation box.

**Note**

Regions in LAMMPS are always 3d geometric objects, regardless of whether the *dimension* of a simulation is 2d or 3d. Thus when using regions in a 2d simulation, you should be careful to define the region so that its intersection with the 2d x-y plane of the simulation has the 2d geometric extent you want.

For style *cone*, an axis-aligned cone is defined which is like a *cylinder* except that two different radii (one at each end) can be defined. Either of the radii (but not both) can be 0.0.

For style *cone* and *cylinder*, the c1,c2 params are coordinates in the 2 other dimensions besides the cylinder axis dimension. For dim = x, c1/c2 = y/z; for dim = y, c1/c2 = x/z; for dim = z, c1/c2 = x/y. Thus the third example above specifies a cylinder with its axis in the y-direction located at x = 2.0 and z = 3.0, with a radius of 5.0, and extending in the y-direction from -5.0 to the upper box boundary.

Added in version 4May2022.

For style *ellipsoid*, an axis-aligned ellipsoid is defined. The ellipsoid has its center at (x,y,z) and is defined by 3 axis-aligned vectors given by A = (a,0,0); B = (0,b,0); C = (0,0,c). Note that although the ellipsoid is specified as axis-aligned it can be rotated via the optional *rotate* keyword.

For style *plane*, a plane is defined which contain the point (px,py,pz) and has a normal vector (nx,ny,nz). The normal vector does not have to be of unit length. The "inside" of the plane is the half-space in the direction of the normal vector; see the discussion of the *side* option below.

For style *prism*, a parallelepiped is defined (it is too hard to spell parallelepiped in an input script!). The parallelepiped has its "origin" at (xlo,ylo,zlo) and is defined by 3 edge vectors starting from the origin given by A = (xhi-xlo,0,0); B = (xy,yhi-ylo,0); C = (xz,yz,zhi-zlo). *Xy,xz,yz* can be 0.0 or positive or negative values and are called "tilt factors" because they are the amount of displacement applied to faces of an originally orthogonal box to transform it into the parallelepiped.

A prism region that will be used with the *create_box* command to define a triclinic simulation box must have tilt factors (xy,xz,yz) that do not skew the box more than half the distance of corresponding the parallel box length. For example, if xlo = 2 and xhi = 12, then the x box length is 10 and the xy tilt factor must be between -5 and 5. Similarly, both xz and yz must be between -(xhi-xlo)/2 and +(yhi-ylo)/2. Note that this is not a limitation, since if the maximum tilt factor is 5 (as in this example), then configurations with tilt = . . . , -15, -5, 5, 15, 25, . . . are all geometrically equivalent.

For style *sphere*, a sphere is defined with its center at (x,y,z) and with radius as its radius.

The *radius* value for styles *sphere* and *cylinder*, and the parameters a,b,c for style *ellipsoid*, can each be specified as an equal-style *variable*. Likewise, for style *sphere* and *ellipsoid* the x-, y-, and z- coordinates of the center of the sphere/ellipsoid can be specified as an equal-style variable. And for style *cylinder* the two center positions c1 and c2 for the location of the cylinder axes can be specified as a equal-style variable. For styles *block*, *cone*, *prism*, and *plane* all properties can be defined via equal-style variables. For style *plane*, the components of the direction vector normal to plane should be either all constants or all defined by equal-style variables.

If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated each timestep, and its value used to determine the radius of the region.

Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent radius or have a time dependent position of the sphere or cylinder region.

**Note**

Whenever a region property, such as a coordinate or an upper/lower bound, is defined via an equal-style variable, the variable should not cause any of the region boundaries to move too far within a single timestep. Otherwise, bad dynamics will occur. "Too far" means a small fraction of the approximate distance of closest approach between two

particles, which for the case of Lennard-Jones particles is the distance of the energy minimum while for granular particles it is their diameter. An example is a rapidly varying direction vector in region plane since a small change in the normal to plane will shift the region surface far away from the region point by a large displacement. Similarly, bad dynamics can also occur for fast changing variables employed in the move/rotate options.

See the *Howto tricilinc* page for a geometric description of triclinic boxes, as defined by LAMMPS, and how to transform these parameters to and from other commonly used triclinic representations.

The *union* style creates a region consisting of the volume of all the listed regions combined. The *intersect* style creates a region consisting of the volume that is common to all the listed regions.

**Note**

The *union* and *intersect* regions operate by invoking methods from their list of sub-regions. Thus you cannot delete the sub-regions after defining a *union* or *intersection* region.

---

The *side* keyword determines whether the region is considered to be inside or outside of the specified geometry. Using this keyword in conjunction with *union* and *intersect* regions, complex geometries can be built up. For example, if the interior of two spheres were each defined as regions, and a *union* style with *side* = out was constructed listing the region-IDs of the 2 spheres, the resulting region would be all the volume in the simulation box that was outside both of the spheres.

The *units* keyword determines the meaning of the distance units used to define the region for any argument above listed as having distance units. It also affects the scaling of the velocity vector specified with the *vel* keyword, the amplitude vector specified with the *wiggle* keyword, and the rotation point specified with the *rotate* keyword, since they each involve a distance metric.

A *box* value selects standard distance units as defined by the *units* command, e.g. Angstroms for units = real or metal. A *lattice* value means the distance units are in lattice spacings. The *lattice* command must have been previously used to define the lattice spacings which are used as follows:

- For style *block*, the lattice spacing in dimension x is applied to xlo and xhi, similarly the spacings in dimensions y,z are applied to ylo/yhi and zlo/zhi.
- For style *cone*, the lattice spacing in argument *dim* is applied to lo and hi. The spacings in the two radial dimensions are applied to c1 and c2. The two cone radii are scaled by the lattice spacing in the dimension corresponding to c1.
- For style *cylinder*, the lattice spacing in argument *dim* is applied to lo and hi. The spacings in the two radial dimensions are applied to c1 and c2. The cylinder radius is scaled by the lattice spacing in the dimension corresponding to c1.
- For style *ellipsoid*, the lattice spacing in dimensions x,y,z are applied to the ellipsoid center x,y,z. The spacing in dimensions x,y,z are applied to the ellipsoid radii a,b,c respectively.
- For style *plane*, the lattice spacing in dimension x is applied to px and nx, similarly the spacings in dimensions y,z are applied to py/ny and pz/nz.
- For style *prism*, the lattice spacing in dimension x is applied to xlo and xhi, similarly for ylo/yhi and zlo/zhi. The lattice spacing in dimension x is applied to xy and xz, and the spacing in dimension y to yz.
- For style *sphere*, the lattice spacing in dimensions x,y,z are applied to the sphere center x,y,z. The spacing in dimension x is applied to the sphere radius.

---

If the *move* or *rotate* keywords are used, the region is "dynamic", meaning its location or orientation changes with time. These keywords cannot be used with a *union* or *intersect* style region. Instead, the keywords should be used to make the individual sub-regions of the *union* or *intersect* region dynamic.

The *move* keyword allows one or more *equal-style variables* to be used to specify the x,y,z displacement of the region, typically as a function of time. A variable is specified as v_name, where name is the variable name. Any of the three variables can be specified as NULL, in which case no displacement is calculated in that dimension.

Note that equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a region displacement that change as a function of time or spans consecutive runs in a continuous fashion. For the latter, see the *start* and *stop* keywords of the *run* command and the *elaplong* keyword of *thermo_style custom* for details.

For example, these commands would displace a region from its initial position, in the positive x direction, effectively at a constant velocity:

```
variable dx equal ramp(0,10)
region 2 sphere 10.0 10.0 0.0 5 move v_dx NULL NULL
```

Note that the initial displacement is 0.0, though that is not required.

Either of these variables would "wiggle" the region back and forth in the y direction:

```
variable dy equal swiggle(0,5,100)
variable dysame equal 5*sin(2*PI*elaplong*dt/100)
region 2 sphere 10.0 10.0 0.0 5 move NULL v_dy NULL
```

The *rotate* keyword rotates the region around a rotation axis $R$ = (Rx,Ry,Rz) that goes through a point $P$ = (Px,Py,Pz). The rotation angle is calculated, presumably as a function of time, by a variable specified as v_theta, where theta is the variable name. The variable should generate its result in radians. The direction of rotation for the region around the rotation axis is consistent with the right-hand rule: if your right-hand thumb points along $R$, then your fingers wrap around the axis in the direction of rotation.

The *move* and *rotate* keywords can be used together. In this case, the displacement specified by the *move* keyword is applied to the $P$ point of the *rotate* keyword which is equivalent to applying the rotation *first* and then the translation.

---

The *open* keyword can be used (multiple times) to indicate that one or more faces of the region are ignored for purposes of particle/wall interactions. This keyword is only relevant for regions used by the *fix wall/region* and *fix wall/gran/region* commands. It can be used to create "open" containers where only some of the region faces are walls. For example, a funnel can be created with a *cone* style region that has an open face at the smaller radius for particles to flow out, or at the larger radius for pouring particles into the cone, or both.

Note that using the *open* keyword partly overrides the *side* keyword, since both exterior and interior surfaces of an open region are tested for particle contacts. The exception to this is a *union* or *intersect* region which includes an open sub-region. In that case the *side* keyword is still used to define the union/intersect region volume, and the *open* settings are only applied to the individual sub-regions that use them.

The indices specified as part of the *open* keyword have the following meanings:

For style *block*, indices 1-6 correspond to the xlo, xhi, ylo, yhi, zlo, zhi surfaces of the block. I.e. 1 is the yz plane at x = xlo, 2 is the yz-plane at x = xhi, 3 is the xz plane at y = ylo, 4 is the xz plane at y = yhi, 5 is the xy plane at z = zlo, 6 is the xy plane at z = zhi). In the second-to-last example above, the region is a box open at both xy planes.

For style *prism*, values 1-6 have the same mapping as for style *block*. I.e. in an untilted *prism*, *open* indices correspond to the xlo, xhi, ylo, yhi, zlo, zhi surfaces.

For style *cylinder*, index 1 corresponds to the flat end cap at the low coordinate along the cylinder axis, index 2 corresponds to the high-coordinate flat end cap along the cylinder axis, and index 3 is the curved cylinder surface. For

example, a *cylinder* region with *open 1 open 2* keywords will be open at both ends (e.g. a section of pipe), regardless of the cylinder orientation.

For style *cone*, the mapping is the same as for style *cylinder*. Index 1 is the low-coordinate flat end cap, index 2 is the high-coordinate flat end cap, and index 3 is the curved cone surface. In the last example above, a *cone* region is defined along the z-axis that is open at the zlo value (e.g. for use as a funnel).

For all other styles, the *open* keyword is ignored. As indicated above, this includes the *intersect* and *union* regions, though their sub-regions can be defined with the *open* keyword.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

**Note**

Currently, only *block* and *sphere* style regions are supported by KOKKOS. The code using the region (such as a fix or compute) must also be supported by KOKKOS or no acceleration will occur.

---

### 1.88.4 Restrictions

A prism cannot be of 0.0 thickness in any dimension; use a small z thickness for 2d simulations. For 2d simulations, the xz and yz parameters must be 0.0.

### 1.88.5 Related commands

*lattice*, *create_atoms*, *delete_atoms*, *group*

### 1.88.6 Default

The option defaults are side = in, units = lattice, and no move or rotation.

## 1.89 region2vmd command

### 1.89.1 Syntax

```
region2vmd filename keyword arg ...
```

- filename = name of file to write VMD script commands to
- zero or more keyword/arg pairs may be appended
- keyword = *region* or *color* or *material* or *command*

```
region region-ID = name of region to translate to VMD graphics
color color-name = set color for following visualized objects
material material-name = set material for following visualized objects
command string = string with custom VMD script command (in quotes)
```

### 1.89.2 Examples

```
region2vmd regions.vmd material Opaque color red region c1 color green region c2
region2vmd vizbox.vmd command "mol new system.lammpstrj waitfor all" region box
region2vmd regdefs.vmd region upper region lower region hole
```

### 1.89.3 Description

Added in version 2Apr2025.

Write a VMD Tcl script file with commands that aim to create a visualization of *regions*. There may be multiple region visualizations stored in a single file.

The visualization is implemented by creating a new (and empty) "VMD molecule" and then assigning a sequence of VMD graphics primitives to represent the region in VMD. Each region will be stored in a separate "VMD molecule" with the name "LAMMPS region <region ID>".

**Visualization of regions with *dump image***

Regions can also be directly visualized within LAMMPS using the *dump image* command. For more information, see *Visualize LAMMPS snapshots*.

The *region2vmd* command is followed by the filename for the resulting VMD script and an arbitrary number of keyword argument pairs to either write out a new *region* visualization, change the *color* or *material* setting, or to insert arbitrary VMD script *command*s. The keywords and arguments are processed in sequence.

The *region* keyword must be followed by a previously defined LAMMPS *region*. Only a limited set region styles and region settings are currently supported. See **Restrictions** below. Unsupported region styles or regions with unsupported settings will be skipped and a corresponding message is printed.

The *color* keyword must be followed by a color name that is defined in VMD. This color will be used by all following region visualizations. The default setting is 'silver'. VMD has the following colors pre-defined:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| blue | red | gray | orange | yellow | tan | silver | green | white | pink | cyan |
| purple | lime | mauve | ochre | iceblue | black | yellow2 | yellow3 | green2 | green3 | cyan2 |
| cyan3 | blue2 | blue3 | violet | violet2 | magenta | magenta2 | red2 | red3 | orange2 | orange3 |

The *material* keyword must be followed by a material name that is defined in VMD. This material will be used by all following visualizations. The default setting is 'Transparent'. VMD has the following materials pre-defined:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Opaque | Transparent | BrushedMetal | Diffuse | Ghost | Glass1 | Glass2 | Glass3 |
| Glossy | HardPlastic | MetallicPastel | Steel | Translucent | Edgy | EdgyShiny | EdgyGlass |
| Goodsell | AOShiny | AOChalky | AOEdgy | BlownGlass | GlassBubble | RTChrome | |

The *command* keyword must be followed by a VMD script command as a single string in quotes. This VMD command will be directly inserted into the created VMD script.

The created file can be loaded into VMD either from the command line with the '-e' flag, or from the command prompt with 'play <script file>', or from the File menu via "Load VMD visualization state".

**Setting the "top" molecule in VMD**

It is usually desirable to have the "molecule" with the LAMMPS trajectory set at "top" molecule in VMD and not one of the "region molecules". The VMD script generated by this region2vmd assumes that this molecule is already loaded and set as the current "top" molecule. Thus at the beginning of the script the index of the top molecule is stored in the VMD variable 'oldtop' and at the end of the script, that "top" molecule is restored. If no molecule is loaded, this can be inserted into the script with a custom command. The molecule index to this new molecules should be assigned to the oldtop variable. This can be done with e.g. `set oldtop [mol new {regions.vmd} waitfor all]`

---

### 1.89.4 Restrictions

Changed in version 11Feb2026.

This command is part of the GRAPHICS package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Only the following region styles are currently supported: *block*, *cone*, *cylinder*, *ellipsoid*, *prism*, and *sphere*. Regions formed from unions or intersections of other regions are not supported.

Rotating regions are currently not supported.

### 1.89.5 Related commands

*region*, *dump image*

### 1.89.6 Defaults

*color* = silver, *material* = Transparent

## 1.90 replicate command

### 1.90.1 Syntax

```
replicate nx ny nz keyword ...
```

nx,ny,nz = replication factors in each dimension

- zero or more keywords may be appended
- keyword = *bbox* or *bond/periodic*

  *bbox* = use a bounding-box algorithm which is faster for large proc counts
  *bond/periodic* = use an algorithm that correctly replicates periodic bond loops

### 1.90.2 Examples

For examples of replicating simple linear polymer chains (periodic or non-periodic) or periodic carbon nanotubes, see examples/replicate.

```
replicate 2 3 2
replicate 2 3 2 bbox
replicate 2 3 2 bond/periodic
```

### 1.90.3 Description

Replicate the current system one or more times in each dimension. For example, replication factors of 2,2,2 will create a simulation with 8x as many atoms by doubling the size of the simulation box in each dimension. A replication factor of 1 leaves the simulation domain unchanged in that dimension.

When the new simulation box is created it is partitioned into a regular 3d grid of rectangular bricks, one per processor, based on the number of processors being used and the settings of the *processors* command. The partitioning can be changed by subsequent *balance* or *fix balance* commands.

All properties of each atom are replicated (except per-atom fix data, see the Restrictions section below). This includes their velocities, which may or may not be desirable. New atom IDs are assigned to new atoms, as are new molecule IDs. Bonds and other topology interactions are created between pairs of new atoms as well as between old and new atoms.

**Note**

The bond discussion which follows only refers to models with permanent covalent bonds typically defined in LAMMPS via a data file. It is not relevant to systems modeled with many-body potentials which can define bonds on-the-fly, based on the current positions of nearby atoms, e.g. models using the *AIREBO* or *ReaxFF* potentials.

If the *bond/periodic* keyword is not specified, bond replication is done by using the image flag for each atom to "unwrap" it out of the periodic box before replicating it. After replication is performed, atoms outside the new periodic box are wrapped back into it. This assigns correct images flags to all atoms in the system. For this to work, all original atoms in the original simulation box must have consistent image flags. This means that if two atoms have a bond between them which crosses a periodic boundary, their respective image flags will differ by 1 in that dimension.

Image flag consistency is not possible if a system has a periodic bond loop, meaning there is a chain of bonds which crosses an entire dimension and re-connects to itself across a periodic boundary. In this case you MUST use the *bond/periodic* keyword to correctly replicate the system. This option zeroes the image flags for all atoms and uses a different algorithm to find new (nearby) bond neighbors in the replicated system. In the final replicated system all image flags are zero (in each dimension).

**Note**

LAMMPS does not check for image flag consistency before performing the replication (it does issue a warning about this before a simulation is run). If the original image flags are inconsistent, the replicated system will also have inconsistent image flags, but will otherwise be correctly replicated. This is NOT the case if there is a periodic bond loop. See the next note.

**Note**

LAMMPS does not check for periodic bond loops. If you use the *bond/periodic* keyword for a system without periodic bond loops, the system will be correctly replicated, but image flag information will be lost (which may or may not be important to your model). If you do not use the *bond/periodic* keyword for a system with periodic bond loops, the replicated system will have invalid bonds (typically very long), resulting in bad dynamics.

If possible, the *bbox* keyword should be used when running on a large number of processors, as it can result in a substantial speed-up for the replication operation. It uses a bounding box to only check atoms in replicas that overlap with each processor's new subdomain when assigning atoms to processors. It also preserves image flag information. The only drawback to the *bbox* option is that it requires a temporary use of more memory. Each processor must be able to store all atoms (and their per-atom data) in the original system, before it is replicated.

**Note**

The algorithm used by the *bond/periodic* keyword builds on the algorithm used by the *bbox* keyword and thus has the same memory requirements. If you specify only the *bond/peridoic* keyword it will internally set the *bbox* keyword as well.

### 1.90.4 Restrictions

A 2d simulation cannot be replicated in the z dimension.

If a simulation is non-periodic in a dimension, care should be used when replicating it in that dimension, as it may generate atoms nearly on top of each other.

If the current simulation was read in from a restart file (before a run is performed), there must not be any fix information stored in the file for individual atoms. Similarly, no fixes can be defined at the time the replicate command is used that require vectors of atom information to be stored. This is because the replicate command does not know how to replicate that information for new atoms it creates.

To work around this restriction two options are possible. (1) Fixes which use the stored data in the restart file can be defined before replication and then deleted via the *unfix* command and re-defined after it. Or (2) the restart file can be converted to a data file (which deletes the stored fix information) and fixes defined after the replicate command. In both these scenarios, the per-atom fix information in the restart file is lost.

### 1.90.5 Related commands

none

### 1.90.6 Default

No settings for using the *bbox* or *bond/periodic* algorithms.

## 1.91 rerun command

### 1.91.1 Syntax

```
rerun file1 file2 ... keyword args ...
```

- file1,file2,... = dump file(s) to read
- one or more keywords may be appended, keyword *dump* must appear and be last

```
keyword = first or last or every or skip or start or stop or post or dump
  first args = Nfirst
    Nfirst = dump timestep to start on
  last args = Nlast
    Nlast = dumptimestep to stop on
  every args = Nevery
```

```
  Nevery = read snapshots matching every this many timesteps
skip args = Nskip
  Nskip = read one out of every Nskip snapshots
start args = Nstart
  Nstart = timestep on which pseudo run will start
stop args = Nstop
  Nstop = timestep to which pseudo run will end
post value = yes or no
dump args = same as read_dump command starting with its field arguments
```

### 1.91.2 Examples

```
rerun dump.file dump x y z vx vy vz
rerun dump1.txt dump2.txt first 10000 every 1000 dump x y z
rerun dump.vels dump x y z vx vy vz box yes format molfile lammpstrj
rerun dump.dcd dump x y z box no format molfile dcd
rerun ../run7/dump.file.gz skip 2 dump x y z box yes
rerun dump.bp dump x y z box no format adios
rerun dump.bp dump x y z vx vy vz format adios timeout 10.0
```

### 1.91.3 Description

Perform a pseudo simulation run where atom information is read one snapshot at a time from a dump file(s), and energies and forces are computed on the shapshot to produce thermodynamic or other output.

This can be useful in the following kinds of scenarios, after an initial simulation produced the dump file:

- Compute the energy and forces of snapshots using a different potential.
- Calculate one or more diagnostic quantities on the snapshots that were not computed in the initial run. These can also be computed with settings not used in the initial run, e.g. computing an RDF via the *compute rdf* command with a longer cutoff than was used initially.
- Calculate the portion of per-atom forces resulting from a subset of the potential. E.g. compute only Coulombic forces. This can be done by only defining only a Coulombic pair style in the rerun script. Doing this in the original script would result in different (bad) dynamics.

Conceptually, using the rerun command is like running an input script that has a loop in it (see the *next* and *jump* commands). Each iteration of the loop reads one snapshot from the dump file via the *read_dump* command, sets the timestep to the appropriate value, and then invokes a *run* command for zero timesteps to simply compute energy and forces, and any other *thermodynamic output* or diagnostic info you have defined. This computation also invokes any fixes you have defined that apply constraints to the system, such as *fix shake* or *fix indent*.

Note that a simulation box must already be defined before using the rerun command. This can be done by the *create_box*, *read_data*, or *read_restart* commands.

Also note that reading per-atom information from dump snapshots is limited to the atom coordinates, velocities and image flags as explained in the *read_dump* command. Other atom properties, which may be necessary to compute energies and forces, such as atom charge, or bond topology information for a molecular system, are not read from (or even contained in) dump files. Thus this auxiliary information should be defined in the usual way, e.g. in a data file read in by a *read_data* command, before using the rerun command.

Also note that the frequency of thermodynamic or dump output from the rerun simulation will depend on settings made in the rerun script, the same as for output from any LAMMPS simulation. See further info below as to what that means if the timesteps for snapshots read from dump files do not match the specified output frequency.

---

If more than one dump file is specified, the dump files are read one after the other in the order specified. It is assumed that snapshot timesteps will be in ascending order. If a snapshot is encountered that is not in ascending order, it will skip the snapshot until it reads one that is. This allows skipping of a duplicate snapshot (same timestep), e.g. that appeared at the end of one file and beginning of the next. However if you specify a series of dump files in an incorrect order (with respect to the timesteps they contain), you may skip large numbers of snapshots.

Note that the dump files specified as part of the *dump* keyword can be parallel files, i.e. written as multiple files either per processor and/or per snapshot. If that is the case they will also be read in parallel which can make the rerun command operate dramatically faster for large systems. See the page for the *read_dump* and *dump* commands which describe how to read and write parallel dump files.

The *first*, *last*, *every*, *skip* keywords determine which snapshots are read from the dump file(s). Snapshots are skipped until they have a timestep >= *Nfirst*. When a snapshot with a timestep > *Nlast* is encountered, the rerun command finishes. Note that the defaults for *first* and *last* are to read all snapshots. If the *every* keyword is set to a value > 0, then only snapshots with timesteps that are a multiple of *Nevery* are read (the first snapshot is always read). If *Nevery* = 0, then this criterion is ignored, i.e. every snapshot is read that meets the other criteria. If the *skip* keyword is used, then after the first snapshot is read, every Nth snapshot is read, where N = *Nskip*. E.g. if *Nskip* = 3, then only 1 out of every 3 snapshots is read, assuming the snapshot timestep is also consistent with the other criteria.

**Note**

Not all dump formats contain the timestep and not all dump readers support reading it. In that case individual snapshots are assigned consecutive timestep numbers starting at 1.

The *start* and *stop* keywords do not affect which snapshots are read from the dump file(s). Rather, they have the same meaning that they do for the *run* command. They only need to be defined if (a) you are using a *fix* command that changes some value over time, and (b) you want the reference point for elapsed time (from start to stop) to be different than the *first* and *last* settings. See the page for individual fixes to see which ones can be used with the *start/stop* keywords. Note that if you define neither of the *start*/*stop* or *first*/*last* keywords, then LAMMPS treats the pseudo run as going from 0 to a huge value (effectively infinity). This means that any quantity that a fix scales as a fraction of elapsed time in the run, will essentially remain at its initial value. Also note that an error will occur if you read a snapshot from the dump file with a timestep value larger than the *stop* setting you have specified.

The *post* keyword can be used to minimize the output to the screen that happens after a *rerun* command, similar to the post keyword of the *run command*. It is set to *no* by default.

The *dump* keyword is required and must be the last keyword specified. Its arguments are passed internally to the *read_dump* command. The first argument following the *dump* keyword should be the *field1* argument of the *read_dump* command. See the *read_dump* page for details on the various options it allows for extracting information from the dump file snapshots, and for using that information to alter the LAMMPS simulation.

---

In general, a LAMMPS input script that uses a rerun command can include and perform all the usual operations of an input script that uses the *run* command. There are a few exceptions and points to consider, as discussed here.

Fixes that perform time integration, such as *fix nve* or *fix npt* are not invoked, since no time integration is performed. Fixes that perturb or constrain the forces on atoms will be invoked, just as they would during a normal run. Examples are *fix indent* and *fix langevin*. So you should think carefully as to whether that makes sense for the manner in which you are reprocessing the dump snapshots.

If you only want the rerun script to perform an analysis that does not involve pair interactions, such as use compute msd to calculated displacements over time, you do not need to define a *pair style*, which may also mean neighbor lists will not need to be calculated which saves time. The *comm_modify cutoff* command can also be used to ensure ghost atoms are acquired from far enough away for operations like bond and angle evaluations, if no pair style is being used.

Every time a snapshot is read, the timestep for the simulation is reset, as if the *reset_timestep* command were used. This command has some restrictions as to what fixes can be defined. See its documentation page for details. For example, the *fix deposit* and *fix dt/reset* fixes are in this category. They also make no sense to use with a rerun command.

If time-averaging fixes like *fix ave/time* are used, they are invoked on timesteps that are a function of their *Nevery*, *Nrepeat*, and *Nfreq* settings. As an example, see the *fix ave/time* page for details. You must ensure those settings are consistent with the snapshot timestamps that are read from the dump file(s). If an averaging fix is not invoked on a timestep it expects to be, LAMMPS will flag an error.

The various forms of LAMMPS output, as defined by the *thermo_style*, *thermo*, *dump*, and *restart* commands occur with specified frequency, e.g. every N steps. If the timestep for a dump snapshot is not a multiple of N, then it will be read and processed, but no output will be produced. If you want output for every dump snapshot, you can simply use N=1 for an output frequency, e.g. for thermodynamic output or new dump file output.

---

### 1.91.4 Restrictions

The *rerun* command is subject to all restrictions of the *read_dump* command.

### 1.91.5 Related commands

*read_dump*

### 1.91.6 Default

The option defaults are first = 0, last = a huge value (effectively infinity), start = same as first, stop = same as last, every = 0, skip = 1, post = no;

## 1.92 reset_atoms command

### 1.92.1 Syntax

```
reset_atoms property arguments ...
```

- property = *id* or *image* or *mol*
- additional arguments depend on the property

```
reset_atoms id keyword value ...
```

  - zero or more keyword/value pairs can be appended
  - keyword = *sort*

    *sort* value = *yes* or *no*

```
reset_atoms image group-ID
```

  - group-ID = ID of group of atoms whose image flags will be reset

```
reset_atoms mol group-ID keyword value ...
```

  - group-ID = ID of group of atoms whose molecule IDs will be reset
  - zero or more keyword/value pairs can be appended
  - keyword = *compress* or *offset* or *single*

```
compress value = yes or no
offset value = Noffset >= -1
single value = yes or no to treat single atoms (no bonds) as molecules
```

### 1.92.2 Examples

```
reset_atoms id
reset_atoms id sort yes
reset_atoms image all
reset_atoms image mobile
reset_atoms mol all
reset_atoms mol all offset 10 single yes
reset_atoms mol solvent compress yes offset 100
reset_atoms mol solvent compress no
```

### 1.92.3 Description

Added in version 22Dec2022.

The *reset_atoms* command resets the values of a specified atom property. In contrast to the *set* command, it does this in a collective manner which resets the values for many atoms in a self-consistent way. This command is often useful when the simulated system has undergone significant modifications like adding or removing atoms or molecules, joining data files, changing bonds, or large-scale diffusion.

The new values can be thought of as a *reset*, similar to values atoms would have if a new data file were being read or a new simulation performed. Note that the *set* command also resets atom properties to new values, but it treats each atom independently.

The *property* setting can be *id* or *image* or *mol*. For *id*, the IDs of all the atoms are reset to contiguous values. For *image*, the image flags of atoms in the specified *group-ID* are reset so that at least one atom in each molecule is in the simulation box (image flag = 0). For *mol*, the molecule IDs of all atoms are reset to contiguous values.

More details on these operations and their arguments or optional keyword/value settings are given below.

---

Property: *id*

Reset atom IDs for the entire system, including all the global IDs stored for bond, angle, dihedral, improper topology data. This will create a set of IDs that are numbered contiguously from 1 to N for a N atoms system.

This can be useful to do after performing a "delete_atoms" command for a molecular system. The delete_atoms compress yes option will not perform this operation due to the existence of bond topology. It can also be useful to do after any simulation which has lost atoms, e.g. due to atoms moving outside a simulation box with fixed boundaries (see the "boundary command"), or due to evaporation (see the "fix evaporate" command).

If the *sort* keyword is used with a setting of *yes*, then the assignment of new atom IDs will be the same no matter how many processors LAMMPS is running on. This is done by first doing a spatial sort of all the atoms into bins and sorting them within each bin. Because the set of bins is independent of the number of processors, this enables a consistent assignment of new IDs to each atom.

This can be useful to do after using the "create_atoms" command and/or "replicate" command. In general those commands do not guarantee assignment of the same atom ID to the same physical atom when LAMMPS is run on different numbers of processors. Enforcing consistent IDs can be useful for debugging or comparing output from two different runs.

Note that the spatial sort requires communication of atom IDs and coordinates between processors in an all-to-all manner. This is done efficiently in LAMMPS, but it is more expensive than how atom IDs are reset without sorting.

Note that whether sorting or not, the resetting of IDs is not a compression, where gaps in atom IDs are removed by decrementing atom IDs that are larger. Instead the IDs for all atoms are erased, and new IDs are assigned so that the atoms owned by an individual processor have consecutive IDs, as the *create_atoms* command explains.

**Note**

If this command is used before a *pair style* is defined, an error about bond topology atom IDs not being found may result. This is because the cutoff distance for ghost atom communication was not sufficient to find atoms in bonds, angles, etc that are owned by other processors. The *comm_modify cutoff* command can be used to correct this issue. Or you can define a pair style before using this command. If you do the former, you should unset the *comm_modify cutoff* after using *reset atoms id* so that subsequent communication is not inefficient.

---

Property: *image*

Reset the image flags of atoms so that at least one atom in each molecule has an image flag of 0. Molecular topology is respected so that if the molecule straddles a periodic simulation box boundary, the images flags of all atoms in the molecule will be consistent. This avoids inconsistent image flags that could result from resetting all image flags to zero with the *set* command.

**Note**

If the system has no bonds, there is no reason to use this command, since image flags for different atoms do not need to be consistent. Use the *set* command with its *image* keyword instead.

Only image flags for atoms in the specified *group-ID* are reset; all others remain unchanged. No check is made for whether the group covers complete molecule fragments and thus whether the command will result in inconsistent image flags.

Molecular fragments are identified by the algorithm used by the *compute fragment/atom* command. For each fragment the average of the largest and the smallest image flag in each direction across all atoms in the fragment is computed and subtracted from the current image flag in the same direction.

This can be a useful operation to perform after running longer equilibration runs of mobile systems where molecules would pass through the system multiple times and thus produce non-zero image flags.

**Note**

Same as explained for the *compute fragment/atom* command, molecules are identified using the current bond topology. This will **not** account for bonds broken by the *bond_style quartic* command, because this bond style does not perform a full update of the bond topology data structures within LAMMPS. In that case, using the *delete_bonds all bond 0 remove* will permanently delete such broken bonds and should thus be used first.

---

Property: *mol*

Reset molecule IDs for a specified group of atoms based on current bond connectivity. This will typically create a new set of molecule IDs for atoms in the group. Only molecule IDs for atoms in the specified *group-ID* are reset; molecule IDs for atoms not in the group are not changed.

For purposes of this operation, molecules are identified by the current bond connectivity in the system, which may or may not be consistent with the current molecule IDs. A molecule in this context is a set of atoms connected to each other with explicit bonds. The specific algorithm used is the one of *compute fragment/atom*. Once the molecules are

identified and a new molecule ID computed for each, this command will update the current molecule ID for all atoms in the group with the new molecule ID. Note that if the group excludes atoms within molecules, one (physical) molecule may become two or more (logical) molecules. For example if the group excludes atoms in the middle of a linear chain, then each end of the chain is considered an independent molecule and will be assigned a different molecule ID.

This can be a useful operation to perform after running reactive molecular dynamics run with *fix bond/react*, *fix bond/create*, or *fix bond/break*, all of which can change molecule topologies. It can also be useful after molecules have been deleted with the *delete_atoms* command or after a simulation which has lost molecules, e.g. via the *fix evaporate* command.

The *compress* keyword determines how new molecule IDs are computed. If the setting is *yes* (the default) and there are N molecules in the group, the new molecule IDs will be a set of N contiguous values. See the *offset* keyword for details on selecting the range of these values. If the setting is *no*, the molecule ID of every atom in the molecule will be set to the smallest atom ID of any atom in the molecule.

The *single* keyword determines whether single atoms (not bonded to another atom) are treated as one-atom molecules or not, based on the *yes* or *no* setting. If the setting is *no* (the default), their molecule IDs are set to 0. This setting can be important if the new molecule IDs will be used as input to other commands such as *compute chunk/atom molecule* or *fix rigid molecule*.

The *offset* keyword is only used if the *compress* setting is *yes*. Its default value is *Noffset* = -1. In that case, if the specified group is *all*, then the new compressed molecule IDs will range from 1 to N. If the specified group is not *all* and the largest molecule ID of atoms outside that group is M, then the new compressed molecule IDs will range from M+1 to M+N, to avoid collision with existing molecule IDs. If an *Noffset* >= 0 is specified, then the new compressed molecule IDs will range from *Noffset*+1 to *Noffset*+N. If the group is not *all* there may be collisions with the molecule IDs of other atoms.

**Note**

Same as explained for the *compute fragment/atom* command, molecules are identified using the current bond topology. This will **not** account for bonds broken by the *bond_style quartic* command, because this bond style does not perform a full update of the bond topology data structures within LAMMPS. In that case, using the *delete_bonds all bond 0 remove* will permanently delete such broken bonds and should thus be used first.

### 1.92.4 Restrictions

The *image* property can only be used when the atom style supports bonds.

### 1.92.5 Related commands

*compute fragment/atom*, *fix bond/react*, *fix bond/create*, *fix bond/break*, *fix evaporate*, *delete_atoms*, *delete_bonds*

### 1.92.6 Defaults

For property *id*, the default keyword setting is sort = no.

For property *mol*, the default keyword settings are compress = yes, single = no, and offset = -1.

## 1.93 reset_timestep command

### 1.93.1 Syntax

```
reset_timestep N keyword values ...
```

- N = timestep number

- zero or more keyword/value pairs may be appended
- keyword = *time*

  *time* value = atime
     atime = accumulated simulation time

### 1.93.2 Examples

```
reset_timestep 0
reset_timestep 4000000
reset_timestep 1000 time 100.0
```

### 1.93.3 Description

Set the timestep counter to the specified value. This command usually comes after the timestep has been set by reading a restart file via the *read_restart* command, or a previous simulation run or minimization advanced the timestep.

The optional *time* keyword allows to also set the accumulated simulation time. This is usually the number of timesteps times the size of the timestep, but when using variable size timesteps with *fix dt/reset* it can differ.

The *read_data* and *create_box* commands set the timestep to 0; the *read_restart* command sets the timestep to the value it had when the restart file was written. The same applies to the accumulated simulation time.

### 1.93.4 Restrictions

This command cannot be used when any fixes are defined that keep track of elapsed time to perform certain kinds of time-dependent operations. Examples are the *fix deposit* and *fix dt/reset* commands. The former adds atoms on specific timesteps. The latter keeps track of accumulated time.

Various fixes use the current timestep to calculate related quantities. If the timestep is reset, this may produce unexpected behavior, but LAMMPS allows the fixes to be defined even if the timestep is reset. For example, commands which thermostat the system, e.g. *fix nvt*, allow you to specify a target temperature which ramps from Tstart to Tstop which may persist over several runs. If you change the timestep, you may induce an instantaneous change in the target temperature.

Resetting the timestep clears flags for *computes* that may have calculated some quantity from a previous run. This means these quantity cannot be accessed by a variable in between runs until a new run is performed. See the *variable* command for more details.

### 1.93.5 Related commands

*rerun*, *timestep*, *fix dt/reset*

### 1.93.6 Default

none

## 1.94 restart command

### 1.94.1 Syntax

```
restart 0
restart N root keyword value ...
restart N file1 file2 keyword value ...
```

- N = write a restart file on timesteps which are multiples of N
- N can be a variable (see below)
- root = filename to which timestep # is appended
- file1,file2 = two full filenames, toggle between them when writing file
- zero or more keyword/value pairs may be appended
- keyword = *fileper* or *nfile*

```
fileper arg = Np
  Np = write one file for every this many processors
nfile arg = Nf
  Nf = write this many files, one from each of Nf processors
```

### 1.94.2 Examples

```
restart 0
restart 1000 poly.restart
restart 1000 restart.*.equil
restart 10000 poly.%.1 poly.%.2 nfile 10
restart v_mystep poly.restart
```

### 1.94.3 Description

Write out a binary restart file with the current state of the simulation on timesteps which are a multiple of N. A value of N = 0 means do not write out any restart files, which is the default. Restart files are written in one (or both) of two modes as a run proceeds. If one filename is specified, a series of filenames will be created which include the timestep in the filename. If two filenames are specified, only 2 restart files will be created, with those names. LAMMPS will toggle between the 2 names as it writes successive restart files.

Note that you can specify the restart command twice, once with a single filename and once with two filenames. This would allow you, for example, to write out archival restart files every 100000 steps using a single filename, and more frequent temporary restart files every 1000 steps, using two filenames. Using restart 0 will turn off both modes of output.

Similar to *dump* files, the restart filename(s) can contain two wild-card characters.

If a "*" appears in the single filename, it is replaced with the current timestep value. This is only recognized when a single filename is used (not when toggling back and forth). Thus, the third example above creates restart files as follows: restart.1000.equil, restart.2000.equil, etc. If a single filename is used with no "*", then the timestep value is appended. E.g. the second example above creates restart files as follows: poly.restart.1000, poly.restart.2000, etc.

If a "%" character appears in the restart filename(s), then one file is written for each processor and the "%" character is replaced with the processor ID from 0 to P-1. An additional file with the "%" replaced by "base" is also written, which contains global information. For example, the files written on step 1000 for filename restart.% would be restart.base.1000, restart.0.1000, restart.1.1000, ..., restart.P-1.1000. This creates smaller files and can be a fast mode of output and subsequent input on parallel machines that support parallel I/O. The optional *fileper* and *nfile* keywords discussed below can alter the number of files written.

Restart files are written on timesteps that are a multiple of N but not on the first timestep of a run or minimization. You can use the *write_restart* command to write a restart file before a run begins. A restart file is not written on the last timestep of a run unless it is a multiple of N. A restart file is written on the last timestep of a minimization if N > 0 and the minimization converges.

**Enabling or disabling restarts may change the trajectory**

Writing a restart file forces a neighbor list rebuild on that timestep, so that all atoms are correctly assigned to their owning subdomains before the file is written. This rebuild happens regardless of the *neigh_modify* *every*, *delay*, and *check* settings. As a result, enabling restart output can change the number of neighbor list builds reported at the end of a run. And because changing when neighbor lists are rebuilt changes the order in which pairwise interactions are summed, it can also lead to small differences in the trajectory (due to the limitations of floating-point math) compared to an otherwise identical run without restart output.

To obtain the *same* number of neighbor list builds (and thus an identical trajectory) whether or not restart output is enabled, use *neigh_modify* with *check* set to *no* and *every* set to a value <*num*> that evenly divides the restart frequency N (with the default *delay* of 0). The neighbor lists are then rebuilt on a fixed schedule that already includes every restart-write step, so the forced rebuild coincides with a rebuild that would have happened anyway and no extra builds are added.

Instead of a numeric value, N can be specified as an *equal-style variable*, which should be specified as v_name, where name is the variable name. In this case, the variable is evaluated at the beginning of a run to determine the next timestep at which a restart file will be written out. On that timestep, the variable will be evaluated again to determine the next timestep, etc. Thus the variable should return timestep values. See the stagger() and logfreq() and stride() math functions for *equal-style variables*, as examples of useful functions to use in this context. Other similar math functions could easily be added as options for *equal-style variables*.

For example, the following commands will write restart files every step from 1100 to 1200, and could be useful for debugging a simulation where something goes wrong at step 1163:

```
variable        s equal stride(1100,1200,1)
restart         v_s tmp.restart
```

---

See the *read_restart* command for information about what is stored in a restart file.

Restart files can be read by a *read_restart* command to restart a simulation from a particular state. Because the file is binary (to enable exact restarts), it may not be readable on another machine. In this case, you can use the *-r command-line switch* to convert a restart file to a data file.

**Note**

Although the purpose of restart files is to enable restarting a simulation from where it left off, not all information about a simulation is stored in the file. For example, the list of fixes that were specified during the initial run is not stored, which means the new input script must specify any fixes you want to use. Even when restart information is stored in the file, as it is for some fixes, commands may need to be re-specified in the new input script, in order to re-use that information. See the *read_restart* command for information about what is stored in a restart file.

---

The optional *nfile* or *fileper* keywords can be used in conjunction with the "%" wildcard character in the specified restart file name(s). As explained above, the "%" character causes the restart file to be written in pieces, one piece for each of P processors. By default P = the number of processors the simulation is running on. The *nfile* or *fileper* keyword can be used to set P to a smaller value, which can be more efficient when running on a large number of processors.

The *nfile* keyword sets P to the specified Nf value. For example, if Nf = 4, and the simulation is running on 100 processors, 4 files will be written, by processors 0,25,50,75. Each will collect information from itself and the next 24 processors and write it to a restart file.

For the *fileper* keyword, the specified value of Np means write one file for every Np processors. For example, if Np = 4, every fourth processor (0,4,8,12,etc) will collect information from itself and the next 3 processors and write it to a restart file.

---

### 1.94.4 Restrictions

none

### 1.94.5 Related commands

*write_restart*, *read_restart*

### 1.94.6 Default

```
restart 0
```

## 1.95 run command

### 1.95.1 Syntax

```
run N keyword values ...
```

- N = # of timesteps
- zero or more keyword/value pairs may be appended
- keyword = *upto* or *start* or *stop* or *pre* or *post* or *every*

```
upto value = none
start value = N1
  N1 = timestep at which 1st run started
stop value = N2
  N2 = timestep at which last run will end
pre value = no or yes
post value = no or yes
every values = M c1 c2 ...
  M = break the run into M-timestep segments and invoke one or more commands␣
↪between each segment
  c1,c2,...,cN = one or more LAMMPS commands, each enclosed in quotes
  c1 = NULL means no command will be invoked
```

### 1.95.2 Examples

```
run 10000
run 1000000 upto
run 100 start 0 stop 1000
run 1000 pre no post yes
run 100000 start 0 stop 1000000 every 1000 "print 'Protein Rg = $r'"
run 100000 every 1000 NULL
```

### 1.95.3 Description

Run or continue dynamics for a specified number of timesteps.

When the *run style* is *respa*, N refers to outer loop (largest) timesteps.

A value of N = 0 is acceptable; only the thermodynamics of the system are computed and printed without taking a timestep.

The *upto* keyword means to perform a run starting at the current timestep up to the specified timestep. E.g. if the current timestep is 10,000 and "run 100000 upto" is used, then an additional 90,000 timesteps will be run. This can be useful for very long runs on a machine that allocates chunks of time and terminate your job when time is exceeded. If you need to restart your script multiple times (reading in the last restart file), you can keep restarting your script with the same run command until the simulation finally completes.

The *start* or *stop* keywords can be used if multiple runs are being performed and you want a *fix* command that changes some value over time (e.g. temperature) to make the change across the entire set of runs and not just a single run. See the page for individual fixes to see which ones can be used with the *start/stop* keywords.

For example, consider this fix followed by 10 run commands:

```
fix         1 all nvt 200.0 300.0 1.0
run         1000 start 0 stop 10000
run         1000 start 0 stop 10000
...
run         1000 start 0 stop 10000
```

The NVT fix ramps the target temperature from 200.0 to 300.0 during a run. If the run commands did not have the start/stop keywords (just "run 1000"), then the temperature would ramp from 200.0 to 300.0 during the 1000 steps of each run. With the start/stop keywords, the ramping takes place over the 10000 steps of all runs together.

The *pre* and *post* keywords can be used to streamline the setup, clean-up, and associated output to the screen that happens before and after a run. This can be useful if you wish to do many short runs in succession (e.g. LAMMPS is being called as a library which is doing other computations between successive short LAMMPS runs).

By default (pre and post = yes), LAMMPS creates neighbor lists, computes forces, and imposes fix constraints before every run. And after every run it gathers and prints timings statistics. If a run is just a continuation of a previous run (i.e. no settings are changed), the initial computation is not necessary; the old neighbor list is still valid as are the forces. So if *pre* is specified as "no" then the initial setup is skipped, except for printing thermodynamic info. Note that if *pre* is set to "no" for the very first run LAMMPS performs, then it is overridden, since the initial setup computations must be done.

**Note**

If your input script "changes" the system between 2 runs, then the initial setup typically needs to be performed to ensure the change is recognized by all parts of the code that are affected. Examples are adding a *fix* or *dump* or *compute*, changing a *neighbor* list parameter, using the *set* command, or writing a restart file via the *write_restart* command, which can migrate atoms between processors. LAMMPS has no easy way to check if this has happened, but it is an error to use the *pre no* option in these cases.

If *post* is specified as "no", the full timing summary is skipped; only a one-line summary timing is printed.

The *every* keyword provides a means of breaking a LAMMPS run into a series of shorter runs. Optionally, one or more LAMMPS commands (c1, c2, ..., cN) will be executed in between the short runs. If used, the *every* keyword must be the last keyword, since it has a variable number of arguments. Each of the trailing arguments is a single LAMMPS command, and each command should be enclosed in quotes, so that the entire command will be treated as a single argument. This will also prevent any variables in the command from being evaluated until it is executed multiple times

during the run. Note that if a command itself needs one of its arguments quoted (e.g. the *print* command), then you can use a combination of single and double quotes, as in the example above or below.

The *every* keyword is a means to avoid listing a long series of runs and interleaving commands in your input script. For example, a *print* command could be invoked or a *fix* could be redefined, e.g. to reset a thermostat temperature. Or this could be useful for invoking a command you have added to LAMMPS that wraps some other code (e.g. as a library) to perform a computation periodically during a long LAMMPS run. See the *Modify* doc page for info about how to add new commands to LAMMPS. See the *Howto couple* page for ideas about how to couple LAMMPS to other codes.

With the *every* option, N total steps are simulated, in shorter runs of M steps each. After each M-length run, the specified commands are invoked. If only a single command is specified as NULL, then no command is invoked. Thus these lines:

```
variable q equal x[100]
run 6000 every 2000 "print 'Coord = $q'"
```

are the equivalent of:

```
variable q equal x[100]
run 2000
print "Coord = $q"
run 2000
print "Coord = $q"
run 2000
print "Coord = $q"
```

which does 3 runs of 2000 steps and prints the x-coordinate of a particular atom between runs. Note that the variable "$q" will be evaluated afresh each time the print command is executed.

Note that by using the line continuation character "&", the run every command can be spread across many lines, though it is still a single command:

```
run 100000 every 1000 &
  "print 'Minimum value = $a'" &
  "print 'Maximum value = $b'" &
  "print 'Temp = $c'" &
  "print 'Press = $d'"
```

If the *pre* and *post* options are set to "no" when used with the *every* keyword, then the first run will do the full setup and the last run will print the full timing summary, but these operations will be skipped for intermediate runs.

**Note**

You might wish to specify a command that exits the run by jumping out of the loop, e.g.

```
variable t equal temp
run 10000 every 100 "if '$t < 300.0' then 'jump SELF afterrun'"
```

However, this will not work. The run command simply executes each command one at a time each time it pauses, then continues the run.

Instead, you should use the *fix halt* command, which has additional options for how to exit the run.

### 1.95.4 Restrictions

When not using the *upto* keyword, the number of specified timesteps N must fit in a signed 32-bit integer, so you are limited to slightly more than 2 billion steps (2^31) in a single run. When using *upto*, N can be larger than a signed 32-bit integer, however the difference between N and the current timestep must still be no larger than 2^31 steps.

However, with or without the *upto* keyword, you can perform successive runs to run a simulation for any number of steps (ok, up to 2^63 total steps). I.e. the timestep counter within LAMMPS is a 64-bit signed integer.

### 1.95.5 Related commands

*minimize*, *run_style*, *temper*, *fix halt*

### 1.95.6 Default

The option defaults are start = the current timestep, stop = current timestep + N, pre = yes, and post = yes.

## 1.96 run_style command

### 1.96.1 Syntax

```
run_style style args
```

- style = *verlet* or *verlet/split* or *verlet/split/rk* or *respa* or *respa/omp*

```
verlet args = none
verlet/split args = none
verlet/split/rk args = none
respa args = N n1 n2 ... keyword values ...
  N = # of levels of rRESPA
  n1, n2, ... = loop factors between rRESPA levels (N-1 values)
  zero or more keyword/value pairings may be appended to the loop factors
  keyword = bond or angle or dihedral or improper or
            pair or inner or middle or outer or hybrid or kspace
    bond value = M
      M = which level (1-N) to compute bond forces in
    angle value = M
      M = which level (1-N) to compute angle forces in
    dihedral value = M
      M = which level (1-N) to compute dihedral forces in
    improper value = M
      M = which level (1-N) to compute improper forces in
    pair value = M
      M = which level (1-N) to compute pair forces in
    inner values = M cut1 cut2
      M = which level (1-N) to compute pair inner forces in
      cut1 = inner cutoff between pair inner and
             pair middle or outer  (distance units)
      cut2 = outer cutoff between pair inner and
             pair middle or outer  (distance units)
    middle values = M cut1 cut2
      M = which level (1-N) to compute pair middle forces in
      cut1 = inner cutoff between pair middle and pair outer (distance units)
      cut2 = outer cutoff between pair middle and pair outer (distance units)
    outer value = M
```

```
      M = which level (1-N) to compute pair outer forces in
    hybrid values = M1 M2 ... (as many values as there are hybrid sub-styles
      M1 = which level (1-N) to compute the first pair_style hybrid sub-style in
      M2 = which level (1-N) to compute the second pair_style hybrid sub-style in
      M3,etc
    kspace value = M
      M = which level (1-N) to compute kspace forces in
```

### 1.96.2 Examples

```
run_style verlet
run_style verlet/split
run_style verlet/split/rk
run_style respa 4 2 2 2 bond 1 dihedral 2 pair 3 kspace 4
run_style respa 4 2 2 2 bond 1 dihedral 2 inner 3 5.0 6.0 outer 4 kspace 4
run_style respa 3 4 2 bond 1 hybrid 2 2 1 kspace 3
```

### 1.96.3 Description

Choose the style of time integrator used for molecular dynamics simulations performed by LAMMPS.

The *verlet* style is the velocity form of the Stoermer-Verlet time integration algorithm (velocity-Verlet)

---

The *verlet/split* style is also a velocity-Verlet integrator, but it splits the force calculation within each timestep over 2 partitions of processors. See the *-partition command-line switch* for info on how to run LAMMPS with multiple partitions.

Specifically, this style performs all computation except the *kspace_style* portion of the force field on the first partition. This include the *pair style*, *bond style*, *neighbor list building*, *fixes* including time integration, and output. The *kspace_style* portion of the calculation is performed on the second partition.

This can lead to a significant speedup, if the number of processors can be easily increased and the fraction of time is spent in computing Kspace interactions is significant, too. The two partitions may have a different number of processors. This is most useful for the PPPM kspace_style when its performance on a large number of processors degrades due to the cost of communication in its 3d FFTs. In this scenario, splitting your P total processors into 2 subsets of processors, P1 in the first partition and P2 in the second partition, can enable your simulation to run faster. This is because the long-range forces in PPPM can be calculated at the same time as pairwise and bonded forces are being calculated *and* the parallel 3d FFTs can be faster to compute when running on fewer processors. Please note that the scenario of using fewer MPI processes to reduce communication overhead can also be implemented through using MPI with OpenMP threads via the INTEL, KOKKOS, or OPENMP package. This alternative option is typically more effective in case of a fixed number of available processors and less complex to execute.

To use the *verlet/split* style, you must define 2 partitions with the *-partition command-line switch*, where partition P1 is either the same size or an integer multiple of the size of the partition P2. Typically having P1 be 3x larger than P2 is a good choice, since the (serial) performance of LAMMPS is often best if the time spent in the `Pair` computation versus `Kspace` is a 3:1 split. The 3d processor layouts in each partition must overlay in the following sense. If P1 is a Px1 by Py1 by Pz1 grid, and P2 = Px2 by Py2 by Pz2, then Px1 must be an integer multiple of Px2, and similarly for Py1 a multiple of Py2, and Pz1 a multiple of Pz2.

Typically the best way to do this is to let the first partition choose its own optimal layout, then require the second partition's layout to match the integer multiple constraint. See the *processors* command with its *part* keyword for a way to control this, e.g.

```
processors * * * part 1 2 multiple
```

You can also use the *partition* command to explicitly specify the processor layout on each partition. E.g. for 2 partitions of 60 and 15 processors each:

```
partition yes 1 processors 3 4 5
partition yes 2 processors 3 1 5
```

When you run in 2-partition mode with the *verlet/split* style, the thermodynamic data for the entire simulation will be output to the log and screen file of the first partition, which are log.lammps.0 and screen.0 by default; see the *-plog and -pscreen command-line switches* to change this. The log and screen file for the second partition will not contain thermodynamic output beyond the first timestep of the run.

See the *Accelerator packages* page for performance details of the speed-up offered by the *verlet/split* style. One important performance consideration is the assignment of logical processors in the 2 partitions to the physical cores of a parallel machine. The *processors* command has options to support this, and strategies are discussed in *Section 5* of the manual.

---

Added in version 4Jul2026.

The requirements for the run style *verlet/split/rk* are similar to that of *verlet/split*, and much of what was written above about the latter applies to the former, including the partition requirements. Note that the use of run style *verlet/split/rk* must be paired with the use of kspace style *pppm/rk*.

The main difference is that for *verlet/split/rk*, the work of the K space processes is restricted to the solution of the Poisson equations for computing the long-range forces via pppm. By contrast, the R space processes carry out the other long-range force computations that the K space processes would have done for *verlet/split*. There are two rationales for considering the use of *verlet/split/rk*.

- *Rationale 1*: One of the most significant bottlenecks to parallel scalability is due to the communication overhead for the 3d FFTs in solving the Poisson equation. The other steps (such as accumulating the charge densities and interpolating the long-range forces) have better parallel scaling, and so their reallocation to the R space processes may result in improved parallel speedup at large enough node counts where the speedup for solving the Poisson equation lags behind the speedup of the other computations (thus the Poisson equation solution is a bottleneck in parallel execution).
- *Rationale 2*: The approach *verlet/split/rk* avoids having to communicate atom-specific information between the R space and K space processes. Thus, re-neighboring, for example, stays confined to the R space processes and does not require inter-R/K communication, and so the communication overhead is reduced.

The implementation of *verlet/split/rk* requires the parallel partitioning to occur in the computation of the long-range forces. Consequently, the use of *verlet/split/rk* requires the use of an */rk* type kspace style like *pppm/rk*.

This approach is based on the enhanced baseline decomposition of *(Dandurand)* and works cited within.

---

The *respa* style implements the rRESPA multi-timescale integrator *(Tuckerman)* with N hierarchical levels, where level 1 is the innermost loop (shortest timestep) and level N is the outermost loop (largest timestep). The loop factor arguments specify what the looping factor is between levels. N1 specifies the number of iterations of level 1 for a single iteration of level 2, N2 is the iterations of level 2 per iteration of level 3, etc. N-1 looping parameters must be specified.

Thus with a 4-level respa setting of "2 2 2" for the 3 loop factors, you could choose to have bond interactions computed 8x per large timestep, angle interactions computed 4x, pair interactions computed 2x, and long-range interactions once per large timestep.

The *timestep* command sets the large timestep for the outermost rRESPA level. Thus if the 3 loop factors are "2 2 2" for 4-level rRESPA, and the outer timestep is set to 4.0 fs, then the inner timestep would be 8x smaller or 0.5 fs. All other LAMMPS commands that specify number of timesteps (e.g. *thermo* for thermo output every N steps, *neigh_modify delay/every* parameters, *dump* every N steps, etc) refer to the outermost timesteps.

The rRESPA keywords enable you to specify at what level of the hierarchy various forces will be computed. If not specified, the defaults are that bond forces are computed at level 1 (innermost loop), angle forces are computed where bond forces are, dihedral forces are computed where angle forces are, improper forces are computed where dihedral forces are, pair forces are computed at the outermost level, and kspace forces are computed where pair forces are. The inner, middle, outer forces have no defaults.

For fixes that support it, the rRESPA level at which a given fix is active, can be selected through the *fix_modify* command.

The *inner* and *middle* keywords take additional arguments for cutoffs that are used by the pairwise force computations. If the 2 cutoffs for *inner* are 5.0 and 6.0, this means that all pairs up to 6.0 apart are computed by the inner force. Those between 5.0 and 6.0 have their force go ramped to 0.0 so the overlap with the next regime (middle or outer) is smooth. The next regime (middle or outer) will compute forces for all pairs from 5.0 outward, with those from 5.0 to 6.0 having their value ramped in an inverse manner.

Note that you can use *inner* and *outer* without using *middle* to split the pairwise computations into two portions instead of three. Unless you are using a very long pairwise cutoff, a 2-way split is often faster than a 3-way split, since it avoids too much duplicate computation of pairwise interactions near the intermediate cutoffs.

Also note that only a few pair potentials support the use of the *inner* and *middle* and *outer* keywords. If not, only the *pair* keyword can be used with that pair style, meaning all pairwise forces are computed at the same rRESPA level. See the doc pages for individual pair styles for details.

Another option for using pair potentials with rRESPA is with the *hybrid* keyword, which requires the use of the *pair_style hybrid or hybrid/overlay* command. In this scenario, different sub-styles of the hybrid pair style are evaluated at different rRESPA levels. This can be useful, for example, to set different timesteps for hybrid coarse-grained/all-atom models. The *hybrid* keyword requires as many level assignments as there are hybrid sub-styles, which assigns each sub-style to a rRESPA level, following their order of definition in the pair_style command. Since the *hybrid* keyword operates on pair style computations, it is mutually exclusive with either the *pair* or the *inner*/*middle*/*outer* keywords.

When using rRESPA (or for any MD simulation) care must be taken to choose a timestep size(s) that ensures the Hamiltonian for the chosen ensemble is conserved. For the constant NVE ensemble, total energy must be conserved. Unfortunately, it is difficult to know *a priori* how well energy will be conserved, and a fairly long test simulation (~10 ps) is usually necessary in order to verify that no long-term drift in energy occurs with the trial set of parameters.

With that caveat, a few rules-of-thumb may be useful in selecting *respa* settings. The following applies mostly to biomolecular simulations using the CHARMM or a similar all-atom force field, but the concepts are adaptable to other problems. Without SHAKE, bonds involving hydrogen atoms exhibit high-frequency vibrations and require a timestep on the order of 0.5 fs in order to conserve energy. The relatively inexpensive force computations for the bonds, angles, impropers, and dihedrals can be computed on this innermost 0.5 fs step. The outermost timestep cannot be greater than 4.0 fs without risking energy drift. Smooth switching of forces between the levels of the rRESPA hierarchy is also necessary to avoid drift, and a 1-2 Angstrom "healing distance" (the distance between the outer and inner cutoffs) works reasonably well. We thus recommend the following settings for use of the *respa* style without SHAKE in biomolecular simulations:

```
timestep  4.0
run_style respa 4 2 2 2 inner 2 4.5 6.0 middle 3 8.0 10.0 outer 4
```

With these settings, users can expect good energy conservation and roughly a 2.5 fold speedup over the *verlet* style with a 0.5 fs timestep.

If SHAKE is used with the *respa* style, time reversibility is lost, but substantially longer time steps can be achieved. For biomolecular simulations using the CHARMM or similar all-atom force field, bonds involving hydrogen atoms exhibit high frequency vibrations and require a time step on the order of 0.5 fs in order to conserve energy. These high frequency modes also limit the outer time step sizes since the modes are coupled. It is therefore desirable to use SHAKE with respa in order to freeze out these high frequency motions and increase the size of the time steps in the respa hierarchy. The following settings can be used for biomolecular simulations with SHAKE and rRESPA:

```
fix             2 all shake 0.000001 500 0 m 1.0 a 1
timestep        4.0
run_style       respa 2 2 inner 1 4.0 5.0 outer 2
```

With these settings, users can expect good energy conservation and roughly a 1.5 fold speedup over the *verlet* style with SHAKE and a 2.0 fs timestep.

For non-biomolecular simulations, the *respa* style can be advantageous if there is a clear separation of time scales - fast and slow modes in the simulation. For example, a system of slowly-moving charged polymer chains could be setup as follows:

```
timestep 4.0
run_style respa 2 8
```

This is two-level rRESPA with an 8x difference between the short and long timesteps. The bonds, angles, dihedrals will be computed every 0.5 fs (assuming real units), while the pair and kspace interactions will be computed once every 4 fs. These are the default settings for each kind of interaction, so no additional keywords are necessary.

Even a LJ system can benefit from rRESPA if the interactions are divided by the inner, middle and outer keywords. A 2-fold or more speedup can be obtained while maintaining good energy conservation. In real units, for a pure LJ fluid at liquid density, with a sigma of 3.0 Angstroms, and epsilon of 0.1 kcal/mol, the following settings seem to work well:

```
timestep  36.0
run_style respa 3 3 4 inner 1 3.0 4.0 middle 2 6.0 7.0 outer 3
```

---

The *respa/omp* style is a variant of *respa* adapted for use with pair, bond, angle, dihedral, improper, or kspace styles with an *omp* suffix. It is functionally equivalent to *respa* but performs additional operations required for managing *omp* styles. For more on *omp* styles see the *Speed omp* doc page. Accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

You can specify *respa/omp* explicitly in your input script, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 1.96.4 Restrictions

The *verlet/split* style can only be used if LAMMPS was built with the REPLICA package. Correspondingly the *respa/omp* style is available only if the OPENMP package was included. See the *Build package* page for more info.

Run style *verlet/split* is not compatible with kspace styles from the INTEL package and it is not compatible with any tip4p, dipole, or spin kspace styles.

Whenever using rRESPA, the user should experiment with trade-offs in speed and accuracy for their system, and verify that they are conserving energy to adequate precision.

### 1.96.5 Related commands

*timestep*, *run*

### 1.96.6 Default

```
run_style verlet
```

For run_style respa, the default assignment of interactions to rRESPA levels is as follows:

- bond forces = level 1 (innermost loop)
- angle forces = same level as bond forces
- dihedral forces = same level as angle forces
- improper forces = same level as dihedral forces
- pair forces = level N (outermost level)
- kspace forces = same level as pair forces
- inner, middle, outer forces = no default

---

**(Dandurand)** Dandurand, Vandierendonck, de Supinski, 39th IEEE IPDPS, June 3-7, (2025).

**(Tuckerman)** Tuckerman, Berne and Martyna, J Chem Phys, 97, p 1990 (1992).

## 1.97 set command

### 1.97.1 Syntax

```
set style ID keyword values ...
```

- style = *atom* or *type* or *mol* or *group* or *region*
- ID = depends on style

```
for style = atom, ID = a range of atom IDs
for style = type, ID = a range of numeric types or a single type label
for style = mol, ID = a range of molecule IDs
for style = group, ID = a group ID
for style = region, ID = a region ID
```

- one or more keyword/value pairs may be appended
- keyword = *angle* or *angmom* or *apip/lambda* or *block* or *bond* or *cc* or *charge* or *density* or *density/disc* or *diameter* or *dihedral* or *dipole* or *dipole/random* or *dpd/theta* or *edpd/cv* or *edpd/temp* or *epsilon* or *image* or *improper* or *length* or *mass* or *mol* or *omega* or *quat* or *quat/random* or *radius/electron* or *rheo/rho* or *rheo/status* or *shape* or *smd/contact/radius* or *smd/mass/density* or *sph/cv* or *sph/e* or *sph/rho* or *spin/atom* or *spin/atom/random* or *spin/electron* or *temperature* or *theta* or *theta/random* or *tri* or *type* or *type/fraction* or *type/ratio* or *type/subset* or *volume* or *vx* or *vy* or *vz* or *x* or *y* or *z* or *i_name* or *d_name* or *i2_name* or *d2_name*

```
  angle value = numeric angle type or angle type label, for all angles between␣
  ↪selected atoms
  angmom values = Lx Ly Lz
    Lx,Ly,Lz = components of angular momentum vector (distance-mass-velocity units)
    any of Lx,Ly,Lz can be an atom-style variable (see below)
  apip/lambda value = fast or precise or float
    fast = switching parameter of fast potential (1)
    precise = switching parameter of fast potential (0)
    float = constant float or atom-style variable (between 0 and 1)
  block value = block1, block2
```

```
  block1,block2 = 2 blockiness parameters for superellipsoids
bond value = numeric bond type or bond type label, for all bonds between selected␣
↪atoms
cc values = index cc
  index = index of a chemical species (1 to Nspecies)
  cc = chemical concentration of tDPD particles for a species (mole/volume units)
  cc can be an atom-style variable (see below)
charge value = atomic charge (charge units)
  value can be an atom-style variable (see below)
density value = particle density for a sphere or ellipsoid (mass/distance^3 units),␣
↪or for a triangle (mass/distance^2 units) or line (mass/distance units) particle
  value can be an atom-style variable (see below)
density/disc value = particle density for a 2d disc or ellipse (mass/distance^2␣
↪units)
  value can be an atom-style variable (see below)
diameter value = diameter of spherical particle (distance units)
  value can be an atom-style variable (see below)
dihedral value = numeric dihedral type or dihedral type label, for all dihedrals␣
↪between selected atoms
dipole values = x y z
  x,y,z = orientation of dipole moment vector
  any of x,y,z can be an atom-style variable (see below)
dipole/random values = seed Dlen
  seed = random # seed (positive integer) for dipole moment orientations
  Dlen = magnitude of dipole moment (dipole units)
dpd/theta value = internal temperature of DPD particles (temperature units)
  value can be an atom-style variable (see below)
  value can be NULL which sets internal temp of each particle to KE temp
edpd/cv value = volumetric heat capacity of eDPD particles (energy/temperature/
↪volume units)
  value can be an atom-style variable (see below)
edpd/temp value = temperature of eDPD particles (temperature units)
  value can be an atom-style variable (see below)
epsilon value = dielectric constant of the medium where the atoms reside
  value can be an atom-style variable (see below)
image values = nx ny nz
  nx,ny,nz = which periodic image of the simulation box the atom is in
  any of nx,ny,nz can be an atom-style variable (see below)
improper value = numeric improper type or improper type label, for all impropers␣
↪between selected atoms
length value = len
  len = length of line segment (distance units)
  len can be an atom-style variable (see below)
mass value = per-atom mass (mass units)
  value can be an atom-style variable (see below)
mol value = molecule ID
  the moleclue ID can be an atom-style variable (see below)
omega values = Wx Wy Wz
  Wx,Wy,Wz = components of angular velocity vector (radians/time units)
  any of Wx,Wy,Wz can be an atom-style variable (see below)
quat values = a b c theta
  a,b,c = unit vector to rotate particle around via right-hand rule
  theta = rotation angle (degrees)
  any of a,b,c,theta values can be an atom-style variable (see below)
```

```
quat/random value = seed
  seed = random # seed (positive integer) for quaternion orientations
radius/electron value = eradius
  eradius = electron radius (or fixed-core radius) (distance units)
  value can be an atom-style variable (see below)
rheo/rho value = density of RHEO particles (mass/distance^3)
  value can be an atom-style variable (see below)
rheo/status value = status or phase of RHEO particles (unitless)
  value can be an atom-style variable (see below)
shape values = Sx Sy Sz
  Sx,Sy,Sz = 3 diameters of ellipsoid (distance units)
  any of Sx,Sy,Sz can be an atom-style variable (see below)
smd/contact/radius value = radius for short range interactions, i.e. contact and␣
↪friction
  value can be an atom-style variable (see below)
smd/mass/density value = set particle mass based on volume by providing a mass␣
↪density
  value can be an atom-style variable (see below)
sph/cv value = heat capacity of SPH particles (need units)
  value can be an atom-style variable (see below)
sph/e value = energy of SPH particles (need units)
  value can be an atom-style variable (see below)
sph/rho value = density of SPH particles (need units)
  value can be an atom-style variable (see below)
spin/atom values = g x y z
  g = magnitude of magnetic spin vector (in Bohr magneton's unit)
  x,y,z = orientation of magnetic spin vector
  any of x,y,z can be an atom-style variable (see below)
spin/atom/random values = seed Dlen
  seed = random # seed (positive integer) for magnetic spin orientations
  Dlen = magnitude of magnetic spin vector (in Bohr magneton's unit)
spin/electron value = espin
  espin = electron spin (+1/-1), 0 = nuclei, 2 = fixed-core, 3 = pseudo-cores (i.e.␣
↪ECP)
  value can be an atom-style variable (see below)
temperature value = temperature for finite-size particles (temperature units)
  value can be an atom-style variable (see below)
theta value = angle (degrees)
  angle = orientation of line segment with respect to x-axis
  value can be an atom-style variable (see below)
theta/random value = seed
  seed = random # seed (positive integer) for line segment orienations
tri value = side
  side = side length of equilateral triangle (distance units)
  value can be an atom-style variable (see below)
type value = numeric atom type or type label
  value can be an atom-style variable (see below)
type/fraction values = type fraction seed
  type = numeric atom type or type label
  fraction = approximate fraction of selected atoms to set to new atom type
  seed = random # seed (positive integer)
type/ratio values = type fraction seed
  type = numeric atom type or type label
  fraction = exact fraction of selected atoms to set to new atom type
```

```
  seed = random # seed (positive integer)
type/subset values = type Nsubset seed
  type = numeric atom type or type label
  Nsubset = exact number of selected atoms to set to new atom type
  seed = random # seed (positive integer)
volume value = particle volume for Peridynamic particle (distance^3 units)
  value can be an atom-style variable (see below)
vx,vy,vz value = atom velocity (velocity units)
  value can be an atom-style variable (see below)
x,y,z value = atom coordinate (distance units)
  value can be an atom-style variable (see below)
i_name value = custom integer vector with name
  value can be an atom-style variable (see below)
d_name value = custom floating-point vector with name
  value can be an atom-style variable (see below)
i2_name value = custom integer array with name
                  column specified as i2_name[N] where N is 1 to Ncol
  value can be an atom-style variable (see below)
d2_name value = custom floating-point array with name
                  column specified as d2_name[N] where N is 1 to Ncol
  value can be an atom-style variable (see below)
```

### 1.97.2 Examples

```
set group solvent type 2
set group solvent type C
set group solvent type/fraction 2 0.5 12393
set group solvent type/fraction C 0.5 12393
set group edge bond 4
set region half charge 0.5
set type 3 charge 0.5
set type H charge 0.5
set type 1*3 charge 0.5
set atom * charge v_atomfile
set atom 100*200 x 0.5 y 1.0
set atom 100 vx 0.0 vy 0.0 vz -1.0
set atom 200 shape 1.5 2.0 4.0 block 2.0 4.0
set atom 1492 type 3
set atom 1492 type H
set atom * i_myVal 5
set atom * d2_Sxyz[1] 6.4
```

### 1.97.3 Description

Set one or more properties of one or more atoms. Since atom properties are initially assigned by the *read_data*, *read_restart* or *create_atoms* commands, this command changes those assignments. This can be useful for overriding the default values assigned by the *create_atoms* command (e.g. charge = 0.0). It can be useful for altering pairwise and molecular force interactions, since force-field coefficients are defined in terms of types. It can be used to change the labeling of atoms by atom type or molecule ID when they are output in *dump* files. It can also be useful for debugging purposes; i.e. positioning an atom at a precise location to compute subsequent forces or energy.

Note that the *style* and *ID* arguments determine which atoms have their properties reset. The remaining keywords specify which properties to reset and what the new values are. Some strings like *type* or *mol* can be used as a style and/or a keyword.

The *fix set* command can be used with similar syntax to this command to reset atom properties once every *N* steps during a simulation using via atom-style variables.

---

This section describes how to select which atoms to change the properties of, via the *style* and *ID* arguments.

Changed in version 28Mar2023: Support for type labels was added for selecting atoms by type

The style *atom* selects all the atoms in a range of atom IDs.

The style *type* selects all the atoms in a range of types or type labels. The style *type* selects atoms in one of two ways. A range of numeric atom types can be specified. Or a single atom type label can be specified, e.g. "C". The style *mol* selects all the atoms in a range of molecule IDs.

In each of the range cases, the range can be specified as a single numeric value, or with a wildcard asterisk to specify a range of values. This takes the form "*" or "*n" or "n*" or "m*n". For example, for the style *type*, if N = the number of atom types, then an asterisk with no numeric values means all types from 1 to N. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to N (inclusive). A middle asterisk means all types from m to n (inclusive). For all the styles except *mol*, the lowest value for the wildcard is 1; for *mol* it is 0.

The style *group* selects all the atoms in the specified group. The style *region* selects all the atoms in the specified geometric region. See the *group* and *region* commands for details of how to specify a group or region.

---

The next section describes the keyword options for which properties to change, for the selected atoms.

Note that except where explicitly prohibited below, all of the keywords allow an *atom-style or atomfile-style variable* to be used as the specified value(s). If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated, and its resulting per-atom value used to determine the value assigned to each selected atom. Note that the per-atom value from the variable will be ignored for atoms that are not selected via the *style* and *ID* settings explained above. A simple way to use per-atom values from the variable to reset a property for all atoms is to use style *atom* with *ID* = "*"; this selects all atom IDs.

Atom-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. They can also include per-atom values, such as atom coordinates. Thus it is easy to specify a time-dependent or spatially-dependent set of per-atom values. As explained on the *variable* doc page, atomfile-style variables can be used in place of atom-style variables, and thus as arguments to the set command. Atomfile-style variables read their per-atoms values from a file.

**Note**

Atom-style and atomfile-style variables return floating point per-atom values. If the values are assigned to an integer variable, such as the molecule ID, then the floating point value is truncated to its integer portion, e.g. a value of 2.6 would become 2.

---

Changed in version 28Mar2023: Support for type labels was added for setting angle types

Keyword *angle* sets the angle type of all angles of selected atoms to the specified value. The value can be a numeric type from 1 to nangletypes. Or it can be a angle type label. See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used. All atoms in a particular angle must be selected atoms in order for the change to be made. The value of nangletypes was set by the *angle types* field in the header of the data file read by the *read_data* command. This keyword does NOT allow use of an atom-style variable.

Keyword *angmom* sets the angular momentum of selected atoms. The particles must be ellipsoids as defined by the *atom_style ellipsoid* command or triangles as defined by the *atom_style tri* command. The angular momentum vector of the particles is set to the 3 specified components.

Changed in version 28Mar2023: Support for type labels was added for setting bond types

Keyword *bond* sets the bond type of all bonds of selected atoms to the specified value. The value can be a numeric type from 1 to nbondtypes. Or it can be a bond type label. See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used. All atoms in a particular bond must be selected atoms in order for the change to be made. The value of nbondtypes was set by the *bond types* field in the header of the data file read by the *read_data* command. This keyword does NOT allow use of an atom-style variable.

Keyword *cc* sets the chemical concentration of a tDPD particle for a specified species as defined by the DPD-MESO package. Currently, only *atom_style tdpd* defines particles with this attribute. An integer for "index" selects a chemical species (1 to Nspecies) where Nspecies is set by the atom_style command. The value for the chemical concentration must be >= 0.0.

Keyword *charge* set the charge of all selected atoms. The *atom style* being used must support the use of atomic charge.

Keyword *density* or *density/disc* also sets the mass of all selected particles, but in a different way. The particles must have a per-atom mass attribute, as defined by the *atom_style* command. If the atom has a radius attribute (see *atom_style sphere*) and its radius is non-zero, its mass is set from the density and particle volume for 3d systems (the input density is assumed to be in mass/distance^3 units). For 2d, the default is for LAMMPS to model particles with a radius attribute as spheres. However, if the *density/disc* keyword is used, then they can be modeled as 2d discs (circles). Their mass is set from the density and particle area (the input density is assumed to be in mass/distance^2 units).

If the atom has a shape attribute (see *atom_style ellipsoid*) and its 3 shape parameters are non-zero, then its mass is set from the density and particle volume (the input density is assumed to be in mass/distance^3 units). The *density/disc* keyword has no effect; it does not (yet) treat 3d ellipsoids as 2d ellipses.

If the atom has a length attribute (see *atom_style line*) and its length is non-zero, then its mass is set from the density and line segment length (the input density is assumed to be in mass/distance units). If the atom has an area attribute (see *atom_style tri*) and its area is non-zero, then its mass is set from the density and triangle area (the input density is assumed to be in mass/distance^2 units).

If none of these cases are valid, then the mass is set to the density value directly (the input density is assumed to be in mass units).

Keyword *diameter* sets the size of the selected atoms. The particles must be finite-size spheres as defined by the *atom_style sphere* command. The diameter of a particle can be set to 0.0, which means they will be treated as point particles. Note that this command does not adjust the particle mass, even if it was defined with a density, e.g. via the *read_data* command.

Changed in version 28Mar2023: Support for type labels was added for setting dihedral types

Keyword *dihedral* sets the dihedral type of all dihedrals of selected atoms to the specified value. The value can be a numeric type from 1 to ndihedraltypes. Or it can be a dihedral type label. See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used. All atoms in a particular dihedral must be selected atoms in order for the change to be made. The value of ndihedraltypes was set by the *dihedral types* field in the header of the data file read by the *read_data* command. This keyword does NOT allow use of an atom-style variable.

Keyword *dipole* uses the specified x,y,z values as components of a vector to set as the orientation of the dipole moment vectors of the selected atoms. The magnitude of the dipole moment is set by the length of this orientation vector.

Keyword *dipole/random* randomizes the orientation of the dipole moment vectors for the selected atoms and sets the magnitude of each to the specified *Dlen* value. For 2d systems, the z component of the orientation is set to 0.0. Random numbers are used in such a way that the orientation of a particular atom is the same, regardless of how many processors are being used. This keyword does NOT allow use of an atom-style variable.

Keyword *dpd/theta* sets the internal temperature of a DPD particle as defined by the DPD-REACT package. If the specified value is a number it must be >= 0.0. If the specified value is NULL, then the kinetic temperature Tkin of each particle is computed as 3/2 k Tkin = KE = 1/2 m v^2 = 1/2 m (vx*vx+vy*vy+vz*vz). Each particle's internal temperature is set to Tkin. If the specified value is an atom-style variable, then the variable is evaluated for each particle.

If a value >= 0.0, the internal temperature is set to that value. If it is < 0.0, the computation of Tkin is performed and the internal temperature is set to that value.

Keywords *edpd/temp* and *edpd/cv* set the temperature and volumetric heat capacity of an eDPD particle as defined by the DPD-MESO package. Currently, only *atom_style edpd* defines particles with these attributes. The values for the temperature and heat capacity must be positive.

Keyword *epsilon* sets the dielectric constant of a particle to be that of the medium where the particle resides as defined by the DIELECTRIC package. Currently, only *atom_style dielectric* defines particles with this attribute. The value for the dielectric constant must be >= 0.0. Note that the set command with this keyword will rescale the particle charge accordingly so that the real charge (e.g., as read from a data file) stays intact. To change the real charges, one needs to use the set command with the *charge* keyword. Care must be taken to ensure that the real and scaled charges and the dielectric constants are consistent.

Keyword *image* sets which image of the simulation box the atom is considered to be in. An image of 0 means it is inside the box as defined. A value of 2 means add 2 box lengths to get the true value. A value of -1 means subtract 1 box length to get the true value. LAMMPS updates these flags as atoms cross periodic boundaries during the simulation. The flags can be output with atom snapshots via the *dump* command. If a value of NULL is specified for any of nx,ny,nz, then the current image value for that dimension is unchanged. For non-periodic dimensions only a value of 0 can be specified. This command can be useful after a system has been equilibrated and atoms have diffused one or more box lengths in various directions. This command can then reset the image values for atoms so that they are effectively inside the simulation box, e.g if a diffusion coefficient is about to be measured via the *compute msd* command. Care should be taken not to reset the image flags of two atoms in a bond to the same value if the bond straddles a periodic boundary (rather they should be different by +/- 1). This will not affect the dynamics of a simulation, but may mess up analysis of the trajectories if a LAMMPS diagnostic or your own analysis relies on the image flags to unwrap a molecule which straddles the periodic box.

Changed in version 28Mar2023: Support for type labels was added for setting improper types

Keyword *improper* sets the improper type of all impropers of selected atoms to the specified value. The value can be a numeric type from 1 to nimpropertypes. Or it can be a improper type label. See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used. All atoms in a particular improper must be selected atoms in order for the change to be made. The value of nimpropertypes was set by the *improper types* field in the header of the data file read by the *read_data* command. This keyword does NOT allow use of an atom-style variable.

Keyword *length* sets the length of selected atoms. The particles must be line segments as defined by the *atom_style line* command. If the specified value is non-zero the line segment is (re)set to a length = the specified value, centered around the particle position, with an orientation along the x-axis. If the specified value is 0.0, the particle will become a point particle. Note that this command does not adjust the particle mass, even if it was defined with a density, e.g. via the *read_data* command.

Keyword *mass* sets the mass of all selected particles. The particles must have a per-atom mass attribute, as defined by the *atom_style* command. See the "mass" command for how to set mass values on a per-type basis.

Keyword *mol* sets the molecule ID for all selected atoms. The *atom style* being used must support the use of molecule IDs.

Keyword *omega* sets the angular velocity of selected atoms. The particles must be spheres as defined by the *atom_style sphere* command. The angular velocity vector of the particles is set to the 3 specified components.

Keyword *quat* uses the specified values to create a quaternion (4-vector) that represents the orientation of the selected atoms. The particles must define a quaternion for their orientation (e.g. ellipsoids, triangles, body particles) as defined by the *atom_style* command. Note that particles defined by *atom_style ellipsoid* have 3 shape parameters. The 3 values must be non-zero for each particle set by this command. They are used to specify the aspect ratios of an ellipsoidal particle, which is oriented by default with its x-axis along the simulation box's x-axis, and similarly for y and z. If this body is rotated (via the right-hand rule) by an angle theta around a unit rotation vector (a,b,c), then the quaternion that represents its new orientation is given by (cos(theta/2), a*sin(theta/2), b*sin(theta/2), c*sin(theta/2)). The theta and a,b,c values are the arguments to the *quat* keyword. LAMMPS normalizes the quaternion in case (a,b,c) was not

specified as a unit vector. For 2d systems, the a,b,c values are ignored, since a rotation vector of (0,0,1) is the only valid choice.

Keyword *quat/random* randomizes the orientation of the quaternion for the selected atoms. The particles must define a quaternion for their orientation (e.g. ellipsoids, triangles, body particles) as defined by the *atom_style* command. Random numbers are used in such a way that the orientation of a particular atom is the same, regardless of how many processors are being used. For 2d systems, only orientations in the xy plane are generated. As with keyword *quat*, for ellipsoidal particles, the 3 shape values must be non-zero for each particle set by this command. This keyword does NOT allow use of an atom-style variable.

Added in version 15Sep2022.

Keyword *radius/electron* uses the specified value to set the radius of electrons or fixed cores.

Keywords *rheo/rho* and *rheo/status* set the density and the status of rheo particles. In particular, one can only set the phase in the status as described by the *RHEO howto page*.

Keyword *shape* sets the size and shape of the selected atoms. The particles must be ellipsoids as defined by the *atom_style ellipsoid* command. The *Sx*, *Sy*, *Sz* settings are the 3 diameters of the ellipsoid in each direction. All 3 can be set to the same value, which means the ellipsoid is effectively a sphere. They can also all be set to 0.0 which means the particle will be treated as a point particle. Note that this command does not adjust the particle mass, even if it was defined with a density, e.g. via the *read_data* command.

Keyword *smd/contact/radius* only applies to simulations with the Smooth Mach Dynamics package MACHDYN. Itsets an interaction radius for computing short-range interactions, e.g. repulsive forces to prevent different individual physical bodies from penetrating each other. Note that the SPH smoothing kernel diameter used for computing long range, nonlocal interactions, is set using the *diameter* keyword.

Added in version 30Mar2026.

Keyword *block* sets the blockiness of the selected atoms. The particles must be ellipsoids as defined by the *atom_style ellipsoid* command. This command is used to define superellipsoid particle shapes for use in granular simulations. The *block1*, *block2* settings are the 2 exponents of the superellipsoid in the vertical and horizontal directions. Vertical sections through the center are superellipses with squareness *block1* and horizontal sections are superellipses with squareness *block2*. If both parameters are set to a value of 2 (the default), the atom is a regular ellipsoid. The keyword *block* should be used together with the keyword *shape* to give the particle the desired shape. If the keyword *block* is given alone, and the *shape* has not been defined, e.g., in a previous *set* command, the 3 diameters would be set to a value of 1 internally. Note that this command does not adjust the particle mass, even if it was defined with a density, e.g. via the *read_data* command.

Keyword *smd/mass/density* sets the mass of all selected particles, but it is only applicable to the Smooth Mach Dynamics package MACHDYN. It assumes that the particle volume has already been correctly set and calculates particle mass from the provided mass density value.

Keywords *sph/cv*, *sph/e*, and *sph/rho* set the heat capacity, energy, and density of smoothed particle hydrodynamics (SPH) particles. See this PDF guide to using SPH in LAMMPS.

**Note**

Note that the SPH PDF guide file has not been updated for many years and thus does not reflect the current *syntax* of the SPH package commands. For that, please refer to the LAMMPS manual.

Changed in version 15Sep2022.

Keyword *spin/atom* uses the specified g value to set the magnitude of the magnetic spin vectors, and the x,y,z values as components of a vector to set as the orientation of the magnetic spin vectors of the selected atoms. This keyword was previously called *spin*.

Changed in version 15Sep2022.

Keyword *spin/atom/random* randomizes the orientation of the magnetic spin vectors for the selected atoms and sets the magnitude of each to the specified *Dlen* value. This keyword does NOT allow use of an atom-style variable. This keyword was previously called *spin/random*.

Added in version 15Sep2022.

Keyword *spin/electron* sets the spin of an electron (+/- 1) or indicates nuclei (=0), fixed-cores (=2), or pseudo-cores (= 3).

Keyword *temperature* sets the temperature of a finite-size particle. Currently, only the GRANULAR package supports this attribute. The temperature must be added using an instance of *fix property/atom* The values for the temperature must be positive.

Keyword *theta* sets the orientation of selected atoms. The particles must be line segments as defined by the *atom_style line* command. The specified value is used to set the orientation angle of the line segments with respect to the x axis.

Keyword *theta/random* randomizes the orientation of theta for the selected atoms. The particles must be line segments as defined by the *atom_style line* command. Random numbers are used in such a way that the orientation of a particular atom is the same, regardless of how many processors are being used. This keyword does NOT allow use of an atom-style variable.

Keyword *tri* sets the size of selected atoms. The particles must be triangles as defined by the *atom_style tri* command. If the specified value is non-zero the triangle is (re)set to be an equilateral triangle in the xy plane with side length = the specified value, with a centroid at the particle position, with its base parallel to the x axis, and the y-axis running from the center of the base to the top point of the triangle. If the specified value is 0.0, the particle will become a point particle. Note that this command does not adjust the particle mass, even if it was defined with a density, e.g. via the *read_data* command.

Changed in version 28Mar2023: Support for type labels was added for setting atom types

Keyword *type* sets the atom type for all selected atoms. A specified value can be either a numeric atom type or an atom type label. When using a numeric type, the specified value must be from 1 to ntypes, where ntypes was set by the *create_box* command or the *atom types* field in the header of the data file read by the *read_data* command. When using a type label it must have been defined previously. See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

Keyword *type/fraction* sets the atom type for a fraction of the selected atoms. The actual number of atoms changed is not guaranteed to be exactly the specified fraction (0 <= *fraction* <= 1), but should be statistically close. Random numbers are used in such a way that a particular atom is changed or not changed, regardless of how many processors are being used. This keyword does NOT allow use of an atom-style variable.

Keywords *type/ratio* and *type/subset* also set the atom type for a fraction of the selected atoms. The actual number of atoms changed will be exactly the requested number. For *type/ratio* the specified fraction (0 <= *fraction* <= 1) determines the number. For *type/subset*, the specified *Nsubset* is the number. An iterative algorithm is used which ensures the correct number of atoms are selected, in a perfectly random fashion. Which atoms are selected will change with the number of processors used. These keywords do not allow use of an atom-style variable.

Keyword *volume* sets the volume of all selected particles. Currently, only the *atom_style peri* command defines particles with a volume attribute. Note that this command does not adjust the particle mass.

Keywords *vx*, *vy*, and *vz* set the velocities of all selected atoms.

Keywords *x*, *y*, *z* set the coordinates of all selected atoms.

Keyword *apip/lambda* sets the switching parameter of an adaptive-precision interatomic potential (*APIP*). The precise potential is used for an atom when its switching parameter $\lambda$ is 0. The fast potential is used for an atom when its switching parameter $\lambda$ is 1. Both potentials are partially used for $\lambda \in (0, 1)$.

Keywords *i_name*, *d_name*, *i2_name*, *d2_name* refer to custom per-atom integer and floating-point vectors or arrays that have been added via the *fix property/atom* command. When that command is used specific names are given to each

attribute which are the "name" portion of these keywords. For arrays *i2_name* and *d2_name*, the column of the array to set must also be included following the name in brackets: e.g. d2_xyz[2] or i2_mySpin[3].

### 1.97.4 Restrictions

You cannot set an atom attribute (e.g. *mol* or *q* or *volume*) if the *atom_style* does not have that attribute.

This command requires inter-processor communication to coordinate the setting of bond types (angle types, etc). This means that the system must be ready to perform a simulation before using one of these keywords (force fields set, atom mass set, etc). This is not necessary for other keywords.

Using the *region* style with the bond (angle, etc) keywords can give unpredictable results if there are bonds (angles, etc) that straddle periodic boundaries. This is because the region may only extend up to the boundary and partner atoms in the bond (angle, etc) may have coordinates outside the simulation box if they are ghost atoms.

### 1.97.5 Related commands

*create_box*, *create_atoms*, *read_data*, *fix set*

### 1.97.6 Default

none

## 1.98 shell command

### 1.98.1 Syntax

```
shell command args
```

- command = *cd* or *mkdir* or *mv* or *rm* or *rmdir* or *putenv* or arbitrary command

```
cd arg = dir
  dir = directory to change to
mkdir args = dir1 dir2 ...
  dir1,dir2 = one or more directories to create
mv args = old new
  old = old filename
  new = new filename or destination folder
rm args = [-f] file1 file2 ...
  -f = turn off warnings (optional)
  file1,file2 = one or more filenames to delete
rmdir args = dir1 dir2 ...
  dir1,dir2 = one or more directories to delete
putenv args = var1=value1 var2=value2
  var=value = one of more definitions of environment variables
anything else is passed as a command to the shell for direct execution
```

### 1.98.2 Examples

```
shell cd sub1
shell cd ..
shell mkdir tmp1 tmp2/tmp3
shell rmdir tmp1 tmp2
shell mv log.lammps hold/log.1
```

(continues on next page)

(continued from previous page)

```
shell rm TMP/file1 TMP/file2
shell putenv LAMMPS_POTENTIALS=../../potentials
shell my_setup file1 10 file2
shell my_post_process 100 dump.out
```

### 1.98.3 Description

Execute a shell command. A few simple file-based shell commands are supported directly, in Unix-style syntax. Any command not listed above is passed as-is to the C-library system() call, which invokes the command in a shell. To use the external executable instead of the built-in version one needs to use a full path, for example */bin/rm* instead of *rm*. The built-in commands will also work on operating systems, that do not - by default - provide the corresponding external executables (like *mkdir* on Windows).

This command provides a ways to invoke custom commands or executables from your input script. For example, you can move files around in preparation for the next section of the input script. Or you can run a program that pre-processes data for input into LAMMPS. Or you can run a program that post-processes LAMMPS output data.

With the exception of *cd*, all commands, including ones invoked via a system() call, are executed by only a single processor, so that files/directories are not being manipulated by multiple processors concurrently which may result in unexpected errors or corrupted files.

The *cd* command changes the current working directory similar to the `cd` command. All subsequent LAMMPS commands that read/write files will use the new directory. All processors execute this command.

The *mkdir* command creates directories similar to the Unix `mkdir -p` command. That is, it will attempt to create the entire path of subdirectories if they do not exist yet.

The *mv* command renames a file and/or moves it to a new directory. It cannot rename files across filesystem boundaries or between drives.

The *rm* command deletes file similar to the Unix `rm` command.

The *rmdir* command deletes directories similar to Unix `rmdir` command. If a directory is not empty, its contents are also removed recursively similar to the Unix `rm -r` command.

The *putenv* command defines or updates an environment variable directly. Since this command does not pass through the shell, no shell variable expansion or globbing is performed, only the usual substitution for LAMMPS variables defined with the *variable* command is performed. The resulting string is then used literally.

Any other command is passed as-is to the shell along with its arguments as one string, invoked by the C-library system() call. For example, these lines in your input script:

```
variable n equal 10
variable foo string file2
shell my_setup file1 $n ${foo}
```

would be the same as invoking

```
my_setup file1 10 file2
```

from a command-line prompt. The executable program "my_setup" is run with 3 arguments: file1 10 file2.

### 1.98.4 Restrictions

LAMMPS will do a best effort to detect errors and print suitable warnings, but due to the nature of delegating commands to the C-library system() call, this is not always reliable.

### 1.98.5 Related commands

none

### 1.98.6 Default

none

## 1.99 special_bonds command

### 1.99.1 Syntax

```
special_bonds keyword values ...
```

- one or more keyword/value pairs may be appended
- keyword = *amber* or *charmm* or *dreiding* or *fene* or *lj/coul* or *lj* or *coul* or *angle* or *dihedral* or *one/five*

```
amber values = none
charmm values = none
dreiding values = none
fene values = none
lj/coul values = w1,w2,w3
  w1,w2,w3 = weights (0.0 to 1.0) on pairwise Lennard-Jones and Coulombic␣
→interactions
lj values = w1,w2,w3
  w1,w2,w3 = weights (0.0 to 1.0) on pairwise Lennard-Jones interactions
coul values = w1,w2,w3
  w1,w2,w3 = weights (0.0 to 1.0) on pairwise Coulombic interactions
angle value = yes or no
dihedral value = yes or no
one/five value = yes or no
```

### 1.99.2 Examples

```
special_bonds amber
special_bonds charmm
special_bonds fene dihedral no
special_bonds lj/coul 0.0 0.0 0.5 angle yes dihedral yes
special_bonds lj 0.0 0.0 0.5 coul 0.0 0.0 0.0 dihedral yes
```

### 1.99.3 Description

Set weighting coefficients for pairwise energy and force contributions between pairs of atoms that are also permanently bonded to each other, either directly or via one or two intermediate bonds. These weighting factors are used by nearly all *pair styles* in LAMMPS that compute simple pairwise interactions. Permanent bonds between atoms are specified by defining the bond topology in the data file read by the *read_data* command. Typically a *bond_style* command is also used to define a bond potential. The rationale for using these weighting factors is that the interaction between a pair of bonded atoms is all (or mostly) specified by the bond, angle, dihedral potentials, and thus the non-bonded Lennard-Jones or Coulombic interaction between the pair of atoms should be excluded (or reduced by a weighting factor).

**Note**

These weighting factors are NOT used by *pair styles* that compute many-body interactions, since the "bonds" that result from such interactions are not permanent, but are created and broken dynamically as atom conformations change. Examples of pair styles in this category are EAM, MEAM, Stillinger-Weber, Tersoff, COMB, AIREBO, and ReaxFF. In fact, it generally makes no sense to define permanent bonds between atoms that interact via these potentials, though such bonds may exist elsewhere in your system, e.g. when using the *pair_style hybrid* command. Thus LAMMPS ignores special_bonds settings when many-body potentials are calculated. Please note, that the existence of explicit bonds for atoms that are described by a many-body potential will alter the neighbor list and thus can render the computation of those interactions invalid, since those pairs are not only used to determine direct pairwise interactions but also neighbors of neighbors and more. The recommended course of action is to remove such bonds, or - if that is not possible - use a special bonds setting of 1.0 1.0 1.0.

**Note**

Unlike some commands in LAMMPS, you cannot use this command multiple times in an incremental fashion: e.g. to first set the LJ settings and then the Coulombic ones. Each time you use this command it sets all the coefficients to default values and only overrides the one you specify, so you should set all the options you need each time you use it. See more details at the bottom of this page.

The Coulomb factors are applied to any Coulomb (charge interaction) term that the potential calculates. The LJ factors are applied to the remaining terms that the potential calculates, whether they represent LJ interactions or not. The weighting factors are a scaling prefactor on the energy and force between the pair of atoms.

A value of 1.0 means include the full interaction without flagging the pair as a "special pair"; a value of 0.0 means exclude the pair completely from the neighbor list, except for pair styles that require a *kspace style* and pair styles *amoeba*, *hippo*, *thole*, *coul/exclude*, and pair styles that include "coul/dsf" or "coul/wolf".

**Note**

To include pairs that would otherwise be excluded (so they are included in the neighbor list for certain analysis compute styles), you can use a very small but non-zero value like 1.0e-100 instead of 0.0. Due to using floating-point math, the computed force, energy, and virial contributions from the pairs will be too small to cause differences.

The first of the 3 coefficients (LJ or Coulombic) is the weighting factor on 1-2 atom pairs, which are pairs of atoms directly bonded to each other. The second coefficient is the weighting factor on 1-3 atom pairs which are those separated by 2 bonds (e.g. the two H atoms in a water molecule). The third coefficient is the weighting factor on 1-4 atom pairs which are those separated by 3 bonds (e.g. the first and fourth atoms in a dihedral interaction). Thus if the 1-2 coefficient is set to 0.0, then the pairwise interaction is effectively turned off for all pairs of atoms bonded to each other. If it is set to 1.0, then that interaction will be at full strength.

**Note**

For purposes of computing weighted pairwise interactions, 1-3 and 1-4 interactions are not defined from the list of angles or dihedrals used by the simulation. Rather, they are inferred topologically from the set of bonds specified when the simulation is defined from a data or restart file (see *read_data* or *read_restart* commands). Thus the set of 1-2,1-3,1-4 interactions that the weights apply to is the same whether angle and dihedral potentials are computed or not, and remains the same even if bonds are constrained, or turned off, or removed during a simulation.

The two exceptions to this rule are (a) if the *angle* or *dihedral* keywords are set to *yes* (see below), or (b) if the

*delete_bonds* command is used with the *special* option that re-computes the 1-2,1-3,1-4 topologies after bonds are deleted; see the *delete_bonds* command for more details.

The *amber* keyword sets the 3 coefficients to 0.0, 0.0, 0.5 for LJ interactions and to 0.0, 0.0, 0.8333 for Coulombic interactions, which is the default for a commonly used version of the AMBER force field, where the last value is really 5/6. See *(Cornell)* for a description of the AMBER force field.

The *charmm* keyword sets the 3 coefficients to 0.0, 0.0, 0.0 for both LJ and Coulombic interactions, which is the default for a commonly used version of the CHARMM force field. Note that in pair styles *lj/charmm/coul/charmm* and *lj/charmm/coul/long* the 1-4 coefficients are defined explicitly, and these pairwise contributions are computed as part of the charmm dihedral style - see the *pair_coeff* and *dihedral_style* commands for more information. See *(MacKerell)* for a description of the CHARMM force field.

The *dreiding* keyword sets the 3 coefficients to 0.0, 0.0, 1.0 for both LJ and Coulombic interactions, which is the default for the Dreiding force field, as discussed in *(Mayo)*.

The *fene* keyword sets the 3 coefficients to 0.0, 1.0, 1.0 for both LJ and Coulombic interactions, which is consistent with a coarse-grained polymer model with *FENE bonds*. See *(Kremer)* for a description of FENE bonds.

The *lj/coul*, *lj*, and *coul* keywords allow the 3 coefficients to be set explicitly. The *lj/coul* keyword sets both the LJ and Coulombic coefficients to the same 3 values. The *lj* and *coul* keywords only set either the LJ or Coulombic coefficients. Use both of them if you wish to set the LJ coefficients to different values than the Coulombic coefficients.

The *angle* keyword allows the 1-3 weighting factor to be ignored for individual atom pairs if they are not listed as the first and last atoms in any angle defined in the simulation or as 1,3 or 2,4 atoms in any dihedral defined in the simulation. For example, imagine the 1-3 weighting factor is set to 0.5 and you have a linear molecule with 4 atoms and bonds as follows: 1-2-3-4. If your data file defines 1-2-3 as an angle, but does not define 2-3-4 as an angle or 1-2-3-4 as a dihedral, then the pairwise interaction between atoms 1 and 3 will always be weighted by 0.5, but different force fields use different rules for weighting the pairwise interaction between atoms 2 and 4. If the *angle* keyword is specified as *yes*, then the pairwise interaction between atoms 2 and 4 will be unaffected (full weighting of 1.0). If the *angle* keyword is specified as *no* which is the default, then the 2,4 interaction will also be weighted by 0.5.

The *dihedral* keyword allows the 1-4 weighting factor to be ignored for individual atom pairs if they are not listed as the first and last atoms in any dihedral defined in the simulation. For example, imagine the 1-4 weighting factor is set to 0.5 and you have a linear molecule with 5 atoms and bonds as follows: 1-2-3-4-5. If your data file defines 1-2-3-4 as a dihedral, but does not define 2-3-4-5 as a dihedral, then the pairwise interaction between atoms 1 and 4 will always be weighted by 0.5, but different force fields use different rules for weighting the pairwise interaction between atoms 2 and 5. If the *dihedral* keyword is specified as *yes*, then the pairwise interaction between atoms 2 and 5 will be unaffected (full weighting of 1.0). If the *dihedral* keyword is specified as *no* which is the default, then the 2,5 interaction will also be weighted by 0.5.

The *one/five* keyword enable calculation and storage of a list of 1-5 neighbors in the molecular topology for each atom. It is required by some pair styles, such as *pair_style amoeba* and *pair_style hippo*.

---

**Note**

LAMMPS stores and maintains a data structure with a list of the first, second, and third neighbors of each atom (within the bond topology of the system). If new bonds are created (or molecules added containing atoms with more special neighbors), the size of this list needs to grow. Note that adding a single bond always adds a new first neighbor but may also induce *many* new second and third neighbors, depending on the molecular topology of your system. Using the *extra/special/per/atom* keyword to either *read_data* or *create_box* reserves empty space in the list for this N additional first, second, or third neighbors to be added. If you do not do this, you may get an error when bonds (or molecules) are added.

---

**Note**

If you reuse this command in an input script, you should set all the options you need each time. This command cannot be used a second time incrementally. E.g. these two commands:

```
special_bonds lj 0.0 1.0 1.0
special_bonds coul 0.0 0.0 1.0
```

are not the same as

```
special_bonds lj 0.0 1.0 1.0 coul 0.0 0.0 1.0
```

In the first case you end up with (after the second command):

```
LJ: 0.0 0.0 0.0
Coul: 0.0 0.0 1.0
```

while only in the second case do you get the desired settings of:

```
LJ: 0.0 1.0 1.0
Coul: 0.0 0.0 1.0
```

This happens because the LJ (and Coul) settings are reset to their default values before modifying them, each time the *special_bonds* command is issued.

### 1.99.4 Restrictions

none

### 1.99.5 Related commands

*delete_bonds*, *fix bond/create*

### 1.99.6 Default

All 3 Lennard-Jones and 3 Coulombic weighting coefficients = 0.0, angle = no, dihedral = no.

---

**(Cornell)** Cornell, Cieplak, Bayly, Gould, Merz, Ferguson, Spellmeyer, Fox, Caldwell, Kollman, JACS 117, 5179-5197 (1995).

**(Kremer)** Kremer, Grest, J Chem Phys, 92, 5057 (1990).

**(MacKerell)** MacKerell, Bashford, Bellott, Dunbrack, Evanseck, Field, Fischer, Gao, Guo, Ha, et al, J Phys Chem, 102, 3586 (1998).

**(Mayo)** Mayo, Olfason, Goddard III, J Phys Chem, 94, 8897-8909 (1990).

## 1.100 suffix command

### 1.100.1 Syntax

```
suffix style args
```

- style = *off* or *on* or *gpu* or *intel* or *kk* or *omp* or *opt* or *hybrid*
- args = for hybrid style, default suffix to be used and alternative suffix

### 1.100.2 Examples

```
suffix off
suffix on
suffix gpu
suffix intel
suffix hybrid intel omp
suffix kk
```

### 1.100.3 Description

This command allows you to use variants of various styles if they exist. In that respect it operates the same as the *-suffix command-line switch*. It also has options to turn off or back on any suffix setting made via the command-line.

The specified style can be *gpu*, *intel*, *kk*, *omp*, *opt* or *hybrid*. These refer to optional packages that LAMMPS can be built with, as described on the *Build package* doc page. The "gpu" style corresponds to the GPU package, the "intel" style to the INTEL package, the "kk" style to the KOKKOS package, the "omp" style to the OPENMP package, and the "opt" style to the OPT package.

These are the variants these packages provide:

- GPU = a handful of pair styles and the PPPM kspace_style, optimized to run on one or more GPUs or multicore CPU/GPU nodes
- INTEL = a collection of pair styles and neighbor routines optimized to run in single, mixed, or double precision on CPUs.
- KOKKOS = a collection of atom, pair, and fix styles optimized to run using the Kokkos library on various kinds of hardware, including GPUs via CUDA and many-core chips via OpenMP or threading.
- OPENMP = a collection of pair, bond, angle, dihedral, improper, kspace, compute, and fix styles with support for OpenMP multi-threading
- OPT = a handful of pair styles, cache-optimized for faster CPU performance
- HYBRID = a combination of two packages can be specified (see below)

As an example, all of the packages provide a *pair_style lj/cut* variant, with style names lj/cut/opt, lj/cut/omp, lj/cut/gpu, lj/cut/intel, or lj/cut/kk. A variant styles can be specified explicitly in your input script, e.g. pair_style lj/cut/gpu. If the suffix command is used with the appropriate style, you do not need to modify your input script. The specified suffix (opt,omp,gpu,intel,kk) is automatically appended whenever your input script command creates a new *atom*, *pair*, *bond*, *angle*, *dihedral*, *improper*, *kspace*, *fix*, *compute*, or *run* style. If the variant version does not exist, the standard version is created.

For "hybrid", two packages are specified. The first is used whenever available. If a style with the first suffix is not available, the style with the suffix for the second package will be used if available. For example, "hybrid intel omp" will use styles from the INTEL package as a first choice and styles from the OPENMP package as a second choice if no INTEL variant is available.

If the specified style is *off*, then any previously specified suffix is temporarily disabled, whether it was specified by a command-line switch or a previous suffix command. If the specified style is *on*, a disabled suffix is turned back on. The use of these 2 commands lets your input script use a standard LAMMPS style (i.e. a non-accelerated variant), which can be useful for testing or benchmarking purposes. Of course this is also possible by not using any suffix commands, and explicitly appending or not appending the suffix to the relevant commands in your input script.

**Note**

The default *run_style* verlet is invoked prior to reading the input script and is therefore not affected by a suffix command in the input script. The KOKKOS package requires "run_style verlet/kk", so when using the KOKKOS package it is necessary to either use the command line "-sf kk" command or add an explicit "run_style verlet" command to the input script.

### 1.100.4 Restrictions

none

### 1.100.5 Related commands

*-suffix command-line switch*

### 1.100.6 Default

none

## 1.101 tad command

### 1.101.1 Syntax

```
tad N t_event T_lo T_hi delta tmax compute-ID keyword value ...
```

- N = # of timesteps to run (not including dephasing/quenching)
- t_event = timestep interval between event checks
- T_lo = temperature at which event times are desired
- T_hi = temperature at which MD simulation is performed
- delta = desired confidence level for stopping criterion
- tmax = reciprocal of lowest expected pre-exponential factor (time units)
- compute-ID = ID of the compute used for event detection
- zero or more keyword/value pairs may be appended
- keyword = *min* or *neb* or *neb_style* or *neb_step* or *neb_log*

```
min values = etol ftol maxiter maxeval
  etol = stopping tolerance for energy (energy units)
  ftol = stopping tolerance for force (force units)
  maxiter = max iterations of minimize
  maxeval = max number of force/energy evaluations
neb values = ftol N1 N2 Nevery
  etol = stopping tolerance for energy (energy units)
  ftol = stopping tolerance for force (force units)
  N1 = max # of iterations (timesteps) to run initial NEB
  N2 = max # of iterations (timesteps) to run barrier-climbing NEB
  Nevery = print NEB statistics every this many timesteps
neb_style value = quickmin or fire
neb_step value = dtneb
  dtneb = timestep for NEB damped dynamics minimization
```

*neb_log* value = file where NEB statistics are printed

### 1.101.2 Examples

```
tad 2000 50 1800 2300 0.01 0.01 event
tad 2000 50 1800 2300 0.01 0.01 event &
    min 1e-05 1e-05 100 100 &
    neb 0.0 0.01 200 200 20 &
    min_style cg &
    neb_style fire &
    neb_log log.neb
```

### 1.101.3 Description

Run a temperature accelerated dynamics (TAD) simulation. This method requires two or more partitions to perform NEB transition state searches.

TAD is described in *this paper* by Art Voter. It is a method that uses accelerated dynamics at an elevated temperature to generate results at a specified lower temperature. A good overview of accelerated dynamics methods (AMD) for such systems is given in *this review paper* from the same group. To quote from the review paper: "The dynamical evolution is characterized by vibrational excursions within a potential basin, punctuated by occasional transitions between basins. The transition probability is characterized by p(t) = k*exp(-kt) where k is the rate constant."

TAD is a suitable AMD method for infrequent-event systems, where in addition, the transition kinetics are well-approximated by harmonic transition state theory (hTST). In hTST, the temperature dependence of transition rates follows the Arrhenius relation. As a consequence a set of event times generated in a high-temperature simulation can be mapped to a set of much longer estimated times in the low-temperature system. However, because this mapping involves the energy barrier of the transition event, which is different for each event, the first event at the high temperature may not be the earliest event at the low temperature. TAD handles this by first generating a set of possible events from the current basin. After each event, the simulation is reflected backwards into the current basin. This is repeated until the stopping criterion is satisfied, at which point the event with the earliest low-temperature occurrence time is selected. The stopping criterion is that the confidence measure be greater than 1-*delta*. The confidence measure is the probability that no earlier low-temperature event will occur at some later time in the high-temperature simulation. hTST provides an lower bound for this probability, based on the user-specified minimum pre-exponential factor (reciprocal of *tmax*).

In order to estimate the energy barrier for each event, the TAD method invokes the *NEB* method. Each NEB replica runs on a partition of processors. The current NEB implementation in LAMMPS restricts you to having exactly one processor per replica. For more information, see the documentation for the *neb* command. In the current LAMMPS implementation of TAD, all the non-NEB TAD operations are performed on the first partition, while the other partitions remain idle. See the *Howto replica* doc page for further discussion of multi-replica simulations.

A TAD run has several stages, which are repeated each time an event is performed. The logic for a TAD run is as follows:

```
while (time remains):
  while (time < tstop):
    until (event occurs):
      run dynamics for t_event steps
      quench
    run neb calculation using all replicas
    compute tlo from energy barrier
    update earliest event
    update tstop
    reflect back into current basin
  execute earliest event
```

Before this outer loop begins, the initial potential energy basin is identified by quenching (an energy minimization, see below) the initial state and storing the resulting coordinates for reference.

Inside the inner loop, dynamics is run continuously according to whatever integrator has been specified by the user, stopping every *t_event* steps to check if a transition event has occurred. This check is performed by quenching the system and comparing the resulting atom coordinates to the coordinates from the previous basin.

A quench is an energy minimization and is performed by whichever algorithm has been defined by the *min_style* command; its default is the CG minimizer. The tolerances and limits for each quench can be set by the *min* keyword. Note that typically, you do not need to perform a highly-converged minimization to detect a transition event.

The event check is performed by a compute with the specified *compute-ID*. Currently there is only one compute that works with the TAD command, which is the *compute event/displace* command. Other event-checking computes may be added. *Compute event/displace* checks whether any atom in the compute group has moved further than a specified threshold distance. If so, an "event" has occurred.

The NEB calculation is similar to that invoked by the *neb* command, except that the final state is generated internally, instead of being read in from a file. The style of minimization performed by NEB is determined by the *neb_style* keyword and must be a damped dynamics minimizer. The tolerances and limits for each NEB calculation can be set by the *neb* keyword. As discussed on the *neb*, it is often advantageous to use a larger timestep for NEB than for normal dynamics. Since the size of the timestep set by the *timestep* command is used by TAD for performing dynamics, there is a *neb_step* keyword which can be used to set a larger timestep for each NEB calculation if desired.

---

A key aspect of the TAD method is setting the stopping criterion appropriately. If this criterion is too conservative, then many events must be generated before one is finally executed. Conversely, if this criterion is too aggressive, high-entropy high-barrier events will be over-sampled, while low-entropy low-barrier events will be under-sampled. If the lowest pre-exponential factor is known fairly accurately, then it can be used to estimate *tmax*, and the value of *delta* can be set to the desired confidence level e.g. *delta* = 0.05 corresponds to 95% confidence. However, for systems where the dynamics are not well characterized (the most common case), it will be necessary to experiment with the values of *delta* and *tmax* to get a good trade-off between accuracy and performance.

A second key aspect is the choice of *t_hi*. A larger value greatly increases the rate at which new events are generated. However, too large a value introduces errors due to anharmonicity (not accounted for within hTST). Once again, for any given system, experimentation is necessary to determine the best value of *t_hi*.

---

Five kinds of output can be generated during a TAD run: event statistics, NEB statistics, thermodynamic output by each replica, dump files, and restart files.

Event statistics are printed to the screen and master log.lammps file each time an event is executed. The quantities are the timestep, CPU time, global event number *N*, local event number *M*, event status, energy barrier, time margin, *t_lo* and *delt_lo*. The timestep is the usual LAMMPS timestep, which corresponds to the high-temperature time at which the event was detected, in units of timestep. The CPU time is the total processor time since the start of the TAD run. The global event number *N* is a counter that increments with each executed event. The local event number *M* is a counter that resets to zero upon entering each new basin. The event status is *E* when an event is executed, and is *D* for an event that is detected, while *DF* is for a detected event that is also the earliest (first) event at the low temperature.

The time margin is the ratio of the high temperature time in the current basin to the stopping time. This last number can be used to judge whether the stopping time is too short or too long (see above).

*t_lo* is the low-temperature event time when the current basin was entered, in units of timestep. del*t_lo* is the time of each detected event, measured relative to *t_lo*. *delt_lo* is equal to the high-temperature time since entering the current basin, scaled by an exponential factor that depends on the hi/lo temperature ratio and the energy barrier for that event.

On lines for executed events, with status *E*, the global event number is incremented by one, the local event number and time margin are reset to zero, while the global event number, energy barrier, and *delt_lo* match the last event with

status *DF* in the immediately preceding block of detected events. The low-temperature event time *t_lo* is incremented by *delt_lo*.

NEB statistics are written to the file specified by the *neb_log* keyword. If the keyword value is "none", then no NEB statistics are printed out. The statistics are written every *Nevery* timesteps. See the *neb* command for a full description of the NEB statistics. When invoked from TAD, NEB statistics are never printed to the screen.

Because the NEB calculation must run on multiple partitions, LAMMPS produces additional screen and log files for each partition, e.g. log.lammps.0, log.lammps.1, etc. For the TAD command, these contain the thermodynamic output of each NEB replica. In addition, the log file for the first partition, log.lammps.0, will contain thermodynamic output from short runs and minimizations corresponding to the dynamics and quench operations, as well as a line for each new detected event, as described above.

After the TAD command completes, timing statistics for the TAD run are printed in each replica's log file, giving a breakdown of how much CPU time was spent in each stage (NEB, dynamics, quenching, etc).

Any *dump files* defined in the input script will be written to during a TAD run at timesteps when an event is executed. This means the requested dump frequency in the *dump* command is ignored. There will be one dump file (per dump command) created for all partitions. The atom coordinates of the dump snapshot are those of the minimum energy configuration resulting from quenching following the executed event. The timesteps written into the dump files correspond to the timestep at which the event occurred and NOT the clock. A dump snapshot corresponding to the initial minimum state used for event detection is written to the dump file at the beginning of each TAD run.

If the *restart* command is used, a single restart file for all the partitions is generated, which allows a TAD run to be continued by a new input script in the usual manner. The restart file is generated after an event is executed. The restart file contains a snapshot of the system in the new quenched state, including the event number and the low-temperature time. The restart frequency specified in the *restart* command is interpreted differently when performing a TAD run. It does not mean the timestep interval between restart files. Instead it means an event interval for executed events. Thus a frequency of 1 means write a restart file every time an event is executed. A frequency of 10 means write a restart file every 10th executed event. When an input script reads a restart file from a previous TAD run, the new script can be run on a different number of replicas or processors.

Note that within a single state, the dynamics will typically temporarily continue beyond the event that is ultimately chosen, until the stopping criterion is satisfied. When the event is eventually executed, the timestep counter is reset to the value when the event was detected. Similarly, after each quench and NEB minimization, the timestep counter is reset to the value at the start of the minimization. This means that the timesteps listed in the replica log files do not always increase monotonically. However, the timestep values printed to the master log file, dump files, and restart files are always monotonically increasing.

---

### 1.101.4 Restrictions

This command can only be used if LAMMPS was built with the REPLICA package. See the *Build package* doc page for more info.

*N* setting must be integer multiple of *t_event*.

Runs restarted from restart files written during a TAD run will only produce identical results if the user-specified integrator supports exact restarts. So *fix nvt* will produce an exact restart, but *fix langevin* will not.

This command cannot be used when any fixes are defined that keep track of elapsed time to perform time-dependent operations. Examples include the "ave" fixes such as *fix ave/chunk*. Also *fix dt/reset* and *fix deposit*.

### 1.101.5 Related commands

*compute event/displace*, *min_modify*, *min_style*, *run_style*, *minimize*, *temper*, *neb*, *prd*

### 1.101.6 Default

The option defaults are *min* = 0.1 0.1 40 50, *neb* = 0.01 100 100 10, *neb_style* = *quickmin*, *neb_step* = the same timestep set by the *timestep* command, and *neb_log* = "none".

---

**(Voter2000)** Sorensen and Voter, J Chem Phys, 112, 9599 (2000)

**(Voter2002)** Voter, Montalenti, Germann, Annual Review of Materials Research 32, 321 (2002).

## 1.102 temper command

### 1.102.1 Syntax

```
temper Nsteps Nevery temp fix-ID seed1 seed2 index
```

- Nsteps = total # of timesteps to run
- Nevery = attempt a tempering swap every this many steps
- temp = initial temperature for this ensemble
- fix-ID = ID of the fix that will control temperature during the run
- seed1 = random # seed used to decide on adjacent temperature to partner with
- seed2 = random # seed for Boltzmann factor in Metropolis swap
- index = which temperature (0 to N-1) I am simulating (optional)

### 1.102.2 Examples

```
temper 100000 100 $t tempfix 0 58728
temper 40000 100 $t tempfix 0 32285 $w
```

### 1.102.3 Description

Run a parallel tempering or replica exchange simulation using multiple replicas (ensembles) of a system. Two or more replicas must be used.

Each replica runs on a partition of one or more MPI processes. Processor partitions are defined at run-time using the *-partition command-line switch*. Note that if you have MPI installed, you can run a multi-replica simulation with more replicas (partitions) than you have physical processors, e.g you can run a 10-replica simulation on one or two processors. You will simply not get the performance speed-up you would see with one or more physical processors per replica. See the *Howto replica* doc page for further discussion.

Each replica's temperature is controlled at a different value by a fix with *fix-ID* that controls temperature. Most thermostat fix styles (with and without included time integration) are supported. The command will print an error message and abort, if the chosen fix is unsupported. The desired temperature is specified by *temp*, which is typically a variable previously set in the input script, so that each partition is assigned a different temperature. See the *variable* command for more details. For example:

```
variable t world 300.0 310.0 320.0 330.0
fix myfix all nvt temp $t $t 100.0
temper 100000 100 $t myfix 3847 58382
```

would define 4 temperatures, and assign one of them to the thermostat used by each replica, and to the temper command.

As the tempering simulation runs for *Nsteps* timesteps, a temperature swap between adjacent ensembles will be attempted every *Nevery* timesteps. If *seed1* is 0, then the swap attempts will alternate between odd and even pairings. If *seed1* is non-zero then it is used as a seed in a random number generator to randomly choose an odd or even pairing each time. Each attempted swap of temperatures is either accepted or rejected based on a Boltzmann-weighted Metropolis criterion which uses *seed2* in the random number generator.

As a tempering run proceeds, multiple log files and screen output files are created, one per replica. By default these files are named `log.lammps.M` and `screen.M` where *M* is the replica number from 0 to *N*-1, with *N* = # of replicas. See the *-log and -screen command-line switches* for info on how to change these names.

The main screen and log file (`log.lammps`) will list information about which temperature is assigned to each replica at each thermodynamic output timestep. E.g. for a simulation with 16 replicas:

```
Running on 16 partitions of processors
Step T0 T1 T2 T3 T4 T5 T6 T7 T8 T9 T10 T11 T12 T13 T14 T15
0    0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
500  1 0 3 2 5 4 6 7 8 9 10 11 12 13 14 15
1000 2 0 4 1 5 3 6 7 8 9 10 11 12 14 13 15
1500 2 1 4 0 5 3 6 7 9 8 10 11 12 14 13 15
2000 2 1 3 0 6 4 5 7 10 8 9 11 12 14 13 15
2500 2 1 3 0 6 4 5 7 11 8 9 10 12 14 13 15
...
```

The column headings *T0* to *TN-1* mean which temperature is currently assigned to the replica 0 to *N*-1. Thus the columns represent replicas and the value in each column is its temperature (also numbered 0 to N-1). For example, a 0 in the fourth column (column *T3*, step 2500) means that the fourth replica is assigned temperature 0, i.e. the lowest temperature. You can verify this time sequence of temperature assignments for the *N*th replica by comparing the *N*th column of screen output to the thermodynamic data in the corresponding `log.lammps.N` or `screen.N` files as time proceeds.

You can have each replica create its own dump file in the following manner:

```
variable rep world 0 1 2 3 4 5 6 7
dump 1 all atom 1000 dump.temper.${rep}
```

**Note**

Each replica's dump file will contain a continuous trajectory for its atoms where the temperature varies over time as swaps take place involving that replica. If you want a series of dump files, each with snapshots (from all replicas) that are all at a single temperature, then you will need to post-process the dump files using the information from the log.lammps file. E.g. you could produce one dump file with snapshots at 300K (from all replicas), another with snapshots at 310K, etc. Note that these new dump files will not contain "continuous trajectories" for individual atoms, because two successive snapshots (in time) may be from different replicas. The reorder_remd_traj python script can do the reordering for you (and additionally also calculates configurational log-weights of trajectory snapshots in the canonical ensemble). The script can be found in the tools/replica directory while instructions on how to use it is available in doc/Tools (in brief) and as a README file in tools/replica (in detail).

The last argument *index* in the temper command is optional and is used when restarting a tempering run from a set of restart files (one for each replica) which had previously swapped to new temperatures. The *index* value (from 0 to N-1, where N is the # of replicas) identifies which temperature the replica was simulating on the timestep the restart files were written. Obviously, this argument must be a variable so that each partition has the correct value. Set the variable to the *N* values listed in the log file for the previous run for the replica temperatures at that timestep. For example if the log file listed the following for a simulation with 5 replicas:

```
500000 2 4 0 1 3
```

then a setting of

```
variable w world 2 4 0 1 3
```

would be used to restart the run with a tempering command like the example above with $w as the last argument.

---

### 1.102.4 Restrictions

This command can only be used if LAMMPS was built with the REPLICA package. See the *Build package* doc page for more info.

### 1.102.5 Related commands

*variable*, *prd*, *neb*

### 1.102.6 Default

none

## 1.103 temper/grem command

### 1.103.1 Syntax

```
temper/grem N M lambda fix-ID thermostat-ID seed1 seed2 index
```

- N = total # of timesteps to run
- M = attempt a tempering swap every this many steps
- lambda = initial lambda for this ensemble
- fix-ID = ID of *fix grem*
- thermostat-ID = ID of the thermostat that controls kinetic temperature
- seed1 = random # seed used to decide on adjacent temperature to partner with
- seed2 = random # seed for Boltzmann factor in Metropolis swap
- index = which temperature (0 to N-1) I am simulating (optional)

### 1.103.2 Examples

```
temper/grem 100000 1000 ${lambda} fxgREM fxnvt 0 58728
temper/grem 40000 100 ${lambda} fxgREM fxnpt 0 32285 ${walkers}
```

### 1.103.3 Description

Run a parallel tempering or replica exchange simulation in LAMMPS partition mode using multiple generalized replicas (ensembles) of a system defined by *fix grem*, which stands for the generalized replica exchange method (gREM) originally developed by *(Kim)*. It uses non-Boltzmann ensembles to sample over first order phase transitions. The is done by defining replicas with an enthalpy dependent effective temperature

Two or more replicas must be used. See the *temper* command for an explanation of how to run replicas on multiple partitions of one or more MPI processes.

This command is a modification of the *temper* command and has the same dependencies, restraints, and input variables which are discussed there in greater detail.

Instead of temperature, this command performs replica exchanges in lambda as per the generalized ensemble enforced by *fix grem*. The desired lambda is specified by *lambda*, which is typically a variable previously set in the input script, so that each partition is assigned a different temperature. See the *variable* command for more details. For example:

```
variable lambda world 400 420 440 460
fix fxnvt all nvt temp 300.0 300.0 100.0
fix fxgREM all grem ${lambda} -0.05 -50000 fxnvt
temper/grem 100000 100 ${lambda} fxgREM fxnvt 3847 58382
```

would define 4 lambdas with constant kinetic temperature but unique generalized temperature, and assign one of them to *fix grem* used by each replica, and to the grem command.

As the gREM simulation runs for *N* timesteps, a swap between adjacent ensembles will be attempted every *M* timesteps. If *seed1* is 0, then the swap attempts will alternate between odd and even pairings. If *seed1* is non-zero then it is used as a seed in a random number generator to randomly choose an odd or even pairing each time. Each attempted swap of temperatures is either accepted or rejected based on a Metropolis criterion, derived for gREM by *(Kim)*, which uses *seed2* in the random number generator.

File management works identical to the *temper* command. Dump files created by this fix contain continuous trajectories and require post-processing to obtain per-replica information.

The last argument *index* in the grem command is optional and is used when restarting a run from a set of restart files (one for each replica) which had previously swapped to new lambda. This is done using a variable. For example if the log file listed the following for a simulation with 5 replicas:

```
500000 2 4 0 1 3
```

then a setting of

```
variable walkers world 2 4 0 1 3
```

would be used to restart the run with a grem command like the example above with ${walkers} as the last argument. This functionality is identical to *temper*.

---

### 1.103.4 Restrictions

This command can only be used if LAMMPS was built with the REPLICA package. See the *Build package* doc page for more info.

This command must be used with *fix grem*.

### 1.103.5 Related commands

*fix grem*, *temper*, *variable*

### 1.103.6 Default

none

**(Kim)** Kim, Keyes, Straub, J Chem Phys, 132, 224107 (2010).

# 1.104 temper/npt command

## 1.104.1 Syntax

```
temper/npt  N M temp fix-ID seed1 seed2 pressure index
```

- N = total # of timesteps to run
- M = attempt a tempering swap every this many steps
- temp = initial temperature for this ensemble
- fix-ID = ID of the fix that will control temperature and pressure during the run
- seed1 = random # seed used to decide on adjacent temperature to partner with
- seed2 = random # seed for Boltzmann factor in Metropolis swap
- pressure = setpoint pressure for the ensemble
- index = which temperature (0 to N-1) I am simulating (optional)

## 1.104.2 Examples

```
temper/npt 100000 100 $t nptfix 0 58728 1
temper/npt 2500000 1000 300 nptfix  0 32285 $p
temper/npt 5000000 2000 $t nptfix 0 12523 1 $w
```

## 1.104.3 Description

Run a parallel tempering or replica exchange simulation using multiple replicas (ensembles) of a system in the isothermal-isobaric (NPT) ensemble. The command temper/npt works like *temper* but requires running replicas in the NPT ensemble instead of the canonical (NVT) ensemble and allows for pressure to be set in the ensembles. These multiple ensembles can run in parallel at different temperatures or different pressures. The acceptance criteria for temper/npt is specific to the NPT ensemble and can be found in references *(Okabe)* and *(Mori)*.

Apart from the difference in acceptance criteria and the specification of pressure, this command works much like the *temper* command. See the documentation on *temper* for information on how the parallel tempering is handled in general.

---

## 1.104.4 Restrictions

This command can only be used if LAMMPS was built with the REPLICA package. See the *Build package* page for more info.

This command should be used with a fix that maintains the isothermal-isobaric (NPT) ensemble.

## 1.104.5 Related commands

*temper*, *variable*, *fix_npt*

## 1.104.6 Default

none

**(Okabe)** T. Okabe, M. Kawata, Y. Okamoto, M. Masuhiro, Chem. Phys. Lett., 335, 435-439 (2001).

**(Mori)** Y. Mori, Y. Okamoto, J. Phys. Soc. Jpn., 7, 074003 (2010).

# 1.105 thermo command

## 1.105.1 Syntax

```
thermo N
```

- N = output thermodynamics every N timesteps
- N can be a variable (see below)

## 1.105.2 Examples

```
thermo 100
```

## 1.105.3 Description

Compute and print thermodynamic info (e.g. temperature, energy, pressure) on timesteps that are a multiple of N and at the beginning and end of a simulation. A value of 0 will only print thermodynamics at the beginning and end.

The content and format of what is printed is controlled by the *thermo_style* and *thermo_modify* commands.

Instead of a numeric value, N can be specified as an *equal-style variable*, which should be specified as v_name, where name is the variable name. In this case, the variable is evaluated at the beginning of a run to determine the next timestep at which thermodynamic info will be written out. On that timestep, the variable will be evaluated again to determine the next timestep, etc. Thus the variable should return timestep values. See the stagger() and logfreq() and stride() math functions for *equal-style variables*, as examples of useful functions to use in this context. Other similar math functions could easily be added as options for *equal-style variables*.

For example, the following commands will output thermodynamic info at timesteps 0, 10, 20, 30, 100, 200, 300, 1000, 2000, *etc*:

```
variable        s equal logfreq(10,3,10)
thermo          v_s
```

## 1.105.4 Restrictions

none

## 1.105.5 Related commands

*thermo_style*, *thermo_modify*

## 1.105.6 Default

```
thermo 0
```

# 1.106 thermo_modify command

## 1.106.1 Syntax

```
thermo_modify keyword value ...
```

- one or more keyword/value pairs may be listed

- keyword = *lost* or *lost/bond* or *warn* or *norm* or *flush* or *line* or *colname* or *format* or *temp* or *press* or *triclinic/general*

```
lost value = error or warn or ignore
lost/bond value = error or warn or ignore
warn value = ignore or reset or default or a number
norm value = yes or no
flush value = yes or no
line value = one or multi or yaml
colname values =  ID string, or auto or default
  string = new column header name
  ID = integer from 1 to N, or integer from -1 to -N, where N = # of quantities␣
↪being output
      or a thermo keyword or reference to compute, fix, property or variable.
format values = line string, int string, float string, ID string, or none
  string = C-style format string
  ID = integer from 1 to N, or integer from -1 to -N, where N = # of quantities␣
↪being output
      or an integer range such as 2*6 (negative values are not allowed)
      or a thermo keyword or reference to compute, fix, property or variable.
temp value = compute ID that calculates a temperature
press value = compute ID that calculates a pressure
triclinic/general arg = yes or no
```

## 1.106.2 Examples

```
thermo_modify lost ignore flush yes
thermo_modify temp myTemp format 3 %15.8g
thermo_modify temp myTemp format line "%ld %g %g %15.8g"
thermo_modify line multi format float %g
thermo_modify line yaml format none
thermo_modify colname 1 Timestep colname -2 Pressure colname f_1[1] AvgDensity
```

## 1.106.3 Description

Set options for how thermodynamic information is computed and printed by LAMMPS.

**Note**

These options apply to the *currently defined* thermo style. When you specify a *thermo_style* command, all thermodynamic settings are restored to their default values, including those previously reset by a thermo_modify command. Thus if your input script specifies a thermo_style command, you should use the thermo_modify command **after** it.

The *lost* keyword determines whether LAMMPS checks for lost atoms each time it computes thermodynamics and what it does if atoms are lost. An atom can be "lost" if it moves across a non-periodic simulation box *boundary* or if it moves more than a box length outside the simulation domain (or more than a processor subdomain length) before reneighboring occurs. The latter case is typically due to bad dynamics (e.g., too large a time step and/or huge forces and velocities). If the value is *ignore*, LAMMPS does not check for lost atoms. If the value is *error* or *warn*, LAMMPS checks and either issues an error or warning. The simulation will exit with an error and continue with a warning. A warning will only be issued once, the first time an atom is lost. This can be a useful debugging option.

The *lost/bond* keyword determines whether LAMMPS throws an error or not if an atom in a bonded interaction (bond, angle, etc) cannot be found when it creates bonded neighbor lists. By default this is a fatal error. However in some

scenarios it may be desirable to only issue a warning or ignore it and skip the computation of the missing bond, angle, etc. An example would be when gas molecules in a vapor are drifting out of the box through a fixed boundary condition (see the *boundary* command). In this case one atom may be deleted before the rest of the molecule is, on a later timestep.

The *warn* keyword allows you to control whether LAMMPS will print warning messages and how many of them. Most warning messages are only printed by MPI rank 0. They are usually pointing out important issues that should be investigated, but LAMMPS cannot determine for certain whether they are an indication of an error.

Some warning messages are printed during a run (or immediately before) each time a specific MPI rank encounters the issue (e.g., bonds that are stretched too far or dihedrals in extreme configurations). The number of these can quickly blow up the size of the log file and screen output. A limit of 100 warning messages is therefore applied by default. The warning count is applied to the entire input file, unless it is reset with a `thermo_modify warn reset` command. If there are more warnings than the limit, LAMMPS will print one final warning that it will not print any additional warning messages.

**Note**

The warning limit is enforced on either the per-processor count or the total count across all MPI processes. For efficiency reasons, however, the total count is only updated at steps with thermodynamic output. Thus when running on a large number of MPI processes in parallel, the total number of warnings printed can be significantly larger than the given limit.

Any number after the keyword *warn* will change the warning limit accordingly. With the value *ignore* all warnings will be suppressed, with the value *always* no limit will be applied and warnings will always be printed, with the value *reset* the internal warning counter will be reset to zero, and with the value *default*, the counter is reset and the limit set to 100. An example usage of either *reset* or *default* would be to re-enable warnings that were disabled or have reached the limit during equilibration, where the warnings would be acceptable while the system is still adjusting, but then change to all warnings for the production run, where they would indicate problems that would require a closer look at what is causing them.

The *norm* keyword determines whether various thermodynamic output values are normalized by the number of atoms or not, depending on whether it is set to *yes* or *no*. Different unit styles have different defaults for this setting (see below). Even if *norm* is set to *yes*, a value is only normalized if it is an "extensive" quantity, meaning that it scales with the number of atoms in the system. For the thermo keywords described by the page for the *thermo_style* command, all energy-related keywords are extensive, such as *pe* or *ebond* or *enthalpy*. Other keywords such as *temp* or *press* are "intensive" meaning their value is independent (in a statistical sense) of the number of atoms in the system and thus are never normalized. For thermodynamic output values extracted from fixes and computes in a *thermo_style custom* command, the page for the individual *fix* or *compute* lists whether the value is "extensive" or "intensive" and thus whether it is normalized. Thermodynamic output values calculated by a variable formula are assumed to be "intensive" and thus are never normalized. You can always include a divide by the number of atoms in the variable formula if this is not the case.

The *flush* keyword invokes a flush operation after thermodynamic info is written to the screen and log file. This ensures the output is updated and not buffered (by the application) even if LAMMPS halts before the simulation completes. Please note that this does not affect buffering by the OS or devices, so you may still lose data in case the simulation stops due to a hardware failure.

The *line* keyword determines whether thermodynamics will be output as a series of numeric values on one line ("one"), in a multi-line format with 3 quantities with text strings per line and a dashed-line header containing the timestep and CPU time ("multi"), or in a YAML format block ("yaml"). This modify option overrides the *one*, *multi*, or *yaml* thermo_style settings.

Added in version 4May2022.

The *colname* keyword can be used to change the default header keyword for a column or field of thermodynamic output. The column names can either be manually set by the user, or automatically generated for certain fixes and computes.

The setting for *ID string* replaces the default text with the provided string. *ID* can be a positive integer when it represents the column number counting from the left, a negative integer when it represents the column number from the right (i.e., $-1$ is the last column/keyword), or a thermo keyword (or compute, fix, property, or variable reference) and then it replaces the string for that specific thermo keyword.

Added in version 10Dec2025.

With a setting of *auto*, certain fixes or computes will generate more descriptive strings as their thermo keywords, which are described in the 'output' section of their documentation. Current commands that automatically generate descriptive thermo output strings include 'fix nvt', 'fix npt', 'fix nph', 'compute reduce', and 'fix bond/react'.

With a *colname* setting of *default*, the default column names are used. The default column names follow the same convention as the arguments provided to the *thermo_style* command.

The *colname* keyword can be used multiple times. If multiple *colname* settings refer to the same keyword, the last setting has precedence. The *auto* keyword will only override user-provided strings for computes and fixes that support automatically generated column names. A setting of *default* clears all previous settings, reverting all values to their default values.

The *format* keyword can be used to change the default numeric format of any of quantities the *thermo_style* command outputs. All the specified format strings are C-style formats (i.e., as used by the C/C++ printf() command). The *line* keyword takes a single argument which is the format string for the entire line of thermo output, with *N* fields, which you must enclose in quotes if it is more than one field. The *int* and *float* keywords take a single format argument and are applied to all integer or floating-point quantities output. The setting for *ID string* also takes a single format argument that is used for the indexed value in each line. The interpretation is the same as for *colname* (i.e., a positive integer is the n-th value corresponding to the n-th thermo keyword, a negative integer is counting backwards, and a string matches the entry with the thermo keyword). For example, the fifth column is output in high precision for "format 5 %20.15g", and the pair energy for "format epair %20.15g". The *ID* field can be a range, such as "3*6", "*", "2*", or "*3"; in such cases, all fields in the range (inclusive) are set to the specified format string. Ranges containing negative numbers are not supported.

The *format* keyword can be used multiple times. The precedence is that for each value in a line of output, the *ID* format (if specified) is used, else the *int* or *float* setting (if specified) is used, else the *line* setting (if specified) for that value is used, else the default setting is used. A setting of *none* clears all previous settings, reverting all values to their default format.

**Note**

The thermo output values *step* and *atoms* are stored internally as 8-byte signed integers, rather than the usual 4-byte signed integers. When specifying the *format int* option you can use a "%d"-style format identifier in the format string and LAMMPS will convert this to the corresponding 8-byte form when it is applied to those keywords. However, when specifying the *line* option or *format ID string* option for *step* and *natoms*, you should specify a format string appropriate for an 8-byte signed integer (i.e., one with "%ld" or "%lld", depending on the platform).

The *temp* keyword is used to determine how thermodynamic temperature is calculated, which is used by all thermo quantities that require a temperature ("temp", "press", "ke", "etotal", "enthalpy", "pxx", etc). The specified compute ID must have been previously defined by the user via the *compute* command and it must be a style of compute that calculates a temperature. As described in the *thermo_style* command, thermo output uses a default compute for temperature with ID = *thermo_temp*. This option allows the user to override the default.

The *press* keyword is used to determine how thermodynamic pressure is calculated, which is used by all thermo quantities that require a pressure ("press", "enthalpy", "pxx", etc). The specified compute ID must have been previously defined by the user via the *compute* command and it must be a style of compute that calculates a pressure. As described in the *thermo_style* command, thermo output uses a default compute for pressure with ID = *thermo_press*. This option allows the user to override the default.

**Note**

If both the *temp* and *press* keywords are used in a single thermo_modify command (or in two separate commands), then the order in which the keywords are specified is important. Note that a *pressure compute* defines its own temperature compute as an argument when it is specified. The *temp* keyword will override this (for the pressure compute being used by thermodynamics), but only if the *temp* keyword comes after the *press* keyword. If the *temp* keyword comes before the *press* keyword, then the new pressure compute specified by the *press* keyword will be unaffected by the *temp* setting.

The *triclinic/general* keyword can only be used with a value of *yes* if the simulation box was created as a general triclinic box. See the *Howto_triclinic* doc page for a detailed explanation of orthogonal, restricted triclinic, and general triclinic simulation boxes.

If this keyword is *yes*, the output of the simulation box edge vectors and the pressure tensor components for the system are affected. These are specified by the *avec,bvec,cvec* and *pxx,pyy,pzz,pxy,pxz,pyz* keywords of the *thermo_style* command. See the *thermo_style* doc page for details.

### 1.106.4 Restrictions

none

### 1.106.5 Related commands

*thermo*, *thermo_style*

### 1.106.6 Default

The option defaults are lost = error, warn = 100, norm = yes for unit style of *lj*, norm = no for unit style of *real* and *metal*, flush = no, temp/press = compute IDs defined by thermo_style, and triclinic/general = no.

The defaults for the line and format options depend on the thermo style. For styles "one" and "custom", the line and format defaults are "one", "%10d", and "%14.8g". For style "multi", the line and format defaults are "multi", "%14d", and "%14.4f". For style "yaml", the line and format defaults are "%d" and "%.15g".

## 1.107 thermo_style command

### 1.107.1 Syntax

```
thermo_style style args
```

- style = *one* or *multi* or *yaml* or *custom*
- args = list of arguments for a particular style

```
one args = none
multi args = none
yaml args = none
custom args = list of keywords
  possible keywords = step, elapsed, elaplong, dt, time,
                      cpu, tpcpu, spcpu, cpuuse, cpuremain, part, timeremain,
                      atoms, temp, press, pe, ke, etotal,
                      evdwl, ecoul, epair, ebond, eangle, edihed, eimp,
                      emol, elong, etail,
                      enthalpy, ecouple, econserve,
                      vol, density,
```

```
                    xlo, xhi, ylo, yhi, zlo, zhi,
                    xy, xz, yz,
                    avecx, avecy, avecz,
                    bvecx, bvecy, bvecz,
                    cvecx, cvecy, cvecz,
                    lx, ly, lz,
                    xlat, ylat, zlat,
                    cella, cellb, cellc, cellalpha, cellbeta, cellgamma,
                    pxx, pyy, pzz, pxy, pxz, pyz,
                    bonds, angles, dihedrals, impropers,
                    fmax, fnorm, nbuild, ndanger,
                    c_ID, c_ID[I], c_ID[I][J],
                    f_ID, f_ID[I], f_ID[I][J],
                    v_name, v_name[I]
step = timestep
elapsed = timesteps since start of this run
elaplong = timesteps since start of initial run in a series of runs
dt = timestep size
time = simulation time
cpu = elapsed CPU time in seconds since start of this run
tpcpu = time per CPU second
spcpu = timesteps per CPU second
cpuuse = CPU utilization in percent (can be > 100% with multi-threading)
cpuremain = estimated CPU time remaining in run
part = which partition (0 to Npartition-1) this is
timeremain = remaining time in seconds on timer timeout.
atoms = # of atoms
temp = temperature
press = pressure
pe = total potential energy
ke = kinetic energy
etotal = total energy (pe + ke)
evdwl = van der Waals pairwise energy (includes etail)
ecoul = Coulombic pairwise energy
epair = pairwise energy (evdwl + ecoul + elong)
ebond = bond energy
eangle = angle energy
edihed = dihedral energy
eimp = improper energy
emol = molecular energy (ebond + eangle + edihed + eimp)
elong = long-range kspace energy
etail = van der Waals energy long-range tail correction
enthalpy = enthalpy (etotal + press*vol)
ecouple = cumulative energy change due to thermo/baro statting fixes
econserve = pe + ke + ecouple = etotal + ecouple
vol = volume
density = mass density of system
xlo,xhi,ylo,yhi,zlo,zhi = box boundaries
xy,xz,yz = box tilt for restricted triclinic (non-orthogonal) simulation boxes
avecx,avecy,avecz = components of edge vector A of the simulation box
bvecx,bvecy,bvecz = components of edge vector B of the simulation box
cvecx,cvecy,cvecz = components of edge vector C of the simulation box
lx,ly,lz = box lengths in x,y,z
xlat,ylat,zlat = lattice spacings as calculated by lattice command
```

```
   cella,cellb,cellc = periodic cell lattice constants a,b,c
   cellalpha, cellbeta, cellgamma = periodic cell angles alpha,beta,gamma
   pxx,pyy,pzz,pxy,pxz,pyz = 6 components of pressure tensor
   bonds,angles,dihedrals,impropers = # of these interactions defined
   fmax = max component of force on any atom in any dimension
   fnorm = length of force vector for all atoms
   nbuild = # of neighbor list builds
   ndanger = # of dangerous neighbor list builds
   c_ID = global scalar value calculated by a compute with ID
   c_ID[I] = Ith component of global vector calculated by a compute with ID, I can␣
↪include wildcard (see below)
   c_ID[I][J] = I,J component of global array calculated by a compute with ID
   f_ID = global scalar value calculated by a fix with ID
   f_ID[I] = Ith component of global vector calculated by a fix with ID, I can␣
↪include wildcard (see below)
   f_ID[I][J] = I,J component of global array calculated by a fix with ID
   v_name = value calculated by an equal-style variable with name
   v_name[I] = value calculated by a vector-style variable with name, I can␣
↪include wildcard (see below)
```

### 1.107.2 Examples

```
thermo_style multi
thermo_style yaml
thermo_style one
thermo_style custom step temp pe etotal press vol
thermo_style custom step temp etotal c_myTemp v_abc
thermo_style custom step temp etotal c_myTemp[*] v_abc
```

### 1.107.3 Description

Set the style and content for printing thermodynamic data to the screen and log files. The units for each column of output corresponding to the list of keywords is determined by the *units* command for the simulation. E.g. energies will be in energy units, temperature in temperature units, pressure in pressure units.

Style *one* prints a single line of thermodynamic info that is the equivalent of "thermo_style custom step temp epair emol etotal press". The line contains only numeric values.

Style *multi* prints a multiple-line listing of thermodynamic info that is the equivalent of "thermo_style custom etotal ke temp pe ebond eangle edihed eimp evdwl ecoul elong press". The listing contains numeric values and a string ID for each quantity.

Added in version 24Mar2022.

Style *yaml* is similar to style *one* but prints the output in YAML format which can be easily read by a variety of script languages and data handling packages. Since LAMMPS may print other output before, after, or in between thermodynamic output, the YAML format content needs to be separated from the rest. All YAML format thermodynamic output can be matched with a regular expression and can thus be extracted with commands like `egrep` as follows:

```
egrep  '^(keywords:|data:$|---$|\.\.\.$|  - \[)' log.lammps > log.yaml
```

Information about processing such YAML files is in the *structured data output howto*.

Style *custom* is the most general setting and allows you to specify which of the keywords listed above you want printed on each thermodynamic timestep. Note that the keywords c_ID, f_ID, v_name are references to *computes*, *fixes*, and equal-style *variables* that have been defined elsewhere in the input script or can even be new styles which users have

added to LAMMPS. See the *Modify* page for details on the latter. Thus the *custom* style provides a flexible means of outputting essentially any desired quantity as a simulation proceeds.

All styles except *custom* have *vol* appended to their list of outputs if the simulation box volume changes during the simulation.

The values printed by the various keywords are instantaneous values, calculated on the current timestep. Time-averaged quantities, which include values from previous timesteps, can be output by using the f_ID keyword and accessing a fix that does time-averaging such as the *fix ave/time* command.

Options invoked by the *thermo_modify* command can be used to set the one- or multi-line format of the print-out, the normalization of thermodynamic output (total values versus per-atom values for extensive quantities (ones which scale with the number of atoms in the system), and the numeric precision of each printed value.

**Note**

When you use a "thermo_style" command, all thermodynamic settings are restored to their default values, including those previously set by a *thermo_modify* command. Thus if your input script specifies a thermo_style command, you should use the thermo_modify command after it.

---

Several of the thermodynamic quantities require a temperature to be computed: "temp", "press", "ke", "etotal", "enthalpy", "pxx", etc. By default this is done by using a *temperature* compute which is created when LAMMPS starts up, as if this command had been issued:

```
compute thermo_temp all temp
```

See the *compute temp* command for details. Note that the ID of this compute is *thermo_temp* and the group is *all*. You can change the attributes of this temperature (e.g. its degrees-of-freedom) via the *compute_modify* command. Alternatively, you can directly assign a new compute (that calculates temperature) which you have defined, to be used for calculating any thermodynamic quantity that requires a temperature. This is done via the *thermo_modify* command.

Several of the thermodynamic quantities require a pressure to be computed: "press", "enthalpy", "pxx", etc. By default this is done by using a *pressure* compute which is created when LAMMPS starts up, as if this command had been issued:

```
compute thermo_press all pressure thermo_temp
```

See the *compute pressure* command for details. Note that the ID of this compute is *thermo_press* and the group is *all*. You can change the attributes of this pressure via the *compute_modify* command. Alternatively, you can directly assign a new compute (that calculates pressure) which you have defined, to be used for calculating any thermodynamic quantity that requires a pressure. This is done via the *thermo_modify* command.

Several of the thermodynamic quantities require a potential energy to be computed: "pe", "etotal", "ebond", etc. This is done by using a *pe* compute which is created when LAMMPS starts up, as if this command had been issued:

```
compute thermo_pe all pe
```

See the *compute pe* command for details. Note that the ID of this compute is *thermo_pe* and the group is *all*. You can change the attributes of this potential energy via the *compute_modify* command.

---

The kinetic energy of the system *ke* is inferred from the temperature of the system with $\frac{1}{2}k_BT$ of energy for each degree of freedom. Thus, using different *compute commands* for calculating temperature, via the *thermo_modify temp* command, may yield different kinetic energies, since different computes that calculate temperature can subtract out different non-thermal components of velocity and/or include different degrees of freedom (translational, rotational, etc).

The potential energy of the system *pe* will include contributions from fixes if the *fix_modify energy yes* option is set for a fix that calculates such a contribution. For example, the *fix wall/lj93* fix calculates the energy of atoms interacting with the wall. See the doc pages for "individual fixes" to see which ones contribute and whether their default *fix_modify energy* setting is *yes* or *no*.

A long-range tail correction *etail* for the van der Waals pairwise energy will be non-zero only if the *pair_modify tail* option is turned on. The *etail* contribution is included in *evdwl*, *epair*, *pe*, and *etotal*, and the corresponding tail correction to the pressure is included in *press* and *pxx*, *pyy*, etc.

---

Here is more information on other keywords whose meaning may not be clear.

The *step*, *elapsed*, and *elaplong* keywords refer to timestep count. *Step* is the current timestep, or iteration count when a *minimization* is being performed. *Elapsed* is the number of timesteps elapsed since the beginning of this run. *Elaplong* is the number of timesteps elapsed since the beginning of an initial run in a series of runs. See the *start* and *stop* keywords for the *run* for info on how to invoke a series of runs that keep track of an initial starting time. If these keywords are not used, then *elapsed* and *elaplong* are the same value.

The *dt* keyword is the current timestep size in time *units*. The *time* keyword is the current elapsed simulation time, also in time *units*, which is simply (step*dt) if the timestep size has not changed and the timestep has not been reset. If the timestep has changed (e.g. via *fix dt/reset*) or the timestep has been reset (e.g. via the "reset_timestep" command), then the simulation time is effectively a cumulative value up to the current point.

The *cpu* keyword is elapsed CPU seconds since the beginning of this run. The *tpcpu* and *spcpu* keywords are measures of how fast your simulation is currently running. The *tpcpu* keyword is simulation time per CPU second, where simulation time is in time *units*. E.g. for metal units, the *tpcpu* value would be picoseconds per CPU second. The *spcpu* keyword is the number of timesteps per CPU second. Both quantities are on-the-fly metrics, measured relative to the last time they were invoked. Thus if you are printing out thermodynamic output every 100 timesteps, the two keywords will continually output the time and timestep rate for the last 100 steps. The *tpcpu* keyword does not attempt to track any changes in timestep size, e.g. due to using the *fix dt/reset* command.

The *cpuuse* keyword represents the CPU utilization in percent on MPI rank 0 for the current run. This should typically be around 100% for single-threaded runs. Smaller values indicate that LAMMPS may be stalling on file I/O, or some other process is competing with LAMMPS for the same CPU. When using multi-threading through the KOKKOS, INTEL, or OPENMP packages the value can be larger than 100% and ideally should be close to *nthreads* x 100%. How close depends on how much of the execution time is spent in multi-threaded parts of the code versus the non-accelerated parts.

The *cpuremain* keyword estimates the CPU time remaining in the current run, based on the time elapsed thus far. It will only be a good estimate if the CPU time/timestep for the rest of the run is similar to the preceding timesteps. On the initial timestep the value will be 0.0 since there is no history to estimate from. For a minimization run performed by the "minimize" command, the estimate is based on the *maxiter* parameter, assuming the minimization will proceed for the maximum number of allowed iterations.

The *part* keyword is useful for multi-replica or multi-partition simulations to indicate which partition this output and this file corresponds to, or for use in a *variable* to append to a filename for output specific to this partition. See discussion of the *-partition command-line switch* for details on running in multi-partition mode.

The *timeremain* keyword is the seconds remaining when a timeout has been configured via the *timer timeout* command. If the timeout timer is inactive, the value of this keyword is 0.0 and if the timer is expired, it is negative. This allows for example to exit loops cleanly, if the timeout is expired with:

```
if "$(timeremain) < 0.0" then "quit 0"
```

The *ecouple* keyword is cumulative energy change in the system due to any thermostatting or barostatting fixes that are being used. A positive value means that energy has been subtracted from the system (added to the coupling reservoir). See the *econserve* keyword for an explanation of why this sign choice makes sense.

The *econserve* keyword is the sum of the potential and kinetic energy of the system as well as the energy that has been transferred by thermostatting or barostatting to their coupling reservoirs – that is, *econserve* = *pe* + *ke* + *ecouple*. Ideally, for a simulation in the NVT, NPH, or NPT ensembles, the *econserve* quantity should remain constant over time even though *etotal* may change.

In LAMMPS, the simulation box can be defined as orthogonal or triclinic (non-orthogonal). See the *Howto_triclinic* doc page for a detailed explanation of orthogonal, restricted triclinic, and general triclinic simulation boxes and how LAMMPS rotates a general triclinic box to be restricted triclinic internally.

The *lx*, *ly*, *lz* keywords are the extent of the simulation box in each dimension. The *xlo*, *xhi*, *ylo*, *yhi*, *zlo*, *zhi* keywords are the lower and upper bounds of the simulation box in each dimension. I.e. *lx* = *xhi* - *xlo*). These 9 values are the same for all 3 kinds of boxes. I.e. for a restricted triclinic box, they are the values as if the box were not tilted. For a general triclinic box, they are the values after it is internally rotated to be a restricted triclinic box.

The *xy*, *xz*, *yz* are the current tilt factors for a triclinic box. They are the same for restricted and general triclinic boxes.

The *avecx*, *avecy*, *avecz*, *bvecx*, *bvecy*, *bvecz*, *cvecx*, *cvecy*, *cvecz* are the components of the 3 edge vectors of the current general simulation box. If it is an orthogonal box the vectors are along the x, y, z coordinate axes. If it is a restricted triclinic box, the **A** vector is along the x axis, the **B** vector is in the xy plane with a +y coordinate, and the **C** vector has a +z coordinate, as explained on the *Howto_triclinic* doc page. If the *thermo_modify triclinic/general* option is set then they are the **A**, **B**, **C** vector which define the general triclinic box.

The *cella*, *cellb*, *cellc*, *cellalpha*, *cellbeta*, *cellgamma* keywords correspond to the usual crystallographic quantities that define the periodic simulation box of a crystalline system. See the *Howto triclinic* page for a precise definition of these quantities in terms of the LAMMPS representation of a restricted triclinic simulation box via *lx*, *ly*, *lz*, *yz*, *xz*, *xy*.

The *pxx,pyy,pzz,pxy,pxz,pyz* keywords are the 6 components of the symmetric pressure tensor for the system. See the *compute pressure* command doc page for details of how it is calculated.

If the *thermo_modify triclinic/general* option is set then the 6 components will be output as values consistent with the orientation of the general triclinic box relative to the standard xyz coordinate axes. If this keyword is not used, the values will be consistent with the orientation of the restricted triclinic box (which aligns with the xyz coordinate axes). As explained on the *Howto_triclinic* doc page, even if the simulation box is created as a general triclinic box, internally LAMMPS uses a restricted triclinic box.

Note that because the pressure tensor components are computed using force vectors and atom coordinates, both of which are rotated in the general versus restricted triclinic representation, the values will typically be different for the two cases.

The *fmax* and *fnorm* keywords are useful for monitoring the progress of an *energy minimization*. The *fmax* keyword calculates the maximum force in any dimension on any atom in the system, or the infinity-norm of the force vector for the system. The *fnorm* keyword calculates the 2-norm or length of the force vector.

The *nbuild* and *ndanger* keywords are useful for monitoring neighbor list builds during a run. Note that both these values are also printed with the end-of-run statistics. The *nbuild* keyword is the number of re-builds during the current run. The *ndanger* keyword is the number of re-builds that LAMMPS considered potentially "dangerous". If atom movement triggered neighbor list rebuilding (see the *neigh_modify* command), then dangerous reneighborings are those that were triggered on the first timestep atom movement was checked for. If this count is non-zero you may wish to reduce the delay factor to ensure no force interactions are missed by atoms moving beyond the neighbor skin distance before a rebuild takes place.

---

For output values from a compute or fix or variable, the bracketed index I used to index a vector, as in *c_ID[I]* or *f_ID[I]* or *v_name[I]*, can be specified using a wildcard asterisk with the index to effectively specify multiple values. This takes the form "*" or "*n" or "n*" or "m*n". If N = the size of the vector, then an asterisk with no numeric values means all indices from 1 to N. A leading asterisk means all indices from 1 to n (inclusive). A trailing asterisk means all indices from n to N (inclusive). A middle asterisk means all indices from m to n (inclusive).

Using a wildcard is the same as if the individual elements of the vector had been listed one by one. E.g. these 2 thermo_style commands are equivalent, since the *compute temp* command creates a global vector with 6 values.

```
compute myTemp all temp
thermo_style custom step temp etotal c_myTemp[*]
thermo_style custom step temp etotal &
             c_myTemp[1] c_myTemp[2] c_myTemp[3] &
             c_myTemp[4] c_myTemp[5] c_myTemp[6]
```

**Note**

For a vector-style variable, only the wildcard forms "*n" or "m*n" are allowed. You must specify the upper bound, because vector-style variable lengths are not determined until the variable is evaluated. If n is specified larger than the vector length turns out to be, zeroes are output for missing vector values.

---

The *c_ID* and *c_ID[I]* and *c_ID[I][J]* keywords allow global values calculated by a compute to be output. As discussed on the *compute* doc page, computes can calculate global, per-atom, local, and per-grid values. Only global values can be referenced by this command. However, per-atom compute values for an individual atom can be referenced in a *equal-style variable* and the variable referenced by thermo_style custom, as discussed below. See the discussion above for how the I in *c_ID[I]* can be specified with a wildcard asterisk to effectively specify multiple values from a global compute vector.

The ID in the keyword should be replaced by the actual ID of a compute that has been defined elsewhere in the input script. See the *compute* command for details. If the compute calculates a global scalar, vector, or array, then the keyword formats with 0, 1, or 2 brackets will reference a scalar value from the compute.

Note that some computes calculate "intensive" global quantities like temperature; others calculate "extensive" global quantities like kinetic energy that are summed over all atoms in the compute group. Intensive quantities are printed directly without normalization by thermo_style custom. Extensive quantities may be normalized by the total number of atoms in the simulation (NOT the number of atoms in the compute group) when output, depending on the *thermo_modify norm* option being used.

The *f_ID* and *f_ID[I]* and *f_ID[I][J]* keywords allow global values calculated by a fix to be output. As discussed on the *fix* doc page, fixes can calculate global, per-atom, local, and per-grid values. Only global values can be referenced by this command. However, per-atom fix values can be referenced for an individual atom in a *equal-style variable* and the variable referenced by thermo_style custom, as discussed below. See the discussion above for how the I in *f_ID[I]* can be specified with a wildcard asterisk to effectively specify multiple values from a global fix vector.

The ID in the keyword should be replaced by the actual ID of a fix that has been defined elsewhere in the input script. See the *fix* command for details. If the fix calculates a global scalar, vector, or array, then the keyword formats with 0, 1, or 2 brackets will reference a scalar value from the fix.

Note that some fixes calculate "intensive" global quantities like timestep size; others calculate "extensive" global quantities like energy that are summed over all atoms in the fix group. Intensive quantities are printed directly without normalization by thermo_style custom. Extensive quantities may be normalized by the total number of atoms in the simulation (NOT the number of atoms in the fix group) when output, depending on the *thermo_modify norm* option being used.

The *v_name* keyword allow the current value of a variable to be output. The name in the keyword should be replaced by the variable name that has been defined elsewhere in the input script. Only equal-style and vector-style variables can be referenced; the latter requires a bracketed term to specify the Ith element of the vector calculated by the variable. However, an equal-style variable can use an atom-style variable in its formula indexed by the ID of an individual atom. This is a way to output a specific atom's per-atom coordinates or other per-atom properties in thermo output. See the *variable* command for details. Note that variables of style *equal* and *vector* and *atom* define a formula which can reference per-atom properties or thermodynamic keywords, or they can invoke other computes, fixes, or variables when evaluated, so this is a very general means of creating thermodynamic output.

Note that equal-style and vector-style variables are assumed to produce "intensive" global quantities, which are thus printed as-is, without normalization by thermo_style custom. You can include a division by "natoms" in the variable formula if this is not the case.

---

### 1.107.4 Restrictions

This command must come after the simulation box is defined by a *read_data*, *read_restart*, or *create_box* command.

### 1.107.5 Related commands

*thermo*, *thermo_modify*, *fix_modify*, *compute temp*, *compute pressure*

### 1.107.6 Default

```
thermo_style one
```

## 1.108 third_order command

Accelerator Variant: third_order/kk

### 1.108.1 Syntax

```
third_order group-ID style delta args keyword value ...
```

- group-ID = ID of group of atoms to displace
- style = *regular* or *eskm*
- delta = finite different displacement length (distance units)
- one or more keyword/arg pairs may be appended

```
keyword = file or binary
  file name = name of output file for the third order tensor
  binary arg = yes or no or gzip
```

### 1.108.2 Examples

```
third_order 1 regular 0.000001
third_order 1 eskm 0.000001
third_order 3 regular 0.00004 file third_order.dat
third_order 5 eskm 0.00000001 file third_order.dat binary yes
```

### 1.108.3 Description

Calculate the third order force constant tensor by finite difference of the selected group,

$$\Phi_{ijk}^{\alpha\beta\gamma} = \frac{\partial^3 U}{\partial x_{i,\alpha} \partial x_{j,\beta} \partial x_{k,\gamma}}$$

where Phi is the third order force constant tensor.

The output of the command is the tensor, three elements at a time. The three elements correspond to the three gamma elements for a specific i/alpha/j/beta/k. The initial five numbers are i, alpha, j, beta, and k respectively.

If the style eskm is selected, the tensor will be using energy units of 10 J/mol. These units conform to eskm style from the dynamical_matrix command, which will simplify operations using dynamical matrices with third order tensors.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 1.108.4 Restrictions

The command collects a 9 times the number of atoms in the group on every single MPI rank, so the memory requirements can be very significant for large systems.

This command is part of the PHONON package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 1.108.5 Related commands

*fix phonon dynamical_matrix*

### 1.108.6 Default

The default settings are file = "third_order.dat", binary = no

## 1.109 timer command

### 1.109.1 Syntax

```
timer args
```

- *args* = one or more of *off* or *loop* or *normal* or *full* or *sync* or *nosync* or *timeout* or *every*

```
off = do not collect or print any timing information
loop = collect only the total time for the simulation loop
normal = collect timer information broken down by sections (default)
full = like normal but also include CPU and thread utilization
sync = explicitly synchronize MPI tasks between sections
nosync = do not synchronize MPI tasks between sections (default)
timeout elapse = set wall time limit to elapse
every Ncheck = perform timeout check every Ncheck steps
```

### 1.109.2 Examples

```
timer full sync
timer timeout 2:00:00 every 100
timer loop
```

### 1.109.3 Description

Select the level of detail at which LAMMPS performs its CPU timings. Multiple keywords can be specified with the *timer* command. For keywords that are mutually exclusive, the last one specified takes precedence.

During a simulation run LAMMPS collects information about how much time is spent in different sections of the code and thus can provide information for determining performance and load imbalance problems. This can be done at different levels of detail and accuracy. For more information about the timing output, see the *Run output* doc page.

The *off* setting will turn all time measurements off. The *loop* setting will only measure the total time for a run and not collect any detailed per section information. With the *normal* setting, timing information for portions of the timestep (pairwise calculations, neighbor list construction, output, etc) are collected as well as information about load imbalances for those sections across MPI processes. The *full* setting adds information about CPU utilization and thread utilization, when multi-threading is enabled.

With the *sync* setting, all MPI tasks are synchronized at each timer call which measures load imbalance for each section more accurately, though it can also slow down the simulation by prohibiting overlapping independent computations on different MPI ranks. Using the *nosync* setting (which is the default) turns this synchronization off.

With the *timeout* keyword a wall time limit can be imposed, that affects the *run* and *minimize* commands. This can be convenient when calculations have to comply with execution time limits, e.g. when running under a batch system when you want to maximize the utilization of the batch time slot, especially for runs where the time per timestep varies much and thus it becomes difficult to predict how many steps a simulation can perform for a given wall time limit. This also applies for difficult to converge minimizations. The timeout *elapse* value should be somewhat smaller than the maximum wall time requested from the batch system, as there is usually some overhead to launch jobs, and it is advisable to write out a restart after terminating a run due to a timeout.

The timeout timer starts when the command is issued. When the time limit is reached, the run or energy minimization will exit on the next step or iteration that is a multiple of the *Ncheck* value which can be set with the *every* keyword. Default is checking every 10 steps. After the timer timeout has expired all subsequent run or minimize commands in the input script will be skipped. The remaining time or timer status can be accessed with the *thermo* variable *timeremain*, which will be zero, if the timeout is inactive (default setting), it will be negative, if the timeout time is expired and positive if there is time remaining and in this case the value of the variable are the number of seconds remaining.

When the *timeout* keyword is used a second time, the timer is restarted with a new time limit. The timeout *elapse* value can be specified as *off* or *unlimited* to impose a no timeout condition (which is the default). The *elapse* setting can be specified as a single number for seconds, two numbers separated by a colon (MM:SS) for minutes and seconds, or as three numbers separated by colons for hours, minutes, and seconds (H:MM:SS).

The *every* keyword sets how frequently during a run or energy minimization the wall clock will be checked. This check count applies to the outer iterations or time steps during minimizations or *r-RESPA runs*, respectively. Checking for timeout too often, can slow a calculation down. Checking too infrequently can make the timeout measurement less accurate, with the run being stopped later than desired.

**Note**

Using the *full* and *sync* options provides the most detailed and accurate timing information, but can also have a negative performance impact due to the overhead of the many required system calls. It is thus recommended to use these settings only when testing tests to identify performance bottlenecks. For calculations with few atoms or a very large number of MPI processes, even the *normal* setting can have a measurable negative performance impact. In those cases you can just use the *loop* or *off* setting.

### 1.109.4 Restrictions

none

### 1.109.5 Related commands

*run post no*, *kspace_modify fftbench*

### 1.109.6 Default

```
timer normal nosync
timer timeout off
timer every 10
```

## 1.110 timestep command

### 1.110.1 Syntax

```
timestep dt
```

- dt = timestep size (time units)

### 1.110.2 Examples

```
timestep 2.0
timestep 0.003
```

### 1.110.3 Description

Set the timestep size for subsequent molecular dynamics simulations. See the *units* command for the time units associated with each choice of units that LAMMPS supports.

The default value for the timestep size also depends on the choice of units for the simulation; see the default values below.

When the *run style* is *respa*, dt is the timestep for the outer loop (largest) timestep.

### 1.110.4 Restrictions

none

### 1.110.5 Related commands

*fix dt/reset*, *run*, *run_style* respa, *units*

### 1.110.6 Default

| choice of *units* | time units | default timestep size |
|---|---|---|
| lj | $\tau$ | 0.005 $\tau$ |
| real | fs | 1.0 fs |
| metal | ps | 0.001 ps |
| si | s | 1.0e-8 s (10 ns) |
| cgs | s | 1.0e-8 s (10 ns) |
| electron | fs | 0.001 fs |
| micro | $\mu$s | 2.0 $\mu$s |
| nano | ns | 0.00045 ns |

## 1.111 uncompute command

### 1.111.1 Syntax

```
uncompute compute-ID
```

- compute-ID = ID of a previously defined compute

### 1.111.2 Examples

```
uncompute 2
uncompute lower-boundary
```

### 1.111.3 Description

Delete a compute that was previously defined with a *compute* command. This also wipes out any additional changes made to the compute via the *compute_modify* command.

### 1.111.4 Restrictions

none

### 1.111.5 Related commands

*compute*

### 1.111.6 Default

none

## 1.112 undump command

### 1.112.1 Syntax

```
undump dump-ID
```

- dump-ID = ID of previously defined dump

### 1.112.2 Examples

```
undump mine
undump 2
```

### 1.112.3 Description

Turn off a previously defined dump so that it is no longer active. This closes the file associated with the dump.

### 1.112.4 Restrictions

none

### 1.112.5 Related commands

*dump*

### 1.112.6 Default

none

## 1.113 unfix command

### 1.113.1 Syntax

```
unfix fix-ID
```

- fix-ID = ID of a previously defined fix

### 1.113.2 Examples

```
unfix 2
unfix lower-boundary
```

### 1.113.3 Description

Delete a fix that was previously defined with a *fix* command. This also wipes out any additional changes made to the fix via the *fix_modify* command.

### 1.113.4 Restrictions

none

### 1.113.5 Related commands

*fix*

### 1.113.6 Default

none

## 1.114 units command

### 1.114.1 Syntax

```
units style
```

- style = *lj* or *real* or *metal* or *si* or *cgs* or *electron* or *micro* or *nano*

### 1.114.2 Examples

```
units metal
units lj
```

### 1.114.3 Description

This command sets the style of units used for a simulation. It determines the units of all quantities specified in the input script and data file, as well as quantities output to the screen, log file, and dump files. Typically, this command is used at the very beginning of an input script.

For all units except *lj*, LAMMPS uses physical constants from www.physics.nist.gov. For the definition of kcal in real units, LAMMPS uses the thermochemical calorie = 4.184 J.

The choice you make for units simply sets some internal conversion factors within LAMMPS. This means that any simulation you perform for one choice of units can be duplicated with any other unit setting LAMMPS supports. In this context "duplicate" means the particles will have identical trajectories and all output generated by the simulation will be identical. This will be the case for some number of timesteps until round-off effects accumulate, since the conversion factors for two different unit systems are not identical to infinite precision.

To perform the same simulation in a different set of units you must change all the unit-based input parameters in your input script and other input files (data file, potential files, etc) correctly to the new units. And you must correctly convert all output from the new units to the old units when comparing to the original results. That is often not simple to do.

Potential or table files may have a `UNITS:` tag included in the first line indicating the unit style those files were created for. If the tag exists, its value will be compared to the chosen unit style and LAMMPS will stop with an error message if there is a mismatch. In some select cases and for specific combinations of unit styles, LAMMPS is capable of automatically converting potential parameters from a file. In those cases, a warning message signaling that an automatic conversion has happened is printed to the screen.

---

For style *lj*, all quantities are unitless. Without loss of generality, LAMMPS sets the fundamental quantities mass, $\sigma$, $\varepsilon$, and the Boltzmann constant $k_B = 1$. The masses, distances, energies you specify are multiples of these fundamental values. The formulas relating the reduced or unitless quantity (with an asterisk) to the same quantity with units is also given. Thus you can use the mass, $\sigma$, and $\varepsilon$ values for a specific material and convert the results from a unitless LJ simulation into physical quantities. Please note that using these three properties as base, your unit of time has to conform to the relation $\varepsilon = \frac{m\sigma^2}{\tau^2}$ since energy is a derived unit (in SI units you equivalently have the relation $1\,\mathrm{J} = 1\frac{\mathrm{kg}\cdot\mathrm{m}^2}{\mathrm{s}^2}$).

- mass = mass or $m$, where $M^* = \frac{M}{m}$

- distance = $\sigma$, where $x^* = \frac{x}{\sigma}$
- time = $\tau$, where $\tau^* = \tau\sqrt{\frac{\varepsilon}{m\sigma^2}}$
- energy = $\varepsilon$, where $E^* = \frac{E}{\varepsilon}$
- velocity = $\frac{\sigma}{\tau}$, where $v^* = v\frac{\tau}{\sigma}$
- force = $\frac{\varepsilon}{\sigma}$, where $f^* = f\frac{\sigma}{\varepsilon}$
- torque = $\varepsilon$, where $t^* = \frac{t}{\varepsilon}$
- temperature = reduced LJ temperature, where $T^* = \frac{Tk_B}{\varepsilon}$
- pressure = reduced LJ pressure, where $p^* = p\frac{\sigma^3}{\varepsilon}$
- dynamic viscosity = reduced LJ viscosity, where $\eta^* = \eta\frac{\sigma^3}{\varepsilon\tau}$
- charge = reduced LJ charge, where $q^* = q\frac{1}{\sqrt{4\pi\varepsilon_0\sigma\varepsilon}}$
- dipole = reduced LJ dipole, moment where $\mu^* = \mu\frac{1}{\sqrt{4\pi\varepsilon_0\sigma^3\varepsilon}}$
- electric field = force/charge, where $E^* = E\frac{\sqrt{4\pi\varepsilon_0\sigma\varepsilon}\sigma}{\varepsilon}$
- density = mass/volume, where $\rho^* = \rho\frac{\sigma^{dim}}{m}$

Note that for LJ units, the default mode of thermodynamic output via the *thermo_style* command is to normalize all extensive quantities by the number of atoms. E.g. potential energy is extensive because it is summed over atoms, so it is output as energy/atom. Temperature is intensive since it is already normalized by the number of atoms, so it is output as-is. This behavior can be changed via the *thermo_modify norm* command.

For style *real*, these are the units:

- mass = grams/mole
- distance = Angstroms
- time = femtoseconds
- energy = kcal/mol
- velocity = Angstroms/femtosecond
- force = (kcal/mol)/Angstrom
- torque = kcal/mol
- temperature = Kelvin
- pressure = atmospheres
- dynamic viscosity = Poise
- charge = multiple of electron charge (1.0 is a proton)
- dipole = charge*Angstroms
- electric field = volts/Angstrom
- density = g/cm^dim

For style *metal*, these are the units:

- mass = grams/mole
- distance = Angstroms

- time = picoseconds
- energy = eV
- velocity = Angstroms/picosecond
- force = eV/Angstrom
- torque = eV
- temperature = Kelvin
- pressure = bars
- dynamic viscosity = Poise
- charge = multiple of electron charge (1.0 is a proton)
- dipole = charge*Angstroms
- electric field = volts/Angstrom
- density = gram/cm^dim

For style *si*, these are the units:

- mass = kilograms
- distance = meters
- time = seconds
- energy = Joules
- velocity = meters/second
- force = Newtons
- torque = Newton-meters
- temperature = Kelvin
- pressure = Pascals
- dynamic viscosity = Pascal*second
- charge = Coulombs (1.6021765e-19 is a proton)
- dipole = Coulombs*meters
- electric field = volts/meter
- density = kilograms/meter^dim

For style *cgs*, these are the units:

- mass = grams
- distance = centimeters
- time = seconds
- energy = ergs
- velocity = centimeters/second
- force = dynes
- torque = dyne-centimeters
- temperature = Kelvin

- pressure = dyne/cm^2 or barye = 1.0e-6 bars
- dynamic viscosity = Poise
- charge = statcoulombs or esu (4.8032044e-10 is a proton)
- dipole = statcoul-cm = 10^18 debye
- electric field = statvolt/cm or dyne/esu
- density = grams/cm^dim

For style *electron*, these are the units:

- mass = atomic mass units
- distance = Bohr
- time = femtoseconds
- energy = Hartrees
- velocity = Bohr/atomic time units [1.03275e-15 seconds]
- force = Hartrees/Bohr
- temperature = Kelvin
- pressure = Pascals
- charge = multiple of electron charge (1.0 is a proton)
- dipole moment = Debye
- electric field = volts/cm

For style *micro*, these are the units:

- mass = picograms
- distance = micrometers
- time = microseconds
- energy = picogram-micrometer^2/microsecond^2
- velocity = micrometers/microsecond
- force = picogram-micrometer/microsecond^2
- torque = picogram-micrometer^2/microsecond^2
- temperature = Kelvin
- pressure = picogram/(micrometer-microsecond^2)
- dynamic viscosity = picogram/(micrometer-microsecond)
- charge = picocoulombs (1.6021765e-7 is a proton)
- dipole = picocoulomb-micrometer
- electric field = volt/micrometer
- density = picograms/micrometer^dim

For style *nano*, these are the units:

- mass = attograms
- distance = nanometers

- time = nanoseconds
- energy = attogram-nanometer^2/nanosecond^2
- velocity = nanometers/nanosecond
- force = attogram-nanometer/nanosecond^2
- torque = attogram-nanometer^2/nanosecond^2
- temperature = Kelvin
- pressure = attogram/(nanometer-nanosecond^2)
- dynamic viscosity = attogram/(nanometer-nanosecond)
- charge = multiple of electron charge (1.0 is a proton)
- dipole = charge-nanometer
- electric field = volt/nanometer
- density = attograms/nanometer^dim

The units command also sets the timestep size and neighbor skin distance to default values for each style:

- For style *lj* these are dt = 0.005 $\tau$ and skin = 0.3 $\sigma$.
- For style *real* these are dt = 1.0 femtoseconds and skin = 2.0 Angstroms.
- For style *metal* these are dt = 0.001 picoseconds and skin = 2.0 Angstroms.
- For style *si* these are dt = 1.0e-8 seconds and skin = 0.001 meters.
- For style *cgs* these are dt = 1.0e-8 seconds and skin = 0.1 centimeters.
- For style *electron* these are dt = 0.001 femtoseconds and skin = 2.0 Bohr.
- For style *micro* these are dt = 2.0 microseconds and skin = 0.1 micrometers.
- For style *nano* these are dt = 0.00045 nanoseconds and skin = 0.1 nanometers.

### 1.114.4 Restrictions

This command cannot be used after the simulation box is defined by a *read_data* or *create_box* command.

### 1.114.5 Related commands

none

### 1.114.6 Default

```
units lj
```

## 1.115 variable command

### 1.115.1 Syntax

```
variable name style args ...
```

- name = name of variable to define

- style = *delete* or *atomfile* or *file* or *format* or *getenv* or *index* or *internal* or *loop* or *python* or *string* or *timer* or *uloop* or *universe* or *world* or *equal* or *vector* or *atom*

```
delete = no args
atomfile arg = filename
file arg = filename
format args = vname fstr
  vname = name of equal-style variable to evaluate
  fstr = C-style format string
getenv arg = one string
index args = one or more strings
internal arg = numeric value
loop args = N
  N = integer size of loop, loop from 1 to N inclusive
loop args = N pad
  N = integer size of loop, loop from 1 to N inclusive
  pad = all values will be same length, e.g. 001, 002, ..., 100
loop args = N1 N2
  N1,N2 = loop from N1 to N2 inclusive
loop args = N1 N2 pad
  N1,N2 = loop from N1 to N2 inclusive
  pad = all values will be same length, e.g. 050, 051, ..., 100
python arg = function
string arg = one string
timer arg = no arguments
uloop args = N
  N = integer size of loop
uloop args = N pad
  N = integer size of loop
  pad = all values will be same length, e.g. 001, 002, ..., 100
universe args = one or more strings
world args = one string for each partition of MPI processes

equal or vector or atom args = one formula containing numbers, thermo keywords,
    math operations, built-in functions, atom values and vectors, compute/fix/
↪variable references
  numbers = 0.0, 100, -5.4, 2.8e-4, etc
  constants = PI, version, on, off, true, false, yes, no
  thermo keywords = vol, ke, press, etc from thermo_style
  math operators = (), -x, x+y, x-y, x*y, x/y, x^y, x%y,
                   x == y, x != y, x < y, x <= y, x > y, x >= y, x && y, x || y, x␣
↪|^ y, !x
  math functions = sqrt(x), exp(x), ln(x), log(x), abs(x), sign(x),
                   sin(x), cos(x), tan(x), asin(x), acos(x), atan(x), atan2(y,x),
                   random(x,y,z), normal(x,y,z), ceil(x), floor(x), round(x),␣
↪ternary(x,y,z),
                   ramp(x,y), stagger(x,y), logfreq(x,y,z), logfreq2(x,y,z),
                   logfreq3(x,y,z), stride(x,y,z), stride2(x,y,z,a,b,c),
                   vdisplace(x,y), swiggle(x,y,z), cwiggle(x,y,z)
  group functions = count(group), mass(group), charge(group),
                    xcm(group,dim), vcm(group,dim), fcm(group,dim),
                    bound(group,dir), gyration(group), ke(group),
                    angmom(group,dim), torque(group,dim),
                    inertia(group,dimdim), omega(group,dim)
```

```
  region functions = count(group,region), mass(group,region), charge(group,region),
                     xcm(group,dim,region), vcm(group,dim,region), fcm(group,dim,
↪region),
                     bound(group,dir,region), gyration(group,region), ke(group,
↪reigon),
                     angmom(group,dim,region), torque(group,dim,region),
                     inertia(group,dimdim,region), omega(group,dim,region)
  special functions = sum(x), min(x), max(x), ave(x), trap(x), slope(x), sort(x),␣
↪rsort(x),
                      gmask(x), rmask(x), grmask(x,y), next(x), is_file(name), is_
↪os(name),
                      extract_setting(name), label2type(kind,label),
                      is_typelabel(kind,label), is_timeout()
  feature functions = is_available(category,feature), is_active(category,feature),
                      is_defined(category,id)
  python function wrapper = py_varname(x,y,z,...)
  atom value = id[i], mass[i], type[i], mol[i], x[i], y[i], z[i], vx[i], vy[i],␣
↪vz[i], fx[i], fy[i], fz[i], q[i]
  atom vector = id, mass, type, mol, radius, q, x, y, z, vx, vy, vz, fx, fy, fz
  custom atom property = i_name, d_name, i_name[i], d_name[i], i2_name[i], d2_
↪name[i], i2_name[i][j], d2_name[i][j]
  compute references = c_ID, c_ID[i], c_ID[i][j], C_ID, C_ID[i], C_ID[i][j]
  fix references = f_ID, f_ID[i], f_ID[i][j], F_ID, F_ID[i], F_ID[i][j]
  variable references = v_name, v_name[i]
  vector initialization = [1,3,7,10] (for vector variables only)
```

### 1.115.2 Examples

```
variable x index run1 run2 run3 run4 run5 run6 run7 run8
variable LoopVar loop $n
variable beta equal temp/3.0
variable b1 equal x[234]+0.5*vol
variable b1 equal "x[234] + 0.5*vol"
variable b equal xcm(mol1,x)/2.0
variable b equal c_myTemp
variable b atom x*y/vol
variable foo string myfile
variable foo internal 3.5
variable myPy python increase
variable f file values.txt
variable temp world 300.0 310.0 320.0 ${Tfinal}
variable x universe 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
variable x uloop 15 pad
variable str format x %.6g
variable myvec vector [1,3,7,10]
variable x delete
```

```
variable start timer
other commands
variable stop timer
print "Elapsed time: $(v_stop-v_start:%.6f)"
```

### 1.115.3 Description

This command assigns one or more strings to a variable name for evaluation later in the input script or during a simulation.

Variables can thus be useful in several contexts. A variable can be defined and then referenced elsewhere in an input script to become part of a new input command. For variable styles that store multiple strings, the *next* command can be used to increment which string is assigned to the variable. Variables of style *equal* store a formula which when evaluated produces a single numeric value which can be output either directly (see the *print*, *fix print*, and *run every* commands) or as part of thermodynamic output (see the *thermo_style* command), or used as input to an averaging fix (see the *fix ave/time* command). Variables of style *vector* store a formula which produces a vector of such values which can be used as input to various averaging fixes, or elements of which can be part of thermodynamic output.

Variables of style *atom* store a formula which when evaluated produces one numeric value per atom which can be output to a dump file (see the *dump custom* command) or used as input to an averaging fix (see the *fix ave/chunk* and *fix ave/atom* commands). Variables of style *atomfile* can be used anywhere in an input script that atom-style variables are used; they get their per-atom values from a file rather than from a formula.

Variables of style *python* can be hooked to Python functions using Python code you provide, so that the variable gets its value from the evaluation of the Python code. Variables of style *internal* are used by a few commands which set their value directly.

**Note**

As discussed on the *Commands parse* doc page, an input script can use "immediate" variables, specified as $(formula) with parenthesis, where the numeric formula has the same syntax as equal-style variables described on this page. This is a convenient way to evaluate a formula immediately without using the variable command to define a named variable and then evaluate that variable. The formula can include a trailing colon and format string which determines the precision with which the numeric value is generated. This is also explained on the *Commands parse* doc page.

In the discussion that follows, the "name" of the variable is the arbitrary string that is the first argument in the variable command. This name can only contain alphanumeric characters and underscores. The "string" is one or more of the subsequent arguments. The "string" can be simple text as in the first example above, it can contain other variables as in the second example, or it can be a formula as in the third example. The "value" is the numeric quantity resulting from evaluation of the string. Note that the same string can generate different values when it is evaluated at different times during a simulation.

**Note**

When an input script line is encountered that defines a variable of style *equal* or *vector* or *atom* or *python* that contains a formula or links to Python code, the formula or Python code is NOT immediately evaluated. Instead, it is evaluated each time the variable is **used**. If you simply want to evaluate a formula in place you can use a so-called immediate variable. as described in the preceding note. Or see the section below about "Immediate Evaluation of Variables" for more details on the topic. This is also true of a *format* style variable since it evaluates another variable when it is invoked.

Variables of style *equal* and *vector* and *atom* can be used as inputs to various other commands which evaluate their formulas as needed, e.g. at different timesteps during a *run*. In this context, variables of style *timer* or *internal* or *python* can be used in place of an equal-style variable, with the following two caveats.

First, internal-style variables require their values be set by code elsewhere in LAMMPS. When a LAMMPS input script or command evaluates an internal-style variable, it must have a current value set (internally) via that mechanism. Second, python-style variables can be used so long as the associated Python function, as defined by the *python* command,

returns a numeric value. When the LAMMPS command evaluates the python-style variable, the Python function will be executed.

**Note**

When a variable command is encountered in the input script and the variable name has already been specified, the command is ignored. This means variables can NOT be re-defined in an input script (with two exceptions, read further). This is to allow an input script to be processed multiple times without resetting the variables; see the *jump* or *include* commands. It also means that using the *command-line switch* -var will override a corresponding index variable setting in the input script.

There are two exceptions to this rule. First, variables of style *string*, *getenv*, *internal*, *equal*, *vector*, *atom*, and *python* ARE redefined each time the command is encountered. This allows these style of variables to be redefined multiple times in an input script. In a loop, this means the formula associated with an *equal* or *atom* style variable can change if it contains a substitution for another variable, e.g. $x or v_x.

Second, as described below, if a variable is iterated on to the end of its list of strings via the *next* command, it is removed from the list of active variables, and is thus available to be re-defined in a subsequent variable command. The *delete* style does the same thing.

Changed in version 4Jul2026.

**Auto-deleted variables can lead to problems**

Special care must be taken when iterated variables, e.g. file style variables are exhausted and thus get deleted **during** a run. For performance reasons, many features in LAMMPS cache how variables are looked up during a run or minimization for the duration of that run or minimization and this can lead to unexpected behavior when variables get auto-deleted. Evaluating a deleted variable returns 0.0 instead of creating an error for practical reasons. But LAMMPS prints a warning when a file or atomfile style variable is exhausted and auto-deleted. This can be avoided by making certain that those variables have additional elements. This condition is rare, but hard to debug, so make certain that when you see the warning about an auto-deleted variable, that this is the intended behavior.

Variables are **not** deleted by the *clear* command with the exception of atomfile-style variables.

---

The *Commands parse* page explains how occurrences of a variable name in an input script line are replaced by the variable's string. The variable name can be referenced as $x if the name "x" is a single character, or as ${LoopVar} if the name "LoopVar" is one or more characters.

As described below, for variable styles *index*, *loop*, *file*, *universe*, and *uloop*, which string is assigned to a variable can be incremented via the *next* command. When there are no more strings to assign, the variable is exhausted and a flag is set that causes the next *jump* command encountered in the input script to be skipped. This enables the construction of simple loops in the input script that are iterated over and then exited from.

As explained above, an exhausted variable can be re-used in an input script. The *delete* style also removes the variable, the same as if it were exhausted, allowing it to be redefined later in the input script or when the input script is looped over. This can be useful when breaking out of a loop via the *if* and *jump* commands before the variable would become exhausted. For example,

```
label       loop
variable    a loop 5
print       "A = $a"
if          "$a > 2" then "jump in.script break"
```

(continues on next page)

(continued from previous page)

```
next        a
jump        in.script loop
label       break
variable    a delete
```

---

The next sections describe in how all the various variable styles are defined and what they store. The styles are listed alphabetically, except for the *equal* and *vector* and *atom* styles, which are explained together after all the others.

Many of the styles store one or more strings. Note that a single string can contain spaces (multiple words), if it is enclosed in quotes in the variable command. When the variable is substituted for in another input script command, its returned string will then be interpreted as multiple arguments in the expanded command.

---

For the *atomfile* style, a filename is provided which contains one or more sets of values, to assign on a per-atom basis to the variable. The format of the file is described below.

When an atomfile-style variable is defined, the file is opened and the first set of per-atom values are read and stored with the variable. This means the variable can then be evaluated as many times as desired and will return those values. There are two ways to cause the next set of per-atom values from the file to be read: use the *next* command or the next() function in an atom-style variable, as discussed below. Unlike most variable styles, which remain defined, atomfile-style variables are **deleted** during a *clear* command.

The rules for formatting the file are as follows. Each time a set of per-atom values is read, a non-blank line is searched for in the file. The file is read line by line but only up to 254 characters are used. The rest are ignored. A comment character "#" can be used anywhere on a line and all text following and the "#" character are ignored; text starting with the comment character is stripped. Blank lines are skipped. The first non-blank line is expected to contain a single integer number as the count $N$ of per-atom lines to follow. $N$ can be the total number of atoms in the system or less, indicating that data for a subset is read. The next N lines must consist of two numbers, the atom-ID of the atom for which a value is set followed by a floating point number with the value. The atom-IDs may be listed in any order.

**Note**

Every time a set of per-atom lines is read, the value of the atomfile variable for **all** atoms is first initialized to 0.0. Thus values for atoms whose ID do not appear in the set in the file will remain at 0.0.

Below is a small example for the atomfile variable file format:

```
# first set
4
# atom-ID value
3 1
4 -4
1 0.5
2 -0.5

# second set
2

2  1.0
4 -1.0
```

---

For the *file* style, a filename is provided which contains a list of strings to assign to the variable, one per line. The strings can be numeric values if desired. See the discussion of the next() function below for equal-style variables, which will convert the string of a file-style variable into a numeric value in a formula.

When a file-style variable is defined, the file is opened and the string on the first line is read and stored with the variable. This means the variable can then be evaluated as many times as desired and will return that string. There are two ways to cause the next string from the file to be read: use the *next* command or the next() function in an equal- or atom-style variable, as discussed below.

The rules for formatting the file are as follows. A comment character "#" can be used anywhere on a line; text starting with the comment character is stripped. Blank lines are skipped. The first "word" of a non-blank line, delimited by white-space, is the "string" assigned to the variable.

---

For the *format* style, an equal-style or compatible variable is specified along with a C-style format string, e.g. "%f" or "%.10g", which must be appropriate for formatting a double-precision floating-point value and may not have extra characters. The default format is "%.15g". This variable style allows an equal-style variable to be formatted precisely when it is evaluated.

Note that if you simply wish to print a variable value with desired precision to the screen or logfile via the *print* or *fix print* commands, you can also do this by specifying an "immediate" variable with a trailing colon and format string, as part of the string argument of those commands. This is explained on the *Commands parse* doc page.

---

For the *getenv* style, a single string is assigned to the variable which should be the name of an environment variable. When the variable is evaluated, it returns the value of the environment variable, or an empty string if it not defined. This style of variable can be used to adapt the behavior of LAMMPS input scripts via environment variable settings, or to retrieve information that has been previously stored with the *shell putenv* command. Note that because environment variable settings are stored by the operating systems, they persist even if the corresponding *getenv* style variable is deleted, and also are set for sub-shells executed by the *shell* command.

---

For the *index* style, one or more strings are specified. Initially, the first string is assigned to the variable. Each time a *next* command is used with the variable name, the next string is assigned. All MPI processes assign the same string to the variable.

Index-style variables with a single string value can also be set by using the *command-line switch -var*.

---

For the *internal* style a numeric value is provided. This value will be assigned to the variable until a LAMMPS command sets it to a new value.

Note however, that most commands which use internal-style variables do not require them to be defined in the input script. They create one or more internal-style variables if they do not already exist. Examples are these commands:

- *create_atoms*
- *fix deposit*
- *compute bond/local*
- *compute angle/local*
- *compute dihedral/local*
- *python* command in conjunction with Python function wrappers used in equal- and atom-style variable formulas

A command which does require an internal-style variable to be defined in the input script is the *fix controller* command, because another (arbitrary) command typically also references the variable.

---

The *loop* style is identical to the *index* style except that the strings are the integers from 1 to N inclusive, if only one argument N is specified. This allows generation of a long list of runs (e.g. 1000) without having to list N strings in the input script. Initially, the string "1" is assigned to the variable. Each time a *next* command is used with the variable name, the next string ("2", "3", etc) is assigned. All MPI processes assign the same string to the variable. The *loop* style can also be specified with two arguments N1 and N2. In this case the loop runs from N1 to N2 inclusive, and the string N1 is initially assigned to the variable. N1 <= N2 and N2 >= 0 is required.

---

For the *python* style a Python function name is provided. This needs to match a function name specified in a *python* command which returns a value to this variable as defined by its *return* keyword. For example these two commands would be self-consistent:

```
variable foo python myMultiply
python myMultiply return v_foo format f file funcs.py
```

The two commands can appear in either order so long as both are specified before the Python function is invoked for the first time.

Each time the variable is evaluated, the associated Python function is invoked, and the value it returns is also returned by the variable. Since the Python function can use other LAMMPS variables as input, or query interal LAMMPS quantities to perform its computation, this means the variable can return a different value each time it is evaluated.

The type of value stored in the variable is determined by the *format* keyword of the *python* command. It can be an integer (i), floating point (f), or string (s) value. As mentioned above, if it is a numeric value (integer or floating point), then the python-style variable can be used in place of an equal-style variable anywhere in an input script, e.g. as an argument to another command that allows for equal-style variables.

A python-style variable can also be used within the formula for an equal-style or atom-style formula in a Python function wrapper, as explained below for variable formulas. In this context, the usage syntax is py_varname(arg1,arg2,...), where varname is the name of the python-style variable. When a Python wrapper function is used in an atom-style formula, it can be invoked once per atom using arguments specific to each atom. The resulting values in the atom-style variable can thus be calculated by Python code.

---

For the *string* style, a single string is assigned to the variable. Two differences between this style and using the *index* style exist: a variable with *string* style can be redefined, e.g. by another command later in the input script, or if the script is read again in a loop. The other difference is that *string* performs variable substitution even if the string parameter is quoted.

---

The *uloop* style is identical to the *universe* style except that the strings are the integers from 1 to N. This allows generation of long list of runs (e.g. 1000) without having to list N strings in the input script.

---

For the *universe* style, one or more strings are specified. There must be at least as many strings as there are processor partitions or "worlds". LAMMPS can be run with multiple partitions via the *-partition command-line switch*. This variable command initially assigns one string to each world. When a *next* command is encountered using this variable, the first processor partition to encounter it, is assigned the next available string. This continues until all the variable strings are consumed. Thus, this command can be used to run 50 simulations on 8 processor partitions. The simulations will be run one after the other on whatever partition becomes available, until they are all finished. Universe-style

variables are incremented using the files "tmp.lammps.variable" and "tmp.lammps.variable.lock" which you will see in your directory during such a LAMMPS run.

---

For the *world* style, one or more strings are specified. There must be one string for each processor partition or "world". LAMMPS can be run with multiple partitions via the *-partition command-line switch*. This variable command assigns one string to each world. All MPI processes in the world are assigned the same string. The next command cannot be used with equal-style variables, since there is only one value per world. This style of variable is useful when you wish to run different simulations on different partitions, or when performing a parallel tempering simulation (see the *temper* command), to assign different temperatures to different partitions.

---

For the *equal* and *vector* and *atom* styles, a single string is specified which represents a formula that will be evaluated afresh each time the variable is used. If you want spaces in the string, enclose it in double quotes so the parser will treat it as a single argument. For *equal*-style variables the formula computes a scalar quantity, which becomes the value of the variable whenever it is evaluated. For *vector*-style variables the formula must compute a vector of quantities, which becomes the value of the variable whenever it is evaluated. The calculated vector can be of length one, but it cannot be a simple scalar value like that produced by an equal-style compute. I.e. the formula for a vector-style variable must have at least one quantity in it that refers to a global vector produced by a compute, fix, or other vector-style variable. For *atom*-style variables the formula computes one quantity for each atom whenever it is evaluated.

Note that *equal*, *vector*, and *atom* variables can produce different values at different stages of the input script or at different times during a run. For example, if an *equal* variable is used in a *fix print* command, different values could be printed each timestep it was invoked. If you want a variable to be evaluated immediately, so that the result is stored by the variable instead of the string, see the section below on "Immediate Evaluation of Variables".

The next command cannot be used with *equal* or *vector* or *atom* style variables, since there is only one string.

The formula for an *equal*, *vector*, or *atom* variable can contain a variety of quantities. The syntax for each kind of quantity is simple, but multiple quantities can be nested and combined in various ways to build up formulas of arbitrary complexity. For example, this is a valid (though strange) variable formula:

```
variable x equal "pe + c_MyTemp / vol^(1/3)"
```

Specifically, a formula can contain numbers, constants, thermo keywords, math operators, math functions, group functions, region functions, special functions, feature functions, Python function wrappers, atom values, atom vectors, custom atom properties, compute references, fix references, and references to other variables.

| | |
|---|---|
| Number | 0.2, 100, 1.0e20, -15.4, etc |
| Constant | PI, version, on, off, true, false, yes, no |
| Thermo keywords | vol, pe, ebond, etc |
| Math operators | (), -x, x+y, x-y, x*y, x/y, x^y, x%y, x == y, x != y, x < y, x <= y, x > y, x >= y, x && y, x \|\| y, x \|^ y, !x |
| Math functions | sqrt(x), exp(x), ln(x), log(x), abs(x), sign(x), sin(x), cos(x), tan(x), asin(x), acos(x), atan(x), atan2(y,x), random(x,y,z), normal(x,y,z), ceil(x), floor(x), round(x), ternary(x,y,z), ramp(x,y), stagger(x,y), logfreq(x,y,z), logfreq2(x,y,z), logfreq3(x,y,z), stride(x,y,z), stride2(x,y,z,a,b,c), vdisplace(x,y), swiggle(x,y,z), cwiggle(x,y,z) |
| Group functions | count(ID), mass(ID), charge(ID), xcm(ID,dim), vcm(ID,dim), fcm(ID,dim), bound(ID,dir), gyration(ID), ke(ID), angmom(ID,dim), torque(ID,dim), inertia(ID,dimdim), omega(ID,dim) |
| Region functions | count(ID,IDR), mass(ID,IDR), charge(ID,IDR), xcm(ID,dim,IDR), vcm(ID,dim,IDR), fcm(ID,dim,IDR), bound(ID,dir,IDR), gyration(ID,IDR), ke(ID,IDR), angmom(ID,dim,IDR), torque(ID,dim,IDR), inertia(ID,dimdim,IDR), omega(ID,dim,IDR) |
| Special functions | sum(x), min(x), max(x), ave(x), trap(x), slope(x), sort(x), rsort(x), gmask(x), rmask(x), grmask(x,y), next(x), is_file(name), is_os(name), extract_setting(name), label2type(kind,label), is_typelabel(kind,label), is_timeout() |
| Feature functions | is_available(category,feature), is_active(category,feature), is_defined(category,id) |
| Python func wrapper | py_varname(x,y,z,...) |
| Atom values | id[i], mass[i], type[i], mol[i], x[i], y[i], z[i], vx[i], vy[i], vz[i], fx[i], fy[i], fz[i], q[i] |
| Atom vectors | id, mass, type, mol, x, y, z, vx, vy, vz, fx, fy, fz, q |
| Custom atom properties | i_name, d_name, i_name[i], d_name[i], i2_name[i], d2_name[i], i2_name[i][j], d_name[i][j] |
| Compute references | c_ID, c_ID[i], c_ID[i][j], C_ID, C_ID[i] |
| Fix references | f_ID, f_ID[i], f_ID[i][j], F_ID, F_ID[i] |
| Other variables | v_name, v_name[i] |

Most of the formula elements produce a scalar value. Some produce a global or per-atom vector of values. Global vectors can be produced by computes or fixes or by other vector-style variables. Per-atom vectors are produced by atom vectors, computes or fixes which output a per-atom vector or array, and variables that are atom-style variables. Math functions that operate on scalar values produce a scalar value; math function that operate on global or per-atom vectors

do so element-by-element and produce a global or per-atom vector.

A formula for equal-style variables cannot use any formula element that produces a global or per-atom vector. A formula for a vector-style variable can use formula elements that produce either a scalar value or a global vector value, but cannot use a formula element that produces a per-atom vector. A formula for an atom-style variable can use formula elements that produce either a scalar value or a per-atom vector, but not one that produces a global vector.

Atom-style variables are evaluated by other commands that define a *group* on which they operate, e.g. a *dump* or *compute* or *fix* command. When they invoke the atom-style variable, only atoms in the group are included in the formula evaluation. The variable evaluates to 0.0 for atoms not in the group.

---

### Numbers, constants, and thermo keywords

Numbers can contain digits, scientific notation (3.0e20,3.0e-20,3.0E20,3.0E-20), and leading minus signs.

Constants are set at compile time and cannot be changed. *PI* will return the number 3.14159265358979323846; *on*, *true* or *yes* will return 1.0; *off*, *false* or *no* will return 0.0; *version* will return a numeric version code of the current LAMMPS version (e.g. version 2 Sep 2015 will return the number 20150902). The corresponding value for newer versions of LAMMPS will be larger, for older versions of LAMMPS will be smaller. This can be used to have input scripts adapt automatically to LAMMPS versions, when non-backwards compatible syntax changes are introduced. Here is an illustrative example (which will not work, since the *version* has been introduced more recently):

```
if $(version<20140513) then "communicate vel yes" else "comm_modify vel yes"
```

The thermo keywords allowed in a formula are those defined by the *thermo_style custom* command. Thermo keywords that require a *compute* to calculate their values such as "temp" or "press", use computes stored and invoked by the *thermo_style* command. This means that you can only use those keywords in a variable if the style you are using with the thermo_style command (and the thermo keywords associated with that style) also define and use the needed compute. Note that some thermo keywords use a compute indirectly to calculate their value (e.g. the enthalpy keyword uses temp, pe, and pressure). If a variable is evaluated directly in an input script (not during a run), then the values accessed by the thermo keyword must be current. See the discussion below about "Variable Accuracy".

---

### Math Operators

Math operators are written in the usual way, where the "x" and "y" in the examples can themselves be arbitrarily complex formulas, as in the examples above. In this syntax, "x" and "y" can be scalar values or per-atom vectors. For example, "ke/natoms" is the division of two scalars, where "vy+vz" is the element-by-element sum of two per-atom vectors of y and z velocities.

Operators are evaluated left to right and have the usual C-style precedence: unary minus and unary logical NOT operator "!" have the highest precedence, exponentiation "^" is next; multiplication and division and the modulo operator "%" are next; addition and subtraction are next; the 4 relational operators "<", "<=", ">", and ">=" are next; the two remaining relational operators "==" and "!=" are next; then the logical AND operator "&&"; and finally the logical OR operator "||" and logical XOR (exclusive or) operator "|^" have the lowest precedence. Parenthesis can be used to group one or more portions of a formula and/or enforce a different order of evaluation than what would occur with the default precedence.

**Note**

Because a unary minus is higher precedence than exponentiation, the formula "-2^2" will evaluate to 4, not -4. This convention is compatible with some programming languages, but not others. As mentioned, this behavior can be easily overridden with parenthesis; the formula "-(2^2)" will evaluate to -4.

The 6 relational operators return either a 1.0 or 0.0 depending on whether the relationship between x and y is TRUE or FALSE. For example the expression x<10.0 in an atom-style variable formula will return 1.0 for all atoms whose x-coordinate is less than 10.0, and 0.0 for the others. The logical AND operator will return 1.0 if both its arguments are non-zero, else it returns 0.0. The logical OR operator will return 1.0 if either of its arguments is non-zero, else it returns 0.0. The logical XOR operator will return 1.0 if one of its arguments is zero and the other non-zero, else it returns 0.0. The logical NOT operator returns 1.0 if its argument is 0.0, else it returns 0.0.

These relational and logical operators can be used as a masking or selection operation in a formula. For example, the number of atoms whose properties satisfy one or more criteria could be calculated by taking the returned per-atom vector of ones and zeroes and passing it to the *compute reduce* command.

---

### Math Functions

Math functions are specified as keywords followed by one or more parenthesized arguments "x", "y", "z", each of which can themselves be arbitrarily complex formulas. In this syntax, the arguments can represent scalar values or global vectors or per-atom vectors. In the latter case, the math operation is performed on each element of the vector. For example, "sqrt(natoms)" is the sqrt() of a scalar, where "sqrt(y*z)" yields a per-atom vector with each element being the sqrt() of the product of one atom's y and z coordinates.

Most of the math functions perform obvious operations. The ln() is the natural log; log() is the base 10 log.

Added in version 4Feb2025.

The sign(x) function returns 1.0 if the value is greater than or equal to 0.0, and -1.0 otherwise.

The random(x,y,z) function takes 3 arguments: x = lo, y = hi, and z = seed. It generates a uniform random number between lo and hi. The normal(x,y,z) function also takes 3 arguments: x = mu, y = sigma, and z = seed. It generates a Gaussian variate centered on mu with variance sigma^2. In both cases the seed is used the first time the internal random number generator is invoked, to initialize it. For equal-style and vector-style variables, every processor uses the same seed so that they each generate the same sequence of random numbers. For atom-style variables, a unique seed is created for each processor, based on the specified seed. This effectively generates a different random number for each atom being looped over in the atom-style variable.

**Note**

Internally, there is just one random number generator for all equal-style and vector-style variables and another one for all atom-style variables. If you define multiple variables (of each style) which use the random() or normal() math functions, then the internal random number generators will only be initialized once, which means only one of the specified seeds will determine the sequence of generated random numbers.

The ceil(), floor(), and round() functions are those in the C math library. Ceil() is the smallest integer not less than its argument. Floor() if the largest integer not greater than its argument. Round() is the nearest integer to its argument.

Added in version 7Feb2024.

Changed in version 22Jul2025: Evaluate only selected argument

The ternary(x,y,z) function is the equivalent of the ternary operator (? and :) in C or C++. It takes 3 arguments. The first argument is a conditional. The result of the function is y if x evaluates to true (non-zero). The result is z if x

evaluates to false (zero). Same as in C or C++ only the selected argument y or z is evaluated, so this function can be used to protect against crashes from evaluating invalid arguments, e.g. in the following example where "qval" is a variable with an expression that may become (slightly) negative due to floating-point math limitations.:

```
variable sqrtofqval equal "ternary(v_qval >= 0, sqrt(v_qval), 0.0)"
```

The ramp(x,y) function uses the current timestep to generate a value linearly interpolated between the specified x,y values over the course of a run, according to this formula:

```
value = x + (y-x) * (timestep-startstep) / (stopstep-startstep)
```

The run begins on startstep and ends on stopstep. Startstep and stopstep can span multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this. If called in between runs or during a *run 0* command, the ramp(x,y) function will return the value of x.

The stagger(x,y) function uses the current timestep to generate a new timestep. X,y > 0 and x > y are required. The generated timesteps increase in a staggered fashion, as the sequence x,x+y,2x,2x+y,3x,3x+y,etc. For any current timestep, the next timestep in the sequence is returned. Thus if stagger(1000,100) is used in a variable by the *dump_modify every* command, it will generate the sequence of output timesteps:

```
100,1000,1100,2000,2100,3000,etc
```

The logfreq(x,y,z) function uses the current timestep to generate a new timestep. X,y,z > 0 and y < z are required. The generated timesteps are on a base-z logarithmic scale, starting with x, and the y value is how many of the z-1 possible timesteps within one logarithmic interval are generated. I.e. the timesteps follow the sequence x,2x,3x,...y*x,x*z,2x*z,3x*z,...y*x*z,x*z^2,2x*z^2,etc. For any current timestep, the next timestep in the sequence is returned. Thus if logfreq(100,4,10) is used in a variable by the *dump_modify every* command, it will generate this sequence of output timesteps:

```
100,200,300,400,1000,2000,3000,4000,10000,20000,etc
```

The logfreq2(x,y,z) function is similar to logfreq, except a single logarithmic interval is divided into y equally-spaced timesteps and all of them are output. Y < z is not required. Thus, if logfreq2(100,18,10) is used in a variable by the *dump_modify every* command, then the interval between 100 and 1000 is divided as 900/18 = 50 steps, and it will generate the sequence of output timesteps:

```
100,150,200,...950,1000,1500,2000,...9500,10000,15000,etc
```

The logfreq3(x,y,z) function generates y points between x and z (inclusive), that are separated by a multiplicative ratio: (z/x)^(1/(y-1)). Constraints are: x,z > 0, y > 1, z-x >= y-1. For eg., if logfreq3(10,25,1000) is used in a variable by the *fix print* command, then the interval between 10 and 1000 is divided into 24 parts with a multiplicative separation of ~1.21, and it will generate the following sequence of output timesteps:

```
10, 13, 15, 18, 22, 27, 32,...384, 465, 563, 682, 826, 1000
```

The stride(x,y,z) function uses the current timestep to generate a new timestep. X,y >= 0 and z > 0 and x <= y are required. The generated timesteps increase in increments of z, from x to y, i.e. it generates the sequence x,x+z,x+2z,...,y. If y-x is not a multiple of z, then similar to the way a for loop operates, the last value will be one that does not exceed y. For any current timestep, the next timestep in the sequence is returned. Thus if stride(1000,2000,100) is used in a variable by the *dump_modify every* command, it will generate the sequence of output timesteps:

```
1000,1100,1200, ... ,1900,2000
```

The stride2(x,y,z,a,b,c) function is similar to the stride() function except it generates two sets of strided timesteps, one at a coarser level and one at a finer level. Thus it is useful for debugging, e.g. to produce output every timestep at the point in simulation when a problem occurs. X,y >= 0 and z > 0 and x <= y are required, as are a,b >= 0 and c > 0 and a < b. Also, a >= x and b <= y are required so that the second stride is inside the first. The generated timesteps increase

in increments of z, starting at x, until a is reached. At that point the timestep increases in increments of c, from a to b, then after b, increments by z are resumed until y is reached. For any current timestep, the next timestep in the sequence is returned. Thus if stride2(1000,2000,100,1350,1360,1) is used in a variable by the *dump_modify every* command, it will generate the sequence of output timesteps:

```
1000,1100,1200,1300,1350,1351,1352, ... 1359,1360,1400,1500, ... ,2000
```

The vdisplace(x,y) function takes 2 arguments: x = value0 and y = velocity, and uses the elapsed time to change the value by a linear displacement due to the applied velocity over the course of a run, according to this formula:

```
value = value0 + velocity*(timestep-startstep)*dt
```

where dt = the timestep size.

The run begins on startstep. Startstep can span multiple runs, using the *start* keyword of the *run* command. See the *run* command for details of how to do this. Note that the *thermo_style* keyword elaplong = timestep-startstep. If used between runs this function will return the value according to the end of the last run or the value of x if used before *any* runs. This function assumes the length of the time step does not change and thus may not be used in combination with *fix dt/reset*.

The swiggle(x,y,z) and cwiggle(x,y,z) functions each take 3 arguments: x = value0, y = amplitude, z = period. They use the elapsed time to oscillate the value by a sin() or cos() function over the course of a run, according to one of these formulas, where omega = 2 PI / period:

```
value = value0 + Amplitude * sin(omega*(timestep-startstep)*dt)
value = value0 + Amplitude * (1 - cos(omega*(timestep-startstep)*dt))
```

where dt = the timestep size.

The run begins on startstep. Startstep can span multiple runs, using the *start* keyword of the *run* command. See the *run* command for details of how to do this. Note that the *thermo_style* keyword elaplong = timestep-startstep. If used between runs these functions will return the value according to the end of the last run or the value of x if used before *any* runs. These functions assume the length of the time step does not change and thus may not be used in combination with *fix dt/reset*.

---

### Group and Region Functions

Group functions are specified as keywords followed by one or two parenthesized arguments. The first argument *ID* is the group-ID. The *dim* argument, if it exists, is *x* or *y* or *z*. The *dir* argument, if it exists, is *xmin*, *xmax*, *ymin*, *ymax*, *zmin*, or *zmax*. The *dimdim* argument, if it exists, is *xx* or *yy* or *zz* or *xy* or *yz* or *xz*.

The group function count() is the number of atoms in the group. The group functions mass() and charge() are the total mass and charge of the group. Xcm() and vcm() return components of the position and velocity of the center of mass of the group. Fcm() returns a component of the total force on the group of atoms. Bound() returns the min/max of a particular coordinate for all atoms in the group. Gyration() computes the radius-of-gyration of the group of atoms. See the *compute gyration* command for a definition of the formula. Angmom() returns components of the angular momentum of the group of atoms around its center of mass. Torque() returns components of the torque on the group of atoms around its center of mass, based on current forces on the atoms. Inertia() returns one of 6 components of the symmetric inertia tensor of the group of atoms around its center of mass, ordered as Ixx,Iyy,Izz,Ixy,Iyz,Ixz. Omega() returns components of the angular velocity of the group of atoms around its center of mass.

Region functions are specified exactly the same way as group functions except they take an extra final argument *IDR* which is the region ID. The function is computed for all atoms that are in both the group and the region. If the group is "all", then the only criteria for atom inclusion is that it be in the region.

---

### Special Functions

Special functions take specific kinds of arguments, meaning their arguments cannot be formulas themselves.

The sum(x), min(x), max(x), ave(x), trap(x), slope(x), sort(x), and rsort(x) functions each take 1 argument which is of the form "c_ID" or "c_ID[N]" or "f_ID" or "f_ID[N]" or "v_name". The first two are computes and the second two are fixes; the ID in the reference should be replaced by the ID of a compute or fix defined elsewhere in the input script. The compute or fix must produce either a global vector or array. If it produces a global vector, then the notation without "[N]" should be used. If it produces a global array, then the notation with "[N]" should be used, where N is an integer, to specify which column of the global array is being referenced. The last form of argument "v_name" is for a vector-style variable where "name" is replaced by the name of the variable.

The sum(x), min(x), max(x), ave(x), trap(x), and slope(x) functions operate on a global vector of inputs and reduce it to a single scalar value. This is analogous to the operation of the *compute reduce* command, which performs similar operations on per-atom and local vectors.

The sort(x) and rsort(x) functions operate on a global vector of inputs and return a global vector of the same length.

The sum() function calculates the sum of all the vector elements. The min() and max() functions find the minimum and maximum element respectively. The ave() function is the same as sum() except that it divides the result by the length of the vector.

The trap() function is the same as sum() except the first and last elements are multiplied by a weighting factor of 1/2 when performing the sum. This effectively implements an integration via the trapezoidal rule on the global vector of data. I.e. consider a set of points, equally spaced by 1 in their x coordinate: (1,V1), (2,V2), ..., (N,VN), where the Vi are the values in the global vector of length N. The integral from 1 to N of these points is trap(). When appropriately normalized by the timestep size, this function is useful for calculating integrals of time-series data, like that generated by the *fix ave/correlate* command.

The slope() function uses linear regression to fit a line to the set of points, equally spaced by 1 in their x coordinate: (1,V1), (2,V2), ..., (N,VN), where the Vi are the values in the global vector of length N. The returned value is the slope of the line. If the line has a single point or is vertical, it returns 1.0e20.

Added in version 27June2024.

The sort(x) and rsort(x) functions sort the data of the input vector by their numeric value: sort(x) sorts in ascending order, rsort(x) sorts in descending order.

The gmask(x) function takes 1 argument which is a group ID. It can only be used in atom-style variables. It returns a 1 for atoms that are in the group, and a 0 for atoms that are not.

The rmask(x) function takes 1 argument which is a region ID. It can only be used in atom-style variables. It returns a 1 for atoms that are in the geometric region, and a 0 for atoms that are not.

The grmask(x,y) function takes 2 arguments. The first is a group ID, and the second is a region ID. It can only be used in atom-style variables. It returns a 1 for atoms that are in both the group and region, and a 0 for atoms that are not in both.

The next(x) function takes 1 argument which is a variable ID (not "v_foo", just "foo"). It must be for a file-style or atomfile-style variable. Each time the next() function is invoked (i.e. each time the equal-style or atom-style variable is evaluated), the following steps occur.

For file-style variables, the current string value stored by the file-style variable is converted to a numeric value and returned by the function. And the next string value in the file is read and stored. Note that if the line previously read from the file was not a numeric string, then it will typically evaluate to 0.0, which is likely not what you want.

For atomfile-style variables, the current per-atom values stored by the atomfile-style variable are returned by the function. And the next set of per-atom values in the file is read and stored.

Since file-style and atomfile-style variables read and store the first line of the file or first set of per-atoms values when they are defined in the input script, these are the value(s) that will be returned the first time the next() function is invoked.

If next() is invoked more times than there are lines or sets of lines in the file, the variable is deleted, similar to how the *next* command operates.

The is_file(name) function is a test whether *name* is a (readable) file and returns 1 in this case, otherwise it returns 0. For that *name* is taken as a literal string and must not have any blanks in it.

The is_os(name) function is a test whether *name* is part of the OS information that LAMMPS collects and provides in the *platform::os_info()* function. The argument *name* is interpreted as a regular expression as documented for the *utils::strmatch()* function. This allows to adapt LAMMPS inputs to the OS it runs on:

```
if $(is_os(^Windows)) then &
  "shell copy ${input_dir}\some_file.txt ." &
else &
  "shell cp ${input_dir}/some_file.txt ."
```

The extract_setting(name) function enables access to basic settings for the LAMMPS executable and the running simulation via calling the *lammps_extract_setting()* library function. For example, the number of processors (MPI ranks) being used by the simulation or the MPI process ID (for this processor) can be queried, or the number of atom types, bond types and so on. For the full list of available keywords *name* and their meaning, see the documentation for extract_setting() via the link in this paragraph.

The label2type(kind,label) function converts type labels into numeric types, using label maps created by the *labelmap* or *read_data* commands. The first argument is the label map kind (atom, bond, angle, dihedral, or improper) and the second argument is the label. The function returns the corresponding numeric type or triggers an error if the queried label does not exist.

Added in version 15Jun2023.

The is_typelabel(kind,label) function has the same arguments as label2type(), but returns 1 if the type label has been assigned, otherwise it returns 0. This function can be used to check if a particular type label already exists in the simulation.

Added in version 29Aug2024.

The is_timeout() function returns 1 when the *timer timeout* has expired otherwise it returns 0. This function can be used to check inputs in combination with the *if command* to execute commands after the timer has expired. Example:

```
variable timeout equal is_timeout()
timer timeout 0:10:00 every 10
run 10000
if ${timeout} then "print 'Timer has expired'"
```

---

### Feature Functions

Feature functions allow probing of the running LAMMPS executable for whether specific features are available, active, or defined. All 3 of the functions take two arguments, a *category* and a category-specific second argument. Both are strings and thus cannot be formulas themselves; only $-style immediate variable expansion is possible. The return value of the functions is either 1.0 or 0.0 depending on whether the function evaluates to true or false, respectively.

The *is_available(category,name)* function queries whether a specific feature is available in the LAMMPS executable that is being run, i.e whether it was included or enabled at compile time.

This supports the following categories: *command*, *compute*, *fix*, *pair_style* and *feature*. For all the categories except *feature* the *name* is a style name, e.g. *nve* for the *fix* category. Note that many LAMMPS input script commands such as *create_atoms* are actually instances of a command style which LAMMPS defines, as opposed to built-in commands. For all of these styles except *command*, appending of active suffixes is also tried before reporting failure.

The *feature* category checks the availability of the following compile-time enabled features: GZIP support, PNG support, JPEG support, FFMPEG support, and C++ exceptions for error handling. Corresponding names are *gzip*, *png*, *jpeg*, *ffmpeg* and *exceptions*.

Example: Only dump in a given format if the compiled binary supports it.

```
if "$(is_available(feature,png))" then "print 'PNG supported'" else "print 'PNG not␣
↪supported'"
if "$(is_available(feature,ffmpeg)" then "dump 3 all movie 25 movie.mp4 type type zoom 1.
↪6 adiam 1.0"
```

The *is_active(category,feature)* function queries whether a specific feature is currently active within LAMMPS. The features are grouped by categories. Supported categories and features are:

- *package*: features = *gpu* or *intel* or *kokkos* or *omp*
- *newton*: features = *pair* or *bond* or *any*
- *pair*: features = *single* or *respa* or *manybody* or *tail* or *shift*
- *comm_style*: features = *brick* or *tiled*
- *min_style*: features = a minimizer style name
- *run_style*: features = a run style name
- *atom_style*: features = an atom style name
- *pair_style*: features = a pair style name
- *bond_style*: features = a bond style name
- *angle_style*: features = an angle style name
- *dihedral_style*: features = a dihedral style name
- *improper_style*: features = an improper style name
- *kspace_style*: features = a kspace style name

Most of the settings are self-explanatory. For the *package* category, a package may have been included in the LAMMPS build, but not have enabled by any input script command, and hence be inactive. The *single* feature in the *pair* category checks whether the currently defined pair style supports a Pair::single() function as needed by compute group/group and others features or LAMMPS. Similarly, the *respa* feature checks whether the inner/middle/outer mode of r-RESPA is supported by the current pair style.

For the categories with *style* in their name, only a single instance of the style is ever active at any time in a LAMMPS simulation. Thus the check is whether the currently active style matches the specified name. This check is also done using suffix flags, if available and enabled.

Example 1: Disable use of suffix for PPPM when using GPU package (i.e. run it on the CPU concurrently while running the pair style on the GPU), but do use the suffix otherwise (e.g. with OPENMP).

```
pair_style lj/cut/coul/long 14.0
if $(is_active(package,gpu)) then "suffix off"
kspace_style pppm
```

Example 2: Use r-RESPA with inner/outer cutoff, if supported by the current pair style, otherwise fall back to using r-RESPA with simply the pair keyword and reducing the outer time step.

```
timestep $(2.0*(1.0+2.0*is_active(pair,respa)))
if $(is_active(pair,respa)) then "run_style respa 4 3 2 2 improper 1 inner 2 5.5 7.0␣
↪outer 3 kspace 4" else "run_style respa 3 3 2 improper 1 pair 2 kspace 3"
```

The *is_defined(category,id)* function checks whether an instance of a style or variable with a specific ID or name is currently defined within LAMMPS. The supported categories are *compute*, *dump*, *fix*, *group*, *region*, and *variable*. Each of these styles (as well as the variable command) can be specified multiple times within LAMMPS, each with a unique *id*. This function checks whether the specified *id* exists. For category *variable", the *id* is the variable name.

---

### Python Function wrapper

A Python function wrapper enables the formula for an equal-style or atom-style variable to invoke functions coded in Python. In the case of an equal-style variable, the Python-coded function will be invoked once. In the case of an atom-style variable, it can be invoked once per atom, if one or more of its arguments include a per-atom quantity, e.g. the position of an atom. As illustrated below, the reason to use a Python function wrapper is to make it easy to pass LAMMPS-related arguments to the Python-coded function associated with a python-style variable.

The syntax for defining a Python function wrapper is

```
py_varname(arg1,arg2,...argN)
```

where *varname* is the name of a python-style variable which couples to a Python-coded function. The function will be passed the zero or more arguments listed in parentheses: *arg1*, *arg2*, ... *argN*. As with Math Functions, each argument can itself be an arbitrarily complex formula.

A Python function wrapper can be used in the following manner by an input script:

```
variable        foo python truncate
python          truncate return v_foo input 1 v_arg format fi here """
def truncate(x):
 return int(x)
"""
variable        xtrunc atom py_foo(x)
variable        ytrunc atom py_foo(y)
variable        ztrunc atom py_foo(z)
dump            1 all custom 100 tmp.dump id x y z v_xtrunc v_ytrunc v_ztrunc
```

The first two commands define a python-style variable *foo* and couple it to the Python-coded function *truncate()* which takes a single floating point argument, and returns its truncated integer value. In this case, the Python code for truncate() is included in the *python* command; it could also be contained in a file. See the *python* command doc page for details.

The next three commands define atom-style variables *xtrunc*, *ytrunc*, and *ztrunc*. Each of them include the same Python function wrapper in their formula, with a different argument. The atom-style variable *xtrunc* will invoke the python-style variable *foo*, which will in turn invoke the Python-coded *truncate()* method. Because *xtrunc* is an atom-style variable, and the argument *x* in the Python function wrapper is a per-atom quantity (the x-coord of each atom), each processor will invoke the *truncate()* method once per atom, for the atoms it owns.

When invoked for the Ith atom, the value of the *arg* internal-style variable, defined by the *python* command, is set to the x-coord of the Ith atom. The call via python-style variable *foo* to the Python *truncate()* function passes the value of the *arg* variable as the function's first (and only) argument. Likewise, the return value of the Python function is stored by the python-style variable *foo* and used in the *xtrunc* atom-style variable formula for the Ith atom.

The resulting per-atom vector for *xtrunc* will thus contain the truncated x-coord of every atom in the system. The dump command includes the truncated xyz coords for each atom in its output.

See the *python* command for more details on options the *python* command can specify as well as examples of more complex Python functions which can be wrapped in this manner. In particular, the Python function can take a variety of arguments, some generated by the *python* command, and others by the arguments of the Python function wrapper.

---

### Atom Values and Vectors

Atom values take an integer argument I from 1 to N, where I is the atom-ID, e.g. x[243], which means use the x coordinate of the atom with ID = 243. Or they can take a variable name, specified as v_name, where name is the name of the variable, like x[v_myIndex]. The variable can be of any style except *vector* or *atom* or *atomfile* variables. The variable is evaluated and the result is expected to be numeric and is cast to an integer (i.e. 3.4 becomes 3), to use an index, which must be a value from 1 to N. Note that a "formula" cannot be used as the argument between the brackets, e.g. x[243+10] or x[v_myIndex+1] are not allowed. To do this a single variable can be defined that contains the needed formula.

Note that the 0 < atom-ID <= N, where N is the largest atom ID in the system. If an ID is specified for an atom that does not currently exist, then the generated value is 0.0.

Atom vectors generate one value per atom, so that a reference like "vx" means the x-component of each atom's velocity will be used when evaluating the variable.

The meaning of the different atom values and vectors is mostly self-explanatory. *Mol* refers to the molecule ID of an atom, and is only defined if an *atom_style* is being used that defines molecule IDs.

Note that many other atom attributes can be used as inputs to a variable by using the *compute property/atom* command and then referencing that compute.

---

### Custom atom properties

Added in version 7Feb2024.

Custom atom properties refer to per-atom integer and floating point vectors or arrays that have been added via the *fix property/atom* command. When that command is used specific names are given to each attribute which are the "name" portion of these references. References beginning with *i* and *d* refer to integer and floating point properties respectively. Per-atom vectors are referenced by *i_name* and *d_name*; per-atom arrays are referenced by *i2_name* and *d2_name*.

The various allowed references to integer custom atom properties in the variable formulas for equal-, vector-, and atom-style variables are listed in the following table. References to floating point custom atom properties are the same; just replace the leading "i" with "d".

| | | |
|---|---|---|
| equal | i_name[I] | element of per-atom vector (I = atom ID) |
| equal | i2_name[I][J] | element of per-atom array (I = atom ID) |
| | | |
| vector | i_name[I] | element of per-atom vector (I = atom ID) |
| vector | i2_name[I][J] | element of per-atom array (I = atom ID) |
| | | |
| atom | i_name | per-atom vector |
| atom | i2_name[I] | column of per-atom array |

The I and J indices in these custom atom property references can be integers or can be a variable name, specified as v_name, where name is the name of the variable. The rules for this syntax are the same as for indices in the "Atom Values and Vectors" discussion above.

---

### Compute References

Compute references access quantities calculated by a *compute*. The ID in the reference should be replaced by the ID of a compute defined elsewhere in the input script.

As discussed on the page for the *compute* command, computes can produce global, per-atom, local, and per-grid values. Only global and per-atom values can be used in a variable. Computes can also produce scalars (global only), vectors, and arrays. See the doc pages for individual computes to see what different kinds of data they produce.

An equal-style variable can only use scalar values, either from global or per-atom data. In the case of per-atom data, this would be a value for a specific atom.

A vector-style variable can use scalar values (same as for equal-style variables), or global vectors of values. The latter can also be a column of a global array.

Atom-style variables can use scalar values (same as for equal-style variables), or per-atom vectors of values. The latter can also be a column of a per-atom array.

The various allowed compute references in the variable formulas for equal-, vector-, and atom-style variables are listed in the following table:

| | | |
|---|---|---|
| equal | c_ID | global scalar |
| equal | c_ID[I] | element of global vector |
| equal | c_ID[I][J] | element of global array |
| equal | C_ID[I] | element of per-atom vector (I = atom ID) |
| equal | C_ID[I][J] | element of per-atom array (I = atom ID) |
| | | |
| vector | c_ID | global vector |
| vector | c_ID[I] | column of global array |
| | | |
| atom | c_ID | per-atom vector |
| atom | c_ID[I] | column of per-atom array |

Note that if an equal-style variable formula wishes to access per-atom data from a compute, it must use capital "C" as the ID prefix and not lower-case "c".

Also note that if a vector- or atom-style variable formula needs to access a scalar value from a compute (i.e. the 5 kinds of values in the first 5 lines of the table), it can not do so directly. Instead, it can use a reference to an equal-style variable which stores the scalar value from the compute.

The I and J indices in these compute references can be integers or can be a variable name, specified as v_name, where name is the name of the variable. The rules for this syntax are the same as for indices in the "Atom Values and Vectors" discussion above.

If a variable containing a compute is evaluated directly in an input script (not during a run), then the values accessed by the compute should be current. See the discussion below about "Variable Accuracy".

---

### Fix References

Fix references access quantities calculated by a *fix*. The ID in the reference should be replaced by the ID of a fix defined elsewhere in the input script.

As discussed on the page for the *fix* command, fixes can produce global, per-atom, local, and per-grid values. Only global and per-atom values can be used in a variable. Fixes can also produce scalars (global only), vectors, and arrays. See the doc pages for individual fixes to see what different kinds of data they produce.

An equal-style variable can only use scalar values, either from global or per-atom data. In the case of per-atom data, this would be a value for a specific atom.

A vector-style variable can use scalar values (same as for equal-style variables), or global vectors of values. The latter can also be a column of a global array.

Atom-style variables can use scalar values (same as for equal-style variables), or per-atom vectors of values. The latter can also be a column of a per-atom array.

The allowed fix references in variable formulas for equal-, vector-, and atom-style variables are listed in the following table:

| | | |
|---|---|---|
| equal | f_ID | global scalar |
| equal | f_ID[I] | element of global vector |
| equal | f_ID[I][J] | element of global array |
| equal | F_ID[I] | element of per-atom vector (I = atom ID) |
| equal | F_ID[I][J] | element of per-atom array (I = atom ID) |
| | | |
| vector | f_ID | global vector |
| vector | f_ID[I] | column of global array |
| | | |
| atom | f_ID | per-atom vector |
| atom | f_ID[I] | column of per-atom array |

Note that if an equal-style variable formula wishes to access per-atom data from a fix, it must use capital "F" as the ID prefix and not lower-case "f".

Also note that if a vector- or atom-style variable formula needs to access a scalar value from a fix (i.e. the 5 kinds of values in the first 5 lines of the table), it can not do so directly. Instead, it can use a reference to an equal-style variable which stores the scalar value from the fix.

The I and J indices in these fix references can be integers or can be a variable name, specified as v_name, where name is the name of the variable. The rules for this syntax are the same as for indices in the "Atom Values and Vectors" discussion above.

Note that some fixes only generate quantities on certain timesteps. If a variable attempts to access the fix on non-allowed timesteps, an error is generated. For example, the *fix ave/time* command may only generate averaged quantities every 100 steps. See the doc pages for individual fix commands for details.

If a variable containing a fix is evaluated directly in an input script (not during a run), then the values accessed by the fix should be current. See the discussion below about "Variable Accuracy".

---

### Variable References

Variable references access quantities stored or calculated by other variables, which will cause those variables to be evaluated. The name in the reference should be replaced by the name of a variable defined elsewhere in the input script.

As discussed on this doc page, equal-style variables generate a single global numeric value, vector-style variables generate a vector of global numeric values, and atom-style and atomfile-style variables generate a per-atom vector of numeric values. All other variables store one or more strings.

The formula for an equal-style variable can use any style of variable including a vector_style or atom-style or atomfile-style. For these 3 styles, a subscript must be used to access a single value from the vector-, atom-, or atomfile-style variable. If a string-storing variable is used, the string is converted to a numeric value. Note that this will typically produce a 0.0 if the string is not a numeric string, which is likely not what you want.

The formula for a vector-style variable can use any style of variable, including atom-style or atomfile-style variables. For these 2 styles, a subscript must be used to access a single value from the atom-, or atomfile-style variable.

The formula for an atom-style variable can use any style of variable, including other atom-style or atomfile-style variables. If it uses a vector-style variable, a subscript must be used to access a single value from the vector-style variable.

The allowed variable references in variable formulas for equal-, vector-, and atom-style variables are listed in the following table. Note that there is no ambiguity as to what a reference means, since referenced variables produce only a global scalar or global vector or per-atom vector.

| | | |
|---|---|---|
| equal | v_name | global scalar from an equal-style variable |
| equal | v_name[I] | element of global vector from a vector-style variable |
| equal | v_name[I] | element of per-atom vector (I = atom ID) from an atom- or atomfile-style variable |
| | | |
| vector | v_name | global scalar from an equal-style variable |
| vector | v_name | global vector from a vector-style variable |
| vector | v_name[I] | element of global vector from a vector-style variable |
| vector | v_name[I] | element of per-atom vector (I = atom ID) from an atom- or atomfile-style variable |
| | | |
| atom | v_name | global scalar from an equal-style variable |
| atom | v_name | per-atom vector from an atom-style or atomfile-style variable |
| atom | v_name[I] | element of global vector from a vector-style variable |
| atom | v_name[I] | element of per-atom vector (I = atom ID) from an atom- or atomfile-style variable |

For the I index, an integer can be specified or a variable name, specified as v_name, where name is the name of the variable. The rules for this syntax are the same as for indices in the "Atom Values and Vectors" discussion above.

---

#### Vector Initialization

Added in version 15Jun2023.

*Vector*-style variables only can be initialized with a special syntax, instead of using a formula. The syntax is a bracketed, comma-separated syntax like the following:

```
variable myvec vector [1,3.5,7,10.2]
```

The 3rd argument formula is replaced by the vector values in brackets, separated by commas. This example creates a 4-length vector with specific numeric values, each of which can be specified as an integer or floating point value. Note that while whitespace can be added before or after individual values, no other mathematical operations can be specified. E.g. "3*10" or "3*v_abc" are not valid vector elements, nor is "10*[1,2,3,4]" valid for the entire vector.

Unlike vector variables specified with formulas, this vector variable is static; its length and values never changes. Its values can be used in other commands (including vector-style variables specified with formulas) via the usual syntax for accessing individual vector elements or the entire vector.

---

### 1.115.4 Immediate Evaluation of Variables

If you want an equal-style variable to be evaluated immediately, it may be the case that you do not need to define a variable at all. See the *Commands parse* page for info on how to use "immediate" variables in an input script, specified as $(formula) with parenthesis, where the formula has the same syntax as equal-style variables described on this page. This effectively evaluates a formula immediately without using the variable command to define a named variable.

More generally, there is a difference between referencing a variable with a leading $ sign (e.g. $x or ${abc}) versus with a leading "v_" (e.g. v_x or v_abc). The former can be used in any input script command, including a variable command. The input script parser evaluates the reference variable immediately and substitutes its value into the command. As explained on the *Commands parse* doc page, you can also use un-named "immediate" variables for this purpose. For example, a string like this $((xlo+xhi)/2+sqrt(v_area)) in an input script command evaluates the string between the parenthesis as an equal-style variable formula.

Referencing a variable with a leading "v_" is an optional or required kind of argument for some commands (e.g. the *fix ave/chunk* or *dump custom* or *thermo_style* commands) if you wish it to evaluate a variable periodically during a run. It can also be used in a variable formula if you wish to reference a second variable. The second variable will be evaluated whenever the first variable is evaluated.

As an example, suppose you use this command in your input script to define the variable "v" as

```
variable v equal vol
```

before a run where the simulation box size changes. You might think this will assign the initial volume to the variable "v". That is not the case. Rather it assigns a formula which evaluates the volume (using the thermo_style keyword "vol") to the variable "v". If you use the variable "v" in some other command like *fix ave/time* then the current volume of the box will be evaluated continuously during the run.

If you want to store the initial volume of the system, you can do it this way:

```
variable v equal vol
variable v0 equal $v
```

The second command will force "v" to be evaluated (yielding the initial volume) and assign that value to the variable "v0". Thus the command

```
thermo_style custom step v_v v_v0
```

would print out both the current and initial volume periodically during the run.

Note that it is a mistake to enclose a variable formula in double quotes if it contains variables preceded by $ signs. For example,

```
variable vratio equal "${vfinal}/${v0}"
```

This is because the quotes prevent variable substitution (explained on the *Commands parse* doc page), and thus an error will occur when the formula for "vratio" is evaluated later.

---

### 1.115.5 Variable Accuracy

Obviously, LAMMPS attempts to evaluate variables which contain formulas (*equal* and *vector* and *atom* style variables) accurately whenever the evaluation is performed. Depending on what is included in the formula, this may require invoking a *compute*, either directly or indirectly via a thermo keyword, or accessing a value previously calculated by a compute, or accessing a value calculated and stored by a *fix*. If the compute is one that calculates the energy or pressure of the system, then the corresponding energy or virial quantities need to be tallied during the evaluation of the interatomic potentials (pair, bond, etc) on any timestep that the variable needs the tallies. An input script can also request variables be evaluated before or after or in between runs, e.g. by including them in a *print* command.

LAMMPS keeps track of all of this as it performs a *run* or *minimize* simulation, as well as in between simulations. An error will be generated if you attempt to evaluate a variable when LAMMPS knows it cannot produce accurate values. For example, if a *thermo_style custom* command prints a variable which accesses values stored by a *fix ave/time* command and the timesteps on which thermo output is generated are not multiples of the averaging frequency used in the fix command, then an error will occur.

However, there are two special cases to be aware when a variable requires invocation of a compute (directly or indirectly). The first is if the variable is evaluated before the first *run* or *minimize* command in the input script. In this case, LAMMPS will generate an error. This is because many computes require initializations which have not yet taken place. One example is the calculation of degrees of freedom for temperature computes. Another example are the computes mentioned above which require tallying of energy or virial quantities; these values are not tallied until the first simulation begins.

The second special case is when a variable that depends on a compute is evaluated in between *run* or *minimize* commands. It is possible for other input script commands issued following the previous run, but before the variable is evaluated, to change the system. For example, the *delete_atoms* command could be used to remove atoms. Since the compute will not re-initialize itself until the next simulation or it may depend on energy/virial computations performed before the system was changed, it will potentially generate an incorrect answer when evaluated. Note that LAMMPS will not generate an error in this case; the evaluated variable may simply be incorrect.

The way to get around both of these special cases is to perform a 0-timestep run before evaluating the variable. For example, these commands

```
# delete_atoms random fraction 0.5 yes all NULL 49839
# run 0 post no
variable t equal temp    # this thermo keyword invokes a temperature compute
print "Temperature of system = $t"
run 1000
```

will generate an error if the "run 1000" command is the first simulation in the input script. If there were a previous run, these commands will print the correct temperature of the system. But if the *delete_atoms* command is uncommented, the printed temperature will be incorrect, because information stored by temperature compute is no longer valid.

Both these issues are resolved, if the "run 0" command is uncommented. This is because the "run 0" simulation will initialize (or re-initialize) the temperature compute correctly.

---

### 1.115.6 Restrictions

Indexing any formula element by global atom ID, such as an atom value, requires the *atom style* to use a global mapping in order to look up the vector indices. By default, only atom styles with molecular information create global maps. The *atom_modify map* command can override the default, e.g. for atomic-style atom styles.

All *universe*- and *uloop*-style variables defined in an input script must have the same number of values.

### 1.115.7 Related commands

*next*, *jump*, *include*, *temper*, *fix print*, *print*

### 1.115.8 Default

none

## 1.116 velocity command

### 1.116.1 Syntax

```
velocity group-ID style args keyword value ...
```

- group-ID = ID of group of atoms whose velocity will be changed
- style = *create* or *set* or *scale* or *ramp* or *zero*

```
create args = temp seed
  temp = temperature value (temperature units)
  seed = random # seed (positive integer)
set args = vx vy vz
  vx,vy,vz = velocity value or NULL (velocity units)
  any of vx,vy,vz can be a variable (see below)
```

```
scale arg = temp
  temp = temperature value (temperature units)
ramp args = vdim vlo vhi dim clo chi
  vdim = vx or vy or vz
  vlo,vhi = lower and upper velocity value (velocity units)
  dim = x or y or z
  clo,chi = lower and upper coordinate bound (distance units)
zero arg = linear or angular
  linear = zero the linear momentum
  angular = zero the angular momentum
```

- zero or more keyword/value pairs may be appended
- keyword = *dist* or *sum* or *mom* or *rot* or *temp* or *bias* or *loop* or *rigid* or *units*

```
dist value = uniform or gaussian
sum value = no or yes
mom value = no or yes
rot value = no or yes
temp value = temperature compute ID
bias value = no or yes
loop value = all or local or geom
rigid value = fix-ID
  fix-ID = ID of rigid body fix
units value = box or lattice
```

### 1.116.2 Examples

```
velocity all create 300.0 4928459 rot yes dist gaussian
velocity border set NULL 4.0 v_vz sum yes units box
velocity flow scale 300.0
velocity flow ramp vx 0.0 5.0 y 5 25 temp mytemp
velocity all zero linear
```

### 1.116.3 Description

Set or change the velocities of a group of atoms in one of several styles. For each style, there are required arguments and optional keyword/value parameters. Not all options are used by each style. Each option has a default as listed below.

The *create* style generates an ensemble of velocities using a random number generator with the specified seed at the specified temperature.

The *set* style sets the velocities of all atoms in the group to the specified values. If any component is specified as NULL, then it is not set. Any of the vx,vy,vz velocity components can be specified as an equal-style or atom-style *variable*. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated, and its value used to determine the velocity component. Note that if a variable is used, the velocity it calculates must be in box units, not lattice units; see the discussion of the *units* keyword below.

Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters or other parameters.

Atom-style variables can specify the same formulas as equal-style variables but can also include per-atom values, such as atom coordinates. Thus it is easy to specify a spatially-dependent velocity field.

The *scale* style computes the current temperature of the group of atoms and then rescales the velocities to the specified temperature.

The *ramp* style is similar to that used by the *compute temp/ramp* command. Velocities ramped uniformly from vlo to vhi are applied to dimension vx, or vy, or vz. The value assigned to a particular atom depends on its relative coordinate value (in dim) from clo to chi. For the example above, an atom with y-coordinate of 10 (1/4 of the way from 5 to 25), would be assigned a x-velocity of 1.25 (1/4 of the way from 0.0 to 5.0). Atoms outside the coordinate bounds (less than 5 or greater than 25 in this case), are assigned velocities equal to vlo or vhi (0.0 or 5.0 in this case).

The *zero* style adjusts the velocities of the group of atoms so that the aggregate linear or angular momentum is zero. No other changes are made to the velocities of the atoms. If the *rigid* option is specified (see below), then the zeroing is performed on individual rigid bodies, as defined by the *fix rigid or fix rigid/small* commands. In other words, zero linear will set the linear momentum of each rigid body to zero, and zero angular will set the angular momentum of each rigid body to zero. This is done by adjusting the velocities of the atoms in each rigid body.

All temperatures specified in the velocity command are in temperature units; see the *units* command. The units of velocities and coordinates depend on whether the *units* keyword is set to *box* or *lattice*, as discussed below.

For all styles, no atoms are assigned z-component velocities if the simulation is 2d; see the *dimension* command.

---

The keyword/value options are used in the following ways by the various styles.

The *dist* keyword is used by *create*. The ensemble of generated velocities can be a *uniform* distribution from some minimum to maximum value, scaled to produce the requested temperature. Or it can be a *gaussian* distribution with a mean of 0.0 and a sigma scaled to produce the requested temperature.

The *sum* keyword is used by all styles, except *zero*. The new velocities will be added to the existing ones if sum = yes, or will replace them if sum = no.

The *mom* and *rot* keywords are used by *create*. If mom = yes, the linear momentum of the newly created ensemble of velocities is zeroed; if rot = yes, the angular momentum is zeroed.

---

If specified, the *temp* keyword is used by *create* and *scale* to specify a *compute* that calculates temperature in a desired way, e.g. by first subtracting out a velocity bias, as discussed on the *Howto thermostat* doc page. If this keyword is not specified, *create* and *scale* calculate temperature using a compute that is defined internally as follows:

```
compute velocity_temp group-ID temp
```

where group-ID is the same ID used in the velocity command. i.e. the group of atoms whose velocity is being altered. This compute is deleted when the velocity command is finished. See the *compute temp* command for details. If the calculated temperature should have degrees-of-freedom removed due to fix constraints (e.g. SHAKE or rigid-body constraints), then the appropriate fix command must be specified before the velocity command is issued.

The *bias* keyword with a *yes* setting is used by *create* and *scale*, but only if the *temp* keyword is also used to specify a *compute* that calculates temperature in a desired way. If the temperature compute also calculates a velocity bias, the bias is subtracted from atom velocities before the *create* and *scale* operations are performed. After the operations, the bias is added back to the atom velocities. See the *Howto thermostat* page for more discussion of temperature computes with biases. Note that the velocity bias is only applied to atoms in the temperature compute specified with the *temp* keyword.

As an example, assume atoms are currently streaming in a flow direction (which could be separately initialized with the *ramp* style), and you wish to initialize their thermal velocity to a desired temperature. In this context thermal velocity means the per-particle velocity that remains when the streaming velocity is subtracted. This can be done using the *create* style with the *temp* keyword specifying the ID of a *compute temp/ramp* or *compute temp/profile* command, and the *bias* keyword set to a *yes* value.

---

The *loop* keyword is used by *create* in the following ways.

If *loop* = *all*, then each processor loops over all atoms in the simulation to create velocities, but only stores velocities for atoms it owns. This can be a slow loop for a large simulation. If atoms were read from a data file, the velocity assigned to a particular atom will be the same, independent of how many MPI processes are being used. This will not be the case if atoms were created using the *create_atoms* command, since atom IDs will likely be assigned to atoms differently.

If *loop* = *local*, then each processor loops over only its atoms to produce velocities. The random number seed is adjusted to give a different set of velocities on each processor. This is a fast loop, but the velocity assigned to a particular atom will depend on which processor owns it. Thus the results will always be different when a simulation is run on a different number of MPI processes.

If *loop* = *geom*, then each processor loops over only its atoms. For each atom a unique random number seed is created, based on the atom's xyz coordinates. A velocity is generated using that seed. This is a fast loop and the velocity assigned to a particular atom will be the same, independent of how many MPI processes are used. However, the set of generated velocities may be more correlated than if the *all* or *local* keywords are used.

**Note**

The *loop geom* keyword will not necessarily assign identical velocities for two simulations run on different machines. This is because the computations based on xyz coordinates are sensitive to tiny differences in the double-precision value for a coordinate as stored on a particular machine.

---

The *rigid* keyword only has meaning when used with the *zero* style. It allows specification of a fix-ID for one of the *rigid-body fix* variants which defines a set of rigid bodies. The zeroing of linear or angular momentum is then performed for each rigid body defined by the fix, as described above.

The *units* keyword is used by *set* and *ramp*. If units = box, the velocities and coordinates specified in the velocity command are in the standard units described by the *units* command (e.g. Angstroms/fs for real units). If units = lattice, velocities are in units of lattice spacings per time (e.g. spacings/fs) and coordinates are in lattice spacings. The *lattice* command must have been previously used to define the lattice spacing.

---

### 1.116.4 Restrictions

Assigning a temperature via the *create* style to a system with *rigid bodies* or *SHAKE constraints* may not have the desired outcome for two reasons. First, the velocity command can be invoked before all of the relevant fixes are created and initialized and the number of adjusted degrees of freedom (DOFs) is known. Thus it is not possible to compute the target temperature correctly. Second, the assigned velocities may be partially canceled when constraints are first enforced, leading to a different temperature than desired. A workaround for this is to perform a *run 0* command, which ensures all DOFs are accounted for properly, and then rescale the temperature to the desired value before performing a simulation. For example:

```
velocity all create 300.0 12345
run 0                          # temperature may not be 300K
velocity all scale 300.0       # now it should be
```

### 1.116.5 Related commands

*fix rigid*, *fix shake*, *lattice*

### 1.116.6 Default

The keyword defaults are dist = uniform, sum = no, mom = yes, rot = no, bias = no, loop = all, and units = lattice. The temp and rigid keywords are not defined by default.

## 1.117 write_coeff command

### 1.117.1 Syntax

```
write_coeff file
```

- file = name of data file to write out

### 1.117.2 Examples

```
write_coeff polymer.coeff
```

### 1.117.3 Description

Write a text format file with the currently defined force field coefficients in a way, that it can be read by LAMMPS with the *include* command. In combination with the nocoeff option of *write_data* this can be used to move the Coeffs sections from a data file into a separate file.

**Note**

The write_coeff command is not yet fully implemented as some pair styles do not output their coefficient information. This means you will need to add/copy this information manually.

---

### 1.117.4 Restrictions

none

### 1.117.5 Related commands

*read_data*, *write_restart*, *write_data*

## 1.118 write_data command

### 1.118.1 Syntax

```
write_data file keyword value ...
```

- file = name of data file to write out
- zero or more keyword/value pairs may be appended
- keyword = *nocoeff* or *nofix* or *nolabelmap* or *triclinic/general* or *types* or *pair*

```
nocoeff = do not write out force field info
nofix = do not write out extra sections read by fixes
nolabelmap = do not write out type labels
```

```
triclinic/general = write data file in general triclinic format
types value = numeric or labels
pair value = ii or ij
  ii = write one line of pair coefficient info per atom type
  ij = write one line of pair coefficient info per IJ atom type pair
```

### 1.118.2 Examples

```
write_data data.polymer
write_data data.polymer.gz
write_data data.*
write_data data.solid triclinic/general
```

### 1.118.3 Description

Write a data file in text format of the current state of the simulation. Data files can be read by the *read data* command to begin a simulation.

Added in version 11Feb2026.

The file may also be a compressed text file (detected by its suffix) if LAMMPS has been compiled with support for *compression commands* and the corresponding compression program is available.

Similar to *dump* files, the data filename can contain a "*" wild-card character. The "*" is replaced with the current timestep value.

**Data in Coeff sections**

The write_data command may not always write all coefficient settings to the corresponding Coeff sections of the data file. This can have one of multiple reasons. 1) The style may be a hybrid style. In that case *no* coeff information is written. 2) A few styles may be missing the code that would write those sections (This is rare these days, but if you come across one, please notify the LAMMPS developers). 3) Some pair styles require a single pair_coeff statement and those are not compatible with data files. 4) The default for write_data is to write a PairCoeff section, which has only entries for atom types i == j. The remaining coefficients would be inferred through the currently selected mixing rule. If there has been a pair_coeff command with i != j, this setting would be lost. LAMMPS will detect this and print a warning message unless *pair ij* is appended to the write_data command. This will request writing a PairIJCoeff section which has information for all pairs of atom types. In cases where the coefficient data in the data file is incomplete, you will need to re-specify that information in your input script that reads the data file.

Because a data file is in text format, if you use a data file written out by this command to restart a simulation, the initial state of the new run will be slightly different than the final state of the old run (when the file was written) which was represented internally by LAMMPS in binary format. A new simulation which reads the data file will thus typically diverge from a simulation that continued in the original input script.

If you want to do more exact restarts, using binary files, see the *restart*, *write_restart*, and *read_restart* commands. You can also convert binary restart files to text data files, after a simulation has run, using the *-r command-line switch*.

**Note**

Only limited information about a simulation is stored in a data file. For example, no information about atom *groups* and *fixes* are stored. *Binary restart files* store more information.

Bond interactions (angle, etc) that have been turned off by the *fix shake* or *delete_bonds* command will be written to a data file as if they are turned on. This means they will need to be turned off again in a new run after the data file is read.

Bonds that are broken (e.g. by a bond-breaking potential) are not written to the data file. Thus these bonds will not exist when the data file is read.

---

Use of the *nocoeff* keyword means no force field parameters are written to the data file. This can be helpful, for example, if you want to make significant changes to the force field or if the force field parameters are read in separately, e.g. from an include file.

Use of the *nofix* keyword means no extra sections read by fixes are written to the data file (see the *fix* option of the *read_data* command for details). For example, this option excludes sections for user-created per-atom properties from *fix property/atom*.

The *nolabelmap* and *types* keywords refer to type labels that may be defined for numeric atom types, bond types, angle types, etc. The label map can be defined in two ways, either by the *labelmap* command or in data files read by the *read_data* command which have sections for Atom Type Labels, Bond Type Labels, Angle Type Labels, etc. See the *Howto type labels* doc page for the allowed syntax of type labels and a general discussion of how type labels can be used.

Use of the *nolabelmap* keyword means that even if type labels exist for a given type-kind (Atoms, Bonds, Angles, etc.), type labels are not written to the data file. By default, they are written if they exist. A type label must be defined for every numeric type (within a given type-kind) to be written to the data file.

Use of the *triclinic/general* keyword will output a data file which specifies a general triclinic simulation box as well as per-atom quantities consistent with the general triclinic box. The latter means that per-atom vectors, such as velocities and dipole moments will be oriented consistent with the 3d rotation implied by the general triclinic box (relative to the associated restricted triclinic box).

This option can only be requested if the simulation box was initially defined to be general triclinic. If if was and the *triclinic/general* keyword is not used, then the data file will specify a restricted triclinic box, since that is the internal format LAMMPS uses for both general and restricted triclinic simulations. See the *Howto triclinic* doc page for more explanation of how general triclinic simulation boxes are supported by LAMMPS. And see the *read_data* doc page for details of how the format is altered for general triclinic data files.

The *types* keyword determines how atom types, bond types, angle types, etc are written into these data file sections: Atoms, Bonds, Angles, etc. The default is the *numeric* setting, even if type label maps exist. If the *labels* setting is used, type labels will be written to the data file, if the corresponding label map exists. Note that when using *types labels*, the *nolabelmap* keyword cannot be used.

The *pair* keyword lets you specify in what format the pair coefficient information is written into the data file. If the value is specified as *ii*, then one line per atom type is written, to specify the coefficients for each of the I=J interactions. This means that no cross-interactions for I != J will be specified in the data file and the pair style will apply its mixing rule, as documented on individual *pair_style* doc pages. Of course this behavior can be overridden in the input script after reading the data file, by specifying additional *pair_coeff* commands for any desired I,J pairs.

If the value is specified as *ij*, then one line of coefficients is written for all I,J pairs where I <= J. These coefficients will include any specific settings made in the input script up to that point. The presence of these I != J coefficients in the data file will effectively turn off the default mixing rule for the pair style. Again, the coefficient values in the data file can be overridden in the input script after reading the data file, by specifying additional *pair_coeff* commands for any desired I,J pairs.

**Note**

Before the data file is written, LAMMPS migrates atoms to their owning subdomains, which deletes any atoms that lie outside of non-periodic boundaries. If this changes the total number of atoms, the *thermo_modify lost* setting determines what happens: with the default *error* setting LAMMPS aborts and does not write the file; with *warn* or *ignore* the stored atom count is reset to the actual number of atoms so that a self-consistent data file is written (with a warning printed for *warn*).

Changed in version 4Jul2026.

Previously, with the *warn* or *ignore* lost-atoms setting, the atom count in the data file header could disagree with the number of atoms actually written (a corrupted data file). The count is now reset so the written file is always self-consistent.

---

### 1.118.4 Restrictions

This command requires inter-processor communication to migrate atoms before the data file is written. This means that your system must be ready to perform a simulation before using this command (force fields setup, atom masses initialized, etc).

To write compressed data files, you must compile LAMMPS with the `-DLAMMPS_GZIP` option. See the *Build settings* doc page for details.

### 1.118.5 Related commands

*read_data*, *write_restart*

### 1.118.6 Default

The option defaults are pair = ii and types = numeric.

## 1.119 write_dump command

### 1.119.1 Syntax

```
write_dump group-ID style file dump-args modify dump_modify-args
```

- group-ID = ID of the group of atoms to be dumped
- style = any of the supported *dump styles*
- file = name of file to write dump info to
- dump-args = any additional args needed for a particular *dump style*
- modify = all args after this keyword are passed to *dump_modify* (optional)
- dump-modify-args = args for *dump_modify* (optional)

### 1.119.2 Examples

```
write_dump all atom dump.atom
write_dump subgroup atom dump.run.bin
write_dump all custom dump.myforce.* id type x y vx fx
write_dump flow custom dump.%.myforce id type c_myF[3] v_ke modify sort id
write_dump all xyz system.xyz modify sort id element O H
write_dump all image snap*.jpg type type size 960 960 modify backcolor white
write_dump all image snap*.jpg element element &
   bond atom 0.3 shiny 0.1 ssao yes 6345 0.2 size 1600 1600  &
   modify backcolor white element C C O H N C C C O H H S O H
write_dump all atom/gz dump.atom.gz modify compression_level 9
write_dump flow custom/zstd dump.%.myforce.zst &
               id type c_myF[3] v_ke &
               modify sort id &
               compression_level 15
```

### 1.119.3 Description

Dump a single snapshot of atom quantities to one or more files for the current state of the system. This is a one-time immediate operation, in contrast to the *dump* command which will will set up a dump style to write out snapshots periodically during a running simulation.

The syntax for this command is mostly identical to that of the *dump* and *dump_modify* commands as if they were concatenated together, with the following exceptions: There is no need for a dump ID or dump frequency and the keyword *modify* is added. The latter is so that the full range of *dump_modify* options can be specified for the single snapshot, just as they can be for multiple snapshots. The *modify* keyword separates the arguments that would normally be passed to the *dump* command from those that would be given the *dump_modify*. Both support optional arguments and thus LAMMPS needs to be able to cleanly separate the two sets of args.

Note that if the specified filename uses wildcard characters "*" or "%", as supported by the *dump* command, they will operate in the same fashion to create the new filename(s). Normally, *dump image* files require a filename with a "*" character for the timestep. That is not the case for the write_dump command; no wildcard "*" character is necessary.

---

### 1.119.4 Restrictions

All restrictions for the *dump* and *dump_modify* commands apply to this command as well, with the exception of the *dump image* filename not requiring a wildcard "*" character, as noted above.

Since dumps are normally written during a *run* or *energy minimization*, the simulation has to be ready to run before this command can be used. Similarly, if the dump requires information from a compute, fix, or variable, the information needs to have been calculated for the current timestep (e.g. by a prior run), else LAMMPS will generate an error message.

For example, it is not possible to dump per-atom energy with this command before a run has been performed, since no energies and forces have yet been calculated. See the *variable* doc page section on Variable Accuracy for more information on this topic.

### 1.119.5 Related commands

*dump*, *dump image*, *dump_modify*

### 1.119.6 Default

The defaults are listed on the doc pages for the *dump* and *dump image* and *dump_modify* commands.

## 1.120 write_molecule command

### 1.120.1 Syntax

```
write_molecule mol-ID file
```

- mol-ID = ID of the molecule template to be written
- file = name of file to write the molecule template to

### 1.120.2 Examples

```
write_molecule mol1 molecule1.mol
write_molecule mol1 molecule1.json
write_molecule twomols template_set%.mol
```

### 1.120.3 Description

Added in version 10Dec2025.

Write the data from a *molecule template* to a molecule file.

The molecule file format is determined by the file name: if the file name ends in `.json` the file will be written in JSON format, otherwise the file is written in the native LAMMPS molecule file format.

When the molecule template contains multiple molecules, as defined by a *molecule command* with multiple molecule files, the filename *must* contain a '%' character. That '%' character will be replaced by the molecule number (starting from 1) and each molecule is written to a separate file.

---

### 1.120.4 Output Format

The output format follows the description of the molecule file format in the *molecule command documentation*. When parsing molecule files there are a few requirement about the order of sections for the native format, but generally the order of list entries can be chosen freely. On writing, however, a specific ordering is enforced following the lists given below. All sections with per-atom data are sorted by the atom-ID value starting from 1. The list of bonds is ordered by the atom-ID of the first atom in the bond. The lists of angles, dihedrals, and impropers are ordered by the atom-ID of the second atom in the definition. For entries with the same atom-ID, the order in which they were defined in the original input data is maintained.

#### Native Format

The native format starts with a title line similar to the following where the angular brackets are replaced with the actual data:

```
# MOLECULE <molecule-ID>, unit = <units setting>, set <set-ID> of <nsets>, <original␣
↪title>
```

This followed by a header section with the following keywords in the listed order:

- *atoms* (required keyword)

- *bonds* (if non-zero)
- *angles* (if non-zero)
- *dihedrals* (if non-zero)
- *impropers* (if non-zero)
- *fragments* (if non-zero)
- *mass* (if present in original input)
- *body* (if present in original input)
- *com* (if present in original input)
- *inertia* (if present in original input)

The header section is followed by individual data sections in the order given below if the corresponding data was provided in the original input or explicitly added later:

- *Coords*
- *Types*
- *Molecules*
- *Fragments*
- *Charges*
- *Diameters*
- *Dipoles*
- *Masses*
- *Bonds*
- *Angles*
- *Dihedrals*
- *Impropers*
- *Special Bond Counts*
- *Special Bonds*
- *Shake Flags*
- *Shake Atoms*
- *Shake Bond Types*
- *Body Integers*
- *Body Doubles*

### JSON Format

The JSON output is written as a text file in strict JSON format with an indentation level of 2 and following the layout given in the JSON schema file available for download at https://download.lammps.org/json/molecule-schema.json

Unlike for the native format output, there is no need to provide any counts for the number of entries for keys containing lists of settings with a specified format, since those can be directly inferred from the data structures after parsing a JSON file. The top level keys are ordered as follows:

- *application*

- *format*
- *revision*
- *schema*
- *title*
- *units*
- *masstotal* (if present in original input)
- *com* (if present in original input)
- *inertia* (if present in original input)
- *coords* (if present in molecule data)
- *types*
- *molecules* (if present in molecule data)
- *fragments* (if present in molecule data)
- *charges* (if present in molecule data)
- *diameters* (if present in molecule data)
- *dipoles* (if present in molecule data)
- *masses* (if present in molecule data)
- *bonds* (if present in molecule data)
- *angles* (if present in molecule data)
- *dihedrals* (if present in molecule data)
- *impropers* (if present in molecule data)
- *special* (if present in original input)
- *shake* (if present in molecule data)
- *body* (if present in molecule data)

---

### 1.120.5 Restrictions

None

### 1.120.6 Related commands

*molecule*

### 1.120.7 Defaults

None

# 1.121 write_restart command

## 1.121.1 Syntax

```
write_restart file keyword value ...
```

- file = name of file to write restart information to
- zero or more keyword/value pairs may be appended
- keyword = *fileper* or *nfile*

```
fileper arg = Np
  Np = write one file for every this many MPI processes
nfile arg = Nf
  Nf = write this many files, one from each of Nf MPI processes
```

## 1.121.2 Examples

```
write_restart restart.equil
write_restart poly.%.* nfile 10
```

## 1.121.3 Description

Write a binary restart file of the current state of the simulation.

During a long simulation, the *restart* command is typically used to output restart files periodically. The write_restart command is useful after a minimization or whenever you wish to write out a single current restart file.

Similar to *dump* files, the restart filename can contain two wild-card characters. If a "*" appears in the filename, it is replaced with the current timestep value. If a "%" character appears in the filename, then one file is written by each processor and the "%" character is replaced with the MPI rank ID from 0 to *P*-1. An additional file with the "%" replaced by "base" is also written, which contains global information. For example, the files written for `filename restart.%` would be `restart.base`, `restart.0`, `restart.1`, ... `restart.P-1`. This creates smaller files and can be a fast mode of output and subsequent input on parallel machines that support parallel I/O. The optional *fileper* and *nfile* keywords discussed below can alter the number of files written.

Restart files can be read by a *read_restart* command to restart a simulation from a particular state. Because the file is binary (to enable exact restarts), it may not be readable on another machine. In this case, you can use the *-r command-line switch* to convert a restart file to a data file.

**Note**

Although the purpose of restart files is to enable restarting a simulation from where it left off, not all information about a simulation is stored in the file. For example, the list of fixes that were specified during the initial run is not stored, which means the new input script must specify any fixes you want to use. Even when restart information is stored in the file, as it is for some fixes, commands may need to be re-specified in the new input script, in order to re-use that information. Details are usually given in the documentation of the respective command. Also, see the *read_restart* command for general information about what is stored in a restart file.

---

The optional *nfile* or *fileper* keywords can be used in conjunction with the "%" wildcard character in the specified restart file name. As explained above, the "%" character causes the restart file to be written in pieces, one piece for each of *P* MPI processes. By default *P* = the number of MPI processes the simulation is running on. The *nfile* or *fileper*

keyword can be used to set *P* to a smaller value, which can be more efficient when running on a large number of MPI processes.

The *nfile* keyword sets *P* to the specified *Nf* value. For example, if *Nf* = 4, and the simulation is running on 100 MPI processes, 4 files will be written, by MPI ranks 0, 25, 50, and 75. Each will collect information from itself and the next 24 MPI processes and write it to a restart file.

For the *fileper* keyword, the specified value of *Np* means write one file for every *Np* MPI processes. For example, if *Np* = 4, every fourth MPI rank (0, 4, 8, 12, *etc.*) will collect information from itself and the next 3 MPI processes and write it to a restart file.

---

### 1.121.4 Restrictions

This command requires inter-processor communication to migrate atoms before the restart file is written. This means that your system must be ready to perform a simulation before using this command (force fields setup, atom masses initialized, *etc.*).

### 1.121.5 Related commands

*restart*, *read_restart*, *write_data*

### 1.121.6 Default

none

CHAPTER

TWO

# FIX STYLES

## 2.1 fix accelerate/cos command

### 2.1.1 Syntax

```
fix ID group-ID accelerate value
```

- ID, group-ID are documented in *fix* command
- accelerate/cos = style name of this fix command
- value = amplitude of acceleration (in unit of velocity/time)

### 2.1.2 Examples

```
fix 1 all accelerate/cos 2.0e-7
```

### 2.1.3 Description

Give each atom a acceleration in x-direction based on its z coordinate. The acceleration is a periodic function along the z-direction:

$$a_x(z) = A \cos\left(\frac{2\pi z}{l_z}\right)$$

where $A$ is the acceleration amplitude, $l_z$ is the $z$-length of the simulation box. At steady state, the acceleration generates a velocity profile:

$$v_x(z) = V \cos\left(\frac{2\pi z}{l_z}\right)$$

The generated velocity amplitude $V$ is related to the shear viscosity $\eta$ by:

$$V = \frac{A\rho}{\eta}\left(\frac{l_z}{2\pi}\right)^2$$

and it can be obtained from ensemble average of the velocity profile:

$$V = \frac{\sum_i 2m_i v_{i,x} \cos\left(\frac{2\pi z_i}{l_z}\right)}{\sum_i m_i},$$

where $m_i$, $v_{i,x}$, and $z_i$ are the mass, $x$-component velocity, and $z$-coordinate of a particle, respectively.

The velocity amplitude $V$ can be calculated with *compute viscosity/cos*, which enables viscosity calculation with periodic perturbation method, as described by *Hess*. Because the applied acceleration drives the system away from equilibration, the calculated shear viscosity is lower than the intrinsic viscosity due to the shear-thinning effect. Extrapolation to zero acceleration should generally be performed to predict the zero-shear viscosity. As the shear stress decreases, the signal-to-noise ratio decreases rapidly, and the simulation time must be extended accordingly to get converged results.

In order to get meaningful results, the group ID of this fix should be all.

---

### 2.1.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to binary restart files. None of the fix_modify options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various output commands. No parameter of this fix can be used with the start/stop keywords of the run command.

This fix is not invoked during energy minimization.

### 2.1.5 Restrictions

This fix is part of the MISC package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Since this fix depends on the $z$-coordinate of atoms, it cannot be used in 2d simulations.

### 2.1.6 Related commands

*compute viscosity/cos*

### 2.1.7 Default

none

---

**(Hess)** Hess, B. Journal of Chemical Physics 2002, 116 (1), 209–217.

## 2.2 fix acks2/reaxff command

Accelerator Variants: *acks2/reaxff/kk*

### 2.2.1 Syntax

```
fix ID group-ID acks2/reaxff Nevery cutlo cuthi tolerance params args
```

- ID, group-ID are documented in *fix* command
- acks2/reaxff = style name of this fix command
- Nevery = perform ACKS2 every this many steps
- cutlo,cuthi = lo and hi cutoff for Taper radius
- tolerance = precision to which charges will be equilibrated
- params = reaxff or a filename
- one or more keywords or keyword/value pairs may be appended

```
keyword = maxiter
  maxiter N = limit the number of iterations to N
```

### 2.2.2 Examples

```
fix 1 all acks2/reaxff 1 0.0 10.0 1.0e-6 reaxff
fix 1 all acks2/reaxff 1 0.0 10.0 1.0e-6 param.acks2 maxiter 500
```

### 2.2.3 Description

Perform the atom-condensed Kohn–Sham DFT to second order (ACKS2) charge equilibration method as described in *(Verstraelen)*. ACKS2 impedes unphysical long-range charge transfer sometimes seen with QEq (e.g., for dissociation of molecules), at increased computational cost. It is typically used in conjunction with the ReaxFF force field model as implemented in the *pair_style reaxff* command, but it can be used with any potential in LAMMPS, so long as it defines and uses charges on each atom. For more technical details about the charge equilibration performed by fix acks2/reaxff, see the *(O'Hearn)* paper.

The ACKS2 method minimizes the electrostatic energy of the system by adjusting the partial charge on individual atoms based on interactions with their neighbors. It requires some parameters for each atom type. If the *params* setting above is the word "reaxff", then these are extracted from the *pair_style reaxff* command and the ReaxFF force field file it reads in. If a file name is specified for *params*, then the parameters are taken from the specified file and the file must contain one line for each atom type. The latter form must be used when performing QeQ with a non-ReaxFF potential. The lines should be formatted as follows:

```
bond_softness
itype chi eta gamma bcut
```

where the first line is the global parameter *bond_softness*. The remaining 1 to Ntypes lines include *itype*, the atom type from 1 to Ntypes, *chi*, the electronegativity in eV, *eta*, the self-Coulomb potential in eV, *gamma*, the valence orbital exponent, and *bcut*, the bond cutoff distance. Note that these 4 quantities are also in the ReaxFF potential file, except that eta is defined here as twice the eta value in the ReaxFF file. Note that unlike the rest of LAMMPS, the units of this fix are hard-coded to be Å, eV, and electronic charge.

The optional *maxiter* keyword allows changing the max number of iterations in the linear solver. The default value is 200.

**Note**

In order to solve the self-consistent equations for electronegativity equalization, LAMMPS imposes the additional constraint that all the charges in the fix group must add up to zero. The initial charge assignments should also satisfy this constraint. LAMMPS will print a warning if that is not the case.

### 2.2.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. This fix computes a global scalar (the number of iterations) for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

This fix is invoked during *energy minimization*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.2.5 Restrictions

This fix is part of the REAXFF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix does not correctly handle interactions involving multiple periodic images of the same atom. Hence, it should not be used for periodic cell dimensions smaller than the non-bonded cutoff radius, which is typically 10 Å for ReaxFF simulations.

This fix may be used in combination with *fix efield* and will apply the external electric field during charge equilibration, but there may be only one fix efield instance used, it may only use a constant electric field, and the electric field vector may only have components in non-periodic directions.

### 2.2.6 Related commands

*pair_style reaxff*, *fix qeq/reaxff*, *fix qtpie/reaxff*, *fix qeq/rel/reaxff*

### 2.2.7 Default

maxiter 200

---

**(O'Hearn)** O'Hearn, Alperen, Aktulga, SIAM J. Sci. Comput., 42(1), C1–C22 (2020).

**(Verstraelen)** Verstraelen, Ayers, Speybroeck, Waroquier, J. Chem. Phys. 138, 074108 (2013).

## 2.3 fix adapt command

### 2.3.1 Syntax

```
fix ID group-ID adapt N attribute args ... keyword value ...
```

- ID, group-ID are documented in *fix* command
- adapt = style name of this fix command
- N = adapt simulation settings every this many timesteps
- one or more attribute/arg pairs may be appended
- attribute = *pair* or *bond* or *angle* or *dihedral* or *improper* or *kspace* or *atom*

```
pair args = pstyle pparam I J v_name
  pstyle = pair style name (e.g., lj/cut)
  pparam = parameter to adapt over time
  I,J = type pair(s) to set parameter for (integer or type label)
  v_name = variable with name that calculates value of pparam
bond args = bstyle bparam I v_name
  bstyle = bond style name (e.g., harmonic)
  bparam = parameter to adapt over time
  I = type bond to set parameter for (integer or type label)
```

```
    v_name = variable with name that calculates value of bparam
  angle args = astyle aparam I v_name
    astyle = angle style name (e.g., harmonic)
    aparam = parameter to adapt over time
    I = type angle to set parameter for (integer or type label)
    v_name = variable with name that calculates value of aparam
  dihedral args = dstyle dparam I v_name
    dstyle = dihedral style name (e.g., quadratic)
    dparam = parameter to adapt over time
    I = type dihedral to set parameter for (integer or type label)
    v_name = variable with name that calculates value of iparam
  improper args = istyle iparam I v_name
    istyle = improper style name (e.g., cvff)
    iparam = parameter to adapt over time
    I = type improper to set parameter for (integer or type label)
    v_name = variable with name that calculates value of iparam
  kspace arg = v_name
    v_name = variable with name that calculates scale factor on k-space terms
  atom args = atomparam v_name
    atomparam = charge or diameter or diameter/disc = parameter to adapt over time
    v_name = variable with name that calculates value of atomparam
```

- zero or more keyword/value pairs may be appended
- keyword = *scale* or *reset* or *mass*

```
  scale value = no or yes
    no = the variable value is the new setting
    yes = the variable value multiplies the original setting
  reset value = no or yes
    no = values will remain altered at the end of a run
    yes = reset altered values to their original values at the end of a run
  mass value = no or yes
    no = mass is not altered by changes in diameter
    yes = mass is altered by changes in diameter
```

### 2.3.2 Examples

```
fix 1 all adapt 1 pair soft a 1 1 v_prefactor
fix 1 all adapt 1 pair soft a 2* 3 v_prefactor
fix 1 all adapt 1 pair lj/cut epsilon * * v_scale1 pair coul/cut scale 3 3 v_scale2␣
↪scale yes reset yes
fix 1 all adapt 10 atom diameter v_size

variable ramp_up equal "ramp(0.01,0.5)"
fix stretch all adapt 1 bond harmonic r0 1 v_ramp_up

labelmap atom 1 c1
fix 1 all adapt 1 pair soft a c1 c1 v_prefactor
```

### 2.3.3 Description

Change or adapt one or more specific simulation attributes or settings over time as a simulation runs. Pair potential and *k*-space and atom attributes which can be varied by this fix are discussed below. Many other fixes can also be used to time-vary simulation parameters (e.g., the *fix deform* command will change the simulation box size/shape and the *fix move* command will change atom positions and velocities in a prescribed manner). Also note that many commands allow variables as arguments for specific parameters, if described in that manner on their doc pages. An equal-style variable can calculate a time-dependent quantity, so this is another way to vary a simulation parameter over time.

If $N$ is specified as 0, the specified attributes are only changed once, before the simulation begins. This is all that is needed if the associated variables are not time-dependent. If $N > 0$, then changes are made every $N$ steps during the simulation, presumably with a variable that is time-dependent.

Depending on the value of the *reset* keyword, attributes changed by this fix will or will not be reset back to their original values at the end of a simulation. Even if *reset* is specified as *yes*, a restart file written during a simulation will contain the modified settings.

If the *scale* keyword is set to *no*, which is the default, then the value of the altered parameter will be whatever the variable generates. If the *scale* keyword is set to *yes*, then the value of the altered parameter will be the initial value of that parameter multiplied by whatever the variable generates (i.e., the variable is now a "scale factor" applied in (presumably) a time-varying fashion to the parameter).

Note that whether scale is *no* or *yes*, internally, the parameters themselves are actually altered by this fix. Make sure you use the *reset yes* option if you want the parameters to be restored to their initial values after the run.

---

The *pair* keyword enables various parameters of potentials defined by the *pair_style* command to be changed, if the pair style supports it. Note that the *pair_style* and *pair_coeff* commands must be used in the usual manner to specify these parameters initially; the fix adapt command simply overrides the parameters.

**Note**

Pair_coeff settings must be made **explicitly** in order for fix adapt to be able to change them. Settings inferred from mixing are not suitable. If necessary all mixed settings can be output to a file using the *write_coeff command* and then the desired mixed pair_coeff settings copied from that file.

The *pstyle* argument is the name of the pair style. If *pair_style hybrid or hybrid/overlay* is used, *pstyle* should be a sub-style name. If there are multiple sub-styles using the same pair style, then *pstyle* should be specified as "style:N", where $N$ is which instance of the pair style you wish to adapt (e.g., the first or second). For example, *pstyle* could be specified as "soft" or "lubricate" or "lj/cut:1" or "lj/cut:2". The *pparam* argument is the name of the parameter to change. This is the current list of pair styles and parameters that can be varied by this fix. See the doc pages for individual pair styles and their energy formulas for the meaning of these parameters:

| | | |
|---|---|---|
| *born* | a,b,c | type pairs |
| *born/coul/long, born/coul/msm* | coulombic_cutoff | type global |
| *born/gauss* | biga0,biga1,r0 | type pairs |
| *buck, buck/coul/cut* | a,c | type pairs |
| *buck/coul/long, buck/coul/msm* | a,c,coulombic_cutoff | type pairs |
| *buck/mdf* | a,c | type pairs |
| *coul/cut, coul/cut/global* | scale | type pairs |
| *coul/cut/soft* | lambda | type pairs |
| *coul/debye* | scale | type pairs |
| *coul/dsf* | coulombic_cutoff | type global |
| *coul/long, coul/msm* | coulombic_cutoff, scale | type pairs |

continues on next page

Table 1 – continued from previous page

| | | |
|---|---|---|
| *coul/long/soft* | scale, lambda, coulombic_cutoff | type pairs |
| *coul/slater/long* | scale | type pairs |
| *coul/streitz* | scale | type pairs |
| *eam, eam/alloy, eam/fs* | scale | type pairs |
| *gauss* | a | type pairs |
| *harmonic/cut* | k, cutoff | type pairs |
| *kim* | scale | type global |
| *lennard/mdf* | A,B | type pairs |
| *lj96/cut* | epsilon,sigma | type pairs |
| *lj/class2* | epsilon,sigma | type pairs |
| *lj/class2/coul/cut, lj/class2/coul/long* | epsilon,sigma,coulombic_cutoff | type pairs |
| *lj/cubic* | epsilon,sigma | type pairs |
| *lj/cut* | epsilon,sigma | type pairs |
| *lj/cut/coul/cut, lj/cut/coul/long, lj/cut/coul/msm* | epsilon,sigma,coulombic_cutoff | type pairs |
| *lj/cut/coul/cut/soft, lj/cut/coul/long/soft* | epsilon,sigma,lambda,coulombic_cut | type pairs |
| *lj/cut/coul/dsf* | cutoff | type global |
| *lj/cut/tip4p/cut* | epsilon,sigma,coulombic_cutoff | type pairs |
| *lj/cut/soft* | epsilon,sigma,lambda | type pairs |
| *lj/expand* | epsilon,sigma,delta | type pairs |
| *lj/lj/gromacs* | epsilon,sigma | type pairs |
| *lj/mdf* | epsilon,sigma | type pairs |
| *lj/pirani* | alpha, beta, gamma, rm, epsilon | type pairs |
| *lj/sf/dipole/sf* | epsilon,sigma,scale | type pairs |
| *lubricate* | mu | global |
| *meam* | scale | type pairs |
| *mie/cut* | epsilon,sigma,gamma_repulsive,gamı | type pairs |
| *morse, morse/smooth/linear* | D0,R0,alpha | type pairs |
| *morse/soft* | D0,R0,alpha,lambda | type pairs |
| *nm/cut* | E0,R0,m,n | type pairs |
| *nm/cut/coul/cut, nm/cut/coul/long* | E0,R0,m,n,coulombic_cutoff | type pairs |
| *pace, pace/extrapolation* | scale | type pairs |
| *pedone* | c0,d0,r0,alpha | type pairs |
| *quip* | scale | type global |
| *snap* | scale | type pairs |
| *spin/dmi* | coulombic_cutoff | type global |
| *spin/exchange* | coulombic_cutoff | type global |
| *spin/magelec* | coulombic_cutoff | type global |
| *spin/neel* | coulombic_cutoff | type global |
| *soft* | a | type pairs |
| *table* | table_cutoff | type pairs |
| *ufm* | epsilon,sigma,scale | type pairs |
| *wf/cut* | epsilon,sigma,nu,mu | type pairs |
| *yukawa* | alpha | type pairs |

**Note**

It is easy to add new pairwise potentials and their parameters to this list. All it typically takes is adding an extract() method to the pair_*.cpp file associated with the potential.

Some parameters are global settings for the pair style (e.g., the viscosity setting "mu" for *pair_style lubricate*). Other parameters apply to atom type pairs within the pair style (e.g., the prefactor *a* for *pair_style soft*).

Note that for many of the potentials, the parameter that can be varied is effectively a prefactor on the entire energy expression for the potential (e.g., the lj/cut epsilon). The parameters listed as "scale" are exactly that, since the energy expression for the *coul/cut* potential (for example) has no labeled prefactor in its formula. To apply an effective prefactor to some potentials, multiple parameters need to be altered. For example, the *Buckingham potential* needs both the $A$ and $C$ terms altered together. To scale the Buckingham potential, you should thus list the pair style twice, once for $A$ and once for $C$.

If a type pair parameter is specified, the $I$ and $J$ settings should be specified to indicate which type pairs to apply it to. If a global parameter is specified, the $I$ and $J$ settings still need to be specified, but are ignored.

Similar to the *pair_coeff command*, $I$ and $J$ can be specified in one of several ways. Explicit numeric values can be used for each, as in the first example above. Or, one or both of the types in the I,J pair can be a *type label*. LAMMPS sets the coefficients for the symmetric $J, I$ interaction to the same values.

A wild-card asterisk can be used in place of or in conjunction with the $I, J$ arguments to set the coefficients for multiple pairs of atom types. This takes the form "*" or "*n" or "m*" or "m*n". If $N$ is the number of atom types, then an asterisk with no numeric values means all types from 1 to $N$. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from m to $N$ (inclusive). A middle asterisk means all types from m to n (inclusive). For the asterisk syntax, note that only type pairs with $I \leq J$ are considered; if asterisks imply type pairs where $J < I$, they are ignored.

IMPORTANT NOTE: If *pair_style hybrid or hybrid/overlay* is being used, then the *pstyle* will be a sub-style name. You must specify $I, J$ arguments that correspond to type pair values defined (via the *pair_coeff* command) for that sub-style.

The *v_name* argument for keyword *pair* is the name of an *equal-style variable* which will be evaluated each time this fix is invoked to set the parameter to a new value. It should be specified as v_name, where name is the variable name. Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify parameters that change as a function of time or span consecutive runs in a continuous fashion. For the latter, see the *start* and *stop* keywords of the *run* command and the *elaplong* keyword of *thermo_style custom* for details.

For example, these commands would change the prefactor coefficient of the *pair_style soft* potential from 10.0 to 30.0 in a linear fashion over the course of a simulation:

```
variable prefactor equal ramp(10,30)
fix 1 all adapt 1 pair soft a * * v_prefactor
```

---

The *bond* keyword uses the specified variable to change the value of a bond coefficient over time, very similar to how the *pair* keyword operates. The only difference is that now a bond coefficient for a given bond type is adapted.

A wild-card asterisk can be used in place of or in conjunction with the bond type argument to set the coefficients for multiple bond types. This takes the form "*" or "*n" or "m*" or "m*n". If $N$ is the number of bond types, then an asterisk with no numeric values means all types from 1 to $N$. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from m to $N$ (inclusive). A middle asterisk means all types from m to n (inclusive).

If *bond_style hybrid* is used, *bstyle* should be a sub-style name. The bond styles that currently work with fix adapt are:

| | | |
|---|---|---|
| *class2* | k2,k3,k4,r0 | type bonds |
| *fene* | k,r0 | type bonds |
| *fene/expand* | k,r0,epsilon,sigma,shift | type bonds |
| *fene/nm* | k,r0 | type bonds |
| *gaussian* | alpha,width,r0 | type bonds |
| *gromos* | k,r0 | type bonds |
| *harmonic* | k,r0 | type bonds |
| *harmonic/restrain* | k | type bonds |
| *harmonic/shift* | k,r0,r1 | type bonds |
| *harmonic/shift/cut* | k,r0,r1 | type bonds |
| *mm3* | k,r0 | type bonds |
| *morse* | d0,alpha,r0 | type bonds |
| *nonlinear* | lamda,epsilon,r0 | type bonds |

---

Added in version 4May2022.

The *angle* keyword uses the specified variable to change the value of an angle coefficient over time, very similar to how the *pair* keyword operates. The only difference is that now an angle coefficient for a given angle type is adapted.

A wild-card asterisk can be used in place of or in conjunction with the angle type argument to set the coefficients for multiple angle types. This takes the form "*" or "*n" or "m*" or "m*n". If *N* is the number of angle types, then an asterisk with no numeric values means all types from 1 to *N*. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from m to *N* (inclusive). A middle asterisk means all types from m to n (inclusive).

If *angle_style hybrid* is used, *astyle* should be a sub-style name. The angle styles that currently work with fix adapt are:

| | | |
|---|---|---|
| *harmonic* | k,theta0 | type angles |
| *charmm* | k,theta0 | type angles |
| *class2* | k2,k3,k4,theta0 | type angles |
| *cosine* | k | type angles |
| *cosine/delta* | k | type angles |
| *cosine/periodic* | k,b,n | type angles |
| *cosine/squared* | k,theta0 | type angles |
| *cosine/squared/restricted* | k,theta0 | type angles |
| *dipole* | k,gamma0 | type angles |
| *fourier* | k,c0,c1,c2 | type angles |
| *fourier/simple* | k,c,n | type angles |
| *gaussian* | alpha,width,theta0 | type angles |
| *mm3* | k,theta0 | type angles |
| *mwlc* | k1,k2,mu,T | type angles |
| *quartic* | k2,k3,k4,theta0 | type angles |
| *spica* | k,theta0 | type angles |

Note that internally, theta0 is stored in radians, so the variable this fix uses to reset theta0 needs to generate values in radians.

---

Added in version 12Jun2025.

The *dihedral* keyword uses the specified variable to change the value of a dihedral coefficient over time, very similar to how the *angle* keyword operates. The only difference is that now a dihedral coefficient for a given dihedral type is adapted.

A wild-card asterisk can be used in place of or in conjunction with the dihedral type argument to set the coefficients for multiple dihedral types. This takes the form "*" or "*n" or "m*" or "m*n". If *N* is the number of dihedral types, then an asterisk with no numeric values means all types from 1 to *N*. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from m to *N* (inclusive). A middle asterisk means all types from m to n (inclusive).

If *dihedral_style hybrid* is used, *dstyle* should be a sub-style name. The dihedral styles that currently work with fix adapt are:

| | | |
|---|---|---|
| *charmm* | k,n,d | type dihedrals |
| *charmmfsw* | k,n,d | type dihedrals |
| *class2* | k1,k2,k3,phi1,phi2,phi3 | type dihedrals |
| *cosine/squared/restricted* | k,phi0 | type dihedrals |
| *helix* | a,b,c | type dihedrals |
| *multi/harmonic* | a1,a2,a3,a4,a5 | type dihedrals |
| *opls* | k1,k2,k3,k4 | type dihedrals |
| *quadratic* | k,phi0 | type dihedrals |

Note that internally, phi0 is stored in radians, so the variable this fix use to reset phi0 needs to generate values in radians.

---

Added in version 2Apr2025.

The *improper* keyword uses the specified variable to change the value of an improper coefficient over time, very similar to how the *angle* keyword operates. The only difference is that now an improper coefficient for a given improper type is adapted.

A wild-card asterisk can be used in place of or in conjunction with the improper type argument to set the coefficients for multiple improper types. This takes the form "*" or "*n" or "m*" or "m*n". If *N* is the number of improper types, then an asterisk with no numeric values means all types from 1 to *N*. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from m to *N* (inclusive). A middle asterisk means all types from m to n (inclusive).

If *improper_style hybrid* is used, *istyle* should be a sub-style name. The improper styles that currently work with fix adapt are:

| | | |
|---|---|---|
| *amoeba* | k | type impropers |
| *class2* | k,chi0 | type impropers |
| *cossq* | k,chi0 | type impropers |
| *cvff* | k,d,n | type impropers |
| *distance* | k2,k4 | type impropers |
| *distharm* | k,d0 | type impropers |
| *fourier* | k,C0,C1,C2 | type impropers |
| *harmonic* | k,chi0 | type impropers |
| *inversion/harmonic* | k,w0 | type impropers |
| *ring* | k,theta0 | type impropers |
| *umbrella* | k,w0 | type impropers |
| *sqdistharm* | k | type impropers |

Note that internally, chi0 and theta0 are stored in radians, so the variable this fix use to reset chi0 or theta0 needs to generate values in radians.

---

The *kspace* keyword used the specified variable as a scale factor on the energy, forces, virial calculated by whatever *k*-space solver is defined by the *kspace_style* command. If the variable has a value of 1.0, then the solver is unaltered.

The *kspace* keyword works this way whether the *scale* keyword is set to *no* or *yes*.

---

The *atom* keyword enables various atom properties to be changed. The *aparam* argument is the name of the parameter to change. This is the current list of atom parameters that can be varied by this fix:

- charge = charge on particle
- diameter or diameter/disc = diameter of particle

The *v_name* argument of the *atom* keyword is the name of an *equal-style variable* which will be evaluated each time this fix is invoked to set, or scale the parameter to a new value. It should be specified as v_name, where name is the variable name. See the discussion above describing the formulas associated with equal-style variables. The new value is assigned to the corresponding attribute for all atoms in the fix group.

If the atom parameter is *diameter* and per-atom density and per-atom mass are defined for particles (e.g., *atom_style granular*), then the mass of each particle is, by default, also changed when the diameter changes. The mass is set from the particle volume for 3d systems (density is assumed to stay constant). For 2d, the default is for LAMMPS to model particles with a radius attribute as spheres. However, if the atom parameter is *diameter/disc*, then the mass is set from the particle area (the density is assumed to be in mass/distance$^2$ units). The mass of the particle may also be kept constant if the *mass* keyword is set to *no*. This can be useful to account for diameter changes that do not involve mass changes (e.g., thermal expansion).

For example, these commands would shrink the diameter of all granular particles in the "center" group from 1.0 to 0.1 in a linear fashion over the course of a 1000-step simulation:

```
variable size equal ramp(1.0,0.1)
fix 1 center adapt 10 atom diameter v_size
```

---

This fix can be used in long simulations which are restarted one or more times to continuously adapt simulation parameters, but it must be done carefully. There are two issues to consider. The first is how to adapt the parameters in a continuous manner from one simulation to the next. The second is how, if desired, to reset the parameters to their original values at the end of the last restarted run.

Note that all the parameters changed by this fix are written into a restart file in their current changed state. A new restarted simulation does not know the original time=0 values, unless the input script explicitly resets the parameters (after the restart file is read) to the original values.

Also note that the time-dependent variable(s) used in the restart script should typically be written as a function of time elapsed since the original simulation began.

With this in mind, if the *scale* keyword is set to *no* (the default) in a restarted simulation, original parameters are not needed. The adapted parameters should seamlessly continue their variation relative to the preceding simulation.

If the *scale* keyword is set to *yes*, then the input script should typically reset the parameters being adapted to their original values, so that the scaling formula specified by the variable will operate correctly. An exception is if the *atom* keyword is being used with *scale yes*. In this case, information is added to the restart file so that per-atom properties in the new run will automatically be scaled relative to their original values. This will only work if the fix adapt command specified in the restart script has the same ID as the one used in the original script.

In a restarted run, if the *reset* keyword is set to *yes*, and the run ends in this script (as opposed to just writing more restart files), parameters will be restored to the values they were at the beginning of the run command in the restart script, which as explained above, may or may not be the original values of the parameters. Again, an exception is if the *atom* keyword is being used with *reset yes* (in all the runs). In that case, the original per-atom parameters are stored in the restart file, and will be restored when the restarted run finally completes.

---

### 2.3.4 Restart, fix_modify, output, run start/stop, minimize info

If the *atom* keyword is used and the *scale* or *reset* keyword is set to *yes*, then this fix writes information to a restart file so that in a restarted run scaling can continue in a seamless manner and/or the per-atom values can be restored, as explained above.

None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

For *rRESPA time integration*, this fix changes parameters on the outermost rRESPA level.

### 2.3.5 Restrictions

none

### 2.3.6 Related commands

*compute ti*, *fix adapt/fep*

### 2.3.7 Default

The option defaults are scale = no, reset = no, mass = yes.

## 2.4 fix adapt/fep command

### 2.4.1 Syntax

```
fix ID group-ID adapt/fep N attribute args ... keyword value ...
```

- ID, group-ID are documented in *fix* command
- adapt/fep = style name of this fix command
- N = adapt simulation settings every this many timesteps
- one or more attribute/arg pairs may be appended
- attribute = *pair* or *kspace* or *atom*

```
pair args = pstyle pparam I J v_name
  pstyle = pair style name (e.g., lj/cut)
  pparam = parameter to adapt over time
  I,J = type pair(s) to set parameter for (integer or type label)
  v_name = variable with name that calculates value of pparam
kspace arg = v_name
  v_name = variable with name that calculates scale factor on K-space terms
atom args = aparam v_name
  aparam = parameter to adapt over time
  I = type(s) to set parameter for (integer or type label)
  v_name = variable with name that calculates value of aparam
```

- zero or more keyword/value pairs may be appended
- keyword = *scale* or *reset* or *after*

```
scale value = no or yes
  no = the variable value is the new setting
  yes = the variable value multiplies the original setting
```

```
reset value = no or yes
  no = values will remain altered at the end of a run
  yes = reset altered values to their original values at the end
  of a run
after value = no or yes
  no = parameters are adapted at timestep N
  yes = parameters are adapted one timestep after N
```

### 2.4.2 Examples

```
fix 1 all adapt/fep 1 pair soft a 1 1 v_prefactor
fix 1 all adapt/fep 1 pair soft a 2* 3 v_prefactor
fix 1 all adapt/fep 1 pair lj/cut epsilon * * v_scale1 coul/cut scale 3 3 v_scale2 scale␣
↪yes reset yes
fix 1 all adapt/fep 10 atom diameter 1 v_size

labelmap atom 1 c1
fix 1 all adapt/fep 1 pair soft a c1 c1 v_prefactor
```

Example input scripts available: examples/PACKAGES/fep

### 2.4.3 Description

Change or adapt one or more specific simulation attributes or settings over time as a simulation runs.

This is an enhanced version of the *fix adapt* command with two differences:

- It is possible to modify the charges of chosen atom types only, instead of scaling all the charges in the system.
- There is a new option *after* for better compatibility with *fix ave/time*.

This version is suited for free energy calculations using *compute ti* or *compute fep*.

If $N$ is specified as 0, the specified attributes are only changed once, before the simulation begins. This is all that is needed if the associated variables are not time-dependent. If $N > 0$, then changes are made every $N$ steps during the simulation, presumably with a variable that is time-dependent.

Depending on the value of the *reset* keyword, attributes changed by this fix will or will not be reset back to their original values at the end of a simulation. Even if *reset* is specified as *yes*, a restart file written during a simulation will contain the modified settings.

If the *scale* keyword is set to *no*, then the value the parameter is set to will be whatever the variable generates. If the *scale* keyword is set to *yes*, then the value of the altered parameter will be the initial value of that parameter multiplied by whatever the variable generates (i.e., the variable is now a "scale factor" applied in (presumably) a time-varying fashion to the parameter). Internally, the parameters themselves are actually altered; make sure you use the *reset yes* option if you want the parameters to be restored to their initial values after the run.

If the *after* keyword is set to *yes*, then the parameters are changed one timestep after the multiple of N. In this manner, if a fix such as "fix ave/time" is used to calculate averages at every N timesteps, all the contributions to the average will be obtained with the same values of the parameters.

---

The *pair* keyword enables various parameters of potentials defined by the *pair_style* command to be changed, if the pair style supports it. Note that the *pair_style* and *pair_coeff* commands must be used in the usual manner to specify these parameters initially; the fix adapt command simply overrides the parameters.

**Note**

Pair_coeff settings must be made **explicitly** in order for fix adapt/fep to be able to change them. Settings inferred from mixing are not suitable. If necessary all mixed settings can be output to a file using the *write_coeff command* and then the desired mixed pair_coeff settings copied from that file.

The *pstyle* argument is the name of the pair style. If *pair_style hybrid or hybrid/overlay* is used, *pstyle* should be a sub-style name. For example, *pstyle* could be specified as "soft" or "lubricate". The *pparam* argument is the name of the parameter to change. This is the current list of pair styles and parameters that can be varied by this fix. See the doc pages for individual pair styles and their energy formulas for the meaning of these parameters:

| | | |
|---|---|---|
| *born* | a,b,c | type pairs |
| *born/gauss* | biga0,biga1,r0 | type pairs |
| *buck, buck/coul/cut, buck/coul/long, buck/coul/msm* | a,c | type pairs |
| *buck/mdf* | a,c | type pairs |
| *coul/cut, coul/cut/global* | scale | type pairs |
| *coul/cut/soft* | lambda | type pairs |
| *coul/debye* | scale | type pairs |
| *coul/long, coul/msm* | scale | type pairs |
| *coul/long/soft* | scale, lambda | type pairs |
| *coul/slater/long* | scale | type pairs |
| *coul/streitz* | scale | type pairs |
| *eam, eam/alloy, eam/fs* | scale | type pairs |
| *harmonic/cut* | k | type pairs |
| *gauss* | a | type pairs |
| *lennard/mdf* | a,b | type pairs |
| *lj/class2* | epsilon,sigma | type pairs |
| *lj/class2/coul/cut, lj/class2/coul/long* | epsilon,sigma | type pairs |
| *lj/cut* | epsilon,sigma | type pairs |
| *lj/cut/soft* | epsilon,sigma,lambda | type pairs |
| *lj/cut/coul/cut, lj/cut/coul/long, lj/cut/coul/msm* | epsilon,sigma | type pairs |
| *lj/cut/coul/cut/soft, lj/cut/coul/long/soft* | epsilon,sigma,lambda | type pairs |
| *lj/cut/tip4p/cut, lj/cut/tip4p/long* | epsilon,sigma | type pairs |
| *lj/cut/tip4p/long/soft* | epsilon,sigma,lambda | type pairs |
| *lj/expand* | epsilon,sigma,delta | type pairs |
| *lj/mdf* | epsilon,sigma | type pairs |
| *lj/sf/dipole/sf* | epsilon,sigma,scale | type pairs |
| *meam* | scale | type pairs |
| *mie/cut* | epsilon,sigma,gamR,gamA | type pairs |
| *morse, morse/smooth/linear* | d0,r0,alpha | type pairs |
| *morse/soft* | d0,r0,alpha,lambda | type pairs |
| *nm/cut* | e0,r0,nn,mm | type pairs |
| *nm/cut/coul/cut, nm/cut/coul/long* | e0,r0,nn,mm | type pairs |
| *pace, pace/extrapolation* | scale | type pairs |
| *snap* | scale | type pairs |
| *soft* | a | type pairs |
| *ufm* | epsilon,sigma,scale | type pairs |

continues on next page

Table 2 – continued from previous page

| | | |
|---|---|---|
| *wf/cut* | epsilon,sigma,nu,mu | type pairs |

**Note**

It is easy to add new potentials and their parameters to this list. All it typically takes is adding an extract() method to the pair_*.cpp file associated with the potential.

Note that for many of the potentials, the parameter that can be varied is effectively a prefactor on the entire energy expression for the potential (e.g., the lj/cut epsilon). The parameters listed as "scale" are exactly that, since the energy expression for the *coul/cut* potential (for example) has no labeled prefactor in its formula. To apply an effective prefactor to some potentials, multiple parameters need to be altered. For example, the *Buckingham potential* needs both the A and C terms altered together. To scale the Buckingham potential, you should thus list the pair style twice, once for A and once for C.

If a type pair parameter is specified, the $I$ and $J$ settings should be specified to indicate which type pairs to apply it to. If a global parameter is specified, the $I$ and $J$ settings still need to be specified, but are ignored.

Similar to the *pair_coeff command*, $I$ and $J$ can be specified in one of several ways. Explicit numeric values can be used for each, as in the first example above. Or, one or both of the types in the I,J pair can be a *type label*. LAMMPS sets the coefficients for the symmetric $J, I$ interaction to the same values.

A wild-card asterisk can be used in place of or in conjunction with the $I, J$ arguments to set the coefficients for multiple pairs of atom types. This takes the form "*" or "*n" or "m*" or "m*n". If $N$ is the number of atom types, then an asterisk with no numeric values means all types from 1 to $N$. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from m to $N$ (inclusive). A middle asterisk means all types from m to n (inclusive). For the asterisk syntax, note that only type pairs with $I \leq J$ are considered; if asterisks imply type pairs where $J < I$, they are ignored.

IMPROTANT NOTE: If *pair_style hybrid or hybrid/overlay* is being used, then the *pstyle* will be a sub-style name. You must specify $I, J$ arguments that correspond to type pair values defined (via the *pair_coeff* command) for that sub-style.

The *v_name* argument for keyword *pair* is the name of an *equal-style variable* which will be evaluated each time this fix is invoked to set the parameter to a new value. It should be specified as v_name, where name is the variable name. Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify parameters that change as a function of time or span consecutive runs in a continuous fashion. For the latter, see the *start* and *stop* keywords of the *run* command and the *elaplong* keyword of *thermo_style custom* for details.

For example, these commands would change the prefactor coefficient of the *pair_style soft* potential from 10.0 to 30.0 in a linear fashion over the course of a simulation:

```
variable prefactor equal ramp(10,30)
fix 1 all adapt 1 pair soft a * * v_prefactor
```

---

The *kspace* keyword used the specified variable as a scale factor on the energy, forces, virial calculated by whatever *k*-space solver is defined by the *kspace_style* command. If the variable has a value of 1.0, then the solver is unaltered.

The *kspace* keyword works this way whether the *scale* keyword is set to *no* or *yes*.

---

The *atom* keyword enables various atom properties to be changed. The *aparam* argument is the name of the parameter to change. This is the current list of atom parameters that can be varied by this fix:

- charge = charge on particle
- diameter = diameter of particle

The *I* argument indicates which atom types are affected. A wild-card asterisk can be used in place of or in conjunction with the *I* argument to set the coefficients for multiple atom types.

The *v_name* argument of the *atom* keyword is the name of an *equal-style variable* which will be evaluated each time this fix is invoked to set the parameter to a new value. It should be specified as v_name, where name is the variable name. See the discussion above describing the formulas associated with equal-style variables. The new value is assigned to the corresponding attribute for all atoms in the fix group.

If the atom parameter is *diameter* and per-atom density and per-atom mass are defined for particles (e.g., *atom_style granular*), then the mass of each particle is also changed when the diameter changes (density is assumed to stay constant).

For example, these commands would shrink the diameter of all granular particles in the "center" group from 1.0 to 0.1 in a linear fashion over the course of a 1000-step simulation:

```
variable size equal ramp(1.0,0.1)
fix 1 center adapt 10 atom diameter * v_size
```

For *rRESPA time integration*, this fix changes parameters on the outermost rRESPA level.

---

### 2.4.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.4.5 Restrictions

The keyword "scale yes" is not supported for scaling per-atom parameters diameter and change. You can use *fix adapt* for those.

### 2.4.6 Related commands

*compute fep*, *fix adapt*, *compute ti*, *pair_style */soft*

### 2.4.7 Default

The option defaults are scale = no, reset = no, after = no.

## 2.5 fix add/heat command

### 2.5.1 Syntax

```
fix ID group-ID add/heat style args keyword values ...
```

- ID, group-ID are documented in *fix* command
- add/heat = style name of this fix command
- style = *constant* or *linear* or *quartic*

```
constant args = rate
  rate = rate of heat flow (energy/time units)
linear args = Ttarget k
  Ttarget = target temperature (temperature units)
  k = prefactor (energy/(time*temperature) units)
quartic args = Ttarget k
  Ttarget = target temperature (temperature units)
  k = prefactor (energy/(time*temperature^4) units)
```

- zero or more keyword/value pairs may be appended to args
- keyword = *overwrite*

```
overwrite value = yes or no
  yes = sets current heat flow of particle
  no = adds to current heat flow of particle
```

### 2.5.2 Examples

```
fix 1 all add/heat constant v_heat
fix 1 all add/heat linear 10.0 1.0 overwrite yes
```

### 2.5.3 Description

This fix adds heat to particles with the temperature attribute every timestep at a given rate. Note that this is an internal temperature of a particle intended for use with non-atomistic models like the discrete element method.

For the *constant* style, heat is added at the specified rate. For the *linear* style, heat is added at a rate of $k(T_{target} - T)$ where $k$ is the specified prefactor, $T_{target}$ is the specified target temperature, and $T$ is the temperature of the atom. This may be more representative of a conductive process. For the *quartic* style, heat is added at a rate of $k(T_{target}^4 - T^4)$, akin to radiative heat transfer.

The rate or temperature can be can be specified as an equal-style or atom-style *variable*. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated each time step, and its value will be used to determine the rate of heat added.

Equal-style variables can specify formulas with various mathematical functions and include *thermo_style* command keywords for the simulation box parameters, time step, and elapsed time to specify time-dependent heating.

Atom-style variables can specify the same formulas as equal-style variables but can also include per-atom values, such as atom coordinates to specify spatially-dependent heating.

If the *overwrite* keyword is set to *yes*, this fix will set the total heat flow on a particle every timestep, overwriting contributions from pair styles or other fixes. If *overwrite* is *no*, this fix will add heat on top of other contributions.

---

### 2.5.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.5.5 Restrictions

This pair style is part of the GRANULAR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix requires that atoms store temperature and heat flow as defined by the *fix property/atom* command or included in certain atom styles, such as atom_style rheo/thermal.

### 2.5.6 Related commands

*fix heat/flow*, *fix property/atom*, *fix rheo/thermal*

### 2.5.7 Default

The default for the *overwrite* keyword is *no*

## 2.6 fix addforce command

Accelerator Variants: *addforce/kk*

### 2.6.1 Syntax

```
fix ID group-ID addforce fx fy fz keyword value ...
```

- ID, group-ID are documented in *fix* command
- addforce = style name of this fix command
- fx,fy,fz = force component values (force units)

  ```
  any of fx,fy,fz can be a variable (see below)
  ```

- zero or more keyword/value pairs may be appended to args
- keyword = *every* or *region* or *energy*

  ```
  every value = Nevery
    Nevery = add force every this many time steps
  region value = region-ID
    region-ID = ID of region atoms must be in to have added force
  energy value = v_name
    v_name = variable with name that calculates the potential energy of each atom in␣
  ↪the added force field
  ```

### 2.6.2 Examples

```
fix kick flow addforce 1.0 0.0 0.0
fix kick flow addforce 1.0 0.0 v_oscillate
fix ff boundary addforce 0.0 0.0 v_push energy v_espace
```

### 2.6.3 Description

Add $(f_x, f_y, f_z)$ to the corresponding component of the force for each atom in the group. This command can be used to give an additional push to atoms in a simulation, such as for a simulation of Poiseuille flow in a channel.

Any of the three quantities defining the force components, namely $f_x$, $f_y$, and $f_z$, can be specified as an equal-style or atom-style *variable*. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated each time step, and its value(s) will be used to determine the force component(s).

Equal-style variables can specify formulas with various mathematical functions and include *thermo_style* command keywords for the simulation box parameters, time step, and elapsed time. Thus, it is easy to specify a time-dependent force field.

Atom-style variables can specify the same formulas as equal-style variables but can also include per-atom values, such as atom coordinates. Thus, it is easy to specify a spatially-dependent force field with optional time-dependence as well.

If the *every* keyword is used, the *Nevery* setting determines how often the forces are applied. The default value is 1, for every time step.

If the *region* keyword is used, the atom must also be in the specified geometric *region* in order to have force added to it.

---

Adding a force to atoms implies a change in their potential energy as they move due to the applied force field. For dynamics via the "run" command, this energy can be optionally added to the system's potential energy for thermodynamic output (see below). For energy minimization via the "minimize" command, this energy must be added to the system's potential energy to formulate a self-consistent minimization problem (see below).

The *energy* keyword is not allowed if the added force is a constant vector $\vec{F} = (f_x, f_y, f_z)$, with all components defined as numeric constants and not as variables. This is because LAMMPS can compute the energy for each atom directly as

$$E = -\vec{x} \cdot \vec{F} = -(xf_x + yf_y + zf_z),$$

so that $-\vec{\nabla}E = \vec{F}$.

The *energy* keyword is optional if the added force is defined with one or more variables, and if you are performing dynamics via the *run* command. If the keyword is not used, LAMMPS will set the energy to 0.0, which is typically fine for dynamics.

The *energy* keyword is required if the added force is defined with one or more variables, and you are performing energy minimization via the "minimize" command. The keyword specifies the name of an atom-style *variable* which is used to compute the energy of each atom as function of its position. Like variables used for $f_x$, $f_y$, $f_z$, the energy variable is specified as v_name, where name is the variable name.

Note that when the *energy* keyword is used during an energy minimization, you must ensure that the formula defined for the atom-style *variable* is consistent with the force variable formulas (i.e., that $-\vec{\nabla}E = \vec{F}$). For example, if the force were a spring-like, $\vec{F} = -k\vec{x}$, then the energy formula should be $E = \frac{1}{2}kx^2$. If you do not do this correctly, the minimization will not converge properly.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

**Note**

The region keyword is supported by Kokkos, but a Kokkos-enabled region must be used. See the region *region* command for more information.

---

### 2.6.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify energy* option is supported by this fix to add the potential energy inferred by the added force to the global potential energy of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify energy no*. Note that this energy is a fictitious quantity but is needed so that the *minimize* command can include the forces added by this fix in a consistent manner (i.e., there is a decrease in potential energy when atoms move in the direction of the added force).

The *fix_modify virial* option is supported by this fix to add the contribution due to the added forces on atoms to both the global pressure and per-atom stress of the system via the *compute pressure* and *compute stress/atom* commands. The former can be accessed by *thermodynamic output*. The default setting for this fix is *fix_modify virial no*.

The *fix_modify respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is adding its forces. Default is the outermost level.

This fix computes a global scalar and a global three-vector of forces, which can be accessed by various *output commands*. The scalar is the potential energy discussed above. The vector is the total force on the group of atoms before the forces on individual atoms are changed by the fix. The scalar and vector values calculated by this fix are "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The forces due to this fix are imposed during an energy minimization, invoked by the *minimize* command. You should not specify force components with a variable that has time-dependence for use with a minimizer, since the minimizer increments the time step as the iteration count during the minimization.

> **Note**
>
> If you want the fictitious potential energy associated with the added forces to be included in the total potential energy of the system (the quantity being minimized), you MUST enable the *fix_modify energy* option for this fix.

### 2.6.5 Restrictions

none

### 2.6.6 Related commands

*fix setforce*, *fix aveforce*

### 2.6.7 Default

The option default for the every keyword is every = 1.

## 2.7 fix addtorque/atom command

### 2.7.1 Syntax

```
fix ID group-ID addtorque/atom tx ty tz keyword value ...
```

- ID, group-ID are documented in *fix* command
- addtorque/atom = style name of this fix command
- tx,ty,tz = torque component values (torque units) .. parsed-literal:

```
any of tx,ty,tz can be a variable (see below)
```

- zero or more keyword/value pairs may be appended to args
- keyword = *every* or *region*

```
every value = Nevery
  Nevery = add torque every this many time steps
region value = region-ID
  region-ID = ID of region atoms must be in to have added torque
```

### 2.7.2 Examples

```
fix kick flow addtorque/atom 1.0 0.0 0.0
fix kick flow addtorque/atom 1.0 0.0 v_oscillate
fix ff boundary addtorque/atom 0.0 0.0 v_push
```

### 2.7.3 Description

Added in version 10Dec2025.

This fix is intended to add a peratom torque of each individual finite-sized atom in the group to the specified values. Unlike *fix addtorque/group*, it does not apply a collective torque to a set of point particles.

Add $(t_x, t_y, t_z)$ to the corresponding component of the torque for each atom in the group. Any of the three quantities defining the torque components, namely $t_x$, $t_y$, and $t_z$, can be specified as an equal-style or atom-style *variable*. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated each time step, and its value(s) will be used to determine the torque component(s).

Equal-style variables can specify formulas with various mathematical functions and include *thermo_style* command keywords for the simulation box parameters, time step, and elapsed time. Thus, it is easy to specify a time-dependent torque field.

Atom-style variables can specify the same formulas as equal-style variables but can also include per-atom values, such as atom coordinates. Thus, it is easy to specify a spatially-dependent torque field with optional time-dependence as well.

If the *every* keyword is used, the *Nevery* setting determines how often the torques are applied. The default value is 1, for every time step.

If the *region* keyword is used, the atom must also be in the specified geometric *region* in order to have torque added to it.

---

### 2.7.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify* *respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is adding its torques. Default is the outermost level.

This fix computes a global three-vector of torques which can be accessed by various *output commands*. The vector is the total torque on the group of atoms before the torques on individual atoms are changed by the fix. The vector values calculated by this fix are "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The torques due to this fix are imposed during an energy minimization, invoked by the *minimize* command. You should not specify torque components with a variable that has time-dependence for use with a minimizer, since the minimizer increments the time step as the iteration count during the minimization.

**Note**

This fix is not (currently) designed to be used with rigid fixes. While it will apply additional torques to all of the atoms in a rigid body as described above, there is not always an easy mapping between these peratom torques and the torque experienced by the body.

### 2.7.5 Restrictions

Fix *addtorque/atom* is part of the GRANULAR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.7.6 Related commands

*fix settorque/atom*, *fix addforce*

### 2.7.7 Default

The option default for the every keyword is every = 1.

## 2.8 fix addtorque/group command

### 2.8.1 Syntax

```
fix ID group-ID addtorque/group Tx Ty Tz
```

- ID, group-ID are documented in *fix* command
- addtorque/group = style name of this fix command
- Tx,Ty,Tz = torque component values (torque units)
- any of Tx,Ty,Tz can be a variable (see below)

### 2.8.2 Examples

```
fix kick bead addtorque/group 2.0 3.0 5.0
fix kick bead addtorque/group 0.0 0.0 v_oscillate
```

### 2.8.3 Description

Changed in version 10Dec2025: Fix *addtorque* was renamed to fix *addtorque/group*

Add a set of forces to each atom in the group such that:

- the components of the total torque applied on the group (around its center of mass) are $T_x$, $T_y$, and $T_z$
- the group would move as a rigid body in the absence of other forces.

This command can be used to drive a group of atoms into rotation by adding forces to the atoms. To apply a torque to individual finite-size atoms, use *fix addtorque/atom* instead.

Any of the three quantities defining the torque components can be specified as an equal-style *variable*, namely *Tx*, *Ty*, *Tz*. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated each timestep, and its value used to determine the torque component.

Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent torque.

**Note**

Fix *addtorque/group* previously was known as fix *addtorque* and was renamed to clarify that the fix operates on a group of atoms as opposed to individual finite-size atoms.

---

### 2.8.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify* *energy* option is supported by this fix to add the potential "energy" inferred by the added torques to the global potential energy of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify energy no*. Note that this is a fictitious quantity but is needed so that the *minimize* command can include the forces added by this fix in a consistent manner (i.e., there is a decrease in potential energy when atoms move in the direction of the added forces).

The *fix_modify* *respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is adding its torque. Default is the outermost level.

This fix computes a global scalar and a global 3-vector, which can be accessed by various *output commands*. The scalar is the potential energy discussed above. The vector is the total torque on the group of atoms before the forces on individual atoms are changed by the fix. The scalar and vector values calculated by this fix are "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The forces due to this fix are imposed during an energy minimization, invoked by the *minimize* command.

**Note**

If you want the fictitious potential energy associated with the added forces to be included in the total potential energy of the system (the quantity being minimized), you MUST enable the *fix_modify* *energy* option for this fix.

**Note**

You should not specify force components with a variable that has time-dependence for use with a minimizer, since the minimizer increments the timestep as the iteration count during the minimization.

### 2.8.5 Restrictions

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.8.6 Related commands

*fix addforce* *fix addtorque/atom*

### 2.8.7 Default

none

## 2.9 fix alchemy command

### 2.9.1 Syntax

```
fix ID group-ID alchemy v_name
```

- ID, group-ID are documented in *fix* command
- alchemy = style name of this fix command
- v_name = variable with name that determines the $\lambda_R$ value

### 2.9.2 Examples

```
fix trans all alchemy v_ramp
```

### 2.9.3 Description

Added in version 28Mar2023.

This fix command enables an "alchemical transformation" to be performed between two systems, whereby one system slowly transforms into the other over the course of a molecular dynamics run. This is useful for measuring thermodynamic differences between two different systems. It also allows transformations that are not easily possible with the *pair style hybrid/scaled*, *fix adapt* or *fix adapt/fep* commands.

Example inputs are included in the `examples/PACKAGES/alchemy` directory for (a) transforming a pure copper system into a copper/aluminum bronze alloy and (b) transforming two water molecules in a box of water into a hydronium and a hydroxyl ion.

The two systems must be defined as *separate replica* and run in separate partitions of processors using the *-partition* command-line switch. Exactly two partitions must be specified, and each partition must use the same number of processors and the same domain decomposition.

Because the forces applied to the atoms are the same mix of the forces from each partition and the simulation starts with the same atom positions across both partitions, they will generate the same trajectory of coordinates for each atom, and the same simulation box size and shape. The latter two conditions are *enforced* by this fix; it exchanges coordinates and box information between the replicas. This is not strictly required, but since MD simulations are an example of a chaotic system, even the tiniest random difference will eventually grow exponentially into an unwanted divergence.

Otherwise, the properties of each atom (type, charge, bond and angle partners, etc.), as well as energy and forces between interacting atoms (pair, bond, angle styles, etc.) can be different in the two systems.

This can be initialized in the same input script by using commands which only apply to one or the other replica. The example scripts use a world-style *variable* command along with *if/then/else* commands for this purpose. The *partition* command can also be used.

```
create_box 2 box
create_atoms 1 box
pair_style eam/alloy
```

(continues on next page)

(continued from previous page)

```
pair_coeff * * AlCu.eam.alloy Cu Al

# replace 5% of copper with aluminum on the second partition only

variable name world pure alloy
if "${name} == alloy" then &
  "set type 1 type/fraction 2 0.05 6745234"
```

Both replicas must define an instance of this fix, but with a different *v_name* variable. The named variable must be an equal-style or equivalent *variable*. The two variables should be defined so that one ramps *down* from 1.0 to 0.0 for the *first* replica (*R=0*) and the other ramps *up* from 0.0 to 1.0 for the *second* replica (*R=1*). A simple way is to do this is linearly, which can be done using the ramp() function of the *variable* command. You could also define a variable which returns a value between 0.0 and 1.0 as a non-linear function of the timestep. Here is a linear example:

```
partition yes 1 variable ramp equal ramp(1.0,0.0)
partition yes 2 variable ramp equal ramp(0.0,1.0)
fix 2 all alchemy v_ramp
```

**Note**

For an alchemical transformation, the two variables should sum to exactly 1.0 at any timestep. LAMMPS does *NOT* check that this is the case.

If you use the `ramp()` function to define the two variables, this fix can easily be used across successive runs in the same input script by ensuring each instance of the *run* command specifies the appropriate *start* or *stop* options.

At each timestep of an MD run, the two instances of this fix evaluate their respective variables as a $\lambda_R$ factor, where $R$ = 0 or 1 for each replica. The forces used by each system for the propagation of their atoms is set to the sum of the forces for the two systems, each scaled by their respective $\lambda_R$ factor. Thus, during the MD run, the system will transform incrementally from the first system to the second system.

**Note**

As mentioned above, the coordinates of the atoms and box size/shape must be exactly the same in the two replicas. Therefore, it is generally not a good idea to initialize the two replicas by reading different data files or creating them individually from scratch. Rather, a single system should be initialized and then desired modifications applied to the system to either replica. If your input script somehow induces the two systems to become different (e.g. by performing *atom_modify sort* differently, or by adding or depositing a different number of atoms), then LAMMPS will detect the mismatch and generate an error. This is done by ensuring that each step the number and ordering of atoms is identical within each pair of processors in the two replicas.

### 2.9.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix stores a global scalar (the current value of $\lambda_R$) and a global vector of length 3 which contains the potential energy of the first partition, the second partition and the combined value, respectively. The global scalar is unitless and "intensive", the vector is in *energy units* and "extensive". These values can be used by any command that uses a global value from a fix as input. See the *output howto* page for an overview of LAMMPS output options.

This fix is not invoked during *energy minimization*.

### 2.9.5 Restrictions

This fix is part of the REPLICA package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

There may be only one instance of this fix in use at a time within each replica.

### 2.9.6 Related commands

*compute pressure/alchemy* command, *fix adapt* command, *fix adapt/fep* command, *pair_style hybrid/scaled* command.

### 2.9.7 Default

none

## 2.10 fix align/self command

### 2.10.1 Syntax

```
fix ID group-ID align/self mode magnitude keyword values
```

- ID, group-ID are documented in *fix* command
- align/self = style name of this fix command
- mode = *dipole* or *quat*
- magnitude = magnitude of self-alignment torque
- zero or one keyword/value pairs may be appended
- keyword = *qvector*

  ```
  qvector value = direction of self-propulsion force in ellipsoid frame
    sx, sy, sz = components of qvector
  ```

### 2.10.2 Examples

```
fix active all align/self dipole 40.0
fix active all align/self quat 15.7 qvector 1.0 0.0 0.0
```

### 2.10.3 Description

Added in version 10Dec2025.

Add a torque to each atom in the group which accounts for the reorientation of each particle toward its own velocity, a generic phenomenon called self-alignment (see *(Baconnier2025)*). The torque is given by :

$$\tau_i = \zeta(\mathbf{e}_i \times \mathbf{v}_i)$$

where $i$ is the particle the torque is being applied to, $\zeta$ is the magnitude of the torque, $\mathbf{e}_i$ is the orientation of the particle, and $\mathbf{v}_i$ is its velocity. The self-alignment term, introduced in *(Shimoyama1996)* with the study of collective motion in systems of self-propelled particles, is an effective torque arising from differential drag in asymmetric rigid bodies.

For mode *dipole*, $e_i$ is just equal to the dipole vectors of the atoms in the group. Therefore, if the dipoles are not unit vectors, the $e_i$ will not be unit vectors.

**Note**

If another command changes the magnitude of the dipole, the applied torque will change accordingly and no warning will be provided by LAMMPS. This is almost never what you want, so ensure you are not changing dipole magnitudes with another LAMMPS fix or pair style. Furthermore, self-propulsion forces (almost) always set $e_i$ to be a unit vector for all times, so it's best to set all the dipole magnitudes to 1.0 unless you have a good reason not to (see the *set* command on how to do this).

For mode *quat*, $e_i$ points in the direction of a unit vector, oriented in the coordinate frame of the ellipsoidal particles, which defaults to point along the x-direction. This default behavior can be changed by via the *qvector* keyword.

The optional *qvector* keyword specifies the direction of self-propulsion via a unit vector (sx,sy,sz). The arguments *sx*, *sy*, and *sz*, are defined within the coordinate frame of the atom's ellipsoid. For instance, for an ellipsoid with long axis along its x-direction, if one wanted the self-propulsion force to also be along this axis, set *sx* equal to 1 and *sy*, *sz* both equal to zero. This keyword may only be specified for mode *quat*.

**Note**

In using keyword *qvector*, the three arguments *sx*, *sy*, and *sz* will be automatically normalized to components of a unit vector internally to avoid users having to explicitly do so themselves. Therefore, in mode *quat*, the vectors $e_i$ will always be of unit length.

---

### 2.10.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. No global or per-atom quantities are stored by this fix for access by various *output commands*.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.10.5 Restrictions

This fix is part of the BROWNIAN package. It is only enabled if LAMMPS was built with that package. See the *Build package* doc page for more info.

The keyword *dipole* requires that atoms store torque as defined by the *atom_style sphere* command, as well as a dipole moment as defined by the *atom_style dipole* command which is part of the DIPOLE package. The keyword *quat* requires that atoms store torque and quaternions as defined by the *atom_style ellipsoid* command.

### 2.10.6 Related commands

*fix propel/self*, *fix brownian*, *fix addtorque/group*

### 2.10.7 Default

none

---

**(Baconnier2025)** P. Baconnier, O. Dauchot, V. Demery, G. Duering, S. Henkes, C. Huepe, and A. Shee, Self-aligning polar active matter, Rev. Mod. Phys. 97, 015007 (2025).

**(Shimoyama1996)** N. Shimoyama, K. Sugawara, T. Mizuguchi, Y. Hayakawa, and M. Sano, Collective Motion in a System of Motile Elements, Phys. Rev. Lett. 76, 3870 (1996).

## 2.11 fix amoeba/bitorsion command

### 2.11.1 Syntax

```
fix ID group-ID ameoba/bitorsion filename
```

- ID, group-ID are documented in *fix* command
- amoeba/bitorsion = style name of this fix command
- filename = force-field file with AMOEBA bitorsion coefficients

### 2.11.2 Examples

```
fix           bit all amoeba/bitorsion bitorsion.ubiquitin.data
read_data     proteinX.data fix bit bitorsions BiTorsions
fix_modify    bit energy yes
```

### 2.11.3 Description

This command enables 5-body torsion/torsion interactions to be added to simulations which use the AMOEBA and HIPPO force fields. It matches how the Tinker MD code computes its torsion/torsion interactions for the AMOEBA and HIPPO force fields. See the *Howto amoeba* doc page for more information about the implementation of AMOEBA and HIPPO in LAMMPS.

Bitorsion interactions add additional potential energy contributions to pairs of overlapping phi-psi dihedrals of amino-acids, which are important to properly represent their conformational behavior. Each bitorsion interaction is thus defined for a 5-tuple of atoms $IJKLM$ with bonds between successive atoms in the list, i.e. two overlapping dihedral interactions for atoms $IJKL$ and $JKLM$.

The examples/amoeba directory has a sample input script and data file for ubiquitin, which illustrates use of the fix amoeba/bitorsion command.

As in the example above, this fix should be used before reading a data file that contains a listing of bitorsion interactions. The *filename* specified should contain the bitorsion parameters for the AMOEBA or HIPPO force field.

The data file read by the *read_data* command must contain the topology of all the bitorsion interactions, similar to the topology data for bonds, angles, dihedrals, etc. Specifically it should have a line like this in its header section:

```
N bitorsions
```

where $N$ is the number of bitorsion 5-body interactions. It should also have a section in the body of the data file like this with $N$ lines:

```
BiTorsions

       1       1       8      10      12      18      20
       2       5      18      20      22      25      27
       [...]
       N       3     314     315     317     318     330
```

The first column is an index from 1 to $N$ to enumerate the bitorsion 5-atom tuples; it is ignored by LAMMPS. The second column is the *type* of the interaction; it is an index into the bitorsion force field file. The remaining 5 columns are the atom IDs of the atoms (in order) for the 5-tuple $IJKLM$, as described above.

Note that the *bitorsions* and *BiTorsions* keywords for the header and body sections match those specified in the *read_data* command following the data file name.

The data file should be generated by using the tools/tinker/tinker2lmp.py conversion script which creates a LAMMPS data file from Tinker input files, including its PRM file which contains the parameters necessary for computing bitorsion interactions. The script must be invoked with the optional "-bitorsion" flag to do this; see the example for the ubiquitin system in the tools/tinker/README file. The same conversion script also creates the file of bitorsion coefficient data which is read by this command.

The potential energy associated with bitorsion interactions can be output as described below. It can also be included in the total potential energy of the system, as output by the *thermo_style* command, if the *fix_modify energy* command is used, as in the example above. See the note below about how to include the bitorsion energy when performing an *energy minimization*.

---

### 2.11.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the list of bitorsion interactions to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The *fix_modify energy* option is supported by this fix to add the potential energy of the bitorsion interactions to both the global potential energy and peratom potential energies of the system as part of *thermodynamic output* or output by the *compute pe/atom* command. The default setting for this fix is *fix_modify energy yes*.

The *fix_modify virial* option is supported by this fix to add the contribution due to the bitorsion interactions to both the global pressure and per-atom stress of the system via the *compute pressure* and *compute stress/atom* commands. The former can be accessed by *thermodynamic output*. The default setting for this fix is *fix_modify virial yes*.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the potential energy discussed above. The scalar value calculated by this fix is "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The forces due to this fix are imposed during an energy minimization, invoked by the *minimize* command.

The *fix_modify respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is adding its forces. Default is the outermost level.

> **Note**
>
> For energy minimization, if you want the potential energy associated with the bitorsion terms forces to be included in the total potential energy of the system (the quantity being minimized), you MUST not disable the *fix_modify energy* option for this fix.

### 2.11.5 Restrictions

To function as expected this fix command must be issued *before* a *read_data* command but *after* a *read_restart* command.

This fix can only be used if LAMMPS was built with the AMOEBA package. See the *Build package* page for more info.

### 2.11.6 Related commands

*fix_modify*, *read_data*

### 2.11.7 Default

none

## 2.12 fix amoeba/pitorsion command

### 2.12.1 Syntax

```
fix ID group-ID ameoba/pitorsion
```

- ID, group-ID are documented in *fix* command
- amoeba/pitorsion = style name of this fix command

### 2.12.2 Examples

```
fix            pit all amoeba/pitorsion
read_data      proteinX.data fix pit "pitorsion types" "PiTorsion Coeffs" &
               fix pit pitorsions PiTorsions
fix_modify     pit energy yes
```

### 2.12.3 Description

This command enables 6-body pitorsion interactions to be added to simulations which use the AMOEBA and HIPPO force fields. It matches how the Tinker MD code computes its pitorsion interactions for the AMOEBA and HIPPO force fields. See the *Howto amoeba* doc page for more information about the implementation of AMOEBA and HIPPO in LAMMPS.

Pitorsion interactions add additional potential energy contributions to 6-tuples of atoms $IJKLMN$ that have a bond between atoms $K$ and $L$, where both $K$ and $L$ are additionally bonded to exactly two other atoms. Namely, $K$ is also bonded to $I$ and $J$, and $L$ is also bonded to $M$ and $N$.

The examples/amoeba directory has a sample input script and data file for ubiquitin, which illustrates use of the fix amoeba/pitorsion command.

As in the example above, this fix should be used before reading a data file that contains a listing of pitorsion interactions.

The data file read by the *read_data* command must contain the topology of all the pitorsion interactions, similar to the topology data for bonds, angles, dihedrals, etc. Specifically, it should have two lines like these in its header section:

```
M pitorsion types
N pitorsions
```

where $N$ is the number of pitorsion 6-body interactions and $M$ is the number of pitorsion types. It should also have two sections in the body of the data file like these with $M$ and $N$ lines each:

```
PiTorsion Coeffs

       1 6.85
       2 10.2
       [...]
       M 6.85
```

```
PiTorsions

```

(continues on next page)

(continued from previous page)

```
      1       1       2       4       3       20      21      24
      2       5       21      23      22      37      38      41
      [...]
      N       7       27      29      28      30      35      36
```

For PiTorsion Coeffs, the first column is an index from 1 to *M* to enumerate the pitorsion types. The second column is the single prefactor coefficient needed for each type.

For PiTorsions, the first column is an index from 1 to *N* to enumerate the pitorsion 6-atom tuples; it is ignored by LAMMPS. The second column is the "type" of the interaction; it is an index into the PiTorsion Coeffs. The remaining 6 columns are the atom IDs of the atoms (in order) for the 6-tuple *IJKLMN*, as described above.

Note that the *pitorsion types* and *pitorsions* and *PiTorsion Coeffs* and *PiTorsions* keywords for the header and body sections of the data file match those specified in the *read_data* command following the data file name.

The data file should be generated by using the tools/tinker/tinker2lmp.py conversion script which creates a LAMMPS data file from Tinker input files, including its PRM file which contains the parameters necessary for computing pitorsion interactions.

The potential energy associated with pitorsion interactions can be output as described below. It can also be included in the total potential energy of the system, as output by the *thermo_style* command, if the *fix_modify energy* command is used, as in the example above. See the note below about how to include the pitorsion energy when performing an *energy minimization*.

---

### 2.12.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the list of pitorsion interactions to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The *fix_modify energy* option is supported by this fix to add the potential energy of the pitorsion interactions to both the global potential energy and peratom potential energies of the system as part of *thermodynamic output* or output by the *compute pe/atom* command. The default setting for this fix is *fix_modify energy yes*.

The *fix_modify virial* option is supported by this fix to add the contribution due to the pitorsion interactions to both the global pressure and per-atom stress of the system via the *compute pressure* and *compute stress/atom* commands. The former can be accessed by *thermodynamic output*. The default setting for this fix is *fix_modify virial yes*.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the potential energy discussed above. The scalar value calculated by this fix is "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The forces due to this fix are imposed during an energy minimization, invoked by the *minimize* command.

The *fix_modify respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is adding its forces. Default is the outermost level.

> **Note**
>
> For energy minimization, if you want the potential energy associated with the pitorsion terms forces to be included in the total potential energy of the system (the quantity being minimized), you MUST not disable the *fix_modify energy* option for this fix.

### 2.12.5 Restrictions

To function as expected this fix command must be issued *before* a *read_data* command but *after* a *read_restart* command.

This fix can only be used if LAMMPS was built with the AMOEBA package. See the *Build package* page for more info.

### 2.12.6 Related commands

*fix_modify*, *read_data*

### 2.12.7 Default

none

## 2.13 fix append/atoms command

### 2.13.1 Syntax

```
fix ID group-ID append/atoms face ... keyword value ...
```

- ID, group-ID are documented in *fix* command
- append/atoms = style name of this fix command
- face = *zhi*
- zero or more keyword/value pairs may be appended
- keyword = *basis* or *size* or *freq* or *temp* or *random* or *units*

```
basis values = M itype
  M = which basis atom
  itype = atom type (1-N) to assign to this basis atom
size args = Lz
  Lz = z size of lattice region appended in a single event(distance units)
freq args = freq
  freq = the number of timesteps between append events
temp args = target damp seed extent
  target = target temperature for the region between zhi-extent and zhi␣
↪(temperature units)
  damp = damping parameter (time units)
  seed = random number seed for langevin kicks
  extent = extent of thermostatted region (distance units)
random args = xmax ymax zmax seed
  xmax, ymax, zmax = maximum displacement in particular direction (distance units)
  seed = random number seed for random displacement
units value = lattice or box
  lattice = the wall position is defined in lattice units
  box = the wall position is defined in simulation box units
```

### 2.13.2 Examples

```
fix 1 all append/atoms zhi size 5.0 freq 295 units lattice
fix 4 all append/atoms zhi size 15.0 freq 5 units box
fix A all append/atoms zhi size 1.0 freq 1000 units lattice
```

### 2.13.3 Description

This fix creates atoms on a lattice, appended on the zhi edge of the system box. This can be useful when a shock or wave is propagating from zlo. This allows the system to grow with time to accommodate an expanding wave. A simulation box must already exist, which is typically created via the *create_box* command. Before using this command, a lattice must also be defined using the *lattice* command.

This fix will automatically freeze atoms on the zhi edge of the system, so that overlaps are avoided when new atoms are appended.

The *basis* keyword specifies an atom type that will be assigned to specific basis atoms as they are created. See the *lattice* command for specifics on how basis atoms are defined for the unit cell of the lattice. By default, all created atoms are assigned type = 1 unless this keyword specifies differently.

The *size* keyword defines the size in $z$ of the chunk of material to be added.

The *random* keyword will give the atoms random displacements around their lattice points to simulate some initial temperature.

The *temp* keyword will cause a region to be thermostatted with a Langevin thermostat on the zhi boundary. The size of the region is measured from zhi and is set with the *extent* argument.

The *units* keyword determines the meaning of the distance units used to define a wall position, but only when a numeric constant is used. A *box* value selects standard distance units as defined by the *units* command (e.g., Å for units = real or metal. A *lattice* value means the distance units are in lattice spacings. The *lattice* command must have been previously used to define the lattice spacings.

---

### 2.13.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.13.5 Restrictions

This fix style is part of the SHOCK package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The boundary on which atoms are added with append/atoms must be shrink/minimum. The opposite boundary may be any boundary type other than periodic.

### 2.13.6 Related commands

*fix wall/piston* command

### 2.13.7 Default

The keyword defaults are size = 0.0, freq = 0, units = lattice. All added atoms are of type 1 unless the basis keyword is used.

## 2.14 fix atc command

Deprecated since version 10Sep2025.

The ATC package and the *fix atc* command have been removed from LAMMPS.

## 2.15 fix atom/swap command

### 2.15.1 Syntax

```
fix ID group-ID atom/swap N X seed T keyword values ...
```

- ID, group-ID are documented in *fix* command
- atom/swap = style name of this fix command
- N = invoke this fix every N steps
- X = number of swaps to attempt every N steps
- seed = random # seed (positive integer)
- T = scaling temperature of the MC swaps (temperature units)
- one or more keyword/value pairs may be appended to args
- keyword = *types* or *mu* or *ke* or *semi-grand* or *region*

```
types values = two or more atom types (1-Ntypes or type label)
mu values = chemical potential of swap types (energy units)
ke value = no or yes
  no = no conservation of kinetic energy after atom swaps
  yes = kinetic energy is conserved after atom swaps
semi-grand value = no or yes
  no = particle type counts and fractions conserved
  yes = semi-grand canonical ensemble, particle fractions not conserved
region value = region-ID
  region-ID = ID of region to use as an exchange/move volume
```

### 2.15.2 Examples

```
fix 2 all atom/swap 1 1 29494 300.0 ke no types 1 2
fix myFix all atom/swap 100 1 12345 298.0 region my_swap_region types 5 6
fix SGMC all atom/swap 1 100 345 1.0 semi-grand yes types 1 2 3 mu 0.0 4.3 -5.0
```

### 2.15.3 Description

This fix performs Monte Carlo swaps of atoms of one given atom type with atoms of the other given atom types. The specified scaling temperature $T$ is used in the Metropolis criterion dictating swap probabilities.

Perform $X$ swaps of atoms of one type with atoms of another type according to a Monte Carlo probability. Swap candidates must be in the fix group, must be in the region (if specified), and must be of one of the listed types. Swaps are attempted between candidates that are chosen randomly with equal probability among the candidate atoms. Swaps are not attempted between atoms of the same type since nothing would happen.

All atoms in the simulation domain can also be moved using regular time integration displacements (e.g., via *fix nvt*), resulting in a hybrid MC+MD simulation, where $X$ MC swap attempts are made once every $N$ MD steps. A

smaller-than-usual timestep size may be needed when running such a hybrid simulation, especially if the swapped atoms are not well equilibrated.

**Note**

To run an MC-only simulation (no MD), you should define no time-integration fix, set the *thermo* command to 1, set $N$ to 1, and set $X$ small enough to see the MC evolution of the system. But if $X$ is too small, the overhead at the start and stop of MC moves each timestep will slow down the simulation.

The *types* keyword is required. At least two atom types must be specified. If not using *semi-grand*, exactly two atom types are required.

The *ke* keyword can be set to *no* to turn off kinetic energy conservation for swaps. The default is *yes*, which means that swapped atoms have their velocities scaled by the ratio of the masses of the swapped atom types. This ensures that the kinetic energy of each atom is the same after the swap as it was before the swap, even though the atom masses have changed.

The *semi-grand* keyword can be set to *yes* to switch to the semi-grand canonical ensemble as discussed in *(Sadigh)*. This means that the total number of each particle type does not need to be conserved. The default is *no*, which means that the only kind of swap allowed exchanges an atom of one type with an atom of a different given type. In other words, the relative mole fractions of the swapped atoms remains constant. Whereas in the semi-grand canonical ensemble, the composition of the system can change. Note that when using *semi-grand*, atoms in the fix group whose type is not listed in the *types* keyword are ineligible for attempted conversion. An attempt is made to switch the selected atom (if eligible) to one of the other listed types with equal probability. Acceptance of each attempt depends upon the Metropolis criterion.

The *mu* keyword allows users to specify chemical potentials. This is required and allowed only when using *semi-grand*. All chemical potentials are absolute, so there is one for each swap type listed following the *types* keyword. In semi-grand canonical ensemble simulations the chemical composition of the system is controlled by the difference in these values. So shifting all values by a constant amount will have no effect on the simulation.

This command may optionally use the *region* keyword to define swap volume. The specified region must have been previously defined with a *region* command. It must be defined with side = *in*. Swap attempts occur only between atoms that are both within the specified region. Swaps are not otherwise attempted.

You should ensure you do not swap atoms belonging to a molecule, or LAMMPS will eventually generate an error when it tries to find those atoms. LAMMPS will warn you if any of the atoms eligible for swapping have a non-zero molecule ID, but does not check for this at the time of swapping.

If not using *semi-grand* this fix checks to ensure all atoms of the given types have the same atomic charge. LAMMPS does not enforce this in general, but it is needed for this fix to simplify the swapping procedure. Successful swaps will swap the atom type and charge of the swapped atoms. Conversely, when using *semi-grand*, it is assumed that all the atom types involved in switches have the same charge. Otherwise, charge would not be conserved. As a consequence, no checks on atomic charges are performed, and successful switches update the atom type but not the atom charge. While it is possible to use *semi-grand* with groups of atoms that have different charges, these charges will not be changed when the atom types change. The same applies for systems with per-atom masses: non *semi-grand* will swap atom masses, but the masses have to be the same each for the atom types. When using *semi-grand* no per-atom masses are changed.

Since this fix computes total potential energies before and after proposed swaps, even complicated potential energy calculations are acceptable, including the following:

- long-range electrostatics ($k$-space)
- many body pair styles
- hybrid pair styles (with restrictions)

- EAM pair styles
- triclinic systems

Some fixes have an associated potential energy. Examples of such fixes include: *efield*, *gravity*, *addforce*, *langevin*, *restrain*, *temp/berendsen*, *temp/rescale*, and *wall fixes*. For that energy to be included in the total potential energy of the system (the quantity used when performing GCMC moves), you **must** enable the *fix_modify* *energy* option for that fix. The doc pages for individual *fix* commands specify if this should be done.

---

### 2.15.4 Dump image info

Added in version 11Feb2026.

Fix *atom/swap* supports the *fix* keyword of *dump image*. The fix will pass geometry information about atoms involved in a swap to *dump image* so that these atoms can be highlighted in the visualization as additional spheres. For how long those additional spheres will be shown depends on the value of the *vizsteps* setting (default is 1000) which can be changed by using the *fix_modify command*. If an atom is involved in multiple swaps, the check on showing the additional graphics depends on the timestep of its last swap.

The color of the additional spheres is by default that of the atom type *before* the swap when using color styles "type" or "element". With color style "const" the default value of "white" can be changed using *dump_modify fcolor*. The transparency is by default fully opaque and can be changed with *dump_modify ftrans*.

The *fflag1* setting of *dump image fix* has no effect.

The *fflag2* setting allows you to set the radius of the added spheres, since the radius is set to zero internally.

---

### 2.15.5 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the state of the fix to *binary restart files*. This includes information about the random number generator seed, the next timestep for MC exchanges, the number of exchange attempts and successes, etc. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

> **Note**
>
> For this to work correctly, the timestep must **not** be changed after reading the restart with *reset_timestep*. The fix will try to detect it and stop with an error.

None of the *fix_modify* options are relevant to this fix.

This fix computes a global vector of length 2, which can be accessed by various *output commands*. The vector values are the following global cumulative quantities:

1. swap attempts
2. swap accepts

The vector values calculated by this fix are "intensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.15.6 Restrictions

This fix is part of the MC package. It is only enabled if LAMMPS was built with that package. See the *Build package* doc page for more info.

When this fix is used with a *hybrid pair style* system, only swaps between atom types of the same sub-style (or combination of sub-styles) are permitted.

This fix can be used with systems that have per-atom masses (e.g. atom style sphere) provided all atoms of the types handled by this fix have the same mass per type. The fix will check for that. In case both, per-type and per-atom masses are present, a warning is printed.

### 2.15.7 Related commands

*fix nvt*, *neighbor*, *fix deposit*, *fix evaporate*, *delete_atoms*, *fix gcmc*, *fix mol/swap*, *fix sgcmc*

### 2.15.8 Default

The option defaults are *ke* = yes, *semi-grand* = no, *mu* = 0.0 for all atom types.

---

**(Sadigh)** B Sadigh, P Erhart, A Stukowski, A Caro, E Martinez, and L Zepeda-Ruiz, Phys. Rev. B, 85, 184203 (2012).

## 2.16 fix atom_weight/apip command

### 2.16.1 Syntax

```
fix ID group-ID atom_weight/apip nevery fast_potential precise_potential lambda_input␣
↪lambda_zone group_lambda_input [no_rescale]
```

- ID, group-ID are documented in *fix* command
- atom_weight/apip = style name of this fix command
- nevery = perform load calculation every this many steps
- fast_potential = *eam* or *ace* for time measurements of the corresponding pair_style or float for constant time
- precise_potential = *ace* for a time measurement of the pair_style pace/apip or float for constant time
- lambda_input = *lambda/input* for a time measurement of pair_style lambda/input/apip or float for constant time
- lambda_zone = *lambda/zone* for a time measurement of pair_style lambda/zone/apip or float for constant time
- group_lambda_input = group-ID of the group for which lambda_input is computed
- no_rescale = do not rescale the work per processor to the measured total force-computation time

### 2.16.2 Examples

```
fix 2 all atom_weight/apip 50 eam ace lambda/input lambda/zone all
fix 2 all atom_weight/apip 50 1e-05 0.0004 4e-06 4e-06 all
fix 2 all atom_weight/apip 50 ace ace 4e-06 4e-06 all no_rescale
```

### 2.16.3 Description

This command approximates the load every atom causes when an adaptive-precision interatomic potential (APIP) according to *(Immel)* is used. This approximated load can be saved as atomic variable and used as input for the dynamic load balancing via the *fix balance* command.

An adaptive-precision potential like *eam/apip* and *pace/apip* is calculated only for a subset of atoms. The switching parameter that determines per atom, which potential energy is used, can be also calculated by *pair_style lambda/input/apip*. A spatial switching zone, that ensures a smooth transition between two different interatomic potentials, can be calculated by *pair_style lambda/zone/apip*. Thus, there are up to four force-subroutines, that are computed only for a subset of atoms and combined via the pair_style *hybrid/overlay*. For all four force-subroutines, the average work per atom is be measured per processor by the corresponding pair_style. This fix extracts these measurements of the pair styles every *nevery* steps. The average compute times are used to calculates a per-atom vector with the approximated atomic weight, whereas the average compute time of the four subroutines contributes only to the load of atoms, for which the corresponding subroutine was calculated. If not disabled via *no_rescale*, the so calculated load is rescaled per processor so that the total atomic compute time matches the also measured total compute time of the whole pair_style. This atomic weight is intended to be used as input for *fix balance*:

```
variable nevery equal 10
fix weight_atom all atom_weight/apip ${nevery} eam ace lambda/input lambda/zone all
variable myweight atom f_weight_atom
fix balance all balance ${nevery} 1.1 rcb weight var myweight
```

Furthermore, this fix provides the over the processors averaged compute time of the four pair_styles, which are used to approximate the atomic weight, as vector.

---

### 2.16.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix produces a per-atom vector that contains the atomic weight of each atom. The per-atom vector can only be accessed on timesteps that are multiples of *nevery*.

Furthermore, this fix computes a global vector of length 4 with statistical information about the four different (possibly) measured compute times per force subroutine. The four values in the vector are as follows:

1. average compute time for one atom using the fast pair_style
2. average compute time for one atom using the precise pair_style
3. average compute time of lambda/input/apip for one atom
4. average compute time of lambda/zone/apip for one atom

The compute times are computed as average of all processors that measured at least one computation of the corresponding style. The vector values calculated by this fix are "intensive" and updated whenever the per-atom vector is computed, i.e., in timesteps that are multiples of *nevery*.

The vector and the per-atom vector can be accessed by various *output commands*.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

---

### 2.16.5 Restrictions

This fix is part of the APIP package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.16.6 Related commands

*fix balance*, *fix lambda/apip*, *fix lambda_thermostat/apip*, *pair_style lambda/zone/apip*, *pair_style lambda/input/apip*, *pair_style eam/apip*, *pair_style pace/apip*,

### 2.16.7 Default

*no_rescale* is not used by default.

---

**(Immel)** Immel, Drautz and Sutmann, J Chem Phys, 162, 114119 (2025)

## 2.17 fix ave/atom command

### 2.17.1 Syntax

```
fix ID group-ID ave/atom Nevery Nrepeat Nfreq value1 value2 ...
```

- ID, group-ID are documented in *fix* command
- ave/atom = style name of this fix command
- Nevery = use input values every this many timesteps
- Nrepeat = # of times to use input values for calculating averages
- Nfreq = calculate averages every this many timesteps
- one or more input values can be listed
- value = *x*, *y*, *z*, *vx*, *vy*, *vz*, *fx*, *fy*, *fz*, c_ID, c_ID[i], f_ID, f_ID[i], v_name

```
x,y,z,vx,vy,vz,fx,fy,fz = atom attribute (position, velocity, force component)
c_ID = per-atom vector calculated by a compute with ID
c_ID[I] = Ith column of per-atom array calculated by a compute with ID, I can␣
↪include wildcard (see below)
f_ID = per-atom vector calculated by a fix with ID
f_ID[I] = Ith column of per-atom array calculated by a fix with ID, I can include␣
↪wildcard (see below)
v_name = per-atom vector calculated by an atom-style variable with name
```

### 2.17.2 Examples

```
fix 1 all ave/atom 1 100 100 vx vy vz
fix 1 all ave/atom 10 20 1000 c_my_stress[1]
fix 1 all ave/atom 10 20 1000 c_my_stress[*]
```

### 2.17.3 Description

Use one or more per-atom vectors as inputs every few timesteps, and average them atom by atom over longer timescales. The resulting per-atom averages can be used by other *output commands* such as the *fix ave/chunk* or *dump custom* commands.

The group specified with the command means only atoms within the group have their averages computed. Results are set to 0.0 for atoms not in the group.

Each input value can be an atom attribute (position, velocity, force component) or can be the result of a *compute* or *fix* or the evaluation of an atom-style *variable*. In the latter cases, the compute, fix, or variable must produce a per-atom vector, not a global quantity or local quantity. If you wish to time-average global quantities from a compute, fix, or variable, then see the *fix ave/time* command.

Each per-atom value of each input vector is averaged independently.

*Computes* that produce per-atom vectors or arrays are those which have the word *atom* in their style name. See the doc pages for individual *fixes* to determine which ones produce per-atom vectors or arrays. *Variables* of style *atom* are the only ones that can be used with this fix since they produce per-atom vectors.

Note that for values from a compute or fix, the bracketed index I can be specified using a wildcard asterisk with the index to effectively specify multiple values. This takes the form "*" or "*n" or "m*" or "m*n". If $N$ is the size of the vector (for *mode* = scalar) or the number of columns in the array (for *mode* = vector), then an asterisk with no numeric values means all indices from 1 to $N$. A leading asterisk means all indices from 1 to n (inclusive). A trailing asterisk means all indices from m to $N$ (inclusive). A middle asterisk means all indices from m to n (inclusive).

Using a wildcard is the same as if the individual columns of the array had been listed one by one. For example, these two fix ave/atom commands are equivalent, since the *compute stress/atom* command creates a per-atom array with six columns:

```
compute my_stress all stress/atom NULL
fix 1 all ave/atom 10 20 1000 c_my_stress[*]
fix 1 all ave/atom 10 20 1000 c_my_stress[1] c_my_stress[2] &
                              c_my_stress[3] c_my_stress[4] &
                              c_my_stress[5] c_my_stress[6]
```

---

The $N_{\text{every}}$, $N_{\text{repeat}}$, and $N_{\text{freq}}$ arguments specify on what timesteps the input values will be used in order to contribute to the average. The final averaged quantities are generated on timesteps that are a multiple of $N_{\text{freq}}$. The average is over $N_{\text{repeat}}$ quantities, computed in the preceding portion of the simulation every $N_{\text{every}}$ timesteps. $N_{\text{freq}}$ must be a multiple of $N_{\text{every}}$ and $N_{\text{every}}$ must be non-zero even if $N_{\text{repeat}}$ is 1. Also, the timesteps contributing to the average value cannot overlap; that is, $N_{\text{repeat}} \times N_{\text{every}}$ cannot exceed $N_{\text{freq}}$.

For example, if $N_{\text{every}} = 2$, $N_{\text{repeat}} = 6$, and $N_{\text{freq}} = 100$, then values on timesteps 90, 92, 94, 96, 98, and 100 will be used to compute the final average on time step 100. Similarly for timesteps 190, 192, 194, 196, 198, and 200 on time step 200, etc.

---

The atom attribute values ($x$, $y$, $z$, $vx$, $vy$, $vz$, $fx$, $fy$, and $fz$) are self-explanatory. Note that other atom attributes can be used as inputs to this fix by using the *compute property/atom* command and then specifying an input value from that compute.

**Note**

The $x$, $y$, and $z$ attributes are values that are re-wrapped inside the periodic box whenever an atom crosses a periodic boundary. Thus, if you time-average an atom that spends half of its time on either side of the periodic box, you will

get a value in the middle of the box. If this is not what you want, consider averaging unwrapped coordinates, which can be provided by the *compute property/atom* command via its *xu*, *yu*, and *zu* attributes.

If a value begins with "c_", a compute ID must follow which has been previously defined in the input script. If no bracketed term is appended, the per-atom vector calculated by the compute is used. If a bracketed term containing an index *I* is appended, the $I^{\text{th}}$ column of the per-atom array calculated by the compute is used. Users can also write code for their own compute styles and *add them to LAMMPS*. See the discussion above for how *I* can be specified with a wildcard asterisk to effectively specify multiple values.

If a value begins with "f_", a fix ID must follow which has been previously defined in the input script. If no bracketed term is appended, the per-atom vector calculated by the fix is used. If a bracketed term containing an index *I* is appended, the $I^{\text{th}}$ column of the per-atom array calculated by the fix is used. Note that some fixes only produce their values on certain timesteps, which must be compatible with $N_{\text{every}}$, else an error will result. Users can also write code for their own fix styles and *add them to LAMMPS*. See the discussion above for how *I* can be specified with a wildcard asterisk to effectively specify multiple values.

If a value begins with "v_", a variable name must follow which has been previously defined in the input script as an *atom-style variable*. Variables of style *atom* can reference thermodynamic keywords or invoke other computes, fixes, or variables when they are evaluated, so this is a very general means of generating per-atom quantities to time average.

---

### 2.17.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global scalar or vector quantities are stored by this fix for access by various *output commands*.

This fix produces a per-atom vector or array which can be accessed by various *output commands*. A vector is produced if only a single quantity is averaged by this fix. If two or more quantities are averaged, then an array of values is produced. The per-atom values can only be accessed on timesteps that are multiples of $N_{\text{freq}}$ since that is when averaging is performed.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.17.5 Restrictions

none

### 2.17.6 Related commands

*compute*, *fix ave/histo*, *fix ave/chunk*, *fix ave/time*, *variable*,

### 2.17.7 Default

none

## 2.18 fix ave/chunk command

### 2.18.1 Syntax

```
fix ID group-ID ave/chunk Nevery Nrepeat Nfreq chunkID value1 value2 ... keyword args ...
```

- ID, group-ID are documented in *fix* command

- ave/chunk = style name of this fix command
- Nevery = use input values every this many timesteps
- Nrepeat = # of times to use input values for calculating averages
- Nfreq = calculate averages every this many timesteps
- chunkID = ID of *compute chunk/atom* command
- one or more input values can be listed
- value = *vx*, *vy*, *vz*, *fx*, *fy*, *fz*, *density/mass*, *density/number*, *mass*, *temp*, c_ID, c_ID[I], f_ID, f_ID[I], v_name

```
vx,vy,vz,fx,fy,fz,mass = atom attribute (velocity, force component, mass)
density/number, density/mass = number or mass density (per volume)
temp = temperature
c_ID = per-atom vector calculated by a compute with ID
c_ID[I] = Ith column of per-atom array calculated by a compute with ID, I can␣
↪include wildcard (see below)
f_ID = per-atom vector calculated by a fix with ID
f_ID[I] = Ith column of per-atom array calculated by a fix with ID, I can include␣
↪wildcard (see below)
v_name = per-atom vector calculated by an atom-style variable with name
```

- zero or more keyword/arg pairs may be appended
- keyword = *norm* or *ave* or *bias* or *adof* or *cdof* or *file* or *append* or *overwrite* or *format* or *title1* or *title2* or *title3*

```
norm arg = all or sample or none = how output on Nfreq steps is normalized
  all = output is sum of atoms across all Nrepeat samples, divided by atom count
  sample = output is sum of Nrepeat sample averages, divided by Nrepeat
  none = output is sum of Nrepeat sample sums, divided by Nrepeat
ave args = one or running or window M
  one = output new average value every Nfreq steps
  running = output cumulative average of all previous Nfreq steps
  window M = output average of M most recent Nfreq steps
bias arg = bias-ID
  bias-ID = ID of a temperature compute that removes a velocity bias for␣
↪temperature calculation
adof value = dof_per_atom
  dof_per_atom = define this many degrees-of-freedom per atom for temperature␣
↪calculation
cdof value = dof_per_chunk
  dof_per_chunk = define this many degrees-of-freedom per chunk for temperature␣
↪calculation
file arg = filename
  filename = file to write results to
append arg = filename
  filename = file to append results to
overwrite arg = none = overwrite output file with only latest output
format arg = string
  string = C-style format string
title1 arg = string
  string = text to print as 1st line of output file
title2 arg = string
  string = text to print as 2nd line of output file
title3 arg = string
```

```
string = text to print as 3rd line of output file
```

### 2.18.2 Examples

```
fix 1 all ave/chunk 10000 1 10000 binchunk c_myCentro title1 "My output values"
fix 1 flow ave/chunk 100 10 1000 molchunk vx vz norm sample file vel.profile
fix 1 flow ave/chunk 100 5 1000 binchunk density/mass ave running
fix 1 flow ave/chunk 100 5 1000 binchunk density/mass ave running
```

**Note**

Changed in version 31May2016.

If you are trying to replace a deprecated fix ave/spatial command with the newer, more flexible fix ave/chunk and *compute chunk/atom* commands, you simply need to split the fix ave/spatial arguments across the two new commands. For example, this command:

```
fix 1 flow ave/spatial 100 10 1000 y 0.0 1.0 vx vz norm sample file vel.profile
```

could be replaced by:

```
compute cc1 flow chunk/atom bin/1d y 0.0 1.0
fix 1 flow ave/chunk 100 10 1000 cc1 vx vz norm sample file vel.profile
```

### 2.18.3 Description

Use one or more per-atom vectors as inputs every few timesteps, sum the values over the atoms in each chunk at each timestep, then average the per-chunk values over longer timescales. The resulting chunk averages can be used by other *output commands* such as *thermo_style custom*, and can also be written to a file.

In LAMMPS, chunks are collections of atoms defined by a *compute chunk/atom* command, which assigns each atom to a single chunk (or no chunk). The ID for this command is specified as chunkID. For example, a single chunk could be the atoms in a molecule or atoms in a spatial bin. See the *compute chunk/atom* page and the *Howto chunk* page for details of how chunks can be defined and examples of how they can be used to measure properties of a system.

Note that if the *compute chunk/atom* command defines spatial bins, the fix ave/chunk command performs a similar computation as the *fix ave/grid* command. However, the per-bin outputs from the fix ave/chunk command are global; each processor stores a copy of the entire set of bin data. By contrast, the *fix ave/grid* command uses a distributed grid where each processor owns a subset of the bins. Thus it is more efficient to use the *fix ave/grid* command when the grid is large and a simulation is run on many processors.

Note that only atoms in the specified group contribute to the summing and averaging calculations. The *compute chunk/atom* command defines its own group as well as an optional region. Atoms will have a chunk ID = 0, meaning they belong to no chunk, if they are not in that group or region. Thus you can specify the "all" group for this command if you simply want to use the chunk definitions provided by chunkID.

Each specified per-atom value can be an atom attribute (position, velocity, force component), a number or mass density, a mass or temperature, or the result of a *compute* or *fix* or the evaluation of an atom-style *variable*. In the latter cases, the compute, fix, or variable must produce a per-atom quantity, not a global quantity. Note that the *compute property/atom* command provides access to any attribute defined and stored by atoms. If you wish to time-average global quantities from a compute, fix, or variable, then see the *fix ave/time* command.

The per-atom values of each input vector are summed and averaged independently of the per-atom values in other input vectors.

*Computes* that produce per-atom quantities are those which have the word *atom* in their style name. See the doc pages for individual *fixes* to determine which ones produce per-atom quantities. *Variables* of style *atom* are the only ones that

can be used with this fix since all other styles of variable produce global quantities.

Note that for values from a compute or fix that produces a per-atom array (multiple values per atom), the bracketed index I can be specified using a wildcard asterisk with the index to effectively specify multiple values. This takes the form "*" or "*n" or "n*" or "m*n". If $N$ = the size of the vector (for *mode* = scalar) or the number of columns in the array (for *mode* = vector), then an asterisk with no numeric values means all indices from 1 to $N$. A leading asterisk means all indices from 1 to n (inclusive). A trailing asterisk means all indices from m to $N$ (inclusive). A middle asterisk means all indices from m to n (inclusive).

Using a wildcard is the same as if the individual columns of the array had been listed one by one. For example, these two fix ave/chunk commands are equivalent, since the *compute property/atom* command creates, in this case, a per-atom array with three columns:

```
compute myAng all property/atom angmomx angmomy angmomz
fix 1 all ave/chunk 100 1 100 cc1 c_myAng[*] file tmp.angmom
fix 2 all ave/chunk 100 1 100 cc1 c_myAng[1] c_myAng[2] c_myAng[3] file tmp.angmom
```

**Note**

This fix works by creating an array of size $N_{\text{chunk}} \times N_{\text{values}}$ on each processor. $N_{\text{chunk}}$ is the number of chunks, which is defined by the *compute chunk/atom* command. $N_{\text{values}}$ is the number of input values specified. Each processor loops over its atoms, tallying its values to the appropriate chunk. Then the entire array is summed across all processors. This means that using a large number of chunks will incur an overhead in memory and computational cost (summing across processors), so be careful to define a reasonable number of chunks.

---

The $N_{\text{every}}$, $N_{\text{repeat}}$, and $N_{\text{freq}}$ arguments specify on what time steps the input values will be accessed and contribute to the average. The final averaged quantities are generated on time steps that are a multiples of $N_{\text{freq}}$. The average is over $N_{\text{repeat}}$ quantities, computed in the preceding portion of the simulation every $N_{\text{every}}$ time steps. $N_{\text{freq}}$ must be a multiple of $N_{\text{every}}$ and $N_{\text{every}}$ must be non-zero even if $N_{\text{repeat}} = 1$. Also, the time steps contributing to the average value cannot overlap (i.e., $N_{\text{repeat}} \times N_{\text{every}}$ cannot exceed $N_{\text{freq}}$).

For example, if $N_{\text{every}} = 2$, $N_{\text{repeat}} = 6$, and $N_{\text{freq}} = 100$, then values on time steps 90, 92, 94, 96, 98, 100 will be used to compute the final average on time step 100. Similarly for time steps 190, 192, 194, 196, 198, 200 on time step 200, etc. If $N_{\text{repeat}} = 1$ and $N_{\text{freq}} = 100$, then no time averaging is done; values are simply generated on time steps 100, 200, etc.

Each input value can also be averaged over the atoms in each chunk. The way the averaging is done across the $N_{\text{repeat}}$ time steps to produce output on the $N_{\text{freq}}$ time steps, and across multiple $N_{\text{freq}}$ outputs, is determined by the *norm* and *ave* keyword settings, as discussed below.

**Note**

To perform per-chunk averaging within a $N_{\text{freq}}$ time window, the number of chunks $N_{\text{chunk}}$ defined by the *compute chunk/atom* command must remain constant. If the *ave* keyword is set to *running* or *window* then $N_{\text{chunk}}$ must remain constant for the duration of the simulation. This fix forces the chunk/atom compute specified by chunkID to hold $N_{\text{chunk}}$ constant for the appropriate time windows, by not allowing it to re-calculate $N_{\text{chunk}}$, which can also affect how it assigns chunk IDs to atoms. This is particularly important to understand if the chunks defined by the *compute chunk/atom* command are spatial bins. If its *units* keyword is set to *box* or *lattice*, then the number of bins $N_{\text{chunk}}$ and size of each bin will be fixed over the $N_{\text{freq}}$ time window, which can affect which atoms are discarded if the simulation box size changes. If its *units* keyword is set to *reduced*, then the number of bins $N_{\text{chunk}}$ will still be fixed, but the size of each bin can vary at each time step if the simulation box size changes (e.g., for an NPT simulation).

---

The atom attribute values (*vx*, *vy*, *vz*, *fx*, *fy*, *fz*, *mass*) are self-explanatory. As noted above, any other atom attributes can be used as input values to this fix by using the *compute property/atom* command and then specifying an input value from that compute.

The *density/number* value means the number density is computed for each chunk (i.e., number/volume). The *density/mass* value means the mass density is computed for each chunk (i.e., total-mass/volume). The output values are in units of 1/volume or mass density (mass/volume). See the *units* command page for the definition of density for each choice of units (e.g., g/cm$^3$). If the chunks defined by the *compute chunk/atom* command are spatial bins, the volume is the bin volume. Otherwise, it is the volume of the entire simulation box.

The *temp* value means the temperature is computed for each chunk, by the formula

$$\mathrm{KE} = \frac{\mathrm{DOF}}{2} k_B T,$$

where KE is the total kinetic energy of the chunk of atoms (sum of $\frac{1}{2}mv^2$), DOF is the the total number of degrees of freedom for all atoms in the chunk, $k_B$ is the Boltzmann constant, and $T$ is the absolute temperature.

The DOF is calculated as $N$*adof + cdof, where $N$ is the number of atoms in the chunk, adof is the number of degrees of freedom per atom, and cdof is the number of degrees of freedom per chunk. By default, adof = 2 or 3 = dimensionality of system, as set via the *dimension* command, and cdof = 0.0. This gives the usual formula for temperature.

Note that currently this temperature only includes translational degrees of freedom for each atom. No rotational degrees of freedom are included for finite-size particles. Also, no degrees of freedom are subtracted for any velocity bias or constraints that are applied, such as *compute temp/partial*, *fix shake*, or *fix rigid*. This is because those degrees of freedom (e.g., a constrained bond) could apply to sets of atoms that are both included and excluded from a specific chunk, and hence the concept is somewhat ill-defined. In some cases, you can use the *adof* and *cdof* keywords to adjust the calculated degrees of freedom appropriately, as explained below.

Also note that a bias can be subtracted from atom velocities before they are used in the above formula for KE, by using the *bias* keyword. This allows, for example, a thermal temperature to be computed after removal of a flow velocity profile.

Note that the per-chunk temperature calculated by this fix and the *compute temp/chunk* command can be different. The compute calculates the temperature for each chunk for a single snapshot. This fix can do that but can also time average those values over many snapshots, or it can compute a temperature as if the atoms in the chunk on different time steps were collected together as one set of atoms to calculate their temperature. The compute allows the center-of-mass velocity of each chunk to be subtracted before calculating the temperature; this fix does not.

If a value begins with "c_", a compute ID must follow which has been previously defined in the input script. If no bracketed integer is appended, the per-atom vector calculated by the compute is used. If a bracketed integer is appended, the Ith column of the per-atom array calculated by the compute is used. Users can also write code for their own compute styles and *add them to LAMMPS*. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

If a value begins with "f_", a fix ID must follow which has been previously defined in the input script. If no bracketed integer is appended, the per-atom vector calculated by the fix is used. If a bracketed integer is appended, the Ith column of the per-atom array calculated by the fix is used. Note that some fixes only produce their values on certain time steps, which must be compatible with $N_{\mathrm{every}}$, else an error results. Users can also write code for their own fix styles and *add them to LAMMPS*. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

If a value begins with "v_", a variable name must follow which has been previously defined in the input script. Variables of style *atom* can reference thermodynamic keywords and various per-atom attributes, or invoke other computes, fixes, or variables when they are evaluated, so this is a very general means of generating per-atom quantities to average within chunks.

---

Additional optional keywords also affect the operation of this fix and its outputs.

The *norm* keyword affects how averaging is done for the per-chunk values that are output every $N_{\text{freq}}$ time steps.

It the *norm* setting is *all*, which is the default, a chunk value is summed over all atoms in all $N_{\text{repeat}}$ samples, as is the count of atoms in the chunk. The averaged output value for the chunk on the $N_{\text{freq}}$ time steps is Total-sum / Total-count. In other words it is an average over atoms across the entire $N_{\text{freq}}$ timescale. For the *density/number* and *density/mass* values, the volume (bin volume or system volume) used in the final normalization will be the volume at the final $N_{\text{freq}}$ time step. For the *temp* values, degrees of freedom and kinetic energy are summed separately across the entire $N_{\text{freq}}$ timescale, and the output value is calculated by dividing those two sums.

If the *norm* setting is *sample*, the chunk value is summed over atoms for each sample, as is the count, and an "average sample value" is computed for each sample (i.e., Sample-sum / Sample-count). The output value for the chunk on the $N_{\text{freq}}$ time steps is the average of the $N_{\text{repeat}}$ "average sample values" (i.e., the sum of $N_{\text{repeat}}$ "average sample values" divided by $N_{\text{repeat}}$). In other words, it is an average of an average. For the *density/number* and *density/mass* values, the volume (bin volume or system volume) used in the per-sample normalization will be the current volume at each sampling step.

If the *norm* setting is *none*, a similar computation as for the *sample* setting is done, except the individual "average sample values" are "summed sample values". A summed sample value is simply the chunk value summed over atoms in the sample, without dividing by the number of atoms in the sample. The output value for the chunk on the $N_{\text{freq}}$ timesteps is the average of the $N_{\text{repeat}}$ "summed sample values" (i.e., the sum of $N_{\text{repeat}}$ "summed sample values" divided by $N_{\text{repeat}}$). For the *density/number* and *density/mass* values, the volume (bin volume or system volume) used in the per-sample sum normalization will be the current volume at each sampling step.

---

The *ave* keyword determines how the per-chunk values produced every $N_{\text{freq}}$ steps are averaged with values produced on previous steps that were multiples of $N_{\text{freq}}$, before they are accessed by another output command or written to a file.

If the *ave* setting is *one*, which is the default, then the chunk values produced on timesteps that are multiples of $N_{\text{freq}}$ are independent of each other; they are output as-is without further averaging.

If the *ave* setting is *running*, then the chunk values produced on timesteps that are multiples of $N_{\text{freq}}$ are summed and averaged in a cumulative sense before being output. Each output chunk value is thus the average of the chunk value produced on that timestep with all preceding values for the same chunk. This running average begins when the fix is defined; it can only be restarted by deleting the fix via the *unfix* command, or re-defining the fix by re-specifying it.

If the *ave* setting is *window*, then the chunk values produced on timesteps that are multiples of $N_{\text{freq}}$ are summed and averaged within a moving "window" of time, so that the last $M$ values for the same chunk are used to produce the output. For example, if $M = 3$ and $N_{\text{freq}} = 1000$, then the output on step 10000 will be the average of the individual chunk values on time steps 8000, 9000, and 10000. Outputs on early steps will average over less than $M$ values if they are not available.

---

The *bias* keyword specifies the ID of a temperature compute that removes a "bias" velocity from each atom, specified as *bias-ID*. It is only used when the *temp* value is calculated, to compute the thermal temperature of each chunk after the translational kinetic energy components have been altered in a prescribed way (e.g., to remove a flow velocity profile). See the doc pages for individual computes that calculate a temperature to see which ones implement a bias.

The *adof* and *cdof* keywords define the values used in the degree of freedom (DOF) formula described above for temperature calculation for each chunk. They are only used when the *temp* value is calculated. They can be used to calculate a more appropriate temperature for some kinds of chunks. Here are three examples:

If spatially binned chunks contain some number of water molecules and *fix shake* is used to make each molecule rigid, then you could calculate a temperature with six degrees of freedom (DOF) (three translational, three rotational) per molecule by setting *adof* to 2.0.

If *compute temp/partial* is used with the *bias* keyword to only allow the $x$ component of velocity to contribute to the temperature, then *adof* = 1.0 would be appropriate.

If each chunk consists of a large molecule, with some number of its bonds constrained by *fix shake* or the entire molecule by *fix rigid/small*, *adof* = 0.0 and *cdof* could be set to the remaining degrees of freedom for the entire molecule (entire chunk in this case), that is, 6 for 3d or 3 for 2d for a rigid molecule.

---

Added in version 17Apr2024: new keyword *append*

The *file* or *append* keywords allow a filename to be specified. If *file* is used, then the filename is overwritten if it already exists. If *append* is used, then the filename is appended to if it already exists, or created if it does not exist. Every $N_{\text{freq}}$ timesteps, a section of chunk info will be written to a text file in the following format. A line with the timestep and number of chunks is written. Then one line per chunk is written, containing the chunk ID ($1 - N_{\text{chunk}}$), an optional original ID value, optional coordinate values for chunks that represent spatial bins, the number of atoms in the chunk, and one or more calculated values. More explanation of the optional values is given below. The number of values in each line corresponds to the number of values specified in the fix ave/chunk command. The number of atoms and the value(s) are summed or average quantities, as explained above.

The *overwrite* keyword will continuously overwrite the output file with the latest output, so that it only contains one timestep worth of output. This option can only be used with the *ave running* setting.

The *format* keyword sets the numeric format of each value when it is printed to a file via the *file* keyword. Note that all values are floating point quantities. The default format is " %g". You can specify a higher precision if desired (e.g., " %20.16g").

The *title1* and *title2* and *title3* keywords allow specification of the strings that will be printed as the first three lines of the output file, assuming the *file* keyword was used. LAMMPS uses default values for each of these, so they do not need to be specified.

By default, these header lines are as follows:

```
# Chunk-averaged data for fix ID and group name
# Timestep Number-of-chunks
# Chunk (OrigID) (Coord1) (Coord2) (Coord3) Ncount value1 value2 ...
```

In the first line, ID and name are replaced with the fix-ID and group name. The second line describes the two values that are printed at the first of each section of output. In the third line the values are replaced with the appropriate value names (e.g., *fx* or c_myCompute[2]).

The words in parenthesis only appear with corresponding columns if the chunk style specified for the *compute chunk/atom* command supports them. The OrigID column is only used if the *compress* keyword was set to *yes* for the *compute chunk/atom* command. This means that the original chunk IDs (e.g., molecule IDs) will have been compressed to remove chunk IDs with no atoms assigned to them. Thus a compressed chunk ID of 3 may correspond to an original chunk ID or molecule ID of 415. The OrigID column will list 415 for the third chunk.

The CoordN columns only appear if a *binning* style was used in the *compute chunk/atom* command. For *bin/1d*, *bin/2d*, and *bin/3d* styles the column values are the center point of the bin in the corresponding dimension. Just Coord1 is used for *bin/1d*, Coord2 is added for *bin/2d*, Coord3 is added for *bin/3d*. For *bin/sphere*, just Coord1 is used, and it is the radial coordinate. For *bin/cylinder*, Coord1 and Coord2 are used. Coord1 is the radial coordinate (away from the cylinder axis), and coord2 is the coordinate along the cylinder axis.

Note that if the value of the *units* keyword used in the *compute chunk/atom command* is *box* or *lattice*, the coordinate values will be in distance *units*. If the value of the *units* keyword is *reduced*, the coordinate values will be in unitless reduced units (0–1). This is not true for the Coord1 value of style *bin/sphere* or *bin/cylinder* which both represent radial dimensions. Those values are always in distance *units*.

---

### 2.18.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix computes a global array of values which can be accessed by various *output commands*. The values can only be accessed on timesteps that are multiples of $N_{\text{freq}}$, since that is when averaging is performed. The global array has # of rows = the number of chunks $N_{\text{chunk}}$, as calculated by the specified *compute chunk/atom* command. The # of columns is $M+1+N_{\text{values}}$, where $M \in \{1,\ldots,4\}$, depending on whether the optional columns for OrigID and CoordN are used, as explained above. Following the optional columns, the next column contains the count of atoms in the chunk, and the remaining columns are the Nvalue quantities. When the array is accessed with a row $I$ that exceeds the current number of chunks, than a 0.0 is returned by the fix instead of an error, since the number of chunks can vary as a simulation runs depending on how that value is computed by the compute chunk/atom command.

The array values calculated by this fix are treated as "intensive", since they are typically already normalized by the count of atoms in each chunk.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.18.5 Restrictions

none

### 2.18.6 Related commands

*compute*, *fix ave/atom*, *fix ave/histo*, *fix ave/time*, *variable*, *fix ave/correlate*, *fix ave/grid*

### 2.18.7 Default

The option defaults are norm = all, ave = one, bias = none, no file output, and title 1,2,3 = strings as described above.

## 2.19 fix ave/correlate command

### 2.19.1 Syntax

```
fix ID group-ID ave/correlate Nevery Nrepeat Nfreq value1 value2 ... keyword args ...
```

- ID, group-ID are documented in *fix* command
- ave/correlate = style name of this fix command
- Nevery = use input values every this many timesteps
- Nrepeat = # of correlation time windows to accumulate
- Nfreq = calculate time window averages every this many timesteps
- one or more input values can be listed
- value = c_ID, c_ID[N], f_ID, f_ID[N], v_name

```
c_ID = global scalar calculated by a compute with ID
c_ID[I] = Ith component of global vector calculated by a compute with ID, I can␣
↪include wildcard (see below)
f_ID = global scalar calculated by a fix with ID
f_ID[I] = Ith component of global vector calculated by a fix with ID, I can include␣
↪wildcard (see below)
v_name = global value calculated by an equal-style variable with name
```

(continues on next page)

(continued from previous page)

```
v_name[I] = Ith component of a vector-style variable with name, I can include␣
↪wildcard (see below)
```

- zero or more keyword/arg pairs may be appended
- keyword = *type* or *ave* or *start* or *prefactor* or *file* or *overwrite* or *title1* or *title2* or *title3*

```
type arg = auto or upper or lower or auto/upper or auto/lower or full or first
  auto = correlate each value with itself
  upper = correlate each value with each succeeding value
  lower = correlate each value with each preceding value
  auto/upper = auto + upper
  auto/lower = auto + lower
  full = correlate each value with every other value, including itself = auto +␣
↪upper + lower
  first = correlate each value with the first value
ave args = one or running
  one = zero the correlation accumulation every Nfreq steps
  running = accumulate correlations continuously
start args = Nstart
  Nstart = start accumulating correlations on this timestep
prefactor args = value
  value = prefactor to scale all the correlation data by
file arg = filename
  filename = name of file to output correlation data to
overwrite arg = none = overwrite output file with only latest output
title1 arg = string
  string = text to print as 1st line of output file
title2 arg = string
  string = text to print as 2nd line of output file
title3 arg = string
  string = text to print as 3rd line of output file
```

### 2.19.2 Examples

```
fix 1 all ave/correlate 5 100 1000 c_myTemp file temp.correlate
fix 1 all ave/correlate 1 50 10000 &
          c_thermo_press[1] c_thermo_press[2] c_thermo_press[3] &
          type upper ave running title1 "My correlation data"
fix 1 all ave/correlate 1 50 10000 c_thermo_press[*]
```

### 2.19.3 Description

Use one or more global scalar values as inputs every few timesteps, calculate time correlations between them at varying time intervals, and average the correlation data over longer timescales. The resulting correlation values can be time integrated by *variables* or used by other *output commands* such as *thermo_style custom*, and can also be written to a file. See the *fix ave/correlate/long* command for an alternate method for computing correlation functions efficiently over very long time windows.

The group specified with this command is ignored. However, note that specified values may represent calculations performed by computes and fixes which store their own "group" definitions.

Each listed value can be the result of a *compute* or *fix* or the evaluation of an equal-style or vector-style *variable*. In each case, the compute, fix, or variable must produce a global quantity, not a per-atom or local quantity. If you wish to

spatial- or time-average or histogram per-atom quantities from a compute, fix, or variable, then see the *fix ave/chunk*, *fix ave/atom*, or *fix ave/histo* commands. If you wish to convert a per-atom quantity into a single global value, see the *compute reduce* command.

The input values must be all scalars. What kinds of correlations between input values are calculated is determined by the *type* keyword as discussed below.

*Computes* that produce global quantities are those which do not have the word *atom* in their style name. Only a few *fixes* produce global quantities. See the doc pages for individual fixes for info on which ones produce such values. *Variables* of style *equal* and *vector* are the only ones that can be used with this fix. Variables of style *atom* cannot be used, since they produce per-atom values.

---

For input values from a compute or fix or variable , the bracketed index I can be specified using a wildcard asterisk with the index to effectively specify multiple values. This takes the form "*" or "*n" or "m*" or "m*n". If $N$ is the size of the vector, then an asterisk with no numeric values means all indices from 1 to $N$. A leading asterisk means all indices from 1 to n (inclusive). A trailing asterisk means all indices from m to $N$ (inclusive). A middle asterisk means all indices from m to n (inclusive).

Using a wildcard is the same as if the individual elements of the vector had been listed one by one. For example, the following two fix ave/correlate commands are equivalent, since the *compute pressure* command creates a global vector with six values:

```
compute myPress all pressure NULL
fix 1 all ave/correlate 1 50 10000 c_myPress[*]
fix 1 all ave/correlate 1 50 10000 &
          c_myPress[1] c_myPress[2] c_myPress[3] &
          c_myPress[4] c_myPress[5] c_myPress[6]
```

**Note**

For a vector-style variable, only the wildcard forms "*n" or "m*n" are allowed. You must specify the upper bound, because vector-style variable lengths are not determined until the variable is evaluated. If n is specified larger than the vector length turns out to be, zeroes are output for missing vector values.

---

The $N_{\text{every}}$, $N_{\text{repeat}}$, and $N_{\text{freq}}$ arguments specify on what timesteps the input values will be used to calculate correlation data. The input values are sampled every $N_{\text{every}}$ time steps. The correlation data for the preceding samples is computed on time steps that are a multiple of $N_{\text{freq}}$. Consider a set of samples from some initial time up to an output timestep. The initial time could be the beginning of the simulation or the last output time; see the *ave* keyword for options. For the set of samples, the correlation value $C_{ij}$ is calculated as:

$$C_{ij}(\Delta t) = \langle V_i(t) V_j(t + \Delta t) \rangle \, ,$$

which is the correlation value between input values $V_i$ and $V_j$, separated by time $\Delta t$. Note that the second value $V_j$ in the pair is always the one sampled at the later time. The average is an average over every pair of samples in the set that are separated by time $\Delta t$. The maximum $\Delta t$ used is of size $(N_{\text{repeat}} - 1)N_{\text{every}}$. Thus the correlation between a pair of input values yields $N_{\text{repeat}}$ correlation data:

$$C_{ij}(0), C_{ij}(N_{\text{every}}), C_{ij}(2N_{\text{every}}), \ldots, C_{ij}\big((N_{\text{repeat}} - 1)N_{\text{every}}\big)$$

For example, if $N_{\text{every}} = 5$, $N_{\text{repeat}} = 6$, and $N_{\text{freq}} = 100$, then values on time steps $0, 5, 10, 15, \ldots, 100$ will be used to compute the final averages on time step 100. Six averages will be computed: $C_{ij}(0)$, $C_{ij}(5)$, $C_{ij}(10)$, $C_{ij}(15)$, $C_{ij}(20)$,

and $C_{ij}(25)$. $C_{ij}(10)$ on time step 100 will be the average of 19 samples, namely $V_i(0)V_j(10)$, $V_i(5)V_j(15)$, $V_i(10)V_j(20)$, $V_i(15)V_j(25), \ldots, V_i(85)V_j(95)$, and $V_i(90)V_j(100)$.

$N_{\text{freq}}$ must be a multiple of $N_{\text{every}}$; $N_{\text{every}}$ and $N_{\text{repeat}}$ must be non-zero. Also, if the *ave* keyword is set to *one* which is the default, then $N_{\text{freq}} \geq (N_{\text{repeat}} - 1)N_{\text{every}}$ is required.

---

If a value begins with "c_", a compute ID must follow which has been previously defined in the input script. If no bracketed term is appended, the global scalar calculated by the compute is used. If a bracketed term is appended, the $I^{\text{th}}$ element of the global vector calculated by the compute is used. See the discussion above for how $I$ can be specified with a wildcard asterisk to effectively specify multiple values.

Note that there is a *compute reduce* command that can sum per-atom quantities into a global scalar or vector which can then be accessed by fix ave/correlate. It can also be a compute defined not in your input script, but by *thermodynamic output* or other fixes such as *fix nvt* or *fix temp/rescale*. See the doc pages for these commands which give the IDs of these computes. Users can also write code for their own compute styles and *add them to LAMMPS*.

If a value begins with "f_", a fix ID must follow which has been previously defined in the input script. If no bracketed term is appended, the global scalar calculated by the fix is used. If a bracketed term is appended, the $I^{\text{th}}$ element of the global vector calculated by the fix is used. See the discussion above for how $I$ can be specified with a wildcard asterisk to effectively specify multiple values.

Note that some fixes only produce their values on certain timesteps, which must be compatible with $N_{\text{every}}$, else an error will result. Users can also write code for their own fix styles and *add them to LAMMPS*.

If a value begins with "v_", a variable name must follow which has been previously defined in the input script. Only equal-style or vector-style variables can be referenced; the latter requires a bracketed term to specify the $I^{\text{th}}$ element of the vector calculated by the variable. See the *variable* command for details. Note that variables of style *equal* or *vector* define a formula which can reference individual atom properties or thermodynamic keywords, or they can invoke other computes, fixes, or variables when they are evaluated, so this is a very general means of specifying quantities to time correlate.

---

Additional optional keywords also affect the operation of this fix.

The *type* keyword determines which pairs of input values are correlated with each other. For $N$ input values $V_i$, with $i \in \{1, \ldots, N\}$, let the number of pairs be $N_{\text{pair}}$. Note that the second value in the pair, $V_i(t)V_j(t + \Delta t)$, is always the one sampled at the later time.

- If *type* is set to *auto* then each input value is correlated with itself (i.e., $C_{ii} = V_i^2$ for $i \in \{1, \ldots, N\}$, so $N_{\text{pair}} = N$).
- If *type* is set to *upper* then each input value is correlated with every succeeding value (i.e., $C_{ij} = V_iV_j$ for $i < j$, so $N_{\text{pair}} = N(N - 1)/2$).
- If *type* is set to *lower* then each input value is correlated with every preceding value (i.e., $C_{ij} = V_iV_j$ for $i > j$, so $N_{\text{pair}} = N(N - 1)/2$).
- If *type* is set to *auto/upper* then each input value is correlated with itself and every succeeding value (i.e., $C_{ij} = V_iV_j$ for $i \geq j$, so $N_{\text{pair}} = N(N + 1)/2$).
- If *type* is set to *auto/lower* then each input value is correlated with itself and every preceding value (i.e., $C_{ij} = V_iV_j$ for $i \leq j$, so $N_{\text{pair}} = N(N + 1)/2$).
- If *type* is set to *full* then each input value is correlated with itself and every other value (i.e., $C_{ij} = V_iV_j$ for $\{i, j\} = \{1, N\}$, so $N_{\text{pair}} = N^2$).
- If *type* is set to *first* then each input value is correlated with the first input value (i.e., $C_{ij} = V_1V_j$ for $\{j\} = \{1, N\}$, so $N_{\text{pair}} = N$).

The *ave* keyword determines what happens to the accumulation of correlation samples every $N_{\text{freq}}$ timesteps. If the *ave* setting is *one*, then the accumulation is restarted or zeroed every $N_{\text{freq}}$ timesteps. Thus the outputs on successive $N_{\text{freq}}$ timesteps are essentially independent of each other. The exception is that the $C_{ij}(0) = V_i(t)V_j(t)$ value at a time step $t$, where $t$ is a multiple of $N_{\text{freq}}$, contributes to the correlation output both at time $t$ and at time $t + N_{\text{freq}}$.

If the *ave* setting is *running*, then the accumulation is never zeroed. Thus the output of correlation data at any timestep is the average over samples accumulated every $N_{\text{every}}$ steps since the fix was defined. It can only be restarted by deleting the fix via the *unfix* command, or by re-defining the fix by re-specifying it.

The *start* keyword specifies what time step the accumulation of correlation samples will begin on. The default is step 0. Setting it to a larger value can avoid adding non-equilibrated data to the correlation averages.

The *prefactor* keyword specifies a constant which will be used as a multiplier on the correlation data after it is averaged. It is effectively a scale factor on $V_iV_j$, which can be used to account for the size of the time window or other unit conversions.

The *file* keyword allows a filename to be specified. Every $N_{\text{freq}}$ steps, an array of correlation data is written to the file. The number of rows is $N_{\text{repeat}}$, as described above. The number of columns is $N_{\text{pair}} + 2$, also as described above. Thus the file ends up to be a series of these array sections.

The *overwrite* keyword will continuously overwrite the output file with the latest output, so that it only contains one timestep worth of output. This option can only be used with the *ave running* setting.

The *title1*, *title2*, and *title3* keywords allow specification of the strings that will be printed as the first three lines of the output file, assuming the *file* keyword was used. LAMMPS uses default values for each of these, so they do not need to be specified.

By default, these header lines are as follows:

```
# Time-correlated data for fix ID
# TimeStep Number-of-time-windows
# Index TimeDelta Ncount valueI*valueJ valueI*valueJ ...
```

In the first line, ID is replaced with the fix-ID. The second line describes the two values that are printed at the first of each section of output. In the third line the value pairs are replaced with the appropriate fields from the fix ave/correlate command.

---

Let $S_{ij}$ be a set of time correlation data for input values $I$ and $J$, namely the $N_{\text{repeat}}$ values:

$$S_{ij} = C_{ij}(0), C_{ij}(N_{\text{every}}), C_{ij}(2N_{\text{every}}), \ldots, C_{ijI}\left((N_{\text{repeat}} - 1)N_{\text{every}}\right)$$

As explained below, these data are output as one column of a global array, which is effectively the correlation matrix.

The *trap* function defined for *equal-style variables* can be used to perform a time integration of this vector of data, using a trapezoidal rule. This is useful for calculating various quantities which can be derived from time correlation data. If a normalization factor is needed for the time integration, it can be included in the variable formula or via the *prefactor* keyword.

---

### 2.19.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix computes a global array of values which can be accessed by various *output commands*. The values can only be accessed on timesteps that are multiples of $N_{\text{freq}}$ since that is when averaging is performed. The global array has # of rows $N_{\text{repeat}}$ and # of columns $N_{\text{pair}} + 2$. The first column has the time $\Delta t$ (in time steps) between the pairs of input values used to calculate the correlation, as described above. The second column has the number of samples contributing to the correlation average, as described above. The remaining Npair columns are for $I, J$ pairs of the $N$ input values, as determined by the *type* keyword, as described above.

- For *type* = *auto*, the $N_{\text{pair}} = N$ columns are ordered: $C_{11}, C_{22}, \ldots, C_{NN}$
- For *type* = *upper*, the $N_{\text{pair}} = N(N-1)/2$ columns are ordered: $C_{12}, C_{13}, \ldots, C_{1N}, C_{23}, \ldots, C_{2N}, C_{34}, \ldots, C_{N-1,N}$
- For *type* = *lower*, the $N_{\text{pair}} = N(N-1)/2$ columns are ordered: $C_{21}, C_{31}, C_{32}, C_{41}, C_{42}, C_{43I}, \ldots, C_{N1}, C_{N2}, \ldots, C_{N,N-1}$
- For *type* = *auto/upper*, the $N_{\text{pair}} = N(N+1)/2$ columns are ordered: $C_{11}, C_{12}, C_{13}, \ldots, C_{1N}, C_{22}, C_{23}, \ldots, C_{2N}, C_{33}, C_{34}, \ldots, C_{N-1,N}, C_{NN}$
- For *type* = *auto/lower*, the $N_{\text{pair}} = N(N+1)/2$ columns are ordered: $C_{11}, C_{21}, C_{22}, C_{31}, C_{32}, C_{33}, C_{41}, \ldots, C_{44}, C_{N1}, C_{N2}, \ldots, C_{N,N-1}, C_{NN}$
- For *type* = *full*, the $N_{\text{pair}} = N^2$ columns are ordered: $C_{11}, C_{12}, \ldots, C_{1N}, C_{21}, C_{22}, \ldots, C_{2N}, C_{31}, \ldots, C_{3N}, \ldots, C_{N1}, \ldots, C_{N,N-1}, C_{NN}$
- For *type* = *first*, the $N_{\text{pair}} = N$ columns are ordered: $C_{11}, C_{12}, \ldots, C_{1N}$

The array values calculated by this fix are treated as extensive. If you need to divide them by the number of atoms, you must do this in a later processing step (e.g., when using them in a *variable*).

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.19.5 Restrictions

none

### 2.19.6 Related commands

*fix ave/correlate/long*, *compute*, *fix ave/time*, *fix ave/atom*, *fix ave/chunk*, *fix ave/histo*, *variable*

### 2.19.7 Default

none

The option defaults are ave = one, type = auto, start = 0, no file output, title 1,2,3 = strings as described above, and prefactor = 1.0.

## 2.20 fix ave/correlate/long command

### 2.20.1 Syntax

```
fix ID group-ID ave/correlate/long Nevery Nfreq value1 value2 ... keyword args ...
```

- ID, group-ID are documented in *fix* command
- ave/correlate/long = style name of this fix command
- Nevery = use input values every this many time steps
- Nfreq = save state of the time correlation functions every this many time steps
- one or more input values can be listed
- value = c_ID, c_ID[N], f_ID, f_ID[N], v_name, v_name[I]

```
c_ID = global scalar calculated by a compute with ID
c_ID[I] = Ith component of global vector calculated by a compute with ID, I can␣
↪include wildcard (see below)
f_ID = global scalar calculated by a fix with ID
```

(continues on next page)

(continued from previous page)

```
f_ID[I] = Ith component of global vector calculated by a fix with ID, I can include␣
↪wildcard (see below)
v_name = global value calculated by an equal-style variable with name
v_name[I] = Ith component of a vector-style variable with name, I can include␣
↪wildcard (see below)
```

- zero or more keyword/arg pairs may be appended
- keyword = *type* or *start* or *file* or *overwrite* or *title1* or *title2* or *ncorr* or *nlen* or *ncount*

```
type arg = auto or upper or lower or auto/upper or auto/lower or full or first
  auto = correlate each value with itself
  upper = correlate each value with each succeeding value
  lower = correlate each value with each preceding value
  auto/upper = auto + upper
  auto/lower = auto + lower
  full = correlate each value with every other value, including itself = auto +␣
↪upper + lower
  first = correlate each value with the first value
start args = Nstart
  Nstart = start accumulating correlations on this time step
file arg = filename
  filename = name of file to output correlation data to
overwrite arg = none = overwrite output file with only latest output
title1 arg = string
  string = text to print as 1st line of output file
title2 arg = string
  string = text to print as 2nd line of output file
ncorr arg = Ncorrelators
  Ncorrelators = number of correlators to store
nlen args = Nlen
  Nlen = length of each correlator
ncount args = Ncount
  Ncount = number of values over which successive correlators are averaged
```

### 2.20.2 Examples

```
fix 1 all ave/correlate/long 5 1000 c_myTemp file temp.correlate
fix 1 all ave/correlate/long 1 10000 &
          c_thermo_press[1] c_thermo_press[2] c_thermo_press[3] &
          type upper title1 "My correlation data" nlen 15 ncount 3
fix 1 all ave/correlate/long 1 10000 c_thermo_press[*]
```

### 2.20.3 Description

This fix is similar in spirit and syntax to the *fix ave/correlate*. However, this fix allows the efficient calculation of time correlation functions on-the-fly over extremely long time windows with little additional CPU overhead, using a multiple-$\tau$ method *(Ramirez)* that decreases the resolution of the stored correlation function with time. It is not a full drop-in replacement.

The group specified with this command is ignored. However, note that specified values may represent calculations performed by computes and fixes which store their own "group" definitions.

Each listed value can be the result of a compute or fix or the evaluation of an equal-style or vector-style variable. The

specified indices can include a wildcard string. See the *fix ave/correlate* page for details on that.

The *Nevery* and *Nfreq* arguments specify on what time steps the input values will be used to calculate correlation data and the frequency with which the time correlation functions will be output to a file, respectively. Note that there is no *Nrepeat* argument, unlike the *fix ave/correlate* command.

The optional keywords *ncorr*, *nlen*, and *ncount* are unique to this command and determine the number of correlation points calculated and the memory and CPU overhead used by this calculation. *Nlen* and *ncount* determine the amount of averaging done at longer correlation times. The default values *nlen* = 16 and *ncount* = 2 ensure that the systematic error of the multiple-$\tau$ correlator is always below the level of the statistical error of a typical simulation (which depends on the ensemble size and the simulation length).

The maximum correlation time (in time steps) that can be reached is given by the formula $(nlen - 1)ncount^{(ncorr-1)}$. Longer correlation times are discarded and not calculated. With the default values of the parameters (*ncorr* = 20, *nlen* = 16 and *ncount* = 2), this corresponds to 7864320 time steps. If longer correlation times are needed, the value of ncorr should be increased. Using *nlen* = 16 and *ncount* = 2, with *ncorr* = 30, the maximum number of steps that can be correlated is 80530636808. If *ncorr* = 40, correlation times in excess of $8 \times 10^{12}$ time steps can be calculated.

The total memory needed for each correlation pair is roughly $4 \times ncorr \times nlen \times 8$ bytes. With the default values of the parameters, this corresponds to about 10 KB.

For the meaning of the additional optional keywords, see the *fix ave/correlate* doc page.

### 2.20.4 Restart, fix_modify, output, run start/stop, minimize info

Contrary to *fix ave/correlate* this fix does **not** provide access to its internal data to various output options. Since this fix in intended for the calculation of time correlation functions over very long MD simulations, the information about this fix is written automatically to binary restart files, so that the time correlation calculation can continue in subsequent simulations. None of the fix_modify options are relevant to this fix.

No parameter of this fix can be used with the start/stop keywords of the run command. This fix is not invoked during energy minimization.

### 2.20.5 Restrictions

This compute is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.20.6 Related commands

*fix ave/correlate*

### 2.20.7 Default

none

The option defaults for keywords that are also keywords for the *fix ave/correlate* command are as follows: type = auto, start = 0, no file output, title 1,2 = strings as described on the *fix ave/correlate* doc page.

The option defaults for keywords unique to this command are as follows: ncorr=20, nlen=16, ncount=2.

---

**(Ramirez)** J. Ramirez, S.K. Sukumaran, B. Vorselaars and A.E. Likhtman, J. Chem. Phys. 133, 154103 (2010).

# 2.21 fix ave/grid command

## 2.21.1 Syntax

```
fix ID group-ID ave/grid Nevery Nrepeat Nfreq Nx Ny Nz value1 value2 ... keyword args ...
```

- ID, group-ID are documented in *fix* command
- ave/grid = style name of this fix command
- Nevery = use input values every this many timesteps
- Nrepeat = # of times to use input values for calculating averages
- Nfreq = calculate averages every this many timesteps
- Nx, Ny, Nz = grid size in each dimension
- one or more per-atom or per-grid input values can be listed
- per-atom value = vx, vy, vz, fx, fy, fz, density/mass, density/number, mass, temp, c_ID, c_ID[I], f_ID, f_ID[I], v_name

```
vx,vy,vz,fx,fy,fz,mass = atom attribute (velocity, force component, mass)
density/number, density/mass = number or mass density (per volume)
temp = temperature
c_ID = per-atom vector calculated by a compute with ID
c_ID[I] = Ith column of per-atom array calculated by a compute with ID, I can␣
↪include wildcard (see below)
f_ID = per-atom vector calculated by a fix with ID
f_ID[I] = Ith column of per-atom array calculated by a fix with ID, I can include␣
↪wildcard (see below)
v_name = per-atom vector calculated by an atom-style variable with name
```

- per-grid value = c_ID:gname:dname, c_ID:gname:dname[I], f_ID:gname:dname, f_ID:gname:dname[I]

```
gname = name of grid defined by compute or fix
dname = name of data field defined by compute or fix
c_ID = per-grid vector calculated by a compute with ID
c_ID[I] = Ith column of per-grid array calculated by a compute with ID, I can␣
↪include wildcard (see below)
f_ID = per-grid vector calculated by a fix with ID
f_ID[I] = Ith column of per-grid array calculated by a fix with ID, I can include␣
↪wildcard (see below)
```

- zero or more keyword/arg pairs may be appended
- keyword = *discard* or *norm* or *ave* or *bias* or *adof* or *cdof*

```
discard arg = yes or no
  yes = discard an atom outside grid in a non-periodic dimension
  no = remap an atom outside grid in a non-periodic dimension to first or last grid␣
↪cell
norm arg = all or sample or none = how output on Nfreq steps is normalized
  all = output is sum of atoms across all Nrepeat samples, divided by atom count
  sample = output is sum of Nrepeat sample averages, divided by Nrepeat
  none = output is sum of Nrepeat sample sums, divided by Nrepeat
ave args = one or running or window M
  one = output new average value every Nfreq steps
```

```
    running = output cumulative average of all previous Nfreq steps
    window M = output average of M most recent Nfreq steps
  bias arg = bias-ID
    bias-ID = ID of a temperature compute that removes a velocity bias for␣
 ↪temperature calculation
  adof value = dof_per_atom
    dof_per_atom = define this many degrees-of-freedom per atom for temperature␣
 ↪calculation
  cdof value = dof_per_grid_cell
    dof_per_grid_cell = add this many degrees-of-freedom per grid_cell for␣
 ↪temperature calculation
```

### 2.21.2 Examples

```
fix 1 all ave/grid 10000 1 10000 10 10 10 fx fy fz c_myMSD[*]
fix 1 flow ave/grid 100 10 1000 20 20 30 f_TTM:grid:data
```

### 2.21.3 Description

Overlay the 2d or 3d simulation box with a uniformly spaced 2d or 3d grid and use it to either (a) time-average per-atom quantities for the atoms in each grid cell, or to (b) time-average per-grid quantities produced by other computes or fixes. This fix operates in either "per-atom mode" (all input values are per-atom) or in "per-grid mode" (all input values are per-grid). You cannot use both per-atom and per-grid inputs in the same command.

The grid created by this command is distributed; each processor owns the grid points that are within its subdomain. This is similar to the *fix ave/chunk* command when it uses chunks from the *compute chunk/atom* command which are 2d or 3d regular bins. However, the per-bin outputs in that case are global; each processor stores a copy of the entire set of bin data. Thus it more efficient to use the fix ave/grid command when the grid is large and a simulation is run on many processors.

For per-atom mode, only atoms in the specified group contribute to the summing and averaging calculations. For per-grid mode, the specified group is ignored.

---

The $N_{\text{every}}$, $N_{\text{repeat}}$, and $N_{\text{freq}}$ arguments specify on what time steps the input values will be accessed and contribute to the average. The final averaged quantities are generated on time steps that are a multiples of $N_{\text{freq}}$. The average is over $N_{\text{repeat}}$ quantities, computed in the preceding portion of the simulation every $N_{\text{every}}$ time steps. $N_{\text{freq}}$ must be a multiple of $N_{\text{every}}$ and $N_{\text{every}}$ must be non-zero even if $N_{\text{repeat}} = 1$. Also, the time steps contributing to the average value cannot overlap (i.e., $N_{\text{repeat}} \times N_{\text{every}}$ cannot exceed $N_{\text{freq}}$).

For example, if $N_{\text{every}} = 2$, $N_{\text{repeat}} = 6$, and $N_{\text{freq}} = 100$, then values on time steps 90,92,94,96,98,100 will be used to compute the final average on timestep 100. Similarly for timesteps 190,192,194,196,198,200 on timestep 200, etc. If $N_{\text{repeat}} = 1$ and $N_{\text{freq}} = 100$, then no time averaging is done; values are simply generated on timesteps 100,200,etc.

In per-atom mode, each input value can also be averaged over the atoms in each grid cell. The way the averaging is done across the $N_{\text{repeat}}$ timesteps to produce output on the $N_{\text{freq}}$ timesteps, and across multiple $N_{\text{freq}}$ outputs, is determined by the *norm* and *ave* keyword settings, as discussed below.

---

The $Nx$, $Ny$, and $Nz$ arguments specify the size of the grid that overlays the simulation box. For 2d simulations, $Nz$ must be 1. The $Nx$, $Ny$, $Nz$ values can be any positive integer. The grid can be very coarse compared to the particle count, or very fine. If one or more of the values = 1, then bins are 2d planes or 1d slices of the simulation domain. Note that if the total number of grid cells is small, it may be more efficient to use the *fix ave/chunk* command which can treat a grid defined by the *compute chunk/atom* command as a global grid where each processor owns a copy of all the grid cells. If $Nx = Ny = Nz = 1$ is used, the same calculation would be more efficiently performed by the *fix ave/atom* command.

If the simulation box size or shape changes during a simulation, the grid always conforms to the size/shape of the current simulation box. If one more dimensions have non-periodic shrink-wrapped boundary conditions, as defined by the *boundary* command, then the grid will extend over the (dynamic) shrink-wrapped extent in each dimension. If the box shape is triclinic, as explained in *Howto triclinic*, then the grid is also triclinic; each grid cell is a small triclinic cell with the same shape as the simulation box.

---

In both per-atom and per-grid mode, input values from a compute or fix that produces an array of values (multiple values per atom or per grid point), the bracketed index I can be specified using a wildcard asterisk with the index to effectively specify multiple values. This takes the form "*" or "*n" or "n*" or "m*n". If N = the number of columns in the array (for *mode* = vector), then an asterisk with no numeric values means all indices from 1 to N. A leading asterisk means all indices from 1 to n (inclusive). A trailing asterisk means all indices from n to N (inclusive). A middle asterisk means all indices from m to n (inclusive).

Using a wildcard is the same as if the individual columns of the array had been listed one by one. E.g. if there were a compute fft/grid command which produced 3 values for each grid point, these two fix ave/grid commands would be equivalent:

```
compute myFFT all fft/grid 10 10 10 ...
fix 1 all ave/grid 100 1 100 10 10 10 c_myFFT:grid:data[*]
fix 2 all ave/grid 100 1 100 10 10 10 c_myFFT:grid:data[*][1] c_myFFT:grid:data[*][2] c_
↪myFFT:grid:data[3]
```

---

*Per-atom mode*:

Each specified per-atom value can be an atom attribute (velocity, force component), a number or mass density, a mass or temperature, or the result of a *compute* or *fix* or the evaluation of an atom-style *variable*. In the latter cases, the compute, fix, or variable must produce a per-atom quantity, not a global quantity. Note that the *compute property/atom* command provides access to any attribute defined and stored by atoms.

The per-atom values of each input vector are summed and averaged independently of the per-atom values in other input vectors.

*Computes* that produce per-atom quantities are those which have the word *atom* in their style name. See the doc pages for individual *fixes* to determine which ones produce per-atom quantities. *Variables* of style *atom* are the only ones that can be used with this fix since all other styles of variable produce global quantities.

---

The atom attribute values (vx,vy,vz,fx,fy,fz,mass) are self-explanatory. As noted above, any other atom attributes can be used as input values to this fix by using the *compute property/atom* command and then specifying an input value from that compute.

The *density/number* value means the number density is computed for each grid cell, i.e. number/volume. The *density/mass* value means the mass density is computed for each grid/cell, i.e. total-mass/volume. The output values are in units of 1/volume or density (mass/volume). See the *units* command page for the definition of density for each choice of units, e.g. gram/cm^3.

The *temp* value computes the temperature for each grid cell, by the formula

$$\mathrm{KE} = \frac{\mathrm{DOF}}{2} k_B T,$$

where KE = total kinetic energy of the atoms in the grid cell ( $\frac{1}{2}mv^2$), DOF = the total number of degrees of freedom for all atoms in the grid cell, $k_B$ = Boltzmann constant, and $T$ = temperature.

The DOF is calculated as N*adof + cdof, where N = number of atoms in the grid cell, adof = degrees of freedom per atom, and cdof = degrees of freedom per grid cell. By default adof = 2 or 3 = dimensionality of system, as set via the *dimension* command, and cdof = 0.0. This gives the usual formula for temperature.

Note that currently this temperature only includes translational degrees of freedom for each atom. No rotational degrees of freedom are included for finite-size particles. Also no degrees of freedom are subtracted for any velocity bias or constraints that are applied, such as *compute temp/partial*, or *fix shake* or *fix rigid*. This is because those degrees of freedom (e.g. a constrained bond) could apply to sets of atoms that are both inside and outside a specific grid cell, and hence the concept is somewhat ill-defined. In some cases, you can use the *adof* and *cdof* keywords to adjust the calculated degrees of freedom appropriately, as explained below.

Also note that a bias can be subtracted from atom velocities before they are used in the above formula for KE, by using the *bias* keyword. This allows, for example, a thermal temperature to be computed after removal of a flow velocity profile.

Note that the per-grid-cell temperature calculated by this fix and the *compute temp/chunk* command (using bins) can be different. The compute calculates the temperature for each chunk for a single snapshot. This fix can do that but can also time average those values over many snapshots, or it can compute a temperature as if the atoms in the grid cell on different timesteps were collected together as one set of atoms to calculate their temperature. The compute allows the center-of-mass velocity of each chunk to be subtracted before calculating the temperature; this fix does not.

If a value begins with "c_", a compute ID must follow which has been previously defined in the input script. If no bracketed integer is appended, the per-atom vector calculated by the compute is used. If a bracketed integer is appended, the Ith column of the per-atom array calculated by the compute is used. Users can also write code for their own compute styles and *add them to LAMMPS*. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

If a value begins with "f_", a fix ID must follow which has been previously defined in the input script. If no bracketed integer is appended, the per-atom vector calculated by the fix is used. If a bracketed integer is appended, the Ith column of the per-atom array calculated by the fix is used. Note that some fixes only produce their values on certain timesteps, which must be compatible with $N_{\text{every}}$, else an error results. Users can also write code for their own fix styles and *add them to LAMMPS*. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

If a value begins with "v_", a variable name must follow which has been previously defined in the input script. Variables of style *atom* can reference thermodynamic keywords and various per-atom attributes, or invoke other computes, fixes, or variables when they are evaluated, so this is a very general means of generating per-atom quantities to average within grid cells.

---

*Per-grid mode*:

The attributes that begin with *c_ID* and *f_ID* both take colon-separated fields *gname* and *dname*. These refer to a grid name and data field name which is defined by the compute or fix. Note that a compute or fix can define one or more grids (of different sizes) and one or more data fields for each of those grids. The sizes of all grids used as values for one instance of this fix must be the same.

The *c_ID:gname:dname* and *c_ID:gname:dname[I]* attributes allow per-grid vectors or arrays calculated by a *compute* to be accessed. The ID in the attribute should be replaced by the actual ID of the compute that has been defined previously in the input script.

If *c_ID:gname:dname* is used as a attribute, then the per-grid vector calculated by the compute is accessed. If *c_ID:gname:dname[I]* is used, then I must be in the range from 1-M, which will access the Ith column of the per-grid array with M columns calculated by the compute. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

The *f_ID:gname:dname* and *f_ID:gname:dname[I]* attributes allow per-grid vectors or arrays calculated by a *fix* to be output. The ID in the attribute should be replaced by the actual ID of the fix that has been defined previously in the input script.

If *f_ID:gname:dname* is used as a attribute, then the per-grid vector calculated by the fix is printed. If *f_ID:gname:dname[I]* is used, then I must be in the range from 1-M, which will print the Ith column of the per-grid with M columns calculated by the fix. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

---

Additional optional keywords also affect the operation of this fix and its outputs. Some are only applicable to per-atom mode. Some are applicable to both per-atom and per-grid mode.

The *discard* keyword is only applicable to per-atom mode. If a dimension of the system is non-periodic, then grid cells will only span the box dimension (fixed or shrink-wrap boundaries as set by the *boundary command* command). An atom may thus be slightly outside the range of grid cells on a particular timestep. If *discard* is set to *yes* (the default), then the atom will be assigned to the closest grid cell (lowest or highest) in that dimension. If *discard* is set to *no* the atom will be ignored.

---

The *norm* keyword is only applicable to per-atom mode. In per-grid mode, the *norm* keyword setting is ignored. The output grid value on an $N_{\text{freq}}$ timestep is the sum of the grid values in each of the $N_{\text{repeat}}$ samples, divided by $N_{\text{repeat}}$.

In per-atom mode, the *norm* keyword affects how averaging is done for the per-grid values that are output on an $N_{\text{freq}}$ timestep. $N_{\text{repeat}}$ samples contribute to the output. The *norm* keyword has 3 possible settings: *all* or *sample* or *none*. *All* is the default.

In the formulas that follow, SumI is the sum of a per-atom property over the CountI atoms in a grid cell for a single sample I, where I varies from 1 to N, and N = Nrepeat. These formulas are used for any per-atom input value listed above, except *density/number*, *density/mass*, and *temp*. Those input values are discussed below.

In per-atom mode, for *norm all* the output grid value on the $N_{\text{freq}}$ timestep is an average over atoms across the entire $N_{\text{freq}}$ timescale:

Output = (Sum1 + Sum2 + … + SumN) / (Count1 + Count2 + … + CountN)

In per-atom mode, for *norm sample* the output grid value on the $N_{\text{freq}}$ timestep is an average of an average:

Output = (Sum1/Count1 + Sum2/Count2 + … + SumN/CountN) / Nrepeat

In per-atom mode, for *norm none* the output grid value on the $N_{\text{freq}}$ timestep is not normalized by the atom counts:

Output = (Sum1 + Sum2 + … SumN) / Nrepeat

For *density/number* and *density/mass*, the output value is the same as in the formulas above for *norm all* and *norm sample*, except that the result is also divided by the grid cell volume. For *norm all*, this will be the volume at the final $N_{\text{freq}}$ timestep. For *norm sample*, the divide-by-volume is done for each sample, using the grid cell volume at the sample timestep. For *norm none*, the output is the same as for *norm all*.

For *temp*, the output temperature uses the formula for kinetic energy KE listed above, and is normalized similarly to the formulas above for *norm all* and *norm sample*, except for the way the degrees of freedom (DOF) are calculated. For *norm none*, the output is the same as for *norm all*.

For *norm all*, the DOF = $N_{\text{repeat}} \times$ *cdof* plus *Count* times *adof*, where *Count* = (Count1 + Count2 + … + CountN). The *cdof* and *adof* keywords are discussed below. The output temperature is computed with all atoms across all samples contributing.

For *norm sample*, the DOF for a single sample = *cdof* plus *Count* times *adof*, where *Count* = CountI for a single sample. The output temperature is the average of *Nsample* temperatures calculated for each sample.

Finally, for all 3 *norm* settings the output count of atoms per grid cell is:

Output count = (Count1 + Count2 + … CountN) / Nrepeat

This count is the same for all per-atom input values, including *density/number*, *density/mass*, and *temp*.

---

The *ave* keyword is applied to both per-atom and per-grid mode. It determines how the per-grid values produced once every $N_{\text{freq}}$ steps are averaged with values produced on previous steps that were multiples of $N_{\text{freq}}$, before they are accessed by another output command.

If the *ave* setting is *one*, which is the default, then the grid values produced on $N_{\text{freq}}$ timesteps are independent of each other; they are output as-is without further averaging.

If the *ave* setting is *running*, then the grid values produced on $N_{\text{freq}}$ timesteps are summed and averaged in a cumulative sense before being output. Each output grid value is thus the average of the grid value produced on that timestep with all preceding values for the same grid value. This running average begins when the fix is defined; it can only be restarted by deleting the fix via the *unfix* command, or re-defining the fix by re-specifying it.

If the *ave* setting is *window*, then the grid values produced on $N_{\text{freq}}$ timesteps are summed and averaged within a moving "window" of time, so that the last M values for the same grid are used to produce the output. E.g. if M = 3 and Nfreq = 1000, then the grid value output on step 10000 will be the average of the grid values on steps 8000,9000,10000. Outputs on early steps will average over less than M values if they are not available.

---

The *bias*, *adof*, and *cdof* keywords are only applicable to per-atom mode.

The *bias* keyword specifies the ID of a temperature compute that removes a "bias" velocity from each atom, specified as *bias-ID*. It is only used when the *temp* value is calculated, to compute the thermal temperature of each grid cell after the translational kinetic energy components have been altered in a prescribed way, e.g. to remove a flow velocity profile. See the doc pages for individual computes that calculate a temperature to see which ones implement a bias.

The *adof* and *cdof* keywords define the values used in the degree of freedom (DOF) formula described above for temperature calculation for each grid cell. They are only used when the *temp* value is calculated. They can be used to calculate a more appropriate temperature in some cases. Here are 3 examples:

If grid cells contain some number of water molecules and *fix shake* is used to make each molecule rigid, then you could calculate a temperature with 6 degrees of freedom (DOF) (3 translational, 3 rotational) per molecule by setting *adof* to 2.0.

If *compute temp/partial* is used with the *bias* keyword to only allow the x component of velocity to contribute to the temperature, then *adof* = 1.0 would be appropriate.

Using *cdof* = -2 or -3 (for 2d or 3d simulations) will subtract out 2 or 3 degrees of freedom for each grid cell, similar to how the *compute temp* command subtracts out 3 DOF for the entire system.

---

### 2.21.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix calculates a per-grid array which has one column for each of the specified input values. The units for each column with be in the *units* for the per-atom or per-grid quantity for the corresponding input value. If the fix is used in per-atom mode, it also calculates a per-grid vector with the count of atoms in each grid cell. The number of rows in the per-grid array and number of values in the per-grid vector (distributed across all processors) is Nx * Ny * Nz.

For access by other commands, the name of the single grid produced by this fix is "grid". The names of its two per-grid datums are "data" for the per-grid array and "count" for the per-grid vector (if using per-atom values). Both datums can be accessed by various *output commands*.

In per-atom mode, the per-grid array values calculated by this fix are treated as "intensive", since they are typically already normalized by the count of atoms in each grid cell.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.21.5 Restrictions

none

### 2.21.6 Related commands

*fix ave/atom*, *fix ave/chunk*

### 2.21.7 Default

The option defaults are discard = yes, norm = all, ave = one, and bias = none.

## 2.22 fix ave/histo command

## 2.23 fix ave/histo/weight command

### 2.23.1 Syntax

```
fix ID group-ID style Nevery Nrepeat Nfreq lo hi Nbin value1 value2 ... keyword args ...
```

- ID, group-ID are documented in *fix* command
- style = *ave/histo* or *ave/histo/weight* = style name of this fix command
- Nevery = use input values every this many timesteps
- Nrepeat = # of times to use input values for calculating histogram
- Nfreq = calculate histogram every this many timesteps
- lo,hi = lo/hi bounds within which to histogram
- Nbin = # of histogram bins
- one or more input values can be listed
- value = *x*, *y*, *z*, *vx*, *vy*, *vz*, *fx*, *fy*, *fz*, c_ID, c_ID[N], f_ID, f_ID[N], v_name

```
x,y,z,vx,vy,vz,fx,fy,fz = atom attribute (position, velocity, force component)
c_ID = scalar or vector calculated by a compute with ID
c_ID[I] = Ith component of vector or Ith column of array calculated by a compute␣
↪with ID, I can include wildcard (see below)
f_ID = scalar or vector calculated by a fix with ID
f_ID[I] = Ith component of vector or Ith column of array calculated by a fix with␣
↪ID, I can include wildcard (see below)
v_name = value(s) calculated by an equal-style or vector-style or atom-style␣
↪variable with name
v_name[I] = value calculated by a vector-style variable with name, I can include␣
↪wildcard (see below)
```

- zero or more keyword/arg pairs may be appended
- keyword = *mode* or *kind* or *file* or *append* or *ave* or *start* or *beyond* or *overwrite* or *title1* or *title2* or *title3*

```
mode arg = scalar or vector
  scalar = all input values are scalars
  vector = all input values are vectors
kind arg = global or peratom or local
file arg = filename
```

```
    filename = name of file to output histogram(s) to
  append arg = filename
    filename = name of file to append histogram(s) to
  ave args = one or running or window
    one = output a new average value every Nfreq steps
    running = output cumulative average of all previous Nfreq steps
    window M = output average of M most recent Nfreq steps
  start args = Nstart
    Nstart = start averaging on this timestep
  beyond arg = ignore or end or extra
    ignore = ignore values outside histogram lo/hi bounds
    end = count values outside histogram lo/hi bounds in end bins
    extra = create 2 extra bins for value outside histogram lo/hi bounds
  overwrite arg = none = overwrite output file with only latest output
  title1 arg = string
    string = text to print as 1st line of output file
  title2 arg = string
    string = text to print as 2nd line of output file
  title3 arg = string
    string = text to print as 3rd line of output file, only for vector mode
```

### 2.23.2 Examples

```
fix 1 all ave/histo 100 5 1000 0.5 1.5 50 c_myTemp file temp.histo ave running
fix 1 all ave/histo 100 5 1000 -5 5 100 c_thermo_press[2] c_thermo_press[3] title1 "My␣
↪output values"
fix 1 all ave/histo 100 5 1000 -5 5 100 c_thermo_press[*]
fix 1 all ave/histo 1 100 1000 -2.0 2.0 18 vx vy vz mode vector ave running beyond extra
fix 1 all ave/histo/weight 1 1 1 10 100 2000 c_XRD[1] c_XRD[2]
```

### 2.23.3 Description

Use one or more values as inputs every few timesteps to create a single histogram. The histogram can then be averaged over longer timescales. The resulting histogram can be used by other *output commands*, and can also be written to a file. The fix ave/histo/weight command has identical syntax to fix ave/histo, except that exactly two values must be specified. See details below.

The group specified with this command is ignored for global and local input values. For per-atom input values, only atoms in the group contribute to the histogram. Note that regardless of the specified group, specified values may represent calculations performed by computes and fixes which store their own "group" definition.

A histogram is simply a count of the number of values that fall within a histogram bin. *Nbins* are defined, with even spacing between *lo* and *hi*. Values that fall outside the lo/hi bounds can be treated in different ways; see the discussion of the *beyond* keyword below.

Each input value can be an atom attribute (position, velocity, force component) or can be the result of a *compute* or *fix* or the evaluation of an equal-style or vector-style or atom-style *variable*. The set of input values can be either all global, all per-atom, or all local quantities. Inputs of different kinds (e.g. global and per-atom) cannot be mixed. Atom attributes are per-atom vector values. See the page for individual "compute" and "fix" commands to see what kinds of quantities they generate.

Note that a compute or fix can produce multiple kinds of data (global, per-atom, local). If LAMMPS cannot unambiguously determine which kind of data to use, the optional *kind* keyword discussed below can force the desired disambiguation.

Note that the output of this command is a single histogram for all input values combined together, not one histogram per input value. See below for details on the format of the output of this fix.

The input values must either be all scalars or all vectors (or arrays), depending on the setting of the *mode* keyword.

If *mode* = scalar, then the input values must be scalars, or vectors with a bracketed term appended, indicating the Ith value of the vector is used.

If *mode* = vector, then the input values must be vectors, or arrays with a bracketed term appended, indicating the Ith column of the array is used.

If the fix ave/histo/weight command is used, exactly two values must be specified. If the values are vectors, they must be the same length. The first value (a scalar or vector) is what is histogrammed into bins, in the same manner the fix ave/histo command operates. The second value (a scalar or vector) is used as a "weight". This means that instead of each value tallying a "1" to its bin, the corresponding weight is tallied. For example, the $N^{\text{th}}$ entry (weight) in the second vector is tallied to the bin corresponding to the $N^{\text{th}}$ entry in the first vector.

---

For input values from a compute or fix or variable, the bracketed index I can be specified using a wildcard asterisk with the index to effectively specify multiple values. This takes the form "*" or "*n" or "m*" or "m*n". If $N$ is the size of the vector (for *mode* = scalar) or the number of columns in the array (for *mode* = vector), then an asterisk with no numeric values means all indices from 1 to $N$. A leading asterisk means all indices from 1 to n (inclusive). A trailing asterisk means all indices from m to $N$ (inclusive). A middle asterisk means all indices from m to n (inclusive).

Using a wildcard is the same as if the individual elements of the vector or columns of the array had been listed one by one. For example, the following two fix ave/histo commands are equivalent, since the *compute com/chunk* command creates a global array with three columns:

```
compute myCOM all com/chunk
fix 1 all ave/histo 100 1 100 -10.0 10.0 100 c_myCOM[*] file tmp1.com mode vector
fix 2 all ave/histo 100 1 100 -10.0 10.0 100 c_myCOM[1] c_myCOM[2] c_myCOM[3] file tmp2.
↪com mode vector
```

**Note**

For a vector-style variable, only the wildcard forms "*n" or "m*n" are allowed. You must specify the upper bound, because vector-style variable lengths are not determined until the variable is evaluated. If n is specified larger than the vector length turns out to be, zeroes are output for missing vector values.

---

The $N_{\text{every}}$, $N_{\text{repeat}}$, and $N_{\text{freq}}$ arguments specify on what time steps the input values will be used in order to contribute to the histogram. The final histogram is generated on time steps that are multiple of $N_{\text{freq}}$. It is averaged over $N_{\text{repeat}}$ histograms, computed in the preceding portion of the simulation every $N_{\text{every}}$ time steps. $N_{\text{freq}}$ must be a multiple of $N_{\text{every}}$ and $N_{\text{every}}$ must be non-zero even if $N_{\text{repeat}}$ is 1. Also, the time steps contributing to the histogram value cannot overlap (i.e., $N_{\text{repeat}} \times N_{\text{every}}$ cannot exceed $N_{\text{freq}}$).

For example, if $N_{\text{every}} = 2$, $N_{\text{repeat}} = 6$, and $N_{\text{freq}} = 100$, then input values on time steps 90, 92, 94, 96, 98, and 100 will be used to compute the final histogram on timestep 100. Similarly for timesteps 190, 192, 194, 196, 198, and 200 on timestep 200, etc. If $N_{\text{repeat}} = 1$ and $N_{\text{freq}} = 100$, then no time averaging of the histogram is done; a histogram is simply generated on timesteps 100, 200, etc.

---

The atom attribute values ($x$, $y$, $z$, $vx$, $vy$, $vz$, $fx$, $fy$, and $fz$) are self-explanatory. Note that other atom attributes can be used as inputs to this fix by using the *compute property/atom* command and then specifying an input value from that compute.

If a value begins with "c_", a compute ID must follow which has been previously defined in the input script. If *mode* = scalar, then if no bracketed term is appended, the global scalar calculated by the compute is used. If a bracketed term is appended, the Ith element of the global vector calculated by the compute is used. If *mode* = vector, then if no bracketed term is appended, the global or per-atom or local vector calculated by the compute is used. If a bracketed term is appended, the Ith column of the global or per-atom or local array calculated by the compute is used. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

Note that there is a *compute reduce* command that can sum per-atom quantities into a global scalar or vector, which can then be accessed by fix ave/histo. It can also be a compute defined not in your input script, but by *thermodynamic output* or other fixes such as *fix nvt* or *fix temp/rescale*. See the doc pages for these commands which give the IDs of these computes. Users can also write code for their own compute styles and *add them to LAMMPS*.

If a value begins with "f_", a fix ID must follow which has been previously defined in the input script. If *mode* = scalar, then if no bracketed term is appended, the global scalar calculated by the fix is used. If a bracketed term is appended, the Ith element of the global vector calculated by the fix is used. If *mode* = vector, then if no bracketed term is appended, the global or per-atom or local vector calculated by the fix is used. If a bracketed term is appended, the $I^{\text{th}}$ column of the global or per-atom or local array calculated by the fix is used. See the discussion above for how $I$ can be specified with a wildcard asterisk to effectively specify multiple values.

Note that some fixes only produce their values on certain timesteps, which must be compatible with $N_{\text{every}}$, else an error will result. Users can also write code for their own fix styles and *add them to LAMMPS*.

If a value begins with "v_", a variable name must follow which has been previously defined in the input script. If *mode* = scalar, then only equal-style or vector-style variables can be used, which both produce global values. In this mode, a vector-style variable requires a bracketed term to specify the $I^{\text{th}}$ element of the vector calculated by the variable. If *mode* = vector, then only vector-style or atom-style variables can be used, which produce a global or per-atom vector respectively. The vector-style variable must be used without a bracketed term. See the *variable* command for details.

Note that variables of style *equal*, *vector*, and *atom* define a formula which can reference individual atom properties or thermodynamic keywords, or they can invoke other computes, fixes, or variables when they are evaluated, so this is a very general means of specifying quantities to histogram.

---

Additional optional keywords also affect the operation of this fix.

If the *mode* keyword is set to *scalar*, then all input values must be global scalars, or elements of global vectors. If the *mode* keyword is set to *vector*, then all input values must be global or per-atom or local vectors, or columns of global or per-atom or local arrays.

The *kind* keyword only needs to be used if any of the specified input computes or fixes produce more than one kind of output (global, per-atom, local). If not, LAMMPS will determine the kind of data all the inputs produce and verify it is all the same kind. If not, an error will be triggered. If a compute or fix produces more than one kind of output, the *kind* keyword should be used to specify which output will be used. The other input arguments must still be consistent.

The *beyond* keyword determines how input values that fall outside the *lo* to *hi* bounds are treated. Values such that *lo* $\leq$ value $\leq$ *hi* are assigned to one bin. Values on a bin boundary are assigned to the lower of the two bins. If *beyond* is set to *ignore* then values $<$ *lo* and values $>$ *hi* are ignored (i.e., they are not binned). If *beyond* is set to *end*, then values $<$ *lo* are counted in the first bin and values $>$ *hi* are counted in the last bin. If *beyond* is set to *extra*, then two extra bins are created so that there are $N_{\text{bins}} + 2$ total bins. Values $<$ *lo* are counted in the first bin and values $>$ *hi* are counted in the last bin ($N_{\text{bins}} + 2$). Values between *lo* and *hi* (inclusive) are counted in bins 2 through $N_{\text{bins}} + 1$. The "coordinate" stored and printed for these two extra bins is *lo* and *hi*.

The *ave* keyword determines how the histogram produced every $N_{\text{freq}}$ steps are averaged with histograms produced on previous steps that were multiples of $N_{\text{freq}}$, before they are accessed by another output command or written to a file.

If the *ave* setting is *one*, then the histograms produced on timesteps that are multiples of $N_{\text{freq}}$ are independent of each other; they are output as-is without further averaging.

If the *ave* setting is *running*, then the histograms produced on timesteps that are multiples of $N_{\text{freq}}$ are summed and averaged in a cumulative sense before being output. Each bin value in the histogram is thus the average of the bin

value produced on that timestep with all preceding values for the same bin. This running average begins when the fix is defined; it can only be restarted by deleting the fix via the *unfix* command, or by re-defining the fix by re-specifying it.

If the *ave* setting is *window*, then the histograms produced on timesteps that are multiples of $N_{\text{freq}}$ are summed within a moving "window" of time, so that the last $M$ histograms are used to produce the output (e.g., if $M = 3$ and $N_{\text{freq}} = 1000$, then the output on step 10000 will be the combined histogram of the individual histograms on steps 8000, 9000, and 10000. Outputs on early steps will be sums over less than $M$ histograms if they are not available.

The *start* keyword specifies what timestep histogramming will begin on. The default is step 0. Often input values can be 0.0 at time 0, so setting *start* to a larger value can avoid including a 0.0 in a running or windowed histogram.

Added in version 17Apr2024: new keyword *append*

The *file* or *append* keywords allow a filename to be specified. If *file* is used, then the filename is overwritten if it already exists. If *append* is used, then the filename is appended to if it already exists, or created if it does not exist. Every *Nfreq* steps, one histogram is written to the file. This includes a leading line that contains the timestep, number of bins, the total count of values contributing to the histogram, the count of values that were not histogrammed (see the *beyond* keyword), the minimum value encountered, and the maximum value encountered. The min/max values include values that were not histogrammed. Following the leading line, one line per bin is written into the file. Each line contains the bin #, the coordinate for the center of the bin (between *lo* and *hi*), the count of values in the bin, and the normalized count. The normalized count is the bin count divided by the total count (not including values not histogrammed), so that the normalized values sum to 1.0 across all bins.

The *overwrite* keyword will continuously overwrite the output file with the latest output, so that it only contains one timestep worth of output. This option can only be used with the *ave running* setting.

The *title1*, *title2*, and *title3* keywords allow specification of the strings that will be printed as the first three lines of the output file, assuming the *file* keyword was used. LAMMPS uses default values for each of these, so they do not need to be specified.

By default, these header lines are as follows:

```
# Histogram for fix ID
# TimeStep Number-of-bins Total-counts Missing-counts Min-value Max-value
# Bin Coord Count Count/Total
```

In the first line, ID is replaced with the fix-ID. The second line describes the six values that are printed at the first of each section of output. The third describes the four values printed for each bin in the histogram.

---

### 2.23.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix produces a global vector and global array which can be accessed by various *output commands*. The values can only be accessed on timesteps that are multiples of $N_{\text{freq}}$ since that is when a histogram is generated. The global vector has four values:

1. total counts in the histogram
2. values that were not histogrammed (see *beyond* keyword)
3. min value of all input values, including ones not histogrammed
4. max value of all input values, including ones not histogrammed

The global array has $N_{\text{bins}}$ rows and three columns. The first column has the bin coordinate, the second column has the count of values in that histogram bin, and the third column has the bin count divided by the total count (not including missing counts), so that the values in the third column sum to 1.0.

The vector and array values calculated by this fix are all treated as intensive. If this is not the case (e.g., due to histogramming per-atom input values), then you will need to account for that when interpreting the values produced by this fix.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.23.5 Restrictions

none

### 2.23.6 Related commands

*compute*, *fix ave/atom*, *fix ave/chunk*, *fix ave/time*, *variable*, *fix ave/correlate*,

### 2.23.7 Default

none

The option defaults are mode = scalar, kind = figured out from input arguments, ave = one, start = 0, no file output, beyond = ignore, and title 1,2,3 = strings as described above.

## 2.24 fix ave/moments command

### 2.24.1 Syntax

```
fix ID group-ID ave/moments Nevery Nrepeat Nfreq value1 value2 ... moment1 moment2 ...␣
↪keyword args ...
```

- ID, group-ID are documented in *fix* command
- ave/moments = style name of this fix command
- Nevery = use input values every this many time steps
- Nrepeat = # of times to use input values for calculating averages
- Nfreq = calculate averages every this many time steps
- one or more input variables can be listed
- value = v_name

```
c_ID = global scalar calculated by a compute with ID
c_ID[I] = Ith component of global vector calculated by a compute with ID, I can␣
↪include wildcard (see below)
f_ID = global scalar calculated by a fix with ID
f_ID[I] = Ith component of global vector calculated by a fix with ID, I can include␣
↪wildcard (see below)
v_name = value calculated by an equal-style variable with name
v_name[I] = value calculated by a vector-style variable with name, I can include␣
↪wildcard (see below)
```

- one or more moments to compute can be listed
- moment = *mean* or *stddev* or *variance* or *skew* or *kurtosis*, see exact definitions below.
- zero or more keyword/arg pairs may be appended

- keyword = *start* or *history*

```
start args = Nstart
  Nstart = invoke first after this time step
history args = Nrecent
  Nrecent = keep a history of up to Nrecent outputs
```

### 2.24.2 Examples

```
fix 1 all ave/moments 1 1000 100 v_volume mean stddev
fix 1 all ave/moments 1 200 1000 v_volume variance kurtosis history 10
```

### 2.24.3 Description

Added in version 12Jun2025.

Using one or more values as input, calculate the moments of the underlying (population) distributions based on samples collected every few time steps over a time step window. The definitions of the moments calculated are given below.

The group specified with this command is ignored. However, note that specified values may represent calculations performed by computes and fixes which store their own "group" definitions.

Each listed value can be the result of a *compute* or *fix* or the evaluation of an equal-style or vector-style *variable*. In each case, the compute, fix, or variable must produce a global quantity, not a per-atom or local quantity. If you wish to spatial- or time-average or histogram per-atom quantities from a compute, fix, or variable, then see the *fix ave/chunk*, *fix ave/atom*, or *fix ave/histo* commands. If you wish to sum a per-atom quantity into a single global quantity, see the *compute reduce* command.

Many *computes* and *fixes* produce global quantities. See their doc pages for details. *Variables* of style *equal* and *vector* are the only ones that can be used with this fix. Variables of style *atom* cannot be used, since they produce per-atom values.

The input values must all be scalars or vectors with a bracketed term appended, indicating the $I^{\text{th}}$ value of the vector is used.

The result of this fix can be accessed as a vector, containing the interleaved moments of each input in order. If M moments are requested, then the moments of input 1 will be the first M values in the vector output by this fix. The moments of input 2 will the next M values, etc. If there are N values, the vector length will be N*M.

---

For input values from a compute or fix or variable, the bracketed index I can be specified using a wildcard asterisk with the index to effectively specify multiple values. This takes the form "*" or "*n" or "m*" or "m*n". If $N$ is the size of the vector, then an asterisk with no numeric values means all indices from 1 to $N$. A leading asterisk means all indices from 1 to n (inclusive). A trailing asterisk means all indices from n to $N$ (inclusive). A middle asterisk means all indices from m to n (inclusive).

Using a wildcard is the same as if the individual elements of the vector or cells of the array had been listed one by one. For examples, see the description of this capability in *fix ave/time*.

---

The $N_{\text{every}}$, $N_{\text{repeat}}$, and $N_{\text{freq}}$ arguments specify on what time steps the input values will be used in order to contribute to the average. The final statistics are generated on time steps that are a multiple of $N_{\text{freq}}$. The average is over a window of up to $N_{\text{repeat}}$ quantities, computed in the preceding portion of the simulation once every $N_{\text{every}}$ time steps.

**Note**

Contrary to most fix ave/* commands, it is not required that Nevery * Nrepeat <= Nfreq. This is to allow the user to choose the time window and number of samples contributing to the output at each Nfreq interval.

For example, if $N_{\text{freq}} = 100$ and $N_{\text{repeat}} = 5$ (and $N_{\text{every}} = 1$), then on step 100 values from time steps 96, 97, 98, 99, and 100 will be used. The fix does not compute its inputs on steps that are not required. If $N_{\text{freq}} = 5$, $N_{\text{repeat}} = 8$ and $N_{\text{every}} = 1$, then values will first be calculated on step 5 from steps 1-5, on step 10 from 3-10, on step 15 from 8-15 and so on, forming a rolling average over timesteps that span a time window larger than Nfreq.

---

If a value begins with "c_", a compute ID must follow which has been previously defined in the input script. If no bracketed term is appended, the global scalar calculated by the compute is used. If a bracketed term is appended, the Ith element of the global vector calculated by the compute is used. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

If a value begins with "f_", a fix ID must follow which has been previously defined in the input script. If no bracketed term is appended, the global scalar calculated by the fix is used. If a bracketed term is appended, the Ith element of the global vector calculated by the fix is used. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

Note that some fixes only produce their values on certain time steps, which must be compatible with *Nevery*, else an error will result. Users can also write code for their own fix styles and *add them to LAMMPS*.

If a value begins with "v_", a variable name must follow which has been previously defined in the input script. Only equal-style or vector-style variables can be used, which both produce global values. Vector-style variables require a bracketed term to specify the Ith element of the vector calculated by the variable.

Note that variables of style *equal* and *vector* define a formula which can reference individual atom properties or thermodynamic keywords, or they can invoke other computes, fixes, or variables when they are evaluated, so this is a very general means of specifying quantities to time average.

---

The moments are output in the order requested in the arguments following the last input. Any number and order of moments can be specified, although it does not make much sense to specify the same moment multiple times. All moments are computed using a correction of the sample estimators used to obtain unbiased cumulants $k_{1..4}$ (see *(Cramer)*). The correction for variance is the standard Bessel correction. For other moments, see *(Joanes)*.

For *mean*, the arithmetic mean $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ is calculated.

For *variance*, the Bessel-corrected sample variance $var = k_2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$ is calculated.

For *stddev*, the Bessel-corrected sample standard deviation $stddev = \sqrt{k_2}$ is calculated.

For *skew*, the adjusted Fisher–Pearson standardized moment $G_1 = \frac{k_3}{k_2^{3/2}} = \frac{k_3}{stddev^3}$ is calculated.

For *kurtosis*, the adjusted Fisher–Pearson standardized moment $G_2 = \frac{k_4}{k_2^2}$ is calculated.

---

Fix invocation and output can be modified by optional keywords.

The *start* keyword specifies that the first computation should be no earlier than the step number given (but will still occur on a multiple of *Nfreq*). The default is step 0. Often input values can be 0.0 at time 0, so setting *start* to a larger value can avoid including a 0.0 in a longer series.

The *history* keyword stores the Nrecent most recent outputs on Nfreq timesteps, so they can be accessed as global outputs of the fix. Nrecent must be >= 1. The default is 1, meaning only the most recent output is accessible. For

example, if history 10 is specified and Nfreq = 1000, then on timestep 20000, the Nfreq outputs from steps 20000, 19000, ... 11000 are available for access. See below for details on how to access the history values.

For example, this will store the outputs of the previous 10 Nfreq time steps, i.e. a window of 10000 time steps:

```
fix 1 all ave/moments 1 200 1000 v_volume mean history 10
```

The previous results can be accessed as values in a global array output by this fix. Each column of the array is the vector output of the N-th preceding Nfreq timestep. For example, assuming a single moment is calculated, the most recent result corresponding to the third input value would be accessed as "f_name[3][1]", "f_name[3][4]" is the 4th most recent and so on. The current vector output is always the first column of the array, corresponding to the most recent result.

To illustrate the utility of keeping output history, consider using this fix in conjunction with *fix halt* to stop a run automatically if a quantity is converged to within some desired tolerance:

```
variable target equal etot
fix aveg all ave/moments 1 200 1000 v_target mean stddev history 10
variable stopcond equal "abs(f_aveg[1]-f_aveg[1][10])<f_aveg[2]"
fix fhalt all halt 1000 v_stopcond == 1
```

In this example, every 1000 time steps, the average and standard deviation of the total energy over the previous 200 time steps are calculated. If the difference between the most recent and 10-th most recent average is lower than the most recent standard deviation, the run is stopped.

---

### 2.24.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

This fix produces a global vector and global array which can be accessed by various *output commands*. The values can be accessed on any time step, but may not be current.

A global vector is produced with the # of elements = number of moments * number of inputs. The moments are output in the order given in the fix definition. An array is produced having # of rows = length of vector output (with an ordering which matches the vector) and # of columns = value of *history*. There is always at least one column.

Each element of the global vector or array can be either "intensive" or "extensive", depending on whether the values contributing to the element are "intensive" or "extensive". If a compute or fix provides the value being time averaged, then the compute or fix determines whether the value is intensive or extensive; see the page for that compute or fix for further info. Values produced by a variable are treated as intensive.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.24.5 Restrictions

This compute is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.24.6 Related commands

*fix ave/time*,

### 2.24.7 Default

The option defaults are history = 1, start = 0.

---

**(Cramer)** Cramer, Mathematical Methods of Statistics, Princeton University Press (1946).

**(Joanes)** Joanes, Gill, The Statistician, 47, 183–189 (1998).

## 2.25 fix ave/spatial command

Deprecated since version 11Dec2015.

The *fix ave/spatial* command has been superseded by *fix ave/chunk*.

## 2.26 fix ave/spatial/sphere command

Deprecated since version 11Dec2015.

The *fix ave/spatial/sphere* command has been superseded by *fix ave/chunk*.

## 2.27 fix ave/time command

### 2.27.1 Syntax

```
fix ID group-ID ave/time Nevery Nrepeat Nfreq value1 value2 ... keyword args ...
```

- ID, group-ID are documented in *fix* command
- ave/time = style name of this fix command
- Nevery = use input values every this many time steps
- Nrepeat = # of times to use input values for calculating averages
- Nfreq = calculate averages every this many time steps
- one or more input values can be listed
- value = c_ID, c_ID[N], f_ID, f_ID[N], v_name

```
c_ID = global scalar or vector calculated by a compute with ID
c_ID[I] = Ith component of global vector or Ith column of global array calculated␣
↪by a compute with ID, I can include wildcard (see below)
f_ID = global scalar or vector calculated by a fix with ID
f_ID[I] = Ith component of global vector or Ith column of global array calculated␣
↪by a fix with ID, I can include wildcard (see below)
v_name = value(s) calculated by an equal-style or vector-style variable with name
v_name[I] = value calculated by a vector-style variable with name, I can include␣
↪wildcard (see below)
```

- zero or more keyword/arg pairs may be appended
- keyword = *mode* or *file* or *append* or *ave* or *start* or *off* or *overwrite* or *format* or *title1* or *title2* or *title3*

```
mode arg = scalar or vector
  scalar = all input values are global scalars
  vector = all input values are global vectors or global arrays
```

```
ave args = one or running or window M
  one = output a new average value every Nfreq steps
  running = output cumulative average of all previous Nfreq steps
  window M = output average of M most recent Nfreq steps
start args = Nstart
  Nstart = start averaging on this time step
off arg = M = do not average this value
  M = value # from 1 to Nvalues
file arg = filename
  filename = name of file to output time averages to
append arg = filename
  filename = name of file to append time averages to
overwrite arg = none = overwrite output file with only latest output
format arg = string
  string = C-style format string
title1 arg = string
  string = text to print as 1st line of output file
title2 arg = string
  string = text to print as 2nd line of output file
title3 arg = string
  string = text to print as 3rd line of output file, only for vector mode
```

### 2.27.2 Examples

```
fix 1 all ave/time 100 5 1000 c_myTemp c_thermo_temp file temp.profile
fix 1 all ave/time 100 5 1000 c_thermo_press[2] ave window 20 &
                              title1 "My output values"
fix 1 all ave/time 100 5 1000 c_thermo_press[*]
fix 1 all ave/time 1 100 1000 f_indent f_indent[1] file temp.indent off 1
```

### 2.27.3 Description

Use one or more global values as inputs every few time steps, and average them over longer timescales. The resulting averages can be used by other *output commands* such as *thermo_style custom*, and can also be written to a file. Note that if no time averaging is done, this command can be used as a convenient way to simply output one or more global values to a file.

The group specified with this command is ignored. However, note that specified values may represent calculations performed by computes and fixes which store their own "group" definitions.

Each listed value can be the result of a *compute* or *fix* or the evaluation of an equal-style or vector-style *variable*. In each case, the compute, fix, or variable must produce a global quantity, not a per-atom or local quantity. If you wish to spatial- or time-average or histogram per-atom quantities from a compute, fix, or variable, then see the *fix ave/chunk*, *fix ave/atom*, or *fix ave/histo* commands. If you wish to sum a per-atom quantity into a single global quantity, see the *compute reduce* command.

*Computes* that produce global quantities are those which do not have the word *atom* in their style name. Only a few *fixes* produce global quantities. See the doc pages for individual fixes for info on which ones produce such values. *Variables* of style *equal* and *vector* are the only ones that can be used with this fix. Variables of style *atom* cannot be used, since they produce per-atom values.

The input values must either be all scalars or all vectors depending on the setting of the *mode* keyword. In both cases, the averaging is performed independently on each input value (i.e., each input scalar is averaged independently or each element of each input vector is averaged independently).

If *mode* = scalar, then the input values must be scalars, or vectors with a bracketed term appended, indicating the $I^{\text{th}}$ value of the vector is used.

If *mode* = vector, then the input values must be vectors, or arrays with a bracketed term appended, indicating the Ith column of the array is used. All vectors must be the same length, which is the length of the vector or number of rows in the array.

---

For input values from a compute or fix or variable, the bracketed index I can be specified using a wildcard asterisk with the index to effectively specify multiple values. This takes the form "*" or "*n" or "m*" or "m*n". If $N$ is the size of the vector (for *mode* = scalar) or the number of columns in the array (for *mode* = vector), then an asterisk with no numeric values means all indices from 1 to $N$. A leading asterisk means all indices from 1 to n (inclusive). A trailing asterisk means all indices from n to $N$ (inclusive). A middle asterisk means all indices from m to n (inclusive).

Using a wildcard is the same as if the individual elements of the vector or columns of the array had been listed one by one. For example, the following two fix ave/time commands are equivalent, since the *compute rdf* command creates, in this case, a global array with three columns, each of length 50:

```
compute myRDF all rdf 50 1 2
fix 1 all ave/time 100 1 100 c_myRDF[*] file tmp1.rdf mode vector
fix 2 all ave/time 100 1 100 c_myRDF[1] c_myRDF[2] c_myRDF[3] file tmp2.rdf mode vector
```

**Note**

For a vector-style variable, only the wildcard forms "*n" or "m*n" are allowed. You must specify the upper bound, because vector-style variable lengths are not determined until the variable is evaluated. If n is specified larger than the vector length turns out to be, zeroes are output for missing vector values.

---

The $N_{\text{every}}$, $N_{\text{repeat}}$, and $N_{\text{freq}}$ arguments specify on what time steps the input values will be used in order to contribute to the average. The final averaged quantities are generated on time steps that are a multiple of $N_{\text{freq}}$. The average is over $N_{\text{repeat}}$ quantities, computed in the preceding portion of the simulation every $N_{\text{every}}$ time steps. $N_{\text{freq}}$ must be a multiple of $N_{\text{every}}$ and $N_{\text{every}}$ must be non-zero even if $N_{\text{repeat}} = 1$. Also, the time steps contributing to the average value cannot overlap (i.e., $N_{\text{repeat}} \times N_{\text{every}}$ cannot exceed $N_{\text{freq}}$).

For example, if $N_{\text{every}} = 2$, $N_{\text{repeat}} = 6$, and $N_{\text{freq}} = 100$, then values on time steps 90, 92, 94, 96, 98, and 100 will be used to compute the final average on time step 100. Similarly for time steps 190, 192, 194, 196, 198, and 200 on time step 200, etc. If $N_{\text{repeat}} = 1$ and $N_{\text{freq}} = 100$, then no time averaging is done; values are simply generated on time steps 100, 200, etc.

---

If a value begins with "c_", a compute ID must follow which has been previously defined in the input script. If *mode* = scalar, then if no bracketed term is appended, the global scalar calculated by the compute is used. If a bracketed term is appended, the Ith element of the global vector calculated by the compute is used. If *mode* = vector, then if no bracketed term is appended, the global vector calculated by the compute is used. If a bracketed term is appended, the Ith column of the global array calculated by the compute is used. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

Note that there is a *compute reduce* command that can sum per-atom quantities into a global scalar or vector, which can then be accessed by fix ave/time. It can also be a compute defined not in your input script, but by *thermodynamic output* or other fixes such as *fix nvt* or *fix temp/rescale*. See the doc pages for these commands which give the IDs of these computes. Users can also write code for their own compute styles and *add them to LAMMPS*.

If a value begins with "f_", a fix ID must follow which has been previously defined in the input script. If *mode* = scalar, then if no bracketed term is appended, the global scalar calculated by the fix is used. If a bracketed term is

appended, the Ith element of the global vector calculated by the fix is used. If *mode* = vector, then if no bracketed term is appended, the global vector calculated by the fix is used. If a bracketed term is appended, the Ith column of the global array calculated by the fix is used. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

Note that some fixes only produce their values on certain time steps, which must be compatible with *Nevery*, else an error will result. Users can also write code for their own fix styles and *add them to LAMMPS*.

If a value begins with "v_", a variable name must follow which has been previously defined in the input script. If *mode* = scalar, then only equal-style or vector-style variables can be used, which both produce global values. In this mode, a vector-style variable requires a bracketed term to specify the Ith element of the vector calculated by the variable. If *mode* = vector, then only a vector-style variable can be used, without a bracketed term. See the *variable* command for details.

Note that variables of style *equal* and *vector* define a formula which can reference individual atom properties or thermodynamic keywords, or they can invoke other computes, fixes, or variables when they are evaluated, so this is a very general means of specifying quantities to time average.

---

Additional optional keywords also affect the operation of this fix.

If the *mode* keyword is set to *scalar*, then all input values must be global scalars, or elements of global vectors. If the *mode* keyword is set to *vector*, then all input values must be global vectors, or columns of global arrays. They can also be global arrays, which are converted into a series of global vectors (one per column), as explained above.

The *ave* keyword determines how the values produced every $N_{\text{freq}}$ steps are averaged with values produced on previous steps that were multiples of $N_{\text{freq}}$, before they are accessed by another output command or written to a file.

If the *ave* setting is *one*, then the values produced on time steps that are multiples of $N_{\text{freq}}$ are independent of each other; they are output as-is without further averaging.

If the *ave* setting is *running*, then the values produced on time steps that are multiples of $N_{\text{freq}}$ are summed and averaged in a cumulative sense before being output. Each output value is thus the average of the value produced on that time step with all preceding values. This running average begins when the fix is defined; it can only be restarted by deleting the fix via the *unfix* command, or by re-defining the fix by re-specifying it.

If the *ave* setting is *window*, then the values produced on time steps that are multiples of *Nfreq* are summed and averaged within a moving "window" of time, so that the last M values are used to produce the output. For example, if $M = 3$ and $N_{\text{freq}} = 1000$, then the output on step 10000 will be the average of the individual values on steps 8000, 9000, and 10000. Outputs on early steps will average over less than $M$ values if they are not available.

The *start* keyword specifies what time step averaging will begin on. The default is step 0. Often input values can be 0.0 at time 0, so setting *start* to a larger value can avoid including a 0.0 in a running or windowed average.

The *off* keyword can be used to flag any of the input values. If a value is flagged, it will not be time averaged. Instead the most recent input value will always be stored and output. This is useful if one of more of the inputs produced by a compute or fix or variable are effectively constant or are simply current values (e.g., they are being written to a file with other time-averaged values for purposes of creating well-formatted output).

Added in version 17Apr2024: new keyword *append*

The *file* or *append* keywords allow a filename to be specified. If *file* is used, then the filename is overwritten if it already exists. If *append* is used, then the filename is appended to if it already exists, or created if it does not exist. Every *Nfreq* steps, one quantity or vector of quantities is written to the file for each input value specified in the fix ave/time command. For *mode* = scalar, this means a single line is written each time output is performed. Thus the file ends up to be a series of lines, i.e. one column of numbers for each input value. For *mode* = vector, an array of numbers is written each time output is performed. The number of rows is the length of the input vectors, and the number of columns is the number of values. Thus the file ends up to be a series of these array sections.

Added in version 4May2022.

If the filename ends in '.yaml' or '.yml' then the output format conforms to the YAML standard which allows easy import that data into tools and scripts that support reading YAML files. The *structured data Howto* contains examples for parsing and plotting such data with very little programming effort in Python using the *pyyaml*, *pandas*, and *matplotlib* packages.

The *overwrite* keyword will continuously overwrite the output file with the latest output, so that it only contains one time step worth of output. This option can only be used with the *ave running* setting.

The *format* keyword sets the numeric format of each value when it is printed to a file via the *file* keyword. Note that all values are floating point quantities. The default format is " %g". You can specify a higher precision if desired (e.g., " %20.16g").

The *title1* and *title2* and *title3* keywords allow specification of the strings that will be printed as the first 2 or 3 lines of the output file, assuming the *file* keyword was used. LAMMPS uses default values for each of these, so they do not need to be specified.

By default, these header lines are as follows for *mode* = scalar:

```
# Time-averaged data for fix ID
# TimeStep value1 value2 ...
```

In the first line, ID is replaced with the fix-ID. In the second line the values are replaced with the appropriate fields from the fix ave/time command. There is no third line in the header of the file, so the *title3* setting is ignored when *mode* = scalar.

By default, these header lines are as follows for *mode* = vector:

```
# Time-averaged data for fix ID
# TimeStep Number-of-rows
# Row value1 value2 ...
```

In the first line, ID is replaced with the fix-ID. The second line describes the two values that are printed at the first of each section of output. In the third line the values are replaced with the appropriate fields from the fix ave/time command.

---

### 2.27.4 Restart, fix_modify, output, run start/stop, minimize info

Added in version 4May2022.

No information about this fix is written to *binary restart files*.

This fix produces a global scalar or global vector or global array which can be accessed by various *output commands*. The values can only be accessed on time steps that are multiples of $N_{\text{freq}}$ since that is when averaging is performed.

A scalar is produced if only a single input value is averaged and *mode* = scalar. A vector is produced if multiple input values are averaged for *mode* = scalar, or a single input value for *mode* = vector. In the first case, the length of the vector is the number of inputs. In the second case, the length of the vector is the same as the length of the input vector. An array is produced if multiple input values are averaged and *mode* = vector. The global array has # of rows = length of the input vectors and # of columns = number of inputs.

If the fix produces a scalar or vector, then the scalar and each element of the vector can be either "intensive" or "extensive", depending on whether the values contributing to the scalar or vector element are "intensive" or "extensive". If the fix produces an array, then all elements in the array must be the same, either "intensive" or "extensive". If a compute or fix provides the value being time averaged, then the compute or fix determines whether the value is intensive or extensive; see the page for that compute or fix for further info. Values produced by a variable are treated as intensive.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.27.5 Restrictions

none

### 2.27.6 Related commands

*compute*, *fix ave/atom*, *fix ave/chunk*, *fix ave/histo*, *variable*, *fix ave/correlate*,

### 2.27.7 Default

The option defaults are mode = scalar, ave = one, start = 0, no file output, format = %g, title 1,2,3 = strings as described above, and no off settings for any input values.

## 2.28 fix aveforce command

Accelerator Variant: *aveforce/kk*

### 2.28.1 Syntax

```
fix ID group-ID aveforce fx fy fz keyword value ...
```

- ID, group-ID are documented in *fix* command
- aveforce = style name of this fix command
- fx,fy,fz = force component values (force units)

  ```
  any of fx,fy,fz can be a variable (see below)
  ```

- zero or more keyword/value pairs may be appended to args
- keyword = *region*

  ```
  region value = region-ID
    region-ID = ID of region atoms must be in to have added force
  ```

### 2.28.2 Examples

```
fix pressdown topwall aveforce 0.0 -1.0 0.0
fix 2 bottomwall aveforce NULL -1.0 0.0 region top
fix 2 bottomwall aveforce NULL -1.0 v_oscillate region top
```

### 2.28.3 Description

Apply an additional external force to a group of atoms in such a way that every atom experiences the same force. This is useful for pushing on wall or boundary atoms so that the structure of the wall does not change over time.

The existing force is averaged for the group of atoms, component by component. The actual force on each atom is then set to the average value plus the component specified in this command. This means each atom in the group receives the same force.

Any of the *fx*, *fy*, or *fz* values can be specified as `NULL`, which means the force in that dimension is not changed. Note that this is not the same as specifying a 0.0 value, since that sets all forces to the same average value without adding in any additional force.

Any of the three quantities defining the force components, namely *fx*, *fy*, and *fz*, can be specified as an equal-style *variable*. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated each timestep, and its value used to determine the average force.

Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent average force.

If the *region* keyword is used, the atom must also be in the specified geometric *region* in order to have force added to it.

---

### 2.28.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify* *respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is adding its forces. Default is the outermost level.

This fix computes a global three-vector of forces, which can be accessed by various *output commands*. This is the total force on the group of atoms before the forces on individual atoms are changed by the fix. The vector values calculated by this fix are "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The forces due to this fix are imposed during an energy minimization, invoked by the *minimize* command. You should not specify force components with a variable that has time-dependence for use with a minimizer, since the minimizer increments the timestep as the iteration count during the minimization.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.28.5 Restrictions

none

### 2.28.6 Related commands

*fix setforce*, *fix addforce*

### 2.28.7 Default

none

# 2.29 fix balance command

## 2.29.1 Syntax

```
fix ID group-ID balance Nfreq thresh style args keyword args ...
```

- ID, group-ID are documented in *fix* command
- balance = style name of this fix command
- Nfreq = perform dynamic load balancing every this many steps
- thresh = imbalance threshold that must be exceeded to perform a re-balance
- style = *shift* or *rcb* or *report*

```
shift args = dimstr Niter stopthresh
  dimstr = sequence of letters containing x or y or z, each not more than once
  Niter = # of times to iterate within each dimension of dimstr sequence
  stopthresh = stop balancing when this imbalance threshold is reached
rcb args = none
report args = none
```

- zero or more keyword/arg pairs may be appended
- keyword = *weight* or *out*

```
weight style args = use weighted particle counts for the balancing
  style = group or neigh or time or var or store
    group args = Ngroup group1 weight1 group2 weight2 ...
      Ngroup = number of groups with assigned weights
      group1, group2, ... = group IDs
      weight1, weight2, ...   = corresponding weight factors
    neigh factor = compute weight based on number of neighbors
      factor = scaling factor (> 0)
    time factor = compute weight based on time spend computing
      factor = scaling factor (> 0)
    var name = take weight from atom-style variable
      name = name of the atom-style variable
    store name = store weight in custom atom property defined by fix property/atom␣
↪command
      name = atom property name (without d_ prefix)
sort arg = no or yes
out arg = filename
  filename = write each processor's subdomain to a file, at each re-balancing
```

## 2.29.2 Examples

```
fix 2 all balance 1000 1.05 shift x 10 1.05
fix 2 all balance 100 0.9 shift xy 20 1.1 out tmp.balance
fix 2 all balance 100 0.9 shift xy 20 1.1 weight group 3 substrate 3.0 solvent 1.0␣
↪solute 0.8 out tmp.balance
fix 2 all balance 100 1.0 shift x 10 1.1 weight time 0.8
fix 2 all balance 100 1.0 shift xy 5 1.1 weight var myweight weight neigh 0.6 weight␣
↪store allweight
fix 2 all balance 1000 1.1 rcb
```

### 2.29.3 Description

This command adjusts the size and shape of processor subdomains within the simulation box, to attempt to balance the number of particles and thus the computational cost (load) evenly across processors. The load balancing is "dynamic" in the sense that re-balancing is performed periodically during the simulation. To perform "static" balancing, before or between runs, see the *balance* command.

Added in version 17Apr2024.

The *report* balance style only computes the load imbalance but does not attempt any re-balancing. This way the load imbalance information can be used otherwise, for instance for stopping a run with *fix halt*.

Load-balancing is typically most useful if the particles in the simulation box have a spatially-varying density distribution or where the computational cost varies significantly between different atoms (e.g., a model of a vapor/liquid interface, or a solid with an irregular-shaped geometry containing void regions, or *hybrid pair style simulations* that combine pair styles with different computational cost). In these cases, the LAMMPS default of dividing the simulation box volume into a regular-spaced grid of 3d bricks, with one equal-volume subdomain per processor, may assign numbers of particles per processor in a way that the computational effort varies significantly. This can lead to poor performance when the simulation is run in parallel.

The balancing can be performed with or without per-particle weighting. With no weighting, the balancing attempts to assign an equal number of particles to each processor. With weighting, the balancing attempts to assign an equal aggregate computational weight to each processor, which typically induces a different number of atoms assigned to each processor.

**Note**

The weighting options listed above are documented with the *balance* command in *this section of the balance command* doc page. That section describes the various weighting options and gives a few examples of how they can be used. The weighting options are the same for both the fix balance and *balance* commands.

Note that the *processors* command allows some control over how the box volume is split across processors. Specifically, for a $P_x \times P_y \times P_z$ grid of processors, it allows choices of $P_x$, $P_y$, and $P_z$ subject to the constraint that $P_x P_y P_z = P$, the total number of processors. This is sufficient to achieve good load-balance for some problems on some processor counts. However, all the processor subdomains will still have the same shape and the same volume.

On a particular time step, a load-balancing operation is only performed if the current "imbalance factor" in particles owned by each processor exceeds the specified *thresh* parameter. The imbalance factor is defined as the maximum number of particles (or weight) owned by any processor, divided by the average number of particles (or weight) per processor. Thus, an imbalance factor of 1.0 is perfect balance.

As an example, for 10000 particles running on 10 processors, if the most heavily loaded processor has 1200 particles, then the imbalance factor is 1.2, meaning there is a 20% imbalance. Note that re-balances can be forced even if the current balance is perfect (1.0) be specifying a *thresh* < 1.0.

**Note**

This command attempts to minimize the imbalance factor, as defined above. But depending on the method a perfect balance (1.0) may not be achieved. For example, "grid" methods (defined below) that create a logical 3d grid cannot achieve perfect balance for many irregular distributions of particles. Likewise, if a portion of the system is a perfect, non-rotated lattice (e.g., the initial system is generated by the *create_atoms* command with no rotations), then "grid" methods may be unable to achieve exact balance. This is because entire lattice planes will be owned or not owned by a single processor.

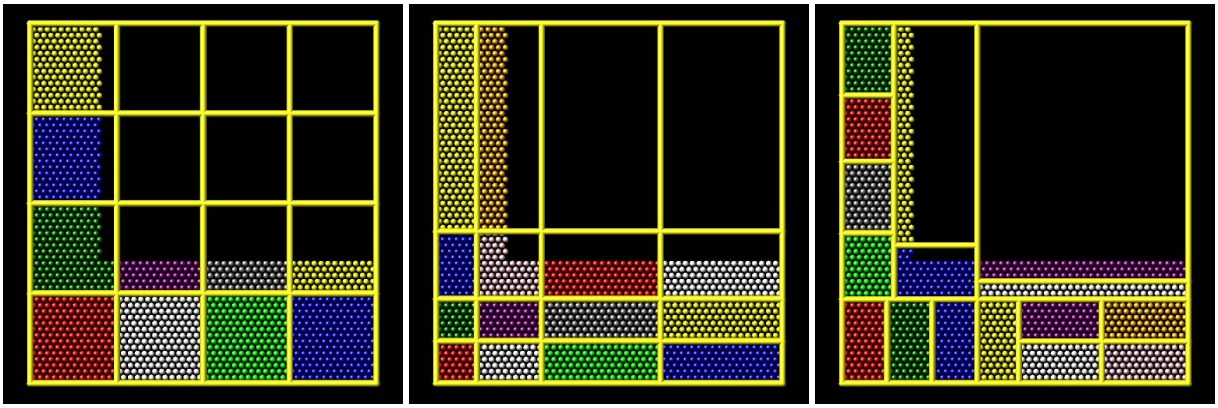
**Note**

The imbalance factor is also an estimate of the maximum speed-up you can hope to achieve by running a perfectly balanced simulation versus an imbalanced one. In the example above, the 10000-particle simulation could run up to 20% faster if it were perfectly balanced, versus when imbalanced. However, computational cost is not strictly proportional to particle count, and changing the relative size and shape of processor subdomains may lead to additional computational and communication overheads (e.g., in the PPPM solver used via the *kspace_style* command). Thus, you should benchmark the run times of a simulation before and after balancing.

---

The method used to perform a load balance is specified by one of the listed styles, which are described in detail below. There are two kinds of styles.

The *shift* style is a "grid" method which produces a logical 3d grid of processors. It operates by changing the cutting planes (or lines) between processors in 3d (or 2d), to adjust the volume (area in 2d) assigned to each processor, as in the following 2d diagram where processor subdomains are shown and atoms are colored by the processor that owns them.



The leftmost diagram is the default partitioning of the simulation box across processors (one sub-box for each of 16 processors); the middle diagram is after a "grid" method has been applied. The *rcb* style is a "tiling" method which does not produce a logical 3d grid of processors. Rather it tiles the simulation domain with rectangular sub-boxes of varying size and shape in an irregular fashion so as to have equal numbers of particles (or weight) in each sub-box, as in the rightmost diagram above.

The "grid" methods can be used with either of the *comm_style* command options, *brick* or *tiled*. The "tiling" methods can only be used with *comm_style tiled*.

When a "grid" method is specified, the current domain partitioning can be either a logical 3d grid or a tiled partitioning. In the former case, the current logical 3d grid is used as a starting point and changes are made to improve the imbalance factor. In the latter case, the tiled partitioning is discarded and a logical 3d grid is created with uniform spacing in all dimensions. This is the starting point for the balancing operation.

When a "tiling" method is specified, the current domain partitioning ("grid" or "tiled") is ignored, and a new partitioning is computed from scratch.

---

The *group-ID* is ignored. However the impact of balancing on different groups of atoms can be affected by using the *group* weight style as described below.

The $N_{\mathrm{freq}}$ setting determines how often a re-balance is performed. If $N_{\mathrm{freq}} > 0$, then re-balancing will occur every $N_{\mathrm{freq}}$ steps. Each time a re-balance occurs, a reneighboring is triggered, so $N_{\mathrm{freq}}$ should not be too small. If $N_{\mathrm{freq}} = 0$, then re-balancing will be done every time reneighboring normally occurs, as determined by the the *neighbor* and *neigh_modify* command settings.

On re-balance steps, re-balancing will only be attempted if the current imbalance factor, as defined above, exceeds the *thresh* setting.

---

The *shift* style invokes a "grid" method for balancing, as described above. It changes the positions of cutting planes between processors in an iterative fashion, seeking to reduce the imbalance factor.

The *dimstr* argument is a string of characters, each of which must be *x* or *y* or *z*. Each character can appear zero or one time, since there is no advantage to balancing on a dimension more than once. You should normally only list dimensions where you expect there to be a density variation in the particles.

Balancing proceeds by adjusting the cutting planes in each of the dimensions listed in *dimstr*, one dimension at a time. For a single dimension, the balancing operation (described below) is iterated on up to $N_{\text{iter}}$ times. After each dimension finishes, the imbalance factor is re-computed, and the balancing operation halts if the *stopthresh* criterion is met.

A re-balance operation in a single dimension is performed using a density-dependent recursive multisectioning algorithm, where the position of each cutting plane (line in 2d) in the dimension is adjusted independently. This is similar to a recursive bisectioning for a single value, except that the bounds used for each bisectioning take advantage of information from neighboring cuts if possible, as well as counts of particles at the bounds on either side of each cuts, which themselves were cuts in previous iterations. The latter is used to infer a density of particles near each of the current cuts. At each iteration, the count of particles on either side of each plane is tallied. If the counts do not match the target value for the plane, the position of the cut is adjusted based on the local density. The low and high bounds are adjusted on each iteration, using new count information, so that they become closer together over time. Thus as the recursion progresses, the count of particles on either side of the plane gets closer to the target value.

The density-dependent part of this algorithm is often an advantage when you re-balance a system that is already nearly balanced. It typically converges more quickly than the geometric bisectioning algorithm used by the *balance* command. However, if can be a disadvantage if you attempt to re-balance a system that is far from balanced, and converge more slowly. In this case you probably want to use the *balance* command before starting a run, so that you begin the run with a balanced system.

Once the re-balancing is complete and final processor subdomains assigned, particles migrate to their new owning processor as part of the normal reneighboring procedure.

**Note**

At each re-balance operation, the bisectioning for each cutting plane (line in 2d) typically starts with low and high bounds separated by the extent of a processor's subdomain in one dimension. The size of this bracketing region shrinks based on the local density, as described above, which should typically be 1/2 or more every iteration. Thus if $N_{\text{iter}}$ is specified as 10, the cutting plane will typically be positioned to better than 1 part in 1000 accuracy (relative to the perfect target position). For $N_{\text{iter}} = 20$, it will be accurate to better than 1 part in a million. Thus there is no need to set $N_{\text{iter}}$ to a large value. This is especially true if you are re-balancing often enough that each time you expect only an incremental adjustment in the cutting planes is necessary. LAMMPS will check if the threshold accuracy is reached (in a dimension) is less iterations than $N_{\text{iter}}$ and exit early.

---

The *rcb* style invokes a "tiled" method for balancing, as described above. It performs a recursive coordinate bisectioning (RCB) of the simulation domain. The basic idea is as follows.

The simulation domain is cut into two boxes by an axis-aligned cut in the longest dimension, leaving one new box on either side of the cut. All the processors are also partitioned into two groups, half assigned to the box on the lower side of the cut, and half to the box on the upper side. If the processor count is odd, one side gets an extra processor. The cut is positioned so that the number of atoms in the lower box is exactly the number that the processors assigned to that box should own for load balance to be perfect. This also makes load balance for the upper box perfect. The positioning is done iteratively, by a bisectioning method. Note that counting atoms on either side of the cut requires communication between all processors at each iteration.

That is the procedure for the first cut. Subsequent cuts are made recursively, in exactly the same manner. The subset of processors assigned to each box make a new cut in the longest dimension of that box, splitting the box, the subset of processors, and the atoms in the box in two. The recursion continues until every processor is assigned a sub-box of the entire simulation domain, and owns the atoms in that sub-box.

---

The *sort* keyword determines whether the communication of per-atom data to other processors during load-balancing will be random or deterministic. Random is generally faster; deterministic will ensure the new ordering of atoms on each processor is the same each time the same simulation is run. This can be useful for debugging purposes. Since the fix balance command is performed during timestepping, the default is *no* so that sorting is not performed.

The *out* keyword writes text to the specified *filename* with the results of each re-balancing operation. The file contains the bounds of the subdomain for each processor after the balancing operation completes. The format of the file is compatible with the Pizza.py *mdump* tool which has support for manipulating and visualizing mesh files. An example is shown here for a balancing by four processors for a 2d problem:

```
ITEM: TIMESTEP
0
ITEM: NUMBER OF NODES
16
ITEM: BOX BOUNDS
0 10
0 10
0 10
ITEM: NODES
1 1 0 0 0
2 1 5 0 0
3 1 5 5 0
4 1 0 5 0
5 1 5 0 0
6 1 10 0 0
7 1 10 5 0
8 1 5 5 0
9 1 0 5 0
10 1 5 5 0
11 1 5 10 0
12 1 10 5 0
13 1 5 5 0
14 1 10 5 0
15 1 10 10 0
16 1 5 10 0
ITEM: TIMESTEP
0
ITEM: NUMBER OF SQUARES
4
ITEM: SQUARES
1 1 1 2 3 4
2 1 5 6 7 8
3 1 9 10 11 12
4 1 13 14 15 16
```

The coordinates of all the vertices are listed in the NODES section, five per processor. Note that the four subdomains share vertices, so there will be duplicate nodes in the list.

The "SQUARES" section lists the node IDs of the four vertices in a rectangle for each processor (1 to 4).

For a 3d problem, the syntax is similar but with eight vertices listed for each processor instead of four, and "SQUARES" replaced by "CUBES".

---

### 2.29.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix computes a global scalar which is the imbalance factor after the most recent re-balance and a global vector of length 3 with additional information about the most recent re-balancing. The three values in the vector are as follows:

1. max # of particles per processor
2. total # iterations performed in last re-balance
3. imbalance factor right before the last re-balance was performed

As explained above, the imbalance factor is the ratio of the maximum number of particles (or total weight) on any processor to the average number of particles (or total weight) per processor.

These quantities can be accessed by various *output commands*. The scalar and vector values calculated by this fix are "intensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

---

### 2.29.5 Restrictions

For 2d simulations, the *z* style cannot be used, nor can *z* appear in *dimstr* for the *shift* style.

Balancing through recursive bisectioning (*rcb* style) requires *comm_style tiled*.

### 2.29.6 Related commands

*group*, *processors*, *balance*, *comm_style*

### 2.29.7 Default

The default setting is sort = no.

## 2.30 fix baoab command

### 2.30.1 Syntax

```
fix ID group-ID baoab Tstart Tstop damp seed keyword values ...
```

- ID, group-ID are documented in *fix* command
- baoab = style name of this fix command
- Tstart, Tstop = desired temperature at start/end of run (temperature units)
- damp = damping parameter (time units)
- seed = random number seed (positive integer)
- zero or more keyword/value pairs may be appended

- keyword = *zero*

```
zero value = no or yes
  no = do not zero the net random momentum each step (default)
  yes = zero the net random momentum each step
```

## 2.30.2 Examples

```
fix 1 all baoab 300.0 300.0 100.0 12345
fix 1 all baoab 1.0 1.0 10.0 48279 zero yes
fix 2 all baoab 300.0 400.0 200.0 77777
```

Added in version 4Jul2026.

## 2.30.3 Description

Apply the BAOAB Langevin integrator of *(Leimkuhler)* to the atoms in the fix group. This fix performs **complete** time integration of Newton's equations of motion coupled to a Langevin heat bath – do **not** combine it with *fix nve*.

The Langevin equation of motion integrated by this fix is

$$m\,d\mathbf{v} = \mathbf{F}_c\,dt - \frac{m}{\tau}\,\mathbf{v}\,dt + \sqrt{\frac{2\,m\,k_B T}{\tau}}\,d\mathbf{W}_t,$$

where $\mathbf{F}_c$ is the conservative force, $\tau$ is the damping time (*damp*), $T$ is the target temperature, and $d\mathbf{W}_t$ is a Wiener increment.

The BAOAB splitting implements the following sub-steps for each MD timestep:

- **B** – half-step velocity kick from conservative forces
- **A** – half-step position drift
- **O** – full-step exact Ornstein-Uhlenbeck (OU) update (thermostat)
- **A** – half-step position drift
- **B** – half-step velocity kick from conservative forces (using new forces)

The O step applies the exact solution of the OU process,

$$\mathbf{v} \leftarrow e^{-\gamma\,dt}\,\mathbf{v} + \sqrt{\frac{k_B T}{m}\left(1 - e^{-2\gamma dt}\right)}\,\mathbf{R}, \qquad \gamma = 1/\mathrm{damp},$$

where $\mathbf{R}$ is a vector of independent standard Gaussian random numbers. This is exact for any timestep, unlike a first-order Euler discretization of the friction and noise.

BAOAB achieves second-order accuracy in configuration space: the stationary distribution of positions converges to the Boltzmann distribution with error $O(dt^2)$, making it significantly more accurate than first-order Langevin integrators (such as the Euler-Maruyama scheme) at the same timestep. This is particularly important for computing configurational averages, free energies, and structural properties.

The target temperature can be ramped linearly from *Tstart* to *Tstop* over the course of a *run* by using the *start* and *stop* keywords of the *run* command.

The *damp* parameter has units of time and sets the relaxation time of the Langevin thermostat. Smaller values couple the system more strongly to the bath (faster thermalization, more friction); larger values give weaker coupling (less friction, closer to NVE dynamics). A typical starting value is 100 times the MD timestep.

If the *zero* keyword is set to *yes*, the net random momentum injected by the O step is subtracted each timestep so that the total momentum of the fix group does not drift. This is useful for bulk systems without periodic boundary conditions or when the group does not span the full system.

The cumulative energy exchanged between the atoms and the Langevin reservoir is always tracked and available via the *ecouple thermo* keyword. The sign convention is that a positive value means energy has flowed from the system into the reservoir (cooling), and negative means the reservoir has heated the system. Correspondingly the `econserve` thermo keyword reports the sum of `etotal` and `ecouple` and thus the conserved quantity of a constant temperature simulation.

This fix supports both per-type masses (`mass` command) and per-atom masses (atom styles such as `sphere`).

This fix can be used with dynamic atom groups.

### 2.30.4 Restart, fix_modify, output, run start/stop, minimize info

The cumulative thermostat energy is written to and read from *binary restart files* so that energy accounting is continuous across restarts.

This fix produces a global scalar (the cumulative thermostat energy) that is extensive and accessible to *thermo_style custom* via the `f_ID` keyword.

This fix can ramp its target temperature over multiple runs using the *start* and *stop* keywords of the *run* command.

This fix is not invoked during *energy minimization*.

### 2.30.5 Restrictions

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix cannot be combined with *fix shake* or *fix rattle*.

Do not combine this fix with *fix nve* or any other time-integration fix on the same group of atoms. LAMMPS will print a warning in this case.

### 2.30.6 Related commands

*fix langevin*, *fix nvt*, *fix gjf* , *fix gle*, *fix gld*

### 2.30.7 Default

The option defaults are zero = no.

---

**(Leimkuhler)** B. Leimkuhler and C. Matthews, "Rational Construction of Stochastic Numerical Methods for Molecular Sampling", Appl. Math. Res. Express, 2013(1), 34-56 (2013). https://doi.org/10.1093/amrx/abs010

## 2.31 fix bocs command

### 2.31.1 Syntax

```
fix ID group-ID bocs keyword values ...
```

- ID, group-ID are documented in *fix* command
- bocs = style name of this fix command
- two or more keyword/value pairs may be appended
- keyword = *temp* or *cgiso* or *tchain* or *pchain* or *mtk* or *tloop* or *ploop*

```
temp values = Tstart Tstop Tdamp
cgiso values = Pstart Pstop Pdamp basis_set args
  basis_set = analytic or linear_spline or cubic_spline
    analytic args = V_avg N_particles N_coeff Coeff_1 Coeff_2 ... Coeff_N
    linear_spline args = input_filename
    cubic_spline args = input_filename
tchain value = N = length of thermostat chain (1 = single thermostat)
pchain value = N = length of thermostat on barostat (0 = no thermostat)
mtk value = yes or no = add MTK adjustment term or not
tloop value = M = number of sub-cycles to perform on thermostat
ploop value = M = number of sub-cycles to perform on barostat
```

### 2.31.2 Examples

```
fix 1 all bocs temp 300.0 300.0 100.0 cgiso 0.986 0.986 1000.0 analytic 66476.015 968 2␣
↪245030.10 8962.20
fix 1 all bocs temp 300.0 300.0 100.0 cgiso 0.986 0.986 1000.0 cubic_spline input_Fv.dat
thermo_modify press 1_press
```

### 2.31.3 Description

These commands incorporate a pressure correction as described by Dunn and Noid *(Dunn1)* to the standard MTK barostat by Martyna et al. *(Martyna)*. The first half of the command mimics a standard *fix npt* command:

```
fix 1 all bocs temp Tstart Tstop Tcoupl cgiso Pstart Pstop Pdamp
```

The two differences are replacing *npt* with *bocs*, and replacing *iso*/*aniso*/etc. with *cgiso*. The rest of the command details what form you would like to use for the pressure correction equation. The choices are: *analytic*, *linear_spline*, or *cubic_spline*.

With either spline method, the only argument that needs to follow it is the name of a file that contains the desired pressure correction as a function of volume. The file must be formatted so each line has:

```
Volume_i, PressureCorrection_i
```

Note both the COMMA and the SPACE separating the volume's value and its corresponding pressure correction. The volumes in the file must be uniformly spaced. Both the volumes and the pressure corrections should be provided in the proper units (e.g., if you are using *units real*, the volumes should all be in $Å^3$ and the pressure corrections should all be in atm). Furthermore, the table should start/end at a volume considerably smaller/larger than you expect your system to sample during the simulation. If the system ever reaches a volume outside of the range provided, the simulation will stop.

With the *analytic* option, the arguments are as follows:

```
... analytic V_avg N_particles N_coeff Coeff_1 Coeff_2 ... Coeff_N
```

Note that *V_avg* and *Coeff_i* should all be in the proper units (e.g., if you are using *units real*, *V_avg* should be in $Å^3$ and the coefficients should all be in atm $\cdot$ $Å^3$).

### 2.31.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the cumulative global energy change to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the fix continues in an uninterrupted fashion.

The *fix_modify* *temp* option is supported by this fix. You can use it to assign a temperature *compute* you have defined to this fix which will be used in its thermostatting procedure, as described above. For consistency, the group used by this fix and by the compute should be the same.

The cumulative energy change in the system imposed by this fix is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* doc page for details.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the same cumulative energy change due to this fix described in the previous paragraph. The scalar value calculated by this fix is "extensive".

This fix can ramp its target temperature over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.31.5 Restrictions

This fix is part of the BOCS package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

As this is computing a (modified) pressure, group-ID should be *all*.

The pressure correction has only been tested for use with an isotropic pressure coupling in 3 dimensions.

By default, LAMMPS will still report the normal value for the pressure if the pressure is printed via a *thermo* command, or if the pressures are written to a file every so often. In order to have LAMMPS report the modified pressure, you must include the *thermo_modify* command given in the examples. For the last argument in the command, you should put XXXX_press, where XXXX is the ID given to the fix bocs command (in the example, the ID of the fix bocs command is 1).

### 2.31.6 Further information

For more details about the pressure correction and the entire BOCS software package, visit the BOCS package on GitHub and read the release paper by Dunn et al. *(Dunn2)* .

---

**(Dunn1)** Dunn and Noid, J Chem Phys, 143, 243148 (2015).

**(Martyna)** Martyna, Tobias, and Klein, J Chem Phys, 101, 4177 (1994).

**(Dunn2)** Dunn, Lebold, DeLyser, Rudzinski, and Noid, J. Phys. Chem. B, 122, 3363 (2018).

## 2.32 fix bond/break command

### 2.32.1 Syntax

```
fix ID group-ID bond/break Nevery bondtype Rmax keyword values ...
```

- ID, group-ID are documented in *fix* command
- bond/break = style name of this fix command
- Nevery = attempt bond breaking every this many steps
- bondtype = type of bonds to break (integer or type label)

- Rmax = bond longer than Rmax can break (distance units)
- zero or more keyword/value pairs may be appended
- keyword = *prob*

```
prob values = fraction seed
  fraction = break a bond with this probability if otherwise eligible
  seed = random number seed (positive integer)
```

### 2.32.2 Examples

```
fix 5 all bond/break 10 2 1.2
fix 5 polymer bond/break 1 1 2.0 prob 0.5 49829
```

### 2.32.3 Description

Break bonds between pairs of atoms as a simulation runs according to specified criteria. This can be used to model the dissolution of a polymer network due to stretching of the simulation box or other deformations. In this context, a bond means an interaction between a pair of atoms computed by the *bond_style* command. Once the bond is broken it will be permanently deleted, as will all angle, dihedral, and improper interactions that bond is part of.

This is different than a *pair-wise* bond-order potential such as Tersoff or AIREBO which infers bonds and many-body interactions based on the current geometry of a small cluster of atoms and effectively creates and destroys bonds and higher-order many-body interactions from timestep to timestep as atoms move.

A check for possible bond breakage is performed every *Nevery* timesteps. If two bonded atoms $i$ and $j$ are farther than the distance *Rmax* from each other, the bond is of type *bondtype*, and both $i$ and $j$ are in the specified fix group, then the bond between $i$ and $j$ is labeled as a "possible" bond to break.

If several bonds involving an atom are stretched, it may have multiple possible bonds to break. Every atom checks its list of possible bonds to break and labels the longest such bond as its "sole" bond to break. After this is done, if atom $i$ is bonded to atom $j$ in its sole bond, and atom $j$ is bonded to atom $j$ in its sole bond, then the bond between $i$ and $j$ is "eligible" to be broken.

Note that these rules mean an atom will only be part of at most one broken bond on a given time step. It also means that if atom $i$ chooses atom $j$ as its sole partner, but atom $j$ chooses atom $k$ as its sole partner (because $R_{jk} > R_{ij}$), then this means atom $i$ will not be part of a broken bond on this time step, even if it has other possible bond partners.

The *prob* keyword can effect whether an eligible bond is actually broken. The *fraction* setting must be a value between 0.0 and 1.0. A uniform random number between 0.0 and 1.0 is generated and the eligible bond is only broken if the random number is less than *fraction*.

When a bond is broken, data structures within LAMMPS that store bond topologies are updated to reflect the breakage. Likewise, if the bond is part of a 3-body (angle) or 4-body (dihedral, improper) interaction, that interaction is removed as well. These changes typically affect pair-wise interactions between atoms that used to be part of bonds, angles, etc.

**Note**

One data structure that is not updated when a bond breaks are the molecule IDs stored by each atom. Even though one molecule becomes two molecules due to the broken bond, all atoms in both new molecules retain their original molecule IDs.

Computationally, each time step this fix is invoked, it loops over all the bonds in the system and computes distances between pairs of bonded atoms. It also communicates between neighboring processors to coordinate which bonds are broken. Moreover, if any bonds are broken, neighbor lists must be immediately updated on the same time step. This is to ensure that any pair-wise interactions that should be turned "on" due to a bond breaking, because they are

no longer excluded by the presence of the bond and the settings of the *special_bonds* command, will be immediately recognized. All of these operations increase the cost of a time step. Thus, you should be cautious about invoking this fix too frequently.

You can dump out snapshots of the current bond topology via the *dump local* command.

**Note**

Breaking a bond typically alters the energy of a system. You should be careful not to choose bond breaking criteria that induce a dramatic change in energy. For example, if you define a very stiff harmonic bond and break it when two atoms are separated by a distance far from the equilibrium bond length, then the two atoms will be dramatically released when the bond is broken. More generally, you may need to thermostat your system to compensate for energy changes resulting from broken bonds (as well as angles, dihedrals, and impropers).

See the *Howto* page on broken bonds for more information on related features in LAMMPS.

---

### 2.32.4 Dump image info

Added in version 11Feb2026.

Fix *bond/break* supports the *fix* keyword of *dump image*. The fix will pass geometry information about atoms involved in a broken bond to *dump image* so that these atoms can be highlighted in the visualization as additional spheres. For how long those additional spheres will be shown depends on the value of the *vizsteps* setting (default is 1000) which can be changed by using the *fix_modify command*. If an atom is involved in multiple broken bonds, the check on showing the additional graphics depends on the timestep of its latest broken bond.

The color of the additional spheres is by default that of the atoms when using color styles "type" or "element". With color style "const" the default value of "white" can be changed using *dump_modify fcolor*. The transparency is by default fully opaque and can be changed with *dump_modify ftrans*.

The *fflag1* setting of *dump image fix* has no effect.

The *fflag2* setting allows you to set the radius of the added spheres, since the radius is set to zero internally.

---

### 2.32.5 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix computes two statistics, which it stores in a global vector of length 2. This vector can be accessed by various *output commands*. The vector values calculated by this fix are "intensive".

The two quantities in the global vector are

(1) number of bonds broken on the most recent breakage time step

(2) cumulative number of bonds broken

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.32.6 Restrictions

This fix is part of the MC package. It is only enabled if LAMMPS was built with that package. See the *Build package* doc page for more info.

### 2.32.7 Related commands

*fix bond/create*, *fix bond/react*, *fix bond/swap*, *dump local*, *special_bonds*

### 2.32.8 Default

The option defaults are prob = 1.0.

## 2.33 fix bond/create command

## 2.34 fix bond/create/angle command

### 2.34.1 Syntax

```
fix ID group-ID style Nevery itype jtype Rmin bondtype keyword values ...
```

- ID, group-ID are documented in *fix* command
- style = *bond/create* or *bond/create/angle*
- Nevery = attempt bond creation every this many steps
- itype,jtype = atoms of itype can bond to atoms of jtype (1-Ntypes or type label)
- Rmin = two atoms separated by less than Rmin can bond (distance units)
- bondtype = type of created bonds (integer or type label)
- zero or more keyword/value pairs may be appended to args
- keyword = *iparam* or *jparam* or *prob* or *atype* or *dtype* or *itype* or *aconstrain* or *molecule*

```
iparam values = maxbond, newtype
  maxbond = max # of bonds of bondtype the itype atom can have
  newtype = change the itype atom to this type when maxbonds exist (1-Ntypes or␣
→type label)
jparam values = maxbond, newtype
  maxbond = max # of bonds of bondtype the jtype atom can have
  newtype = change the jtype atom to this type when maxbonds exist (1-Ntypes or␣
→type label)
prob values = fraction seed
  fraction = create a bond with this probability if otherwise eligible
  seed = random number seed (positive integer)
atype value = angletype
  angletype = type of created angles (integer or type label)
dtype value = dihedraltype
  dihedraltype = type of created dihedrals (integer or type label)
itype value = impropertype
  impropertype = type of created impropers (integer or type label)
aconstrain value = amin amax
  amin = minimal angle at which new bonds can be created
  amax = maximal angle at which new bonds can be created
molecule value = off or inter or intra
  off = allow both inter- and intramolecular reactions (default)
  inter = search for reactions between molecules with different IDs
  intra = search for reactions within the same molecule
```

### 2.34.2 Examples

```
fix 5 all bond/create 10 1 2 0.8 1
fix 5 all bond/create 1 3 3 0.8 1 prob 0.5 85784 iparam 2 3
fix 5 all bond/create 1 3 3 0.8 1 prob 0.5 85784 iparam 2 3 atype 1 dtype 2
fix 5 all bond/create 10 13 25 7 28 iparam 1 15 jparam 1 27 prob 0.2 91322 molecule inter
fix 5 all bond/create/angle 10 1 2 1.122 1 aconstrain 120 180 prob 1 4928459 iparam 2 1␣
↪jparam 2 2

labelmap atom 1 c1 2 n2
labelmap bond 1 c1-n2
fix 5 all bond/create 10 c1 n2 0.8 c1-n2
```

### 2.34.3 Description

Create bonds between pairs of atoms as a simulation runs according to specified criteria. This can be used to model the cross-linking of polymers, the formation of a percolation network, etc. In this context, a bond means an interaction between a pair of atoms computed by the *bond_style* command. Once the bond is created it will be permanently in place. Optionally, the creation of a bond can also create angle, dihedral, and improper interactions that the bond is part of. See the discussion of the *atype*, *dtype*, and *itype* keywords below.

This process is different than a *pair-wise* bond-order potential such as Tersoff or AIREBO, which infer bonds and many-body interactions based on the current geometry of a small cluster of atoms and effectively create and destroy bonds and higher-order many-body interactions from time step to time step as the atoms move.

A check for possible new bonds is performed every *Nevery* time steps. If two atoms $i$ and $j$ are within a distance *Rmin* of each other, atom $i$ is of type *itype*, atom $j$ is of type *jtype*, and both $i$ and $j$ are in the specified fix group, then if a bond does not already exist between atoms $i$ and $j$, and if both $i$ and $j$ meet their respective *maxbond* requirements (explained below), then $i$ and $j$ are labeled as a "possible" bond pair.

If several atoms are close to an atom, it may have multiple possible bond partners. Every atom checks its list of possible bond partners and labels the closest such partner as its "sole" bond partner. After this is done, if atom $i$ has atom $j$ as its sole partner and atom $j$ has atom $i$ as its sole partner, then the $i$, $j$ bond is "eligible" to be formed.

Note that these rules mean that an atom will only be part of at most one created bond on a given time step. It also means that if atom $i$ chooses atom $j$ as its sole partner, but atom $j$ chooses atom $k$ as its sole partner (because $R_{jk} < R_{ij}$), then atom $i$ will not form a bond on this time step, even if it has other possible bond partners.

It is permissible to have *itype* = *jtype*. *Rmin* must be $\leq$ the pair-wise cutoff distance between *itype* and *jtype* atoms, as defined by the *pair_style* command.

The *iparam* and *jparam* keywords can be used to limit the bonding functionality of the participating atoms. Each atom keeps track of how many bonds of *bondtype* it already has. If atom $i$ of type *itype* already has *maxbond* bonds (as set by the *iparam* keyword), then it will not form any more, and likewise for atom $j$. If *maxbond* is set to 0, then there is no limit on the number of bonds that can be formed with that atom.

The *newtype* value for *iparam* and *jparam* can be used to change the atom type of atom $i$ or $j$ when it reaches *maxbond* number of bonds of type *bondtype*. This means it can now interact in a pair-wise fashion with other atoms in a different way by specifying different *pair_coeff* coefficients. If you do not wish the atom type to change, simply specify *newtype* as *itype* or *jtype*.

The *prob* keyword can also affect whether an eligible bond is actually created. The *fraction* setting must be a value between 0.0 and 1.0. A uniform random number between 0.0 and 1.0 is generated and the eligible bond is only created if the random number is less than *fraction*.

The *molecule* keyword can be used to force the reaction to be intermolecular, intramolecular or either. When the value is set to *off*, molecule IDs are not considered when searching for reactions (default). When the value is set to *inter*,

atoms must have different molecule IDs in order to be considered for the reaction. When the value is set to *intra*, only atoms with the same molecule ID are considered for the reaction.

The *aconstrain* keyword is only available with the fix bond/create/angle command. It allows one to specify minimum and maximum angles *amin* and *amax*, respectively, between the two prospective bonding partners and a third particle that is already bonded to one of the two partners. Such a criterion can be important when new angles are defined together with the formation of a new bond. Without a restriction on the permissible angle, and for stiffer angle potentials, very large energies can arise and lead to unphysical behavior.

Any bond that is created is assigned a bond type of *bondtype*.

When a bond is created, data structures within LAMMPS that store bond topologies are updated to reflect the creation. If the bond is part of new 3-body (angle) or 4-body (dihedral, improper) interactions, you can choose to create new angles, dihedrals, and impropers as well using the *atype*, *dtype*, and *itype* keywords. All of these changes typically affect pair-wise interactions between atoms that are now part of new bonds, angles, etc.

**Note**

One data structure that is not updated when a bond breaks are the molecule IDs stored by each atom. Even though two molecules become one molecule due to the created bond, all atoms in the new molecule retain their original molecule IDs.

If the *atype* keyword is used and if an angle potential is defined via the *angle_style* command, then any new 3-body interactions inferred by the creation of a bond will create new angles of type *angletype*, with parameters assigned by the corresponding *angle_coeff* command. Likewise, the *dtype* and *itype* keywords will create new dihedrals and impropers of type *dihedraltype* and *impropertype*.

**Note**

To create a new bond, the internal LAMMPS data structures that store this information must have space for it. When LAMMPS is initialized from a data file, the list of bonds is scanned and the maximum number of bonds per atom is tallied. If some atom will acquire more bonds than this limit as this fix operates, then the "extra bond per atom" parameter must be set to allow for it. Ditto for "extra angle per atom", "extra dihedral per atom", and "extra improper per atom" if angles, dihedrals, or impropers are being added when bonds are created. See the *read_data* or *create_box* command for more details. Note that a data file with no atoms can be used if you wish to add non-bonded atoms via the *create atoms* command (e.g., for a percolation simulation).

**Note**

LAMMPS stores and maintains a data structure with a list of the first, second, and third neighbors of each atom (within the bond topology of the system) for use in weighting pair-wise interactions for bonded atoms. Note that adding a single bond always adds a new first neighbor but may also induce **many** new second and third neighbors, depending on the molecular topology of your system. The "extra special per atom" parameter must typically be set to allow for the new maximum total size (first + second + third neighbors) of this per-atom list. There are two ways to do this. See the *read_data* or *create_box* commands for details.

**Note**

Even if you do not use the *atype*, *dtype*, or *itype* keywords, the list of topological neighbors is updated for atoms affected by the new bond. This in turn affects which neighbors are considered for pair-wise interactions, using the weighting rules set by the *special_bonds* command. Consider a new bond created between atoms $i$ and $j$. If

$j$ has a bonded neighbor $k$, then $k$ becomes a second neighbor of $i$. Even if the *atype* keyword is not used to create angle $\angle ijk$, the pair-wise interaction between $i$ and $k$ could potentially be turned off or weighted by the 1–3 weighting specified by the *special_bonds* command. This is the case even if the "angle yes" option was used with that command. The same is true for third neighbors (1–4 interactions), the *dtype* keyword, and the "dihedral yes" option used with the *special_bonds* command.

Note that even if your simulation starts with no bonds, you must define a *bond_style* and use the *bond_coeff* command to specify coefficients for the *bondtype*. Similarly, if new atom types are specified by the *iparam* or *jparam* keywords, they must be within the range of atom types allowed by the simulation and pair-wise coefficients must be specified for the new types.

Computationally, each time step this fix is invoked, it loops over neighbor lists and computes distances between pairs of atoms in the list. It also communicates between neighboring processors to coordinate which bonds are created. Moreover, if any bonds are created, neighbor lists must be immediately updated on the same time step. This is to ensure that any pair-wise interactions that should be turned "off" due to a bond creation, because they are now excluded by the presence of the bond and the settings of the *special_bonds* command, will be immediately recognized. All of these operations increase the cost of a time step. Thus, you should be cautious about invoking this fix too frequently.

You can dump out snapshots of the current bond topology via the *dump local* command.

**Note**

Creating a bond typically alters the energy of a system. You should be careful not to choose bond creation criteria that induce a dramatic change in energy. For example, if you define a very stiff harmonic bond and create it when two atoms are separated by a distance far from the equilibrium bond length, then the two atoms will oscillate dramatically when the bond is formed. More generally, you may need to thermostat your system to compensate for energy changes resulting from created bonds (and angles, dihedrals, impropers).

---

### 2.34.4 Dump image info

Added in version 11Feb2026.

Fixes *bond/create* and *bond/create/angle* support the *fix* keyword of *dump image*. The fixes will pass geometry information about atoms involved in a created bond to *dump image* so that these atoms can be highlighted in the visualization as additional spheres. For how long those additional spheres will be shown depends on the value of the *vizsteps* setting (default is 1000) which can be changed by using the *fix_modify command*. If an atom is involved in the creation of multiple bonds, the check on showing the additional graphics depends on the timestep of its last bond creation.

The color of the additional spheres is by default that of the atoms when using color styles "type" or "element". With color style "const" the default value of "white" can be changed using *dump_modify fcolor*. The transparency is by default fully opaque and can be changed with *dump_modify ftrans*.

The *fflag1* setting of *dump image fix* has no effect.

The *fflag2* setting allows you to set the radius of the added spheres, since the radius is set to zero internally.

---

### 2.34.5 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix computes two statistics which it stores in a global vector of length 2, which can be accessed by various *output commands*. The vector values calculated by this fix are "intensive".

The two quantities in the global vector are

(1) number of bonds created on the most recent creation time step

(2) cumulative number of bonds created

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.34.6 Restrictions

This fix is part of the MC package. It is only enabled if LAMMPS was built with that package. See the *Build package* doc page for more info.

### 2.34.7 Related commands

*fix bond/break*, *fix bond/react*, *fix bond/swap*, *dump local*, *special_bonds*

### 2.34.8 Default

The option defaults are iparam = (0,itype), jparam = (0,jtype), and prob = 1.0.

## 2.35 fix bond/react command

### 2.35.1 Syntax

```
fix ID group-ID bond/react common_keyword values &
  react react-ID react-group-ID Nevery Rmin Rmax template-ID(pre-reacted) template-
↪ID(post-reacted) map_file individual_keyword values &
  react react-ID react-group-ID Nevery Rmin Rmax template-ID(pre-reacted) template-
↪ID(post-reacted) map_file individual_keyword values &
  react react-ID react-group-ID Nevery Rmin Rmax template-ID(pre-reacted) template-
↪ID(post-reacted) map_file individual_keyword values &
  ...
```

- ID, group-ID are documented in *fix* command.
- bond/react = style name of this fix command
- the common keyword/values may be appended directly after 'bond/react'
- common keywords apply to all reaction specifications
- common_keyword = *stabilization* or *reset_mol_ids* or *rate_limit* or *max_rxn* or *shuffle_seed* or *file*

```
stabilization values = stabilize group_prefix xmax
  stabilize = yes or no
    yes = perform reaction site stabilization
    no = no reaction site stabilization (default)
  group_prefix = user-assigned prefix for the dynamic group of atoms not currently␣
↪involved in a reaction
  xmax = value that is used by an internally-created nve/limit integrator
reset_mol_ids values = yes or no or molmap
  yes = update molecule IDs based on new global topology (default)
  no = do not update molecule IDs
  molmap = customize how molecule IDs are updated
rate_limit values = react-ID_1 react-ID_2 ... react-ID_N Nlimit Nsteps
  react-IDs = one or more names of the reactions to include in rate limit
```

```
    Nlimit = maximum number of reactions allowed to occur within interval
    Nsteps = the interval (number of timesteps) over which to count reactions
  max_rxn values = react-ID_1 react-ID_2 ... react-ID_N Nlimit
    react-IDs = one or more names of the reactions to include in rate limit
    Nlimit = maximum total number of reactions allowed to occur
  shuffle_seed value = seed
    seed = random # seed (positive integer) for choosing between eligible reactions
  file value = filename
    filename = name of the JSON file that records reaction occurrences
```

- react = mandatory argument indicating new reaction specification
- react-ID = user-assigned name for the reaction
- react-group-ID = only atoms in this group are considered for the reaction
- Nevery = attempt reaction every this many steps
- Rmin = initiator atoms must be separated by more than Rmin to initiate reaction (distance units)
- Rmax = initiator atoms must be separated by less than Rmax to initiate reaction (distance units)
- template-ID(pre-reacted) = ID of a molecule template containing pre-reaction topology
- template-ID(post-reacted) = ID of a molecule template containing post-reaction topology
- map_file = name of file specifying corresponding atom-IDs in the pre- and post-reacted templates
- zero or more individual keyword/value pairs may be appended to each react argument
- individual_keyword = *prob* or *stabilize_steps* or *custom_charges* or *rescale_charges* or *molecule* or *modify_create*

```
  prob values = fraction seed
    fraction = initiate reaction with this probability if otherwise eligible
    seed = random number seed (positive integer)
  stabilize_steps value = timesteps
    timesteps = number of time steps to apply the internally-created nve/limit fix to␣
↪reacting atoms
  custom_charges value = no or fragment-ID
    no = update all atomic charges (default)
    fragment-ID = ID of molecule fragment whose charges are updated
  rescale_charges value = no or yes
    no = do not rescale atomic charges (default)
    yes = rescale charges such that total charge does not change during reaction
  molecule value = off or inter or intra
    off = allow both inter- and intramolecular reactions (default)
    inter = search for reactions between molecules with different IDs
    intra = search for reactions within the same molecule
  modify_create values = keyword arg
    fit arg = all or fragment-ID
      all = use all eligible atoms for create-atoms fit (default)
      fragment-ID = ID of molecule fragment used for create-atoms fit
    overlap value = R
      R = only insert atom/molecule if further than R from existing particles␣
↪(distance units)
```

### 2.35.2 Examples

For unabridged example scripts and files, see examples/PACKAGES/reaction.

```
molecule mol1 pre_reacted_topology.txt
molecule mol2 post_reacted_topology.txt
fix rxns all bond/react react diels_alder all 1 0 3.25 mol1 mol2 map_file.txt

molecule mol1 pre_reacted_rxn1.txt
molecule mol2 post_reacted_rxn1.txt
molecule mol3 pre_reacted_rxn2.txt
molecule mol4 post_reacted_rxn2.txt
fix 5 all bond/react stabilization yes nvt_grp .03 &
  react myrxn1 all 1 0 3.25 mol1 mol2 map_file_rxn1.txt prob 0.50 12345 &
  react myrxn2 all 1 0 2.75 mol3 mol4 map_file_rxn2.txt prob 0.25 12345
fix 6 nvt_grp_REACT nvt temp 300 300 100 # set thermostat after bond/react
```

### 2.35.3 Description

Initiate complex covalent bonding (topology) changes. These topology changes will be referred to as 'reactions' throughout this documentation. Topology changes are defined in pre- and post-reaction molecule templates and can include creation and deletion of bonds, angles, dihedrals, impropers, bond types, angle types, dihedral types, atom types, or atomic charges. In addition, reaction by-products or other molecules can be identified and deleted. Finally, atoms can be created and inserted at specific positions relative to the reaction site.

Fix bond/react does not use quantum mechanical (e.g., *fix qmmm*) or pairwise bond-order potential (e.g., *Tersoff* or *AIREBO*) methods to determine bonding changes a priori. Rather, it uses a distance-based probabilistic criteria to effect predetermined topology changes in simulations using standard force fields.

This fix was created to facilitate the dynamic creation of polymeric, amorphous or highly cross-linked systems. A suggested workflow for using this fix is

(1) identify a reaction to be simulated

(2) build a molecule template of the reaction site before the reaction has occurred

(3) build a molecule template of the reaction site after the reaction has occurred

(4) create a map that relates the template-atom-IDs of each atom between pre- and post-reaction molecule templates

(5) fill a simulation box with molecules and run a simulation with fix bond/react.

**Note**

Added in version 15Sep2022.

*Type labels* allow for molecule templates and data files to use alphanumeric atom types that match those of a force field. Input files that use type labels are inherently compatible with each other and portable between different simulations. Therefore, it is highly recommended to use type labels to specify atom, bond, etc. types when using fix bond/react.

Only one 'fix bond/react' command can be used at a time. Multiple reactions can be simultaneously applied by specifying multiple *react* arguments to a single 'fix bond/react' command. This syntax is necessary because the "common" keywords are applied to all reactions.

The *stabilization* keyword enables reaction site stabilization. Reaction site stabilization is performed by including reacting atoms in an internally-created fix *nve/limit* time integrator for a set number of time steps given by the *stabilize_steps* keyword. While reacting atoms are being time integrated by the internal nve/limit, they are prevented from

being involved in any new reactions. The *xmax* value keyword should typically be set to the maximum distance that non-reacting atoms move during the simulation.

Fix bond/react creates and maintains two important dynamic groups of atoms when using the *stabilization* keyword. The first group contains all atoms currently involved in a reaction; this group is automatically time-integrated by an internally-created *nve/limit* integrator. The second group contains all atoms currently not involved in a reaction. This group should be controlled by a thermostat in order to time integrate the system. The name of this group of non-reacting atoms is created by appending '_REACT' to the group-ID argument of the *stabilization* keyword, as shown in the second example above.

**Note**

When using reaction stabilization, you should generally **not** have a separate thermostat that acts on the "all" group.

The group-ID set using the *stabilization* keyword can be an existing static group or a previously-unused group-ID. It cannot be specified as "all". If the group-ID is previously unused, the fix bond/react command creates a *dynamic group* that is initialized to include all atoms. If the group-ID is that of an existing static group, the group is used as the parent group of new, internally-created dynamic group. In both cases, this new dynamic group is named by appending '_REACT' to the group-ID (e.g., nvt_grp_REACT). By specifying an existing group, you may thermostat constant-topology parts of your system separately. The dynamic group contains only atoms not involved in a reaction at a given time step, and therefore should be used by a subsequent system-wide time integrator such as *fix nvt*, *fix npt*, or *fix nve*, as shown in the second example above (full examples can be found in examples/PACKAGES/reaction). The time integration command should be placed after the fix bond/react command due to the internal dynamic grouping performed by fix bond/react.

**Note**

If the group-ID is an existing static group, react-group-IDs should also be specified as this static group or a subset.

Added in version 2Apr2025: New *molmap* option

If the *reset_mol_ids* keyword is set to *yes* (default), the *reset_atoms mol* command is invoked after a reaction occurs, to ensure that molecule IDs are consistent with the new bond topology. The group-ID used for *reset_atoms mol* is the group-ID for this fix. Resetting molecule IDs is necessarily a global operation, so it can be slow for very large systems. If the *reset_mol_ids* keyword is set to *no*, molecule IDs are not updated. If the *reset_mol_ids* keyword is set to *molmap*, molecule IDs are updated consistently with the molecule IDs listed in the *Molecules* section of the pre- and post-reaction templates. If a post-reaction atom has the same molecule ID as one or more pre-reaction atoms in the templates, then the post-reaction simulation atom will be assigned the same simulation molecule ID that those corresponding pre-reaction simulation atoms had before the reaction. The *molmap* option is only guaranteed to work correctly if all the pre-reaction atoms that have equivalent template molecule IDs also have equivalent molecule IDs in the simulation. No check is performed to test for this consistency. For post-reaction atoms that have a template molecule ID that does not exist in pre-reaction template, they are assigned a new molecule ID that does not currently exist in the simulation.

The *rate_limit* keyword can enforce an upper limit on the overall rate of one or more reactions. The number of reaction occurrences is limited to Nlimit within an interval of Nsteps timesteps. No reactions are permitted to occur within the first Nsteps timesteps of the first run after reading a data file. The reactions to sum over are listed by reaction name (react-ID). The number of reaction occurrences is calculated by summing over the listed reactions. This sum is limited to Nlimit, which can be specified with an equal-style *variable*. Reaction occurrences are chosen randomly from all eligible reaction sites of all listed reactions. By default, a hardware-based random number source is used if available; reactions are chosen deterministically if a positive integer is specified for the 'shuffle_seed' keyword. Multiple *rate_limit* keywords can be specified. This keyword is useful when multiple *react* arguments define similar types of reactions, and the relative rates between two or more types of reactions must be enforced.

The *max_rxn* keyword can enforce an upper limit on the overall number of one or more reactions. The reactions to sum over are listed by reaction name (react-ID). The number of reaction occurrences is calculated by summing over the listed reactions. This sum is limited to Nlimit. Reaction occurrences are chosen randomly from all eligible reaction sites of all listed reactions. By default, a hardware-based random number source is used if available; reactions are chosen deterministically if a positive integer is specified for the 'shuffle_seed' keyword. Multiple *max_rxn* keywords can be specified.

Added in version 10Dec2025.

The *file* keyword can be used to dump information about each reaction that occurs during the simulation. The atom IDs, types, and coordinates of all atoms in the reaction site are printed out on the timestep that the reaction is initiated. The output file follows the *JSON dump molecules format*, with one extra key added to each molecule object to identify the reaction. The added key is "reaction" and its value is the reaction name (react-ID). Here is an example output for a hypothetical reaction involving one water molecule:

```
{
    "application": "LAMMPS",
    "units": "real",
    "format": "dump",
    "style": "molecules",
    "revision": 1,
    "title": "fix bond/react",
    "timesteps": [
        {
            "timestep": 1,
            "molecules": [
                {
                    "reaction": "water_dissociation",
                    "types": {
                        "format": ["atom-id", "type"],
                        "data": [
                            [1368, "H"],
                            [1366, "O"],
                            [1367, "H"]
                        ]
                    },
                    "coords": {
                        "format": ["atom-id", "x", "y", "z"],
                        "data": [
                            [1368, 26.787767440427466, 29.785528640296768, 25.
↪85197353660144],
                            [1366, 26.641801222582824, 29.868106247702887, 24.
↪91285138212243],
                            [1367, 25.69611192416744, 30.093425787807448, 24.
↪914380215672846]
                        ]
                    }
                }
            ]
        }
    ]
}
```

The following comments pertain to each *react* argument (in other words, they can be customized for each reaction, or reaction step):

A check for possible new reaction sites is performed every *Nevery* time steps. *Nevery* can be specified with an equal-style *variable*, whose value is rounded up to the nearest integer.

Three physical conditions must be met for a reaction to occur. First, an initiator atom pair must be identified within the reaction distance cutoffs. Second, the topology surrounding the initiator atom pair must match the topology of the pre-reaction template. Only atom types and bond connectivity are used to identify a valid reaction site (not bond types, etc.). Finally, any reaction constraints listed in the map file (see below) must be satisfied. If all of these conditions are met, the reaction site is eligible to be modified to match the post-reaction template.

An initiator atom pair will be identified if several conditions are met. First, a pair of atoms *i* and *j* within the specified react-group-ID of type *itype* and *jtype* must be separated by a distance between *Rmin* and *Rmax*. *Rmin* and *Rmax* can be specified with equal-style *variables*. For example, these reaction cutoffs can be functions of the reaction conversion using the following commands:

```
variable rmax equal 0 # initialize variable before bond/react
fix myrxn all bond/react react myrxn1 all 1 0 v_rmax mol1 mol2 map_file.txt
variable rmax equal 3+f_myrxn[1]/100 # arbitrary function of reaction count
```

The following criteria are used if multiple candidate initiator atom pairs are identified within the cutoff distance:

(1) If the initiator atoms in the pre-reaction template are not 1–2 neighbors (i.e., not directly bonded) the closest potential partner is chosen.

(2) Otherwise, if the initiator atoms in the pre-reaction template are 1–2 neighbors (i.e. directly bonded) the farthest potential partner is chosen.

(3) Then, if both an atom *i* and atom *j* have each other as initiator partners, these two atoms are identified as the initiator atom pair of the reaction site.

Note that it can be helpful to select unique atom types for the initiator atoms: if an initiator atom pair is identified, as described in the previous steps, but it does not correspond to the same pair specified in the pre-reaction template, an otherwise eligible reaction could be prevented from occurring. Once this unique initiator atom pair is identified for each reaction, there could be two or more reactions that involve the same atom on the same time step. If this is the case, only one such reaction is permitted to occur. This reaction is chosen randomly from all potential reactions involving the overlapping atom. This capability allows, for example, different reaction pathways to proceed from identical reaction sites with user-specified probabilities.

The pre-reacted molecule template is specified by a molecule command. This molecule template file contains a sample reaction site and its surrounding topology. As described below, the initiator atom pairs of the pre-reacted template are specified by atom ID in the map file. The pre-reacted molecule template should contain as few atoms as possible while still completely describing the topology of all atoms affected by the reaction (which includes all atoms that change atom type or connectivity, and all bonds that change bond type). For example, if the force field contains dihedrals, the pre-reacted template should contain any atom within three bonds of reacting atoms.

Some atoms in the pre-reacted template that are not reacting may have missing topology with respect to the simulation. For example, the pre-reacted template may contain an atom that, in the simulation, is currently connected to the rest of a long polymer chain. These are referred to as edge atoms, and are specified in the map file in the EdgeIDs section. All pre-reaction template atoms should be linked to an initiator atom, via at least one path that does not involve edge atoms. When the pre-reaction template contains edge atoms, not all atoms, bonds, etc. specified in the reaction templates will be updated. Specifically, topology that involves only atoms that are "too near" to template edges will not be updated. The definition of "too near the edge" depends on which interactions are defined in the simulation. If the simulation has defined dihedrals, atoms within two bonds of edge atoms are considered "too near the edge." If the simulation defines angles, but not dihedrals, atoms within one bond of edge atoms are considered "too near the edge." If just bonds are defined, only edge atoms are considered "too near the edge."

**Note**

Small molecules (i.e., ones that have all their atoms contained within the reaction templates) never have edge atoms.

Note that some care must be taken when a building a molecule template for a given simulation. All atom types in the pre-reacted template must be the same as those of a potential reaction site in the simulation. A detailed discussion of matching molecule template atom types with the simulation is provided on the *molecule* command page. It is highly recommended to use *Type labels* (added in version 15Sep2022) in both molecule templates and data files, which automates the process of syncing atom types between different input files.

Wildcard atoms match to any atom type in the simulation. Wildcard atoms can be used to reduce the number of reaction templates needed to model a set of similar reactions. Wildcard atoms are specified in the Wildcards section of the map file. The atom types of wildcard atoms in the simulation are not updated. Any bond, angle, dihedral, or improper, that is defined in the reaction templates and contains a wildcard atom, will be updated by inferring its type from its constituent atom types. To use wildcard atoms, a specific *type label* format is necessary to infer the types of higher-order interactions. Bond, angle, dihedral, and improper type labels must contain their constituent atom types delimited by hyphens, e.g., 'c2-c2-c2-n' for a dihedral that contains three atoms of type 'c2' and one atom of 'n'. Certain symmetries are considered to account for equivalent ways of writing higher-order interactions. Type labels for bonds, angles, and dihedrals are assumed to be equivalent to those written in reverse order. For example, an angle with type label 'c1-c2-n' is equivalent to 'n-c2-c1'. Symmetries for impropers are more complex and are described on the doc page for each improper style in the 'Symmetry convention' section.

The post-reacted molecule template contains a sample of the reaction site and its surrounding topology after the reaction has occurred. It must contain the same number of atoms as the pre-reacted template (unless there are created atoms). A one-to-one correspondence between the atom IDs in the pre- and post-reacted templates is specified in the map file as described below. Note that during a reaction, an atom, bond, etc. type may change to one that was previously not present in the simulation. These new types must also be defined during the setup of a given simulation. A discussion of correctly handling this is also provided on the *molecule* command page.

**Note**

When a reaction occurs, it is possible that the resulting topology/atom (e.g., special bonds, dihedrals) exceeds that of the existing system and reaction templates. As when inserting molecules, enough space for this increased topology/atom must be reserved by using the relevant "extra" keywords to the *read_data* or *create_box* commands.

The map file is a text document with the following format:

A map file has a header and a body. The header of the map file contains one mandatory keyword and six optional keywords. The mandatory keyword is *equivalences*:

```
N equivalences = # of atoms N in the reaction molecule templates
```

The optional keywords are *edgeIDs*, *wildcards*, *deleteIDs*, *createIDs*, *chiralIDs*, and *constraints*:

```
N edgeIDs = # of edge atoms N in the pre-reacted molecule template
N wildcards = # of atoms with wildcard atom types N
N deleteIDs = # of atoms N that are deleted
N createIDs = # of atoms N that are created
N chiralIDs = # of chiral centers N
N constraints = # of reaction constraints N
```

The body of the map file contains two mandatory sections and six optional sections. The first mandatory section begins with the keyword "InitiatorIDs" and lists the two atom IDs of the initiator atom pair in the pre-reacted molecule template. The second mandatory section begins with the keyword "Equivalences" and lists a one-to-one correspondence between atom IDs of the pre- and post-reacted templates. The first column is an atom ID of the pre-reacted molecule template,

and the second column is the corresponding atom ID of the post-reacted molecule template. The first optional section begins with the keyword "EdgeIDs" and lists the atom IDs of edge atoms in the pre-reacted molecule template. The second optional section begins with the keyword "Wildcards" and lists the pre-reaction atom IDs of atoms that have wildcard atom types. The third optional section begins with the keyword "DeleteIDs" and lists the atom IDs of pre-reaction template atoms to delete. The fourth optional section begins with the keyword "CreateIDs" and lists the atom IDs of the post-reaction template atoms to create. The fifth optional section begins with the keyword "ChiralIDs" lists the atom IDs of chiral atoms whose handedness should be enforced. The sixth optional section begins with the keyword "Constraints" and lists additional criteria that must be satisfied in order for the reaction to occur. Currently, there are six types of constraints available, as discussed below: "distance", "angle", "dihedral", "arrhenius", "rmsd", and "custom".

A sample map file is given below:

```
# this is a map file

7 equivalences
2 edgeIDs

InitiatorIDs

3
5

EdgeIDs

1
7

Equivalences

1   1
2   2
3   3
4   4
5   5
6   6
7   7
```

A user-specified set of atoms can be deleted by listing their pre-reaction template IDs in the DeleteIDs section. A deleted atom must still be included in the post-reaction molecule template, in which it cannot be bonded to an atom that is not deleted. In addition to deleting unwanted reaction by-products, this feature can be used to remove specific topologies, such as small rings, that may be otherwise indistinguishable.

Atoms can be created by listing their post-reaction template IDs in the CreateIDs section. A created atom should not be included in the pre-reaction template. The inserted positions of created atoms are determined by the coordinates of the post-reaction template, after optimal translation and rotation of the post-reaction template to the reaction site (using a fit with atoms that are neither created nor deleted). The *modify_create* keyword can be used to modify the default behavior when creating atoms. The *modify_create* keyword has two sub-keywords, *fit* and *overlap*. One or more of the sub-keywords may be used after the *modify_create* keyword. The *fit* sub-keyword can be used to specify which post-reaction atoms are used for the optimal translation and rotation of the post-reaction template. The fragment-ID value of the *fit* sub-keyword must be the name of a molecule fragment defined in the post-reaction *molecule* template, and only atoms in this fragment are used for the fit. Atoms are created only if no current atom in the simulation is within a distance $R$ of any created atom, including the effect of periodic boundary conditions if applicable. $R$ is defined by

the *overlap* sub-keyword. Note that the default value for *R* is 0.0, which will allow atoms to strongly overlap if you are inserting where other atoms are present. The molecule ID of a created atom is zero, unless the *reset_mol_ids molmap* option is used. The velocity of each created atom is initialized in a random direction with a magnitude calculated from the instantaneous temperature of the reaction site.

**Note**

The 'Coords' section must be included in the post-reaction template when creating atoms because these coordinates are used to determine where new atoms are inserted.

The handedness of atoms that are chiral centers can be enforced by listing their IDs in the ChiralIDs section. A chiral atom must be bonded to four atoms with mutually different atom types. This feature uses the coordinates and types of the involved atoms in the pre-reaction template to determine handedness. Three atoms bonded to the chiral center are arbitrarily chosen, to define an oriented plane, and the relative position of the fourth bonded atom determines the chiral center's handedness.

Any number of additional constraints may be specified in the Constraints section of the map file. The constraint of type "distance" has syntax as follows:

`distance` *ID1 ID2 rmin rmax*

where "distance" is the required keyword, *ID1* and *ID2* are pre-reaction atom IDs (or molecule-fragment IDs, see below), and these two atoms must be separated by a distance between *rmin* and *rmax* for the reaction to occur.

The constraint of type "angle" has the following syntax:

`angle` *ID1 ID2 ID3 amin amax*

where "angle" is the required keyword, *ID1*, *ID2* and *ID3* are pre-reaction atom IDs (or molecule-fragment IDs, see below), and these three atoms must form an angle between *amin* and *amax* for the reaction to occur (where *ID2* is the central atom). Angles must be specified in degrees. This constraint can be used to enforce a certain orientation between reacting molecules.

The constraint of type "dihedral" has the following syntax:

`dihedral` *ID1 ID2 ID3 ID4 amin amax amin2 amax2*

where "dihedral" is the required keyword, and *ID1*, *ID2*, *ID3* and *ID4* are pre-reaction atom IDs (or molecule-fragment IDs, see below). Dihedral angles are calculated in the interval $(-180°, 180°]$. Refer to the *dihedral style* documentation for further details on convention. If *amin* is less than *amax*, these four atoms must form a dihedral angle greater than *amin* **and** less than *amax* for the reaction to occur. If *amin* is greater than *amax*, these four atoms must form a dihedral angle greater than *amin* **or** less than *amax* for the reaction to occur. Angles must be specified in degrees. Optionally, a second range of permissible angles *amin2* to *amax2* can be specified.

For the 'distance', 'angle', and 'dihedral' constraints (explained above), atom IDs can be replaced by pre-reaction molecule-fragment IDs. The molecule-fragment ID must begin with a letter. The location of the ID is the geometric center of all atom positions in the fragment. The molecule fragment must have been defined in the *molecule* command for the pre-reaction template.

The constraint of type 'arrhenius' imposes an additional reaction probability according to the modified Arrhenius equation,

$$k = AT^n e^{-E_a/k_B T}.$$

The Arrhenius constraint has the following syntax:

`arrhenius` *A n E_a seed*

where "arrhenius" is the required keyword, $A$ is the pre-exponential factor, $n$ is the exponent of the temperature dependence, $E_a$ is the activation energy (*units* of energy), and *seed* is a random number seed. The temperature is defined as

the instantaneous temperature averaged over all atoms in the reaction site and is calculated in the same manner as for example *compute temp/chunk*. Currently, there are no options for additional temperature averaging or velocity-biased temperature calculations. A uniform random number between 0 and 1 is generated using *seed*; if this number is less than the result of the Arrhenius equation above, the reaction is permitted to occur.

The constraint of type 'rmsd' has the following syntax:

`rmsd` *RMSDmax molfragment*

where "rmsd" is the required keyword, and *RMSDmax* is the maximum root-mean-square deviation between atom positions of the pre-reaction template and the local reaction site (distance units), after optimal translation and rotation of the pre-reaction template. Optionally, the name of a molecule fragment (of the pre-reaction template) can be specified by *molfragment*. If a molecule fragment is specified, only atoms that are part of this molecule fragment are used to determine the RMSD. A molecule fragment must have been defined in the *molecule* command for the pre-reaction template. For example, the molecule fragment could consist of only the backbone atoms of a polymer chain. This constraint can be used to enforce a specific relative position and orientation between reacting molecules.

Changed in version 22Dec2022.

The constraint of type "custom" has the following syntax:

`custom` *varstring*

where 'custom' is the required keyword, and *varstring* is a variable expression. The expression must be a valid equal-style variable formula that can be read by the *variable* command, after any special reaction functions are evaluated. If the resulting expression is zero, the reaction is prevented from occurring; otherwise, it is permitted to occur. There are three special reaction functions available, 'rxnbond', 'rxnsum', and 'rxnave'. The 'rxnbond' function allows per-bond values to be included in the variable strings of the custom constraint. The 'rxnbond' function has two mandatory arguments. The first argument is the ID of a previously defined 'compute bond/local' command. This 'compute bond/local' must compute only one value, e.g. bond force. This value is returned by the 'rxnbond' function. The second argument is the name of a molecule fragment in the pre-reaction template. The fragment must contain exactly two atoms, corresponding to the atoms involved in the bond whose value should be calculated. An example of a constraint that uses the force experienced by a bond is provided below. When using 'rxnbond', at least one atom in the fragment must be an initiator atom. The 'rxnsum' and 'rxnave' functions operate over the atoms in a given reaction site, and have one mandatory argument and one optional argument. The mandatory argument is the identifier for an atom-style variable. The second, optional argument is the name of a molecule fragment in the pre-reaction template, and can be used to operate over a subset of atoms in the reaction site. The 'rxnsum' function sums the atom-style variable over the reaction site, while the 'rxnave' returns the average value. For example, a constraint on the total potential energy of atoms involved in the reaction can be imposed as follows:

```
compute 1 all pe/atom # in LAMMPS input script
variable my_pe atom c_1 # in LAMMPS input script
```

```
custom "rxnsum(v_my_pe) > 100" # in Constraints section of map file
```

The above example prevents the reaction from occurring unless the total potential energy of the reaction site is above 100. As a second example, this time using the 'rxnbond' function, consider a modified Arrhenius constraint that depends on the bond force of a specific bond:

```
# in LAMMPS input script

compute bondforce all bond/local force

compute ke_atom all ke/atom
variable ke atom c_ke_atom

```

(continues on next page)

(continued from previous page)

```
variable E_a equal 100.0 # activation energy
variable l0 equal 1.0 # characteristic length
```

```
# in Constraints section of map file

custom "exp(-(v_E_a-rxnbond(c_bondforce,bond1frag)*v_l0)/(2/3*rxnave(v_ke))) > random(0,
↪1,12345)"
```

By using an inequality and the 'random(x,y,z)' function, the left-hand side can be interpreted as the probability of the reaction occurring, similar to the 'arrhenius' constraint above.

By default, all constraints must be satisfied for the reaction to occur. In other words, constraints are evaluated as a series of logical values using the logical AND operator "&&". More complex logic can be achieved by explicitly adding the logical AND operator "&&" or the logical OR operator "||" after a given constraint command. If a logical operator is specified after a constraint, it must be placed after all constraint parameters, on the same line as the constraint (one per line). Similarly, parentheses can be used to group constraints. The expression that results from concatenating all constraints should be a valid logical expression that can be read by the *variable* command after converting each constraint to a logical value. Because exactly one constraint is allowed per line, having a valid logical expression implies that left parentheses "(" should only appear before a constraint, and right parentheses ")" should only appear after a constraint and before any logical operator.

Once a reaction site has been successfully identified, data structures within LAMMPS that store bond topology are updated to reflect the post-reacted molecule template. All force fields with fixed bonds, angles, dihedrals or impropers are supported.

A few capabilities to note:

(1) You may specify as many *react* arguments as desired. For example, you could break down a complicated reaction mechanism into several reaction steps, each defined by its own *react* argument.

(2) While typically a bond is formed or removed between the initiator atoms specified in the pre-reacted molecule template, this is not required.

(3) By reversing the order of the pre- and post-reacted molecule templates in another *react* argument, you can allow for the possibility of one or more reverse reactions.

The optional keywords deal with the probability of a given reaction occurring as well as the stable equilibration of each reaction site as it occurs.

The *prob* keyword can affect whether or not an eligible reaction actually occurs. The fraction setting must be a value between 0.0 and 1.0, and can be specified with an equal-style *variable*. A uniform random number between 0.0 and 1.0 is generated and the eligible reaction only occurs if the random number is less than the fraction.

The *stabilize_steps* keyword allows for the specification of how many time steps a reaction site is stabilized before being returned to the overall system thermostat. In order to produce the most physical behavior, this "reaction site equilibration time" should be tuned to be as small as possible while retaining stability for a given system or reaction step. After a limited number of case studies, this number has been set to a default of 60 time steps. Ideally, it should be individually tuned for each fix reaction step. Note that in some situations, decreasing rather than increasing this parameter will result in an increase in stability.

The *custom_charges* keyword can be used to specify which atoms' atomic charges are updated. When the value is set to *no*, all atomic charges are updated to those specified by the post-reaction template (default). Otherwise, the value should be the name of a molecule fragment defined in the pre-reaction molecule template. In this case, only the atomic charges of atoms in the molecule fragment are updated.

Added in version 22Dec2022.

The *rescale_charges* keyword can be used to ensure the total charge of the system does not change as reactions occur. When the argument is set to *yes*, a fixed value is added to the charges of post-reaction atoms such that their total charge equals that of the pre-reaction site. If only a subset of atomic charges are updated via the *custom_charges* keyword, this rescaling is applied to the subset. This keyword could be useful for systems that contain different molecules with the same reactive site, if the partial charges on the reaction site vary from molecule to molecule, or when removing reaction by-products.

The *molecule* keyword can be used to force the reaction to be intermolecular, intramolecular or either. When the value is set to *off*, molecule IDs are not considered when searching for reactions (default). When the value is set to *inter*, the initiator atoms must have different molecule IDs in order to be considered for the reaction. When the value is set to *intra*, only initiator atoms with the same molecule ID are considered for the reaction.

A few other considerations:

Optionally, you can enforce additional behaviors on reacting atoms. For example, it may be beneficial to force reacting atoms to remain at a certain temperature. For this, you can use the internally-created dynamic group named "bond_react_MASTER_group", which consists of all atoms currently involved in a reaction. For example, adding the following command would add an additional thermostat to the group of all currently-reacting atoms:

```
fix 1 bond_react_MASTER_group temp/rescale 1 300 300 10 1
```

**Note**

This command must be added after the fix bond/react command, and will apply to all reactions.

Computationally, each time step this fix is invoked, it loops over neighbor lists (for bond-forming reactions) and computes distances between pairs of atoms in the list. It also communicates between neighboring processors to coordinate which bonds are created and/or removed. All of these operations increase the cost of a time step. Thus, you should be cautious about invoking this fix too frequently.

You can dump out snapshots of the current bond topology via the *dump local* command.

---

### 2.35.4 Dump image info

Added in version 11Feb2026.

Fix *bond/react* supports the *fix* keyword of *dump image*. The fix will pass geometry information about atoms involved in a reaction to *dump image* so that these atoms can be highlighted in the visualization through additional spheres. For how long those additional spheres will be shown depends on the value of the *vizsteps* setting (default is 1000) which can be changed by using the *fix_modify command*. If an atom is involved in multiple reactions, the check on showing the additional graphics depends on the timestep of its last reaction.

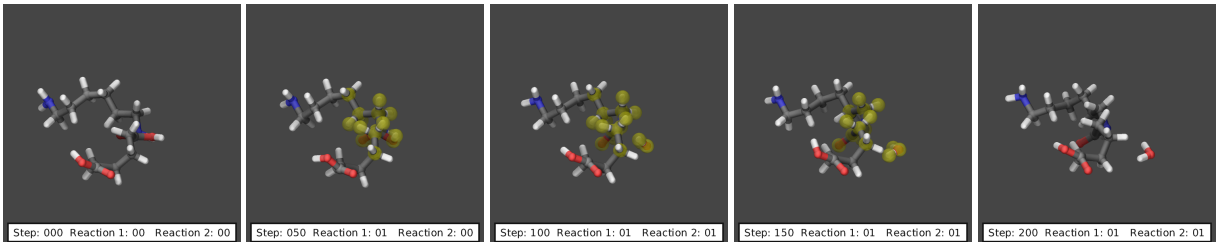
The color of the additional spheres is by default that of the atom type when using color styles "type" or "element". With color style "const" the default value of "white" can be changed using *dump_modify fcolor*. The transparency is by default fully opaque and can be changed with *dump_modify ftrans*.

The *fflag1* setting of *dump image fix* has no effect.

The *fflag2* setting allows you to set the radius of the added spheres, since the radius is set to zero internally.

The following commands can be added to the `examples/PACKAGES/reaction/tiny_nylon/in.tiny_nylon.stabilized` input to highlight the reacting atoms with transparent yellow spheres and will create images like those shown below:

```
variable ndump index 10
# display simulation step and reaction counts
fix labels all graphics/labels ${ndump} text &
     "Step: $(step:%03.0f)  Reaction 1: $(f_myrxns[1]:%02.0f)   Reaction 2: $(f_
↪myrxns[2]:%02.0f)" &
     300 24 0 transcolor none backcolor silver fontcolor black framecolor black
# output highlighted atoms for 10 dumps
fix_modify myrxns vizsteps $(10*v_ndump)
dump viz all image ${ndump} nylon-*.png element element size 600 600 center s 0.6 0.55 0.
↪6 &
           shiny 0.1 fsaa yes ssao yes 231541 0.6 zoom 6.0 view 70 -10 box no 0.1 &
           fix myrxns const 0.0 1.0 fix labels const 1 0
dump_modify viz pad 4 element C C O H H O N H N H O &
           backcolor darkgray ftrans myrxns 0.5 fcolor myrxns yellow
run 200
```




### 2.35.5 Restart, fix_modify, output, run start/stop, minimize info

Cumulative reaction counts for each reaction are written to *binary restart files*. These values are associated with the reaction name (react-ID). Additionally, internally-created per-atom properties are stored to allow for smooth restarts.

The *vizsteps* option of the *fix_modify* command is supported by this fix to control on which timesteps reacted atoms are highlighted in *dump image* output.

This fix computes one statistic for each *react* argument that it stores in a global vector, of length (number of react arguments), that can be accessed by various *output commands*. The vector values calculated by this fix are "intensive".

There is one quantity in the global vector for each *react* argument:

(1) cumulative number of reactions that occurred

Added in version 10Dec2025.

This fix supports automatically generated thermo column names when using *thermo_modify colname auto*. The thermo column names are "f_", followed by the fix ID, followed by a colon, followed by the react-ID. E.g., the first example in the Examples section above would print a thermo column name of "f_rxns:diels_alder", compared to the default column output name of "f_rxns[1]".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

When fix bond/react is "*unfixed*", all internally-created groups are deleted. Therefore, fix bond/react can only be unfixed after unfixing all other fixes that use any group created by fix bond/react.

### 2.35.6 Restrictions

This fix is part of the REACTION package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.35.7 Related commands

*fix bond/create*, *fix bond/break*, *fix bond/swap*, *dump local*, *special_bonds*, *fix_modify*

### 2.35.8 Default

The option defaults are *stabilization* = no, *prob* = 1.0, *stabilize_steps* = 60, *reset_mol_ids* = yes, *custom_charges* = no, *molecule* = off, *modify_create* = *fit all*, *vizsteps* = 1000

---

**(Gissinger2017)** Gissinger, Jensen and Wise, Polymer, 128, 211-217 (2017).

**(Gissinger2020)** Gissinger, Jensen and Wise, Macromolecules, 53, 22, 9953-9961 (2020).

**(Gissinger2024)** Gissinger, Jensen and Wise, Computer Physics Communications, 304, 109287 (2024).

## 2.36 fix bond/swap command

### 2.36.1 Syntax

```
fix ID group-ID bond/swap Nevery fraction cutoff seed
```

- ID, group-ID are documented in *fix* command
- bond/swap = style name of this fix command
- Nevery = attempt bond swapping every this many steps
- fraction = fraction of group atoms to consider for swapping
- cutoff = distance at which swapping will be considered (distance units)
- seed = random # seed (positive integer)

### 2.36.2 Examples

```
fix 1 all bond/swap 50 0.5 1.3 598934
```

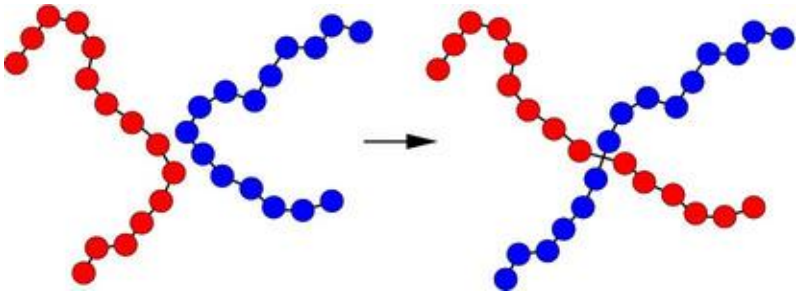
### 2.36.3 Description

In a simulation of polymer chains this command attempts to swap a pair of bonds, as illustrated below. This is done via Monte Carlo rules using the Boltzmann acceptance criterion, typically with the goal of equilibrating the polymer system more quickly. This fix is designed for use with idealized bead-spring polymer chains where each polymer is a linear chain of monomers, but LAMMPS does not check that is the case for your system.

Here are two use cases for this fix.

The first use case is for swapping bonds on two different chains, effectively grafting the end of one chain onto the other chain and vice versa. The purpose is to equilibrate the polymer chain conformations more rapidly than dynamics alone would do it, by enabling instantaneous large conformational changes in a dense polymer melt. The polymer chains should thus more rapidly converge to the proper end-to-end distances and radii of gyration.

A schematic of the kinds of bond swaps that can occur in this use case is shown here:



On the left, the red and blue chains have two monomers A1 and B1 close to each other, which are currently bonded to monomers A2 and B2 respectively within their own chains. The bond swap operation will attempt to delete the A1-A2 and B1-B2 bonds and replace them with A1-B2 and B1-A2 bonds. If the swap is energetically favorable, the two chains on the right are the result and each polymer chain has undergone a dramatic conformational change. This reference, *(Sides)* provides more details on the algorithm's effectiveness for this use case.

The second use case is a collection of polymer chains with some fraction of their sites identified as "sticker" sites. Initially each polymer chain is isolated from the others in a topological sense, and there is an intra-chain bond between every pair of sticker sites on the same chain. Over time, bonds swap so that inter-molecular sticker bonds are created. This models a vitrification-style process whereby the polymer chains all become interconnected. For this use case, if angles are defined they should not include bonds between sticker sites.

**Note**

For the sticker site model, you should set the newton flag for bonds to "off", via the *newton on off* command ("on" is the default for the 2nd argument). This is to ensure appropriate randomness in bond selection because the I,J bond will be stored by both atom I and atom J. LAMMPS cannot check for this, because it is not aware that a sticker site model is being used.

---

The bond swapping operation is invoked once every *Nevery* timesteps. If any bond in the entire system is swapped, a re-build of the neighbor lists is triggered, since a swap alters the list of which neighbors are considered for pairwise interaction. At each invocation, each processor considers a random specified *fraction* of its atoms as potential swapping monomers for this timestep. Choosing a small *fraction* value can reduce the likelihood of a reverse swap occurring soon after an initial swap.

For each monomer A1, its neighbors are looped over as B1 monomers. For each A1,B1 an additional double loop of bond partners A2 of A1, and bond partners B2 of B1 a is performed. For each pair of A1-A2 and B1-B2 bonds to be eligible for swapping, the following 4 criteria must be met:

1. All 4 monomers must be in the fix group.
2. All 4 monomers must be owned by the processor (not ghost atoms). This ensures that another processor does not attempt to swap bonds involving the same atoms on the same timestep. Note that this also means that bond pairs which straddle processor boundaries are not eligible for swapping on this step.
3. The distances between 4 pairs of atoms – (A1,A2), (B1,B2), (A1,B2), (B1,A2) – must all be less than the specified *cutoff*.
4. The molecule IDs of A1 and B1 must be the same (see below).

If an eligible B1 partner is found, the energy change due to swapping the two bonds is computed. This includes changes in pairwise, bond, and angle energies due to the altered connectivity of the 2 chains. Dihedral and improper interactions are not allowed to be defined when this fix is used.

If the energy decreases due to the swap operation, the bond swap is accepted. If the energy increases it is accepted with probability exp(-delta/kT) where delta is the increase in energy, k is the Boltzmann constant, and T is the current

temperature of the system.

**Note**

Whether the swap is accepted or rejected, no other swaps are attempted by this processor on this timestep. No other eligible 4-tuples of atoms are considered. This means that each processor will perform either a single swap or none on timesteps this fix is invoked.

---

The criterion for matching molecule IDs is how the first use case described above can be simulated while conserving chain lengths. This is done by setting up the molecule IDs for the polymer chains in a specific way, typically in the data file, read by the *read_data* command.

Consider a system of 6-mer chains. You have 2 choices. If the molecule IDs for monomers on each chain are set to 1,2,3,4,5,6 then swaps will conserve chain length. For a particular monomer there will be only one other monomer on another chain which is a potential swap partner. If the molecule IDs for monomers on each chain are set to 1,2,3,3,2,1 then swaps will conserve chain length but swaps will be able to occur at either end of a chain. Thus for a particular monomer there will be 2 possible swap partners on another chain. In this scenario, swaps can also occur within a single chain, i.e. the two ends of a chain swap with each other.

**Note**

If your simulation uses molecule IDs in the usual way, where all monomers on a single chain are assigned the same ID (different for each chain), then swaps will only occur within the same chain. If you assign the same molecule ID to all monomers in all chains then inter-chain swaps will occur, but they will not conserve chain length. Neither of these scenarios is probably what you want for this fix.

**Note**

When a bond swap occurs the image flags of monomers in the new polymer chains can become inconsistent. See the *dump* command for a discussion of image flags. This is not an issue for running dynamics, but can affect calculation of some diagnostic quantities or the printing of unwrapped coordinates to a dump file.

For the second use case described above, the molecule IDs for all sticker sites should be the same.

---

This fix computes a temperature each time it is invoked for use by the Boltzmann criterion. To do this, the fix creates its own compute of style *temp*, as if this command had been issued:

```
compute fix-ID_temp all temp
```

See the *compute temp* command for details. Note that the ID of the new compute is the fix-ID with underscore + "temp" appended and the group for the new compute is "all", so that the temperature of the entire system is used.

Note that this is NOT the compute used by thermodynamic output (see the *thermo_style* command) with ID = *thermo_temp*. This means you can change the attributes of this fix's temperature (e.g. its degrees-of-freedom) via the *compute_modify* command or print this temperature during thermodynamic output via the *thermo_style custom* command using the appropriate compute-ID. It also means that changing attributes of *thermo_temp* will have no effect on this fix.

---

### 2.36.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. Because the state of the random number generator is not saved in restart files, this means you cannot do "exact" restarts with this fix, where the simulation continues on the same as if no restart had taken place. However, in a statistical sense, a restarted simulation should produce the same behavior. Also note that each processor generates possible swaps independently of other processors. Thus if you repeat the same simulation on a different number of processors, the specific swaps performed will be different.

The *fix_modify* *temp* option is supported by this fix. You can use it to assign a *compute* you have defined to this fix which will be used to compute the temperature for the Boltzmann criterion.

This fix computes two statistical quantities as a global 2-vector of output, which can be accessed by various *output commands*. The first component of the vector is the cumulative number of swaps performed by all processors. The second component of the vector is the cumulative number of swaps attempted (whether accepted or rejected). Note that a swap "attempt" only occurs when swap partners meeting the criteria described above are found on a particular timestep. The vector values calculated by this fix are "intensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.36.5 Restrictions

This fix is part of the MC package. It is only enabled if LAMMPS was built with that package. See the *Build package* doc page for more info.

This fix requires using an atom style with molecule IDs.

The settings of the "special_bond" command must be 0,1,1 in order to use this fix, which is typical of bead-spring chains with FENE or harmonic bonds. This means that pairwise interactions between bonded atoms are turned off, but are turned on between atoms two or three hops away along the chain backbone.

Currently, energy changes in dihedral and improper interactions due to a bond swap are not considered. Thus a simulation that uses this fix cannot use a dihedral or improper potential.

### 2.36.6 Related commands

*fix atom/swap*

### 2.36.7 Default

none

---

**(Sides)** Sides, Grest, Stevens, Plimpton, J Polymer Science B, 42, 199-208 (2004).

## 2.37 fix box/relax command

### 2.37.1 Syntax

```
fix ID group-ID box/relax keyword value ...
```

- ID, group-ID are documented in *fix* command
- box/relax = style name of this fix command

  one or more keyword value pairs may be appended
  keyword = *iso* or *aniso* or *tri* or *x* or *y* or *z* or *xy* or *yz* or *xz* or *couple* or *nreset*
  ↪or *vmax* or *dilate* or *scaleyz* or *scalexz* or *scalexy* or *fixedpoint*

```
  iso or aniso or tri value = Ptarget = desired pressure (pressure units)
  x or y or z or xy or yz or xz value = Ptarget = desired pressure (pressure units)
  couple = none or xyz or xy or yz or xz
  nreset value = reset reference cell every this many minimizer iterations
  vmax value = fraction = max allowed volume change in one iteration
  dilate value = all or partial
  scaleyz value = yes or no = scale yz with lz
  scalexz value = yes or no = scale xz with lz
  scalexy value = yes or no = scale xy with ly
  fixedpoint values = x y z
    x,y,z = perform relaxation dilation/contraction around this point (distance␣
↪units)
```

### 2.37.2 Examples

```
fix 1 all box/relax iso 0.0 vmax 0.001
fix 2 water box/relax aniso 0.0 dilate partial
fix 2 ice box/relax tri 0.0 couple xy nreset 100
```

### 2.37.3 Description

Apply an external pressure or stress tensor to the simulation box during an *energy minimization*. This allows the box size and shape to vary during the iterations of the minimizer so that the final configuration will be both an energy minimum for the potential energy of the atoms, and the system pressure tensor will be close to the specified external tensor. Conceptually, specifying a positive pressure is like squeezing on the simulation box; a negative pressure typically allows the box to expand.

---

The external pressure tensor is specified using one or more of the *iso*, *aniso*, *tri*, *x*, *y*, *z*, *xy*, *xz*, *yz*, and *couple* keywords. These keywords give you the ability to specify all 6 components of an external stress tensor, and to couple various of these components together so that the dimensions they represent are varied together during the minimization.

Orthogonal simulation boxes have 3 adjustable dimensions (x,y,z). Triclinic (non-orthogonal) simulation boxes have 6 adjustable dimensions (x,y,z,xy,xz,yz). The *create_box*, *read data*, and *read_restart* commands specify whether the simulation box is orthogonal or non-orthogonal (triclinic) and explain the meaning of the xy,xz,yz tilt factors.

The target pressures *Ptarget* for each of the 6 components of the stress tensor can be specified independently via the *x*, *y*, *z*, *xy*, *xz*, *yz* keywords, which correspond to the 6 simulation box dimensions. For example, if the *y* keyword is used, the y-box length will change during the minimization. If the *xy* keyword is used, the xy tilt factor will change. A box dimension will not change if that component is not specified.

Note that in order to use the *xy*, *xz*, or *yz* keywords, the simulation box must be triclinic, even if its initial tilt factors are 0.0.

When the size of the simulation box changes, all atoms are re-scaled to new positions, unless the keyword *dilate* is specified with a value of *partial*, in which case only the atoms in the fix group are re-scaled. This can be useful for leaving the coordinates of atoms in a solid substrate unchanged and controlling the pressure of a surrounding fluid.

The *scaleyz*, *scalexz*, and *scalexy* keywords control whether or not the corresponding tilt factors are scaled with the associated box dimensions when relaxing triclinic periodic cells. The default values *yes* will turn on scaling, which corresponds to adjusting the linear dimensions of the cell while preserving its shape. Choosing *no* ensures that the tilt factors are not scaled with the box dimensions. See below for restrictions and default values in different situations. In older versions of LAMMPS, scaling of tilt factors was not performed. The old behavior can be recovered by setting all three scale keywords to *no*.

The *fixedpoint* keyword specifies the fixed point for cell relaxation. By default, it is the center of the box. Whatever point is chosen will not move during the simulation. For example, if the lower periodic boundaries pass through (0,0,0), and

this point is provided to *fixedpoint*, then the lower periodic boundaries will remain at (0,0,0), while the upper periodic boundaries will move twice as far. In all cases, the particle positions at each iteration are unaffected by the chosen value, except that all particles are displaced by the same amount, different on each iteration.

**Note**

Applying an external pressure to tilt dimensions *xy*, *xz*, *yz* can sometimes result in arbitrarily large values of the tilt factors, i.e. a dramatically deformed simulation box. This typically indicates that there is something badly wrong with how the simulation was constructed. The two most common sources of this error are applying a shear stress to a liquid system or specifying an external shear stress tensor that exceeds the yield stress of the solid. In either case the minimization may converge to a bogus conformation or not converge at all. Also note that if the box shape tilts to an extreme shape, LAMMPS will run less efficiently, due to the large volume of communication needed to acquire ghost atoms around a processor's irregular-shaped subdomain. For extreme values of tilt, LAMMPS may also lose atoms and generate an error.

**Note**

Performing a minimization with this fix is not a mathematically well-defined minimization problem. This is because the objective function being minimized changes if the box size/shape changes. In practice this means the minimizer can get "stuck" before you have reached the desired tolerance. The solution to this is to restart the minimizer from the new adjusted box size/shape, since that creates a new objective function valid for the new box size/shape. Repeat as necessary until the box size/shape has reached its new equilibrium.

---

The *couple* keyword allows two or three of the diagonal components of the pressure tensor to be "coupled" together. The value specified with the keyword determines which are coupled. For example, *xz* means the *Pxx* and *Pzz* components of the stress tensor are coupled. *Xyz* means all 3 diagonal components are coupled. Coupling means two things: the instantaneous stress will be computed as an average of the corresponding diagonal components, and the coupled box dimensions will be changed together in lockstep, meaning coupled dimensions will be dilated or contracted by the same percentage every timestep. The *Ptarget* values for any coupled dimensions must be identical. *Couple xyz* can be used for a 2d simulation; the *z* dimension is simply ignored.

---

The *iso*, *aniso*, and *tri* keywords are simply shortcuts that are equivalent to specifying several other keywords together.

The keyword *iso* means couple all 3 diagonal components together when pressure is computed (hydrostatic pressure), and dilate/contract the dimensions together. Using "iso Ptarget" is the same as specifying these 4 keywords:

```
x Ptarget y Ptarget z Ptarget couple xyz
```

The keyword *aniso* means *x*, *y*, and *z* dimensions are controlled independently using the *Pxx*, *Pyy*, and *Pzz* components of the stress tensor as the driving forces, and the specified scalar external pressure. Using "aniso Ptarget" is the same as specifying these 4 keywords:

```
x Ptarget y Ptarget z Ptarget couple none
```

The keyword *tri* means *x*, *y*, *z*, *xy*, *xz*, and *yz* dimensions are controlled independently using their individual stress components as the driving forces, and the specified scalar pressure as the external normal stress. Using "tri Ptarget" is the same as specifying these 7 keywords:

```
x Ptarget y Ptarget z Ptarget xy 0.0 yz 0.0 xz 0.0 couple none
```

The *vmax* keyword can be used to limit the fractional change in the volume of the simulation box that can occur in one iteration of the minimizer. If the pressure is not settling down during the minimization this can be because the volume is fluctuating too much. The specified fraction must be greater than 0.0 and should be << 1.0. A value of 0.001 means the volume cannot change by more than 1/10 of a percent in one iteration when *couple xyz* has been specified. For any other case it means no linear dimension of the simulation box can change by more than 1/10 of a percent.

---

With this fix, the potential energy used by the minimizer is augmented by an additional energy provided by the fix. The overall objective function then is:

$$E = U + P_t\,(V - V_0) + E_{strain}$$

where $U$ is the system potential energy, $P_t$ is the desired hydrostatic pressure, $V$ and $V_0$ are the system and reference volumes, respectively. $E_{strain}$ is the strain energy expression proposed by Parrinello and Rahman *(Parrinello1981)*. Taking derivatives of $E$ w.r.t. the box dimensions, and setting these to zero, we find that at the minimum of the objective function, the global system stress tensor **P** will satisfy the relation:

$$\mathbf{P} = P_t\mathbf{I} + \mathbf{S}_t\,\left(\mathbf{h}_0^{-1}\right)^t\,\mathbf{h}_{0d}$$

where **I** is the identity matrix, $\mathbf{h_0}$ is the box dimension tensor of the reference cell, and :$\mathbf{h_{0d}}$ is the diagonal part of $\mathbf{h_0}$. $\mathbf{S_t}$ is a symmetric stress tensor that is chosen by LAMMPS so that the upper-triangular components of **P** equal the stress tensor specified by the user.

This equation only applies when the box dimensions are equal to those of the reference dimensions. If this is not the case, then the converged stress tensor will not equal that specified by the user. We can resolve this problem by periodically resetting the reference dimensions. The keyword *nreset* controls how often this is done. If this keyword is not used, or is given a value of zero, then the reference dimensions are set to those of the initial simulation domain and are never changed. A value of *nstep* means that every *nstep* minimization steps, the reference dimensions are set to those of the current simulation domain. Note that resetting the reference dimensions changes the objective function and gradients, which sometimes causes the minimization to fail. This can be resolved by changing the value of *nreset*, or simply continuing the minimization from a restart file.

**Note**

As normally computed, pressure includes a kinetic- energy or temperature-dependent component; see the *compute pressure* command. However, atom velocities are ignored during a minimization, and the applied pressure(s) specified with this command are assumed to only be the virial component of the pressure (the non-kinetic portion). Thus if atoms have a non-zero temperature and you print the usual thermodynamic pressure, it may not appear the system is converging to your specified pressure. The solution for this is to either (a) zero the velocities of all atoms before performing the minimization, or (b) make sure you are monitoring the pressure without its kinetic component. The latter can be done by outputting the pressure from the pressure compute this command creates (see below) or a pressure compute you define yourself.

**Note**

Because pressure is often a very sensitive function of volume, it can be difficult for the minimizer to equilibrate the system the desired pressure with high precision, particularly for solids. Some techniques that seem to help are (a) use the "min_modify line quadratic" option when minimizing with box relaxations, (b) minimize several times in succession if need be, to drive the pressure closer to the target pressure, (c) relax the atom positions before relaxing the box, and (d) relax the box to the target hydrostatic pressure before relaxing to a target shear stress state. Also note that some systems (e.g. liquids) will not sustain a non-hydrostatic applied pressure, which means the minimizer will not converge.

This fix computes a temperature and pressure each timestep. The temperature is used to compute the kinetic contribution to the pressure, even though this is subsequently ignored by default. To do this, the fix creates its own computes of style "temp" and "pressure", as if these commands had been issued:

```
compute fix-ID_temp group-ID temp
compute fix-ID_press group-ID pressure fix-ID_temp virial
```

See the *compute temp* and *compute pressure* commands for details. Note that the IDs of the new computes are the fix-ID + underscore + "temp" or fix_ID + underscore + "press", and the group for the new computes is the same as the fix group. Also note that the pressure compute does not include a kinetic component.

Note that these are NOT the computes used by thermodynamic output (see the *thermo_style* command) with ID = *thermo_temp* and *thermo_press*. This means you can change the attributes of this fix's temperature or pressure via the *compute_modify* command or print this temperature or pressure during thermodynamic output via the *thermo_style custom* command using the appropriate compute-ID. It also means that changing attributes of *thermo_temp* or *thermo_press* will have no effect on this fix.

### 2.37.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify temp* and *press* options are supported by this fix. You can use them to assign a *compute* you have defined to this fix which will be used in its temperature and pressure calculation, as described above. Note that as described above, if you assign a pressure compute to this fix that includes a kinetic energy component it will affect the minimization, most likely in an undesirable way.

**Note**

If both the *temp* and *press* keywords are used in a single thermo_modify command (or in two separate commands), then the order in which the keywords are specified is important. Note that a *pressure compute* defines its own temperature compute as an argument when it is specified. The *temp* keyword will override this (for the pressure compute being used by fix box/relax), but only if the *temp* keyword comes after the *press* keyword. If the *temp* keyword comes before the *press* keyword, then the new pressure compute specified by the *press* keyword will be unaffected by the *temp* setting.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the pressure-volume energy, plus the strain energy, if it exists, as described above. The energy values reported at the end of a minimization run under "Minimization stats" include this energy, and so differ from what LAMMPS normally reports as potential energy. This fix does not support the *fix_modify energy* option, because that would result in double-counting of the fix energy in the minimization energy. Instead, the fix energy can be explicitly added to the potential energy using one of these two variants:

```
variable emin equal pe+f_1

variable emin equal pe+f_1/atoms
```

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

This fix is invoked during *energy minimization*, but not for the purpose of adding a contribution to the energy or forces being minimized. Instead it alters the simulation box geometry as described above.

### 2.37.5 Restrictions

Only dimensions that are available can be adjusted by this fix. Non-periodic dimensions are not available. *z*, *xz*, and *yz*, are not available for 2D simulations. *xy*, *xz*, and *yz* are only available if the simulation domain is non-orthogonal. The *create_box*, *read data*, and *read_restart* commands specify whether the simulation box is orthogonal or non-orthogonal (triclinic) and explain the meaning of the xy,xz,yz tilt factors.

The *scaleyz yes* and *scalexz yes* keyword/value pairs can not be used for 2D simulations. *scaleyz yes*, *scalexz yes*, and *scalexy yes* options can only be used if the second dimension in the keyword is periodic, and if the tilt factor is not coupled to the barostat via keywords *tri*, *yz*, *xz*, and *xy*.

### 2.37.6 Related commands

*fix npt*, *minimize*

### 2.37.7 Default

The keyword defaults are dilate = all, vmax = 0.0001, nreset = 0.

---

**(Parrinello1981)** Parrinello and Rahman, J Appl Phys, 52, 7182 (1981).

## 2.38 fix brownian command

## 2.39 fix brownian/sphere command

## 2.40 fix brownian/asphere command

### 2.40.1 Syntax

```
fix ID group-ID style_name temp seed keyword args
```

- ID, group-ID are documented in *fix* command
- style_name = *brownian* or *brownian/sphere* or *brownian/asphere*
- temp = temperature
- seed = random number generator seed
- one or more keyword/value pairs may be appended
- keyword = *rng* or *dipole* or *gamma_r_eigen* or *gamma_t_eigen* or *gamma_r* or *gamma_t* or *rotation_style* or *rotation_temp* or *planar_rotation*

```
rng value = uniform or gaussian or none
  uniform = use uniform random number generator
  gaussian = use gaussian random number generator
  none = turn off noise
dipole value = mux and muy and muz for brownian/asphere
  mux, muy, and muz = update orientation of dipole having direction (mux,*muy*,
↪*muz*) in body frame of rigid body
gamma_r_eigen values = gr1 and gr2 and gr3 for brownian/asphere
  gr1, gr2, and gr3 = diagonal entries of body frame rotational friction tensor
gamma_r values = gr for brownian/sphere
  gr = magnitude of the (isotropic) rotational friction tensor
gamma_t_eigen values = gt1 and gt2 and gt3 for brownian/asphere
```

```
  gt1, gt2, and gt3 = diagonal entries of body frame translational friction tensor
gamma_t values = gt for brownian and brownian/sphere
   gt = magnitude of the (isotropic) translational friction tensor
rotation_style values = geometric or projection for brownian/sphere
   geometric = geometric, rotation-based integration scheme
   projection = projection-based integration scheme
rotation_temp values = T for brownian/sphere and brownian/asphere
   T = rotation temperature, which can be different than temp when out of␣
↪equilibrium
planar_rotation values = none (constrains rotational diffusion to be in xy plane if␣
↪in 3D)
```

## 2.40.2 Examples

```
fix 1 all brownian 1.0 12908410 gamma_t 1.0
fix 1 all brownian 1.0 12908410 gamma_t 3.0 rng gaussian
fix 1 all brownian/sphere 1.0 1294019 gamma_t 3.0 gamma_r 1.0
fix 1 all brownian/sphere 1.0 19581092 gamma_t 1.0 gamma_r 0.3 rng none
fix 1 all brownian/sphere 1.0 19581092 gamma_t 1.0 gamma_r 0.3 rng gaussian rotation_
↪style projection
fix 1 all brownian/asphere 1.0 1294019 gamma_t_eigen 1.0 2.0 3.0 gamma_r_eigen 4.0 7.0 8.
↪0 rng gaussian
fix 1 all brownian/asphere 1.0 1294019 gamma_t_eigen 1.0 2.0 3.0 gamma_r_eigen 4.0 7.0 8.
↪0 dipole 1.0 0.0 0.0
```

## 2.40.3 Description

Perform Brownian Dynamics time integration to update position, velocity, dipole orientation (for spheres) and quaternion orientation (for ellipsoids, with optional dipole update as well) of all particles in the fix group in each timestep. Brownian Dynamics uses Newton's laws of motion in the limit that inertial forces are negligible compared to viscous forces. The stochastic equation of motion for the center of mass positions is

$$d\mathbf{r} = \gamma_t^{-1}\mathbf{F}dt + \sqrt{2k_BT}\gamma_t^{-1/2}d\mathbf{W}_t,$$

in the lab-frame (i.e., $\gamma_t$ is not diagonal, but only depends on orientation and so the noise is still additive).

The overdamped rotational motion for the spherical and ellipsoidal particles results from random rotations instead of translations, which are chosen in such a way that the motion reproduces the equilibrium Boltzmann distribution for the case of conservative torques (see *(Hoefling)*, *(Ilie)*, and *(Delong)*).

For the style *brownian*, only the positions of the particles are updated. This is therefore suitable for point particle simulations.

For the style *brownian/sphere*, the positions of the particles are updated, and a dipole slaved to the spherical orientation is also updated. This style therefore requires the hybrid atom style *atom_style dipole* and *atom_style sphere*. The equation of motion for the dipole is

$$\mu(t+dt) = \mathrm{R}(\omega dt)\,\mu(t)\,,$$

where $\mathrm{R}(\omega dt)$ is a rotation matrix with axis $n = \omega/|\omega|$ and angle $\theta = |\omega|dt$ (see *(Hoefling)*). For small angles, the action of the rotation matrix can be cast into a tangential increment $\omega \times \mu dt$ and subsequent projection to preserve the magnitude $|\mu(t)|$ of the dipole (see *(Ilie)*):

$$\mu(t+dt) = |\mu(t)|\,\frac{\mu(t)+\omega\times\mu dt}{|\mu(t)+\omega\times\mu dt|}\,.$$

For suitable time step $dt$, both expressions were shown to correctly reproduce the Boltzmann distribution of orientations and rotational diffusion moments when

$$\omega = \frac{\mathbf{T}}{\gamma_r} + \sqrt{\frac{2k_B T_{rot}}{\gamma_r}\frac{d\mathbf{W}}{dt}},$$

with $d\mathbf{W}$ being a random number with zero mean and variance $dt$ and $T_{rot}$ is *rotation_temp*. The geometric integration scheme, however, accepts time steps that can be an order of magnitude larger (see *(Hoefling)*).

For the style *brownian/asphere*, the center of mass positions and the quaternions of ellipsoidal particles are updated. This fix style is suitable for equations of motion where the rotational and translational friction tensors can be diagonalized in a certain (body) reference frame. In this case, the rotational equation of motion is updated via the quaternion

$$\mathbf{q}(t + dt) = \frac{\mathbf{q}(t) + d\mathbf{q}}{\|\mathbf{q}(t) + d\mathbf{q}\|}$$

which correctly reproduces a Boltzmann distribution of orientations and rotational diffusion moments [see *(Ilie)*] when the quaternion step is given by

$$d\mathbf{q} = \Psi\omega dt$$

where $\Psi$ has rows $(-q_1, -q_2, -q_3)$, $(q_0, -q_3, q_2)$, $(q_3, q_0, -q_1)$, and $(-q_2, q_1, q_0)$. $\omega$ is evaluated in the body frame of reference where the friction tensor is diagonal. See *(Delong)* for more details of a similar algorithm.

---

**Note**

This integrator does not by default assume a relationship between the rotational and translational friction tensors, though such a relationship should exist in the case of no-slip boundary conditions between the particles and the surrounding (implicit) solvent. For example, in the case of spherical particles, the condition $\gamma_t = 3\gamma_r/\sigma^2$ must be explicitly accounted for by setting *gamma_t* to 3x and *gamma_r* to x (where $\sigma$ is the sphere's diameter). A similar (though more complex) relationship holds for ellipsoids and rod-like particles. The translational diffusion and rotational diffusion are given by *temp/gamma_t* and *rotation_temp/gamma_r*.

---

**Note**

Temperature computation using the *compute temp* will not correctly compute the temperature of these overdamped dynamics since we are explicitly neglecting inertial effects. Furthermore, this time integrator does not add the stochastic terms or viscous terms to the force and/or torques. Rather, they are just added in to the equations of motion to update the degrees of freedom.

---

If the *rng* keyword is used with the *uniform* value, then the noise is generated from a uniform distribution (see *(Dunweg)* for why this works). This is the same method of noise generation as used in *fix_langevin*.

If the *rng* keyword is used with the *gaussian* value, then the noise is generated from a Gaussian distribution. Typically this added complexity is unnecessary, and one should be fine using the *uniform* value for reasons argued in *(Dunweg)*.

If the *rng* keyword is used with the *none* value, then the noise terms are set to zero.

The *gamma_t* keyword sets the (isotropic) translational viscous damping. Required for (and only compatible with) *brownian* and *brownian/sphere*. The units of *gamma_t* are mass/time.

The *gamma_r* keyword sets the (isotropic) rotational viscous damping. Required for (and only compatible with) *brownian/sphere*. The units of *gamma_r* are mass*length**2/time.

The *gamma_r_eigen*, and *gamma_t_eigen* keywords are the eigenvalues of the rotational and viscous damping tensors (having the same units as their isotropic counterparts). Required for (and only compatible with) *brownian/asphere*. For a 2D system, the first two values of *gamma_r_eigen* must be *inf* (only rotation in *x*–*y* plane), and the third value of *gamma_t_eigen* must be *inf* (only diffusion in the *x*–*y* plane).

If the *dipole* keyword is used, then the dipole moments of the particles are updated as described above. Only compatible with *brownian/asphere* (as *brownian/sphere* updates dipoles automatically).

Added in version 11Feb2026.

If the *rotation_style* keyword is used with the *geometric* value, then the geometric, rotation-based integration scheme (*(Hoefling)*) is used. If the keyword is used with the *projection* value, the linearized, projection-based scheme (*(Ilie)*) is used. Only compatible with *brownian/sphere*.

Note: *rotation_style projection* reproduces the legacy behavior (the former default).

If the *rotation_temp* keyword is used, then the rotational diffusion will occur at this prescribed temperature instead of *temp*. Only compatible with *brownian/sphere* and *brownian/asphere*.

If the *planar_rotation* keyword is used, then rotation is constrained to the *x*– *y* plane in a 3D simulation. Only compatible with *brownian/sphere* and *brownian/asphere* in 3D.

---

**Note**

For style *brownian/asphere*, the components *gamma_t_eigen* = (x,x,x) and *gamma_r_eigen* = (y,y,y), the dynamics will replicate those of the *brownian/sphere* style with *gamma_t* = x and *gamma_r* = y.

---

### 2.40.4 Restart, fix_modify, output, run start/stop, minimize info

Changed in version 4Jul2026.

This fix writes the state of its random number generator to *binary restart files*, so that a simulation continued from a restart reproduces the same stochastic trajectory (provided the number of MPI processes is unchanged). No global or per-atom quantities are stored by this fix for access by various *output commands*.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.40.5 Restrictions

The style *brownian/sphere* fix requires that atoms store torque and angular velocity (omega) as defined by the *atom_style sphere* command. The style *brownian/asphere* fix requires that atoms store torque and quaternions as defined by the *atom_style ellipsoid* command. If the *dipole* keyword is used, they must also store a dipole moment as defined by the *atom_style dipole* command.

This fix is part of the BROWNIAN package. It is only enabled if LAMMPS was built with that package. See the *Build package* doc page for more info.

### 2.40.6 Related commands

*fix propel/self*, *fix langevin*, *fix nve/sphere*,

### 2.40.7 Default

Changed in version 11Feb2026.

The default for *rng* is *uniform*. The default for *rotation_style* is *geometric*. The default for the rotational and translational friction tensors are the identity tensor.

---

**(Hoefling)** Höfling and Straube, Physical Review Research, 7, 043034 (2025).

**(Ilie)** Ilie, Briels, den Otter, Journal of Chemical Physics, 142, 114103 (2015).

**(Delong)** Delong, Usabiaga, Donev, Journal of Chemical Physics. 143, 144107 (2015).

**(Dunweg)** Dunweg and Paul, Int J of Modern Physics C, 2, 817-27 (1991).

## 2.41 fix charge/regulation command

### 2.41.1 Syntax

```
fix ID group-ID charge/regulation cation_type anion_type keyword value(s)
```

- ID, group-ID are documented in fix command
- charge/regulation = style name of this fix command
- cation_type = atom type of free cations (integer or type label)
- anion_type = atom type of free anions (integer or type label)
- zero or more keyword/value pairs may be appended

```
keyword = pH, pKa, pKb, pIp, pIm, pKs, acid_type, base_type, lunit_nm, temp,␣
↪tempfixid, nevery, nmc, rxd, seed, tag, group, onlysalt, pmcmoves
pH value = pH of the solution (can be specified as an equal-style variable)
pKa value = acid dissociation constant (in the -log10 representation)
pKb value = base dissociation constant (in the -log10 representation)
pIp value = activity (effective concentration) of free cations (in the -log10␣
↪representation)
pIm value = activity (effective concentration) of free anions (in the -log10␣
↪representation)
pKs value = solvent self-dissociation constant (in the -log10 representation)
acid_type = atom type of acid groups (integer or type label)
base_type  = atom type of base groups (integer or type label)
lunit_nm value = unit length used by LAMMPS (# in the units of nanometers)
temp value = temperature
tempfixid value = fix ID of temperature thermostat
nevery value = invoke this fix every nevery steps
nmc value = number of charge regulation MC moves to attempt every nevery steps
rxd value = cutoff distance for acid/base reaction
seed value = random # seed (positive integer)
tag value = yes or no (yes: The code assign unique tags to inserted ions; no: The␣
↪tag of all inserted ions is "0")
```

```
group value = group-ID, inserted ions are assigned to group group-ID. Can be used␣
↪multiple times to assign inserted ions to multiple groups.
onlysalt values = flag charge_cation charge_anion.
   flag = yes or no (yes: the fix performs only ion insertion/deletion, no: perform␣
↪acid/base dissociation and ion insertion/deletion)
   charge_cation, charge_anion = value of cation/anion charge, must be an integer␣
↪(only specify if flag = yes)
pmcmoves values = pmcA pmcB pmcI -  MC move fractions for acid ionization (pmcA),␣
↪base ionization (pmcB) and free ion exchange (pmcI)
```

### 2.41.2 Examples

```
fix chareg all charge/regulation 1 2 acid_type 3 base_type 4 pKa 5.0 pKb 6.0 pH 7.0 pIp␣
↪3.0 pIm 3.0 nevery 200 nmc 200 seed 123 tempfixid fT
fix chareg all charge/regulation 1 2 pIp 3 pIm 3 onlysalt yes 2 -1 seed 123 tag yes temp␣
↪1.0

labelmap atom 1 H+ 2 OH-
fix chareg all charge/regulation H+ OH- pIp 3 pIm 3 onlysalt yes 2 -1 seed 123 tag yes␣
↪temp 1.0
```

### 2.41.3 Description

This fix performs Monte Carlo (MC) sampling of charge regulation and exchange of ions with a reservoir as discussed in *(Curk1)* and *(Curk2)*. The implemented method is largely analogous to the grand-reaction ensemble method in *(Landsgesell)*. The implementation is parallelized, compatible with existing LAMMPS functionalities, and applicable to any system utilizing discrete, ionizable groups or surface sites.

Specifically, the fix implements the following three types of MC moves, which discretely change the charge state of individual particles and insert ions into the systems: $\mathrm{A} \rightleftharpoons \mathrm{A}^- + \mathrm{X}^+$, $\mathrm{B} \rightleftharpoons \mathrm{B}^+ + \mathrm{X}^-$, and $\emptyset \rightleftharpoons Z^-\mathrm{X}^{Z^+} + Z^+\mathrm{X}^{-Z^-}$. In the former two types of reactions, Monte Carlo moves alter the charge value of specific atoms (A, B) and simultaneously insert a counterion to preserve the charge neutrality of the system, modeling the dissociation/association process. The last type of reaction performs grand canonical MC exchange of ion pairs with a (fictitious) reservoir.

In our implementation "acid" refers to particles that can attain charge $q = \{0, -1\}$ and "base" to particles with $q = \{0, 1\}$, whereas the MC exchange of free ions allows any integer charge values of $Z^+$ and $Z^-$.

Here we provide several practical examples for modeling charge regulation effects in solvated systems. An acid ionization reaction ($\mathrm{A} \rightleftharpoons \mathrm{A}^- + \mathrm{H}^+$) can be defined via a single line in the input file

```
fix acid_reaction all charge/regulation 2 3 acid_type 1 pH 7.0 pKa 5.0 pIp 7.0 pIm 7.0
```

where the fix attempts to charge A (discharge $\mathrm{A}^-$) to $\mathrm{A}^-$ (A) and insert (delete) a proton (atom type 2). Besides, the fix implements self-ionization reaction of water $\emptyset \rightleftharpoons \mathrm{H}^+ + \mathrm{OH}^-$.

However, this approach is highly inefficient at pH $\approx$ 7 when the concentration of both protons and hydroxyl ions is low, resulting in a relatively low acceptance rate of MC moves.

A more efficient way is to allow salt ions to participate in ionization reactions, which can be easily achieved via

```
fix acid_reaction2 all charge/regulation 4 5 acid_type 1 pH 7.0 pKa 5.0 pIp 2.0 pIm 2.0
```

where particles of atom type 4 and 5 are the salt cations and anions, both at activity (effective concentration) of $10^{-2}$ mol/l, see *(Curk1)* and *(Landsgesell)* for more details.

We could have simultaneously added a base ionization reaction ($\mathrm{B} \rightleftharpoons \mathrm{B}^+ + \mathrm{OH}^-$)

```
fix acid_base_reaction all charge/regulation 2 3 acid_type 1 base_type 6 pH 7.0 pKa 5.0␣
↪pKb 6.0 pIp 7.0 pIm 7.0
```

where the fix will attempt to charge B (discharge $B^+$) to $B^+$ (B) and insert (delete) a hydroxyl ion $OH^-$ of atom type 3.

Dissociated ions and salt ions can be combined into a single particle type, which reduces the number of necessary MC moves and increases sampling performance, see *(Curk1)*. The $H^+$ and monovalent salt cation ($S^+$) are combined into a single particle type, $X^+ = \{H^+, S^+\}$. In this case "pIp" refers to the effective concentration of the combined cation type $X^+$ and its value is determined by $10^{-pIp} = 10^{-pH} + 10^{-pSp}$, where $10^{-pSp}$ is the effective concentration of salt cations. For example, at pH=7 and pSp=6 we would find pIp~5.958 and the command that performs reactions with combined ions could read,

```
fix acid_reaction_combined all charge/regulation 2 3 acid_type 1 pH 7.0 pKa 5.0 pIp 5.
↪958 pIm 5.958
```

If neither the acid or the base type is specified, for example,

```
fix salt_reaction all charge/regulation 4 5 pIp 2.0 pIm 2.0
```

the fix simply inserts or deletes an ion pair of a free cation (atom type 4) and a free anion (atom type 5) as done in a conventional grand-canonical MC simulation. Multivalent ions can be inserted (deleted) by using the *onlysalt* keyword.

This fix is compatible with LAMMPS packages such as MOLECULE or RIGID. The acid and base particles can be part of larger molecules or rigid bodies. Free ions that are inserted to or deleted from the system must be defined as single particles (no bonded interactions allowed) and cannot be part of larger molecules or rigid bodies. If an atom style with molecule IDs is used, all inserted ions have a molecule ID equal to zero.

Note that LAMMPS implicitly assumes a constant number of particles (degrees of freedom). Since using this fix alters the total number of particles during the simulation, any thermostat used by LAMMPS, such as NVT or Langevin, must use a dynamic calculation of system temperature. This can be achieved by specifying a dynamic temperature compute (e.g. dtemp) and using it with the desired thermostat, e.g. a Langevin thermostat:

```
compute dtemp all temp
compute_modify dtemp dynamic/dof yes
fix fT all langevin 1.0 1.0 1.0 123
fix_modify fT temp dtemp
```

The units of pH, pKa, pKb, pIp, pIm are considered to be in the standard -log10 representation assuming reference concentration $\rho_0$ = mol/l. For example, in the dilute ideal solution limit, the concentration of free cations will be $c_1 = 10^{-pIp}$mol/l. To perform the internal unit conversion, the the value of the LAMMPS unit length must be specified in nanometers via *lunit_nm*. The default value is set to the Bjerrum length in water at room temperature (0.71 nm), *lunit_nm* = 0.71.

The temperature used in MC acceptance probability is set by *temp*. This temperature should be the same as the temperature set by the molecular dynamics thermostat. For most purposes, it is probably best to use *tempfixid* keyword which dynamically sets the temperature equal to the chosen MD thermostat temperature, in the example above we assumed the thermostat fix-ID is *fT*. The inserted particles attain a random velocity corresponding to the specified temperature. Using *tempfixid* overrides any fixed temperature set by *temp*.

The *rxd* keyword can be used to restrict the inserted/deleted counterions to a specific radial distance from an acid or base particle that is currently participating in a reaction. This can be used to simulate more realist reaction dynamics. If *rxd* = 0 or *rxd* > *L* / 2, where *L* is the smallest box dimension, the radial restriction is automatically turned off and free ion can be inserted or deleted anywhere in the simulation box.

If the *tag yes* is used, every inserted atom gets a unique tag ID, otherwise, the tag of every inserted atom is set to 0. *tag yes* might cause an integer overflow in very long simulations since the tags are unique to every particle and thus increase with every successful particle insertion.

The *pmcmoves* keyword sets the relative probability of attempting the three types of MC moves (reactions): acid charging, base charging, and ion pair exchange. The fix only attempts to perform particle charging MC moves if *acid_type* or *base_type* is defined. Otherwise fix only performs free ion insertion/deletion. For example, if *acid_type* is not defined, *pmcA* is automatically set to 0. The vector *pmcmoves* is automatically normalized, for example, if set to *pmcmoves* 0 0.33 0.33, the vector would be normalized to [0,0.5,0.5].

The *only_salt* option can be used to perform multivalent grand-canonical ion-exchange moves. If *only_salt yes* is used, no charge exchange is performed, only ion insertion/deletion (*pmcmoves* is set to [0,0,1]), but ions can be multivalent. In the example above, an MC move would consist of three ion insertion/deletion to preserve the charge neutrality of the system.

The *group* keyword can be used to add inserted particles to a specific group-ID. All inserted particles are automatically added to group *all*.

### 2.41.4 Output

This fix computes a global vector of length 8, which can be accessed by various output commands. The vector values are the following global quantities:

1. cumulative MC attempts
2. cumulative MC successes
3. current # of neutral acid atoms
4. current # of -1 charged acid atoms
5. current # of neutral base atoms
6. current # of +1 charged base atoms
7. current # of free cations
8. current # of free anions

### 2.41.5 Restrictions

This fix is part of the MC package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The *atom_style*, used must contain the charge property and have per atom type masses, for example, the style could be *charge* or *full*. Only usable for 3D simulations. Atoms specified as free ions cannot be part of rigid bodies or molecules and cannot have bonding interactions. The scheme is limited to integer charges, any atoms with non-integer charges will not be considered by the fix.

All interaction potentials used must be continuous, otherwise the MD integration and the particle exchange MC moves do not correspond to the same equilibrium ensemble. For example, if an lj/cut pair style is used, the LJ potential must be shifted so that it vanishes at the cutoff. This can be easily achieved using the *pair_modify* command, i.e., by using: *pair_modify shift yes*.

**Note**

Region restrictions are not yet implemented.

### 2.41.6 Related commands

*fix gcmc*, *fix atom/swap*

### 2.41.7 Default

pH = 7.0; pKa = 100.0; pKb = 100.0; pIp = 5.0; pIm = 5.0; pKs = 14.0; acid_type = -1; base_type = -1; lunit_nm = 0.71; temp = 1.0; nevery = 100; nmc = 100; rxd = 0; seed = 0; tag = no; onlysalt = no, pmcmoves = [1/3, 1/3, 1/3], group-ID = all

---

**(Curk1)** T. Curk, J. Yuan, and E. Luijten, "Accelerated simulation method for charge regulation effects", JCP 156 (2022).

**(Curk2)** T. Curk and E. Luijten, "Charge-regulation effects in nanoparticle self-assembly", PRL 126 (2021)

**(Landsgesell)** J. Landsgesell, P. Hebbeker, O. Rud, R. Lunkad, P. Kosovan, and C. Holm, "Grand-reaction method for simulations of ionization equilibria coupled to ion partitioning", Macromolecules 53, 3007-3020 (2020).

## 2.42 fix cmap command

Accelerator Variants: *cmap/kk*

### 2.42.1 Syntax

```
fix ID group-ID cmap filename
```

- ID, group-ID are documented in *fix* command
- cmap = style name of this fix command
- filename = force-field file with CMAP coefficients

### 2.42.2 Examples

```
fix           myCMAP all cmap ../potentials/cmap36.data
read_data     proteinX.data fix myCMAP crossterm CMAP
fix_modify    myCMAP energy yes
```

### 2.42.3 Description

This command enables CMAP 5-body interactions to be added to simulations which use the CHARMM force field. These are relevant for any CHARMM model of a peptide or protein sequences that is 3 or more amino-acid residues long; see *(Buck)* and *(Brooks)* for details, including the analytic energy expressions for CMAP interactions. The CMAP 5-body terms add additional potential energy contributions to pairs of overlapping phi-psi dihedrals of amino-acids, which are important to properly represent their conformational behavior.

The examples/cmap directory has a sample input script and data file for a small peptide, that illustrates use of the fix cmap command.

As in the example above, this fix should be used before reading a data file that contains a listing of CMAP interactions. The *filename* specified should contain the CMAP parameters for a particular version of the CHARMM force field. Two such files are including in the lammps/potentials directory: charmm22.cmap and charmm36.cmap.

The data file read by the "read_data" must contain the topology of all the CMAP interactions, similar to the topology data for bonds, angles, dihedrals, etc. Specially it should have a line like this in its header section:

```
N crossterms
```

where N is the number of CMAP 5-body interactions. It should also have a section in the body of the data file like this with N lines:

```
CMAP

      1       1       8      10      12      18      20
      2       5      18      20      22      25      27
      [...]
      N       3     314     315     317     318     330
```

The first column is an index from 1 to N to enumerate the CMAP 5-atom tuples; it is ignored by LAMMPS. The second column is the "type" of the interaction; it is an index into the CMAP force field file. The remaining 5 columns are the atom IDs of the atoms in the two 4-atom dihedrals that overlap to create the CMAP interaction. Note that the "crossterm" and "CMAP" keywords for the header and body sections match those specified in the read_data command following the data file name; see the *read_data* page for more details.

A data file containing CMAP 5-body interactions can be generated from a PDB file using the charmm2lammps.pl script in the tools/ch2lmp directory of the LAMMPS distribution. The script must be invoked with the optional "-cmap" flag to do this; see the tools/ch2lmp/README file for more information. The same conversion script also creates the file of CMAP coefficient data which is read by this command.

The potential energy associated with CMAP interactions can be output as described below. It can also be included in the total potential energy of the system, as output by the *thermo_style* command, if the *fix_modify energy* command is used, as in the example above. See the note below about how to include the CMAP energy when performing an *energy minimization*.

---

### 2.42.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the list of CMAP cross-terms to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The *fix_modify energy* option is supported by this fix to add the potential energy of the CMAP interactions to both the global potential energy and peratom potential energies of the system as part of *thermodynamic output* or output by the *compute pe/atom* command. The default setting for this fix is *fix_modify energy yes*.

The *fix_modify virial* option is supported by this fix to add the contribution due to the CMAP interactions to both the global pressure and per-atom stress of the system via the *compute pressure* and *compute stress/atom* commands. The former can be accessed by *thermodynamic output*. The default setting for this fix is *fix_modify virial yes*.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the potential energy discussed above. The scalar value calculated by this fix is "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The forces due to this fix are imposed during an energy minimization, invoked by the *minimize* command.

The *fix_modify respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is adding its forces. Default is the outermost level.

> **Note**
>
> For energy minimization, if you want the potential energy associated with the CMAP terms forces to be included in the total potential energy of the system (the quantity being minimized), you MUST not disable the *fix_modify energy* option for this fix.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.42.5 Restrictions

To function as expected this fix command must be issued *before* a *read_data* command but *after* a *read_restart* command.

This fix can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* page for more info.

### 2.42.6 Related commands

*fix_modify*, *read_data*

### 2.42.7 Default

none

---

**(Buck)** Buck, Bouguet-Bonnet, Pastor, MacKerell Jr., Biophys J, 90, L36 (2006).

**(Brooks)** Brooks, Brooks, MacKerell Jr., J Comput Chem, 30, 1545 (2009).

## 2.43 fix colvars command

Accelerator Variants: *colvars/kk*

### 2.43.1 Syntax

```
fix ID group-ID colvars *configfile* keyword value ...
```

- *ID*, *group-ID* are documented in *fix* command
- "colvars" = style name of this fix command
- *configfile* = configuration file for Colvars (use "*none*" to provide it inline)
- keyword = *output* or *input* or *unwrap* or *tstat* or *seed*

```
output value = state filename/prefix for Colvars (default: "out")
input value = input state filename/prefix for Colvars (optional, default: "NULL")
unwrap value = "yes" or "no" (default: "yes")
tstat value = fix ID of thermostat applied to relevant atoms (default: "NULL")
seed value = seed for random number generator (default: 1966)
```

### 2.43.2 Examples

```
# Create the fix using a config file, set prefix for its output files
fix Colvars all colvars colvars.inp output ${JOB}

# Communicate the LAMMPS target temperature to the Colvars module
fix_modify Colvars tstat NPT

# Add a new restraint specific to this LAMMPS run
fix_modify Colvars config """
harmonic {
    name restraint
    colvars distance1 distance2
    centers ${ref1} ${ref2}
    forceConstant ${kappa}
}"""
```

### 2.43.3 Description

This fix interfaces LAMMPS to the collective variables Colvars library, which allows to accelerate sampling of rare events and the computation of free energy surfaces and potentials of mean force (PMFs) for any set of collective variables using a variety of sampling methods (e.g. umbrella-sampling, metadynamics, ABF...).

This documentation describes only the "fix colvars" command itself in a LAMMPS script. The Colvars library is fully documented in the included PDF manual or in the webpage https://colvars.github.io/colvars-refman-lammps/colvars-refman-lammps.html.

The Colvars library is developed at https://github.com/Colvars/colvars. A detailed discussion of its implementation is in *(Fiorin)*; additional citations for specific features are printed at runtime if these features are used.

There are example scripts on the Colvars website as well as in the `examples/PACKAGES/colvars` directory in the LAMMPS source tree.

---

The only required argument to the fix is the name of the Colvars configuration file. The contents of this file are independent from the MD engine in which the Colvars library has been integrated, save for the units that are specific to each engine. In LAMMPS, the units used by Colvars are consistent with those specified by the *units* command.

Added in version Colvars_2023-06-04: The special value "*none*" (lowercase) initializes an empty Colvars module, which allows loading configuration dynamically using *fix_modify* (see below).

The *group-ID* entry is ignored. "fix colvars" will always apply to the entire system, but specific atoms will be selected based on selection keywords in the Colvars configuration file or files. There is no need to define multiple "fix colvars" instances and it is not allowed.

The "output" keyword allows to specify the prefix of output files generated by Colvars, for example "*output*.colvars.traj" or "output.pmf". Supplying an empty string suppresses any file output from Colvars to file, except for data saved into the LAMMPS *binary restart* files.

The "input" keyword allows to specify an optional state file that contains the restart information needed to continue a previous simulation state. However, because "fix colvars" records its state in LAMMPS *binary restart* files, this is usually not needed when using the *read_restart* command.

The *unwrap* keyword controls whether wrapped or unwrapped coordinates are passed to the Colvars library for calculation of the collective variables and the resulting forces. The default is *yes*, i.e. the image flags are used to reconstruct the absolute atom positions. Setting this to *no* will use the current local coordinates that are wrapped back into the simulation cell at each re-neighboring step instead. For information about when and how this affects results, please see https://colvars.github.io/colvars-refman-lammps/colvars-refman-lammps.html#sec:colvar_atom_groups_wrapping.

The *tstat* keyword can be either "NULL" or the label of a thermostatting fix that thermostats all atoms in the fix colvars group. This will be used to provide the colvars module with the current thermostat target temperature.

The *seed* keyword contains the seed for the random number generator that will be used in the colvars module.

---

**Note**

Fix colvars/kk is not really ported to KOKKOS, since the colvars library has not been ported to KOKKOS. It merely has some optimizations to reduce the data transfers between host and device for KOKKOS with GPUs.

---

### 2.43.4 Restarting

This fix writes the current state of the Colvars module into *binary restart files*. This is in addition to the text-mode ".colvars.state" state file that is written by the Colvars module itself. The information contained in both files is identical, and the binary LAMMPS restart file is also used by fix colvars when *read_restart* is called in a LAMMPS script. In that case, there is typically no need to specify the *input* keyword.

As long as LAMMPS binary restarts will be used to continue a simulation, it is safe to delete the ".colvars.state" files to save space. However, when a LAMMPS simulation is restarted using *read_data*, the Colvars state file must be available and loaded via the "input" keyword or via a "fix_modify Colvars load" command (see below).

When restarting, the fix and the Colvars module should be created and configured using the original configuration file(s).

### 2.43.5 Output

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the energy due to all external potentials defined in the Colvars configuration. The scalar value calculated by this fix is "extensive".

Aside from the state information in a ".colvars.state" file, other output files are produced by Colvars depending on the type of simulation. For this reason, the "output" keyword is required for fix colvars.

Added in version 30Mar2026.

This fix provides access to a global array of the current values of each collective variable computed in the Colvars module. The number of rows of the global array is equal to the number of collective variables and the columns are:

- *f_ID[I][1]* for a scalar-valued collective variable
- *f_ID[I][1], ..., f_ID[I][N]* for a vector-valued collective variable

### 2.43.6 Controlling Colvars via *fix_modify*

Added in version Colvars_2023-06-04.

The *fix_modify* command may be used on "fix colvars" in either one of two ways:

(1) Provide updated values for the fix parameters, such as *output*, *input*, *unwrap*, *tstat* and *seed*. Additionally, the *fix_modify energy* keyword is supported by this fix to add the energy change from the biasing force added by Colvars to the global potential energy of the system as part of *thermodynamic output* (the default is *fix_modify energy no*). For example, in a multi-step LAMMPS script involving multiple thermostats (e.g. fix nvt followed by fix npt), Colvars can read a new thermostat's target temperature like this:

```
fix NVT all nvt ...
fix Colvars all colvars <configfile> output equil1 tstat NVT
run <NUMSTEPS>
unfix nvt
fix NPT all n ...
fix_modify Colvars tstat NPT
fix_modify Colvars output equil2
```

(2) Call one of the scripting functions provided by the Colvars module itself (a full list is available in the Colvars doc). The arguments to these functions are provided as strings and passed to Colvars.

LAMMPS variables referenced by their string representation "${variable}" will be expanded immediately. Note also that this variable expansion *will also happen within quotes*, similar to what the *mdi* command provides. This feature makes it possible to use the values of certain LAMMPS variables in Colvars configuration strings. For example, to synchronize the LAMMPS and Colvars dump frequencies:

```
variable freq index 10000
dump myDump all atom/zstd ${freq} dump.atom.zstd
fix_modify Colvars config "colvarsTrajFrequency ${freq}"
```

**Note**

Although it is possible to use *fix_modify* at any time, its results will only reflect the state of the Colvars module at the end of the most recent "run" or "minimize" command. Any new configuration added via "fix_modify Colvars configfile" or "fix_modify Colvars config" will only be loaded when the simulation resumes. Configuration files or strings will be parsed in the same sequence as they were provided in the LAMMPS script.

### 2.43.7 Restrictions

This fix is provided by the COLVARS package and is only available if LAMMPS was built with that package (default in most builds). Some of the features also require code available from the LEPTON package. See the *Build package* page for more info.

There can only be one Colvars instance defined at a time. Since the interface communicates only the minimum required amount of information, and the Colvars module itself can handle an arbitrary number of collective variables, this is not a limitation of functionality.

### 2.43.8 Related commands

*fix smd*, *fix spring*, *fix plumed*

---

**(Fiorin)** Fiorin, Klein, Henin, Mol. Phys. 111, 3345 (2013) https://doi.org/10.1080/00268976.2013.813594

https://colvars.github.io/colvars-refman-lammps/colvars-refman-lammps.html

## 2.44 fix controller command

### 2.44.1 Syntax

```
fix ID group-ID controller Nevery alpha Kp Ki Kd pvar setpoint cvar
```

- ID, group-ID are documented in *fix* command

- controller = style name of this fix command
- Nevery = invoke controller every this many timesteps
- alpha = coupling constant for PID equation (see units discussion below)
- Kp = proportional gain in PID equation (unitless)
- Ki = integral gain in PID equation (unitless)
- Kd = derivative gain in PID equation (unitless)
- pvar = process variable of form c_ID, c_ID[I], f_ID, f_ID[I], or v_name

```
c_ID = global scalar calculated by a compute with ID
c_ID[I] = Ith component of global vector calculated by a compute with ID
f_ID = global scalar calculated by a fix with ID
f_ID[I] = Ith component of global vector calculated by a fix with ID
v_name = value calculated by an equal-style variable with name
```

- setpoint = desired value of process variable (same units as process variable)
- cvar = name of control variable

### 2.44.2 Examples

```
fix 1 all controller 100 1.0 0.5 0.0 0.0 c_thermo_temp 1.5 tcontrol
fix 1 all controller 100 0.2 0.5 0 100.0 v_pxxwall 1.01325 xwall
fix 1 all controller 10000 0.2 0.5 0 2000 v_avpe -3.785 tcontrol
```

### 2.44.3 Description

This fix enables control of a LAMMPS simulation using a control loop feedback mechanism known as a proportional-integral-derivative (PID) controller. The basic idea is to define a "process variable" which is a quantity that can be monitored during a running simulation. A desired target value is chosen for the process variable. A "control variable" is also defined which is an adjustable attribute of the running simulation, which the process variable will respond to. The PID controller continuously adjusts the control variable based on the difference between the process variable and the target.

Here are examples of ways in which this fix can be used. The examples/pid directory contains a script that implements the simple thermostat.

| Goal | process variable | control variable |
|---|---|---|
| Simple thermostat | instantaneous T | thermostat target T |
| Find melting temperature | average PE per atom | thermostat target T |
| Control pressure in non-periodic system | force on wall | position of wall |

**Note**

For this fix to work, the control variable must actually induce a change in a running LAMMPS simulation. Typically this will only occur if there is some other command (e.g. a thermostat fix) which uses the control variable as an input parameter. This could be done directly or indirectly, e.g. the other command uses a variable as input whose formula uses the control variable. The other command should alter its behavior dynamically as the variable changes.

**Note**

If there is a command you think could be used in this fashion, but does not currently allow a variable as an input parameter, please notify the LAMMPS developers. It is often not difficult to enable a command to use a variable as an input parameter.

The group specified with this command is ignored. However, note that the process variable may be defined by calculations performed by computes and fixes which store their own "group" definitions.

The PID controller is invoked once each *Nevery* timesteps.

The PID controller is implemented as a discretized version of the following dynamic equation:

$$\frac{dc}{dt} = -\alpha(K_p e + K_i \int_0^t e\,dt + K_d \frac{de}{dt})$$

where $c$ is the continuous time analog of the control variable, $e$ =*pvar-setpoint* is the error in the process variable, and $\alpha$, $K_p$, $K_i$ , and $K_d$ are constants set by the corresponding keywords described above. The discretized version of this equation is:

$$c_n = c_{n-1} - \alpha \left( K_p \tau e_n + K_i \tau^2 \sum_{i=1}^{n} e_i + K_d (e_n - e_{n-1}) \right)$$

where $\tau$ = `Nevery·timestep` is the time interval between updates, and the subscripted variables indicate the values of $c$ and $e$ at successive updates.

From the first equation, it is clear that if the three gain values $K_p$, $K_i$, $K_d$ are dimensionless constants, then $\alpha$ must have units of [unit *cvar*]/[unit *pvar*]/[unit time] e.g. [ eV/K/ps ]. The advantage of this unit scheme is that the value of the constants should be invariant under a change of either the MD timestep size or the value of *Nevery*. Similarly, if the LAMMPS *unit style* is changed, it should only be necessary to change the value of $\alpha$ to reflect this, while leaving $K_p$, $K_i$, and $K_d$ unaltered.

When choosing the values of the four constants, it is best to first pick a value and sign for $\alpha$ that is consistent with the magnitudes and signs of *pvar* and *cvar*. The magnitude of $K_p$ should then be tested over a large positive range keeping $K_i = K_d = 0$. A good value for $K_p$ will produce a fast response in *pvar*, without overshooting the *setpoint*. For many applications, proportional feedback is sufficient, and so $K_i = K_d = 0$ can be used. In cases where there is a substantial lag time in the response of *pvar* to a change in *cvar*, this can be counteracted by increasing $K_d$. In situations where *pvar* plateaus without reaching *setpoint*, this can be counteracted by increasing $K_i$. In the language of Charles Dickens, $K_p$ represents the error of the present, $K_i$ the error of the past, and $K_d$ the error yet to come.

Because this fix updates *cvar*, but does not initialize its value, the initial value $c_0$ is that assigned by the user in the input script via the *internal-style variable* command. This value is used (by every other LAMMPS command that uses the variable) until this fix performs its first update of *cvar* after *Nevery* timesteps. On the first update, the value of the derivative term is set to zero, because the value of $e_{n-1}$ is not yet defined.

---

The process variable *pvar* can be specified as the output of a *compute* or *fix* or the evaluation of a *variable*. In each case, the compute, fix, or variable must produce a global quantity, not a per-atom or local quantity.

If *pvar* begins with "c_", a compute ID must follow which has been previously defined in the input script and which generates a global scalar or vector. See the individual *compute* doc page for details. If no bracketed integer is appended, the scalar calculated by the compute is used. If a bracketed integer is appended, the Ith value of the vector calculated by the compute is used. Users can also write code for their own compute styles and *add them to LAMMPS*.

If *pvar* begins with "f_", a fix ID must follow which has been previously defined in the input script and which generates a global scalar or vector. See the individual *fix* page for details. Note that some fixes only produce their values on certain timesteps, which must be compatible with when fix controller references the values, or else an error results.

If no bracketed integer is appended, the scalar calculated by the fix is used. If a bracketed integer is appended, the Ith value of the vector calculated by the fix is used. Users can also write code for their own fix style and *add them to LAMMPS*.

If *pvar* begins with "v_", a variable name must follow which has been previously defined in the input script. Only equal-style variables can be referenced. See the *variable* command for details. Note that variables of style *equal* define a formula which can reference individual atom properties or thermodynamic keywords, or they can invoke other computes, fixes, or variables when they are evaluated, so this is a very general means of specifying the process variable.

The target value *setpoint* for the process variable must be a numeric value, in whatever units *pvar* is defined for.

The control variable *cvar* must be the name of an *internal-style variable* previously defined in the input script. Note that it is not specified with a "v_" prefix, just the name of the variable. It must be an internal-style variable, because this fix updates its value directly. Note that other commands can use an equal-style versus internal-style variable interchangeably.

---

### 2.44.4 Restart, fix_modify, output, run start/stop, minimize info

Currently, no information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix produces a global vector with 3 values which can be accessed by various *output commands*. The values can be accessed on any timestep, though they are only updated on timesteps that are a multiple of *Nevery*.

The three values are the most recent updates made to the control variable by each of the 3 terms in the PID equation above. The first value is the proportional term, the second is the integral term, the third is the derivative term.

The units of the vector values will be whatever units the control variable is in. The vector values calculated by this fix are "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.44.5 Restrictions

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.44.6 Related commands

*fix adapt*

### 2.44.7 Default

none

## 2.45 fix damping/cundall command

### 2.45.1 Syntax

```
fix ID group-ID damping/cundall gamma_l gamma_a keyword values ...
```

- ID, group-ID are documented in *fix* command
- damping/cundall = style name of this fix command
- gamma_l = linear damping coefficient (dimensionless)

- gamma_a = angular damping coefficient (dimensionless)
- zero or more keyword/value pairs may be appended

```
keyword = scale
  scale values = type ratio or v_name
    type = atom type (1-N)
    ratio = factor to scale the damping coefficients by
    v_name = reference to atom style variable name
```

### 2.45.2 Examples

```
fix 1 all damping/cundall 0.8 0.8
fix 1 all damping/cundall 0.8 0.5 scale 3 2.5
fix a all damping/cundall 0.8 0.5 scale v_radscale
```

### 2.45.3 Description

Add damping force and torque to finite-size spherical particles in the group following the model of *Cundall, 1987*, as implemented in other granular physics code (e.g., *Yade-DEM*, *PFC*).

The damping is constructed to always have negative mechanical power with respect to the current velocity/angular velocity to ensure dissipation of kinetic energy. If used without additional thermostatting (to add kinetic energy to the system), it has the effect of slowly (or rapidly) freezing the system; hence it can also be used as a simple energy minimization technique.

The magnitude of the damping force/torque $F_d/T_d$ is a fraction $\gamma \in [0;1]$ of the current force/torque $F/T$ on the particle. Damping is applied component-by-component in each direction $k \in \{x,y,z\}$:

$$F_{dk} = -\gamma_l \, F_k \, \mathrm{sign}(F_k v_k)$$

$$T_{dk} = -\gamma_a \, T_k \, \mathrm{sign}(T_k \omega_k)$$

The larger the coefficients, the faster the kinetic energy is reduced.

If the optional keyword *scale* is used, $\gamma_l$ and $\gamma_a$ can be scaled up or down by the specified factor for atoms. This factor can be set for different atom types and thus the *scale* keyword used multiple times followed by the atom type and the associated scale factor. Alternately the scaling factor can be computed for each atom (e.g. based on its radius) by using an *atom-style variable*.

**Note**

The damping force/torque is computed based on the force/torque at the moment this fix is invoked. Any force/torque added after this fix, e.g., by *fix addforce* or *fix addtorque/group* will not be damped. When performing simulations with gravity, invoking *fix gravity* after this fix will maintain the specified gravitational acceleration.

**Note**

This scheme is dependent on the coordinates system and does not correspond to realistic physical processes. It is constructed for numerical convenience and efficacy.

This non-viscous damping presents the following advantages:

1. damping is independent of velocity, equally damping regions with distinct natural frequencies,

2. damping affects acceleration and vanishes for steady uniform motion of the particles,
3. damping parameter $\gamma$ is dimensionless and does not require scaling.

---

### 2.45.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The *fix_modify* *respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is modifying forces/torques. Default is the outermost level.

The forces/torques due to this fix are imposed during an energy minimization, invoked by the *minimize* command. This fix should only be used with damped dynamics minimizers that allow for non-conservative forces. See the *min_style* command for details.

### 2.45.5 Restrictions

This fix is part of the GRANULAR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix requires that atoms store torque and a radius as defined by the *atom_style sphere* command.

### 2.45.6 Related commands

*fix viscous*, *fix viscous/sphere*

### 2.45.7 Default

none

### 2.45.8 References

**(Cundall, 1987)** Cundall, P. A. Distinct Element Models of Rock and Soil Structure, in Analytical and Computational Methods in Engineering Rock Mechanics, Ch. 4, pp. 129-163. E. T. Brown, ed. London: Allen & Unwin., 1987.

**(PFC)** PFC Particle Flow Code 6.0 Documentation. Itasca Consulting Group.

**(Yade-DEM)** V. Smilauer et al. (2021), Yade Documentation 3rd ed. The Yade Project. DOI:10.5281/zenodo.5705394 (https://yade-dem.org/doc/)

## 2.46 fix deform command

Accelerator Variants: *deform/kk*

### 2.46.1 Syntax

```
fix ID group-ID deform N parameter style args ... keyword value ...
```

- ID, group-ID are documented in *fix* command
- N = perform box deformation every this many timesteps
- one or more parameter/args sequences may be appended

```
parameter = x or y or z or xy or xz or yz
  x, y, z args = style value(s)
    style = final or delta or scale or vel or erate or trate or volume or wiggle or␣
↪variable or pressure or pressure/mean
      final values = lo hi
        lo hi = box boundaries at end of run (distance units)
      delta values = dlo dhi
        dlo dhi = change in box boundaries at end of run (distance units)
      scale values = factor
        factor = multiplicative factor for change in box length at end of run
      vel value = V
        V = change box length at this velocity (distance/time units),
            effectively an engineering strain rate
      erate value = R
        R = engineering strain rate (1/time units)
      trate value = R
        R = true strain rate (1/time units)
      volume value = none = adjust this dim to preserve volume of system
      wiggle values = A Tp
        A = amplitude of oscillation (distance units)
        Tp = period of oscillation (time units)
      variable values = v_name1 v_name2
        v_name1 = variable with name1 for box length change as function of time
        v_name2 = variable with name2 for change rate as function of time

  xy, xz, yz args = style value
    style = final or delta or vel or erate or erate/rescale or trate or wiggle or␣
↪variable
      final value = tilt
        tilt = tilt factor at end of run (distance units)
      delta value = dtilt
        dtilt = change in tilt factor at end of run (distance units)
      vel value = V
        V = change tilt factor at this velocity (distance/time units),
            effectively an engineering shear strain rate
      erate value = R
        R = engineering shear strain rate (1/time units)
      erate/rescale value = R
        R = engineering shear strain rate (1/time units)
      trate value = R
        R = true shear strain rate (1/time units)
      wiggle values = A Tp
        A = amplitude of oscillation (distance units)
        Tp = period of oscillation (time units)
      variable values = v_name1 v_name2
        v_name1 = variable with name1 for tilt change as function of time
        v_name2 = variable with name2 for change rate as function of time
```

- zero or more keyword/value pairs may be appended
- keyword = *remap* or *flip* or *units* or *couple* or *vol/balance/p* or *max/rate* or *normalize/pressure*

```
remap value = x or v or none
  x = remap coords of atoms in group into deforming box
  v = remap velocities of atoms in group when they cross periodic boundaries
```

```
    none = no remapping of x or v
  flip value = yes or no
    allow or disallow box flips when it becomes highly skewed
  units value = lattice or box
    lattice = distances are defined in lattice units
    box = distances are defined in simulation box units
```

### 2.46.2 Examples

```
fix 1 all deform 1 x final 0.0 9.0 z final 0.0 5.0 units box
fix 1 all deform 1 x trate 0.1 y volume z volume
fix 1 all deform 1 xy erate 0.001 remap v
fix 1 all deform 10 y delta -0.5 0.5 xz vel 1.0
```

### 2.46.3 Description

Change the volume and/or shape of the simulation box during a dynamics run. Orthogonal simulation boxes have 3 adjustable parameters (x,y,z). Triclinic (non-orthogonal) simulation boxes have 6 adjustable parameters (x,y,z,xy,xz,yz). Any or all of them can be adjusted independently and simultaneously.

The *fix deform/pressure* command extends this command with additional keywords and arguments. The rest of this page explains the options common to both commands. The *fix deform/pressure* page explains the options available ONLY with the fix deform/pressure command. Note that a simulation can define only a single deformation command: fix deform or fix deform/pressure.

Both these fixes can be used to perform non-equilibrium MD (NEMD) simulations of a continuously strained system. See the *fix nvt/sllod* and *compute temp/deform* commands for more details. Note that simulation of a continuously extended system (extensional flow) can be modeled using the *UEF package* and its *fix commands*.

**Inconsistent trajectories due to image flags**

When running long simulations while shearing the box or using a high shearing rate, it is possible that the image flags used for storing unwrapped atom positions will "wrap around". When LAMMPS is compiled with the default settings, case image flags are limited to a range of $-512 \leq i \leq 511$, which will overflow when atoms starting at zero image flag value have passed through a periodic box dimension more than 512 times.

Changing the *size of LAMMPS integer types* to the "bigbig" setting can make this overflow much less likely, since it increases the image flag value range to $-1,048,576 \leq i \leq 1\,048\,575$

---

For N > 0, the box is updated at the end of the time step every N steps. For N = 0, the box is updated immediately after position updates on every time step. The latter is required for accurate NEMD simulations with *fix nvt/sllod*, but is not compatible with the *variable* style. Note, if using N = 0 with *run_style* "respa" the box will be updated in the inner loop.

For the *x*, *y*, *z* parameters, the associated dimension cannot be shrink-wrapped. For the *xy*, *yz*, *xz* parameters, the associated second dimension cannot be shrink-wrapped. Dimensions not varied by this command can be periodic or non-periodic. Dimensions corresponding to unspecified parameters can also be controlled by a *fix npt* or *fix nph* command.

The size and shape of the simulation box at the beginning of the simulation run were either specified by the *create_box* or *read_data* or *read_restart* command used to setup the simulation initially if it is the first run, or they are the values from the end of the previous run. The *create_box*, *read data*, and *read_restart* commands specify whether the simulation box is orthogonal or non-orthogonal (triclinic) and explain the meaning of the xy,xz,yz tilt factors. If fix deform changes the xy,xz,yz tilt factors, then the simulation box must be triclinic, even if its initial tilt factors are 0.0.

As described below, the desired simulation box size and shape at the end of the run are determined by the parameters of the fix deform command. Every Nth timestep during the run, the simulation box is expanded, contracted, or tilted to ramped values between the initial and final values.

---

For the *x*, *y*, and *z* parameters, this is the meaning of their styles and values.

The *final*, *delta*, *scale*, *vel*, and *erate* styles all change the specified dimension of the box via "constant displacement" which is effectively a "constant engineering strain rate". This means the box dimension changes linearly with time from its initial to final value.

For style *final*, the final lo and hi box boundaries of a dimension are specified. The values can be in lattice or box distance units. See the discussion of the units keyword below.

For style *delta*, plus or minus changes in the lo/hi box boundaries of a dimension are specified. The values can be in lattice or box distance units. See the discussion of the units keyword below.

For style *scale*, a multiplicative factor to apply to the box length of a dimension is specified. For example, if the initial box length is 10, and the factor is 1.1, then the final box length will be 11. A factor less than 1.0 means compression.

For style *vel*, a velocity at which the box length changes is specified in units of distance/time. This is effectively a "constant engineering strain rate", where rate = V/L0 and L0 is the initial box length. The distance can be in lattice or box distance units. See the discussion of the units keyword below. For example, if the initial box length is 100 Angstroms, and V is 10 Angstroms/ps, then after 10 ps, the box length will have doubled. After 20 ps, it will have tripled.

The *erate* style changes a dimension of the box at a "constant engineering strain rate". The units of the specified strain rate are 1/time. See the *units* command for the time units associated with different choices of simulation units, e.g. picoseconds for "metal" units). Tensile strain is unitless and is defined as delta/L0, where L0 is the original box length and delta is the change relative to the original length. The box length L as a function of time will change as

```
L(t) = L0 (1 + erate*dt)
```

where dt is the elapsed time (in time units). Thus if *erate* R is specified as 0.1 and time units are picoseconds, this means the box length will increase by 10% of its original length every picosecond. I.e. strain after 1 ps = 0.1, strain after 2 ps = 0.2, etc. R = -0.01 means the box length will shrink by 1% of its original length every picosecond. Note that for an "engineering" rate the change is based on the original box length, so running with R = 1 for 10 picoseconds expands the box length by a factor of 11 (strain of 10), which is different that what the *trate* style would induce.

The *trate* style changes a dimension of the box at a "constant true strain rate". Note that this is not an "engineering strain rate", as the other styles are. Rather, for a "true" rate, the rate of change is constant, which means the box dimension changes non-linearly with time from its initial to final value. The units of the specified strain rate are 1/time. See the *units* command for the time units associated with different choices of simulation units, e.g. picoseconds for "metal" units). Tensile strain is unitless and is defined as delta/L0, where L0 is the original box length and delta is the change relative to the original length.

The box length L as a function of time will change as

```
L(t) = L0 exp(trate*dt)
```

where dt is the elapsed time (in time units). Thus if *trate* R is specified as ln(1.1) and time units are picoseconds, this means the box length will increase by 10% of its current (not original) length every picosecond. I.e. strain after 1 ps = 0.1, strain after 2 ps = 0.21, etc. R = ln(2) or ln(3) means the box length will double or triple every picosecond. R = ln(0.99) means the box length will shrink by 1% of its current length every picosecond. Note that for a "true" rate the change is continuous and based on the current length, so running with R = ln(2) for 10 picoseconds does not expand the box length by a factor of 11 as it would with *erate*, but by a factor of 1024 since the box length will double every picosecond.

Note that to change the volume (or cross-sectional area) of the simulation box at a constant rate, you can change multiple dimensions via *erate* or *trate*. E.g. to double the box volume in a picosecond picosecond, you could set "x erate M",

"y erate M", "z erate M", with M = pow(2,1/3) - 1 = 0.26, since if each box dimension grows by 26%, the box volume doubles. Or you could set "x trate M", "y trate M", "z trate M", with M = ln(1.26) = 0.231, and the box volume would double every picosecond.

The *volume* style changes the specified dimension in such a way that the box volume remains constant while other box dimensions are changed explicitly via the styles discussed above. For example, "x scale 1.1 y scale 1.1 z volume" will shrink the z box length as the x,y box lengths increase, to keep the volume constant (product of x,y,z lengths). If "x scale 1.1 z volume" is specified and parameter *y* is unspecified, then the z box length will shrink as x increases to keep the product of x,z lengths constant. If "x scale 1.1 y volume z volume" is specified, then both the y,z box lengths will shrink as x increases to keep the volume constant (product of x,y,z lengths). In this case, the y,z box lengths shrink so as to keep their relative aspect ratio constant.

For solids or liquids, note that when one dimension of the box is expanded via fix deform (i.e. tensile strain), it may be physically undesirable to hold the other 2 box lengths constant (unspecified by fix deform) since that implies a density change. Using the *volume* style for those 2 dimensions to keep the box volume constant may make more physical sense, but may also not be correct for materials and potentials whose Poisson ratio is not 0.5. An alternative is to use *fix npt aniso* with zero applied pressure on those 2 dimensions, so that they respond to the tensile strain dynamically.

The *wiggle* style oscillates the specified box length dimension sinusoidally with the specified amplitude and period. I.e. the box length L as a function of time is given by

```
L(t) = L0 + A sin(2*pi t/Tp)
```

where L0 is its initial length. If the amplitude A is a positive number the box initially expands, then contracts, etc. If A is negative then the box initially contracts, then expands, etc. The amplitude can be in lattice or box distance units. See the discussion of the units keyword below.

The *variable* style changes the specified box length dimension by evaluating a variable, which presumably is a function of time. The variable with *name1* must be an *equal-style variable* and should calculate a change in box length in units of distance. Note that this distance is in box units, not lattice units; see the discussion of the *units* keyword below. The formula associated with variable *name1* can reference the current timestep. Note that it should return the "change" in box length, not the absolute box length. This means it should evaluate to 0.0 when invoked on the initial timestep of the run following the definition of fix deform. It should evaluate to a value > 0.0 to dilate the box at future times, or a value < 0.0 to compress the box. The exception would be if the run command uses the *pre no* option.

The variable *name2* must also be an *equal-style variable* and should calculate the rate of box length change, in units of distance/time, i.e. the time-derivative of the *name1* variable. This quantity is used internally by LAMMPS to reset atom velocities when they cross periodic boundaries. It is computed internally for the other styles, but you must provide it when using an arbitrary variable.

Here is an example of using the *variable* style to perform the same box deformation as the *wiggle* style formula listed above, where we assume that the current timestep = 0.

```
variable A equal 5.0
variable Tp equal 10.0
variable displace equal "v_A * sin(2*PI * step*dt/v_Tp)"
variable rate equal "2*PI*v_A/v_Tp * cos(2*PI * step*dt/v_Tp)"
fix 2 all deform 1 x variable v_displace v_rate remap v
```

For the *scale*, *vel*, *erate*, *trate*, *volume*, *wiggle*, and *variable* styles, the box length is expanded or compressed around its mid point.

---

For the *xy*, *xz*, and *yz* parameters, this is the meaning of their styles and values. Note that changing the tilt factors of a triclinic box does not change its volume.

The *final*, *delta*, *vel*, and *erate* styles all change the shear strain at a "constant engineering shear strain rate". This means the tilt factor changes linearly with time from its initial to final value.

For style *final*, the final tilt factor is specified. The value can be in lattice or box distance units. See the discussion of the units keyword below.

For style *delta*, a plus or minus change in the tilt factor is specified. The value can be in lattice or box distance units. See the discussion of the units keyword below.

For style *vel*, a velocity at which the tilt factor changes is specified in units of distance/time. This is effectively an "engineering shear strain rate", where rate = V/L0 and L0 is the initial box length perpendicular to the direction of shear. The distance can be in lattice or box distance units. See the discussion of the units keyword below. For example, if the initial tilt factor is 5 Angstroms, and the V is 10 Angstroms/ps, then after 1 ps, the tilt factor will be 15 Angstroms. After 2 ps, it will be 25 Angstroms.

The *erate* style changes a tilt factor at a "constant engineering shear strain rate". The units of the specified shear strain rate are 1/time. See the *units* command for the time units associated with different choices of simulation units, e.g. picoseconds for "metal" units). Shear strain is unitless and is defined as offset/length, where length is the box length perpendicular to the shear direction (e.g. y box length for xy deformation) and offset is the displacement distance in the shear direction (e.g. x direction for xy deformation) from the unstrained orientation.

The tilt factor T as a function of time will change as

```
T(t) = T0 + L0*erate*dt
```

where T0 is the initial tilt factor, L0 is the original length of the box perpendicular to the shear direction (e.g. y box length for xy deformation), and dt is the elapsed time (in time units). Thus if *erate* R is specified as 0.1 and time units are picoseconds, this means the shear strain will increase by 0.1 every picosecond. I.e. if the xy shear strain was initially 0.0, then strain after 1 ps = 0.1, strain after 2 ps = 0.2, etc. Thus the tilt factor would be 0.0 at time 0, 0.1*ybox at 1 ps, 0.2*ybox at 2 ps, etc, where ybox is the original y box length. R = 1 or 2 means the tilt factor will increase by 1 or 2 every picosecond. R = -0.01 means a decrease in shear strain by 0.01 every picosecond.

Changed in version 11Feb2026.

The *erate/rescale* style operates similarly to the *erate* style with a specified strain rate in units of 1/time. The difference is that the change in the tilt factor will depend on the current length of the box perpendicular to the shear direction, *L*, instead of the original length, *L0*. It will also account for the effects of elongation and shearing in other directions, with evolution as follows. The xy and yz tilt factors, $T_{\alpha\beta}$, will evolve according to

$$\frac{dT_{\alpha\beta}(t)}{dt} = L_\beta \cdot \text{erate} + \frac{1}{L_\alpha} \cdot \frac{dL_\alpha(t)}{dt} \cdot T_{\alpha\beta}(t)$$

where $\alpha$ and $\beta$ denote the two directions (e.g. x and y, or y and z). The xz tilt factor has an additional term which accounts for xy shear with yz tilt, and evolves as

$$\frac{dT_{xz}(t)}{dt} = L_z \cdot \text{erate} + \frac{1}{L_x} \cdot \frac{dL_x(t)}{dt} \cdot T_{xz}(t) + \frac{1}{L_y} \cdot \frac{dT_{xy}(t)}{dt} \cdot T_{yz}(t)$$

If xy is also *erate/rescale*, then $\frac{dT_{xy}(t)}{dt}$ is calculated as the first term on the RHS of the previous equation so that it is equal to $\text{erate}_{xy}$. With x/y/z = *trate* and xy/xz/yz = *erate/rescale*, this is equivalent to evolving the lattice vectors, $\mathbf{v} \in \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$, according to

$$\frac{d\mathbf{v}(t)}{dt} = \mathbf{v}(t) \cdot \nabla\mathbf{u}$$

where $\nabla\mathbf{u}$ is the flow tensor which remains constant and is given by

$$\nabla\mathbf{u} = \begin{bmatrix} \text{trate}_x & 0 & 0 \\ \text{erate}_{xy} & \text{trate}_y & 0 \\ \text{erate}_{xz} & \text{erate}_{yz} & \text{trate}_z \end{bmatrix}$$

The tilt factors are integrated in two half-steps before and after any changes to the x, y and z lengths, with the first half-step taking the order xz, yz, xy and the second half-step taking the reverse order so that the full step is reversible. This

option may be useful in scenarios where $L$ changes in time. It is required by *fix nvt/sllod* in such cases for calculating viscosity of mixed flows. Note, under mixed xy and yz shear, the xz component can be set to use the *erate/rescale* style with a value of zero in order to maintain a constant $\nabla \mathbf{u}$.

The *trate* style changes a tilt factor at a "constant true shear strain rate". Note that this is not an "engineering shear strain rate", as the other styles are. Rather, for a "true" rate, the rate of change is constant, which means the tilt factor changes non-linearly with time from its initial to final value. The units of the specified shear strain rate are 1/time. See the *units* command for the time units associated with different choices of simulation units, e.g. picoseconds for "metal" units). Shear strain is unitless and is defined as offset/length, where length is the box length perpendicular to the shear direction (e.g. y box length for xy deformation) and offset is the displacement distance in the shear direction (e.g. x direction for xy deformation) from the unstrained orientation.

The tilt factor T as a function of time will change as

```
T(t) = T0 exp(trate*dt)
```

where T0 is the initial tilt factor and dt is the elapsed time (in time units). Thus if *trate* R is specified as ln(1.1) and time units are picoseconds, this means the shear strain or tilt factor will increase by 10% every picosecond. I.e. if the xy shear strain was initially 0.1, then strain after 1 ps = 0.11, strain after 2 ps = 0.121, etc. R = ln(2) or ln(3) means the tilt factor will double or triple every picosecond. R = ln(0.99) means the tilt factor will shrink by 1% every picosecond. Note that the change is continuous, so running with R = ln(2) for 10 picoseconds does not change the tilt factor by a factor of 10, but by a factor of 1024 since it doubles every picosecond. Note that the initial tilt factor must be non-zero to use the *trate* option.

Note that shear strain is defined as the tilt factor divided by the perpendicular box length. The *erate* and *trate* styles control the tilt factor, but assume the perpendicular box length remains constant. If this is not the case (e.g. it changes due to another fix deform parameter), then this effect on the shear strain is ignored.

The *wiggle* style oscillates the specified tilt factor sinusoidally with the specified amplitude and period. I.e. the tilt factor T as a function of time is given by

```
T(t) = T0 + A sin(2*pi t/Tp)
```

where T0 is its initial value. If the amplitude A is a positive number the tilt factor initially becomes more positive, then more negative, etc. If A is negative then the tilt factor initially becomes more negative, then more positive, etc. The amplitude can be in lattice or box distance units. See the discussion of the units keyword below.

The *variable* style changes the specified tilt factor by evaluating a variable, which presumably is a function of time. The variable with *name1* must be an *equal-style variable* and should calculate a change in tilt in units of distance. Note that this distance is in box units, not lattice units; see the discussion of the *units* keyword below. The formula associated with variable *name1* can reference the current timestep. Note that it should return the "change" in tilt factor, not the absolute tilt factor. This means it should evaluate to 0.0 when invoked on the initial timestep of the run following the definition of fix deform.

The variable *name2* must also be an *equal-style variable* and should calculate the rate of tilt change, in units of distance/time, i.e. the time-derivative of the *name1* variable. This quantity is used internally by LAMMPS to reset atom velocities when they cross periodic boundaries. It is computed internally for the other styles, but you must provide it when using an arbitrary variable.

Here is an example of using the *variable* style to perform the same box deformation as the *wiggle* style formula listed above, where we assume that the current timestep = 0.

```
variable A equal 5.0
variable Tp equal 10.0
variable displace equal "v_A * sin(2*PI * step*dt/v_Tp)"
variable rate equal "2*PI*v_A/v_Tp * cos(2*PI * step*dt/v_Tp)"
fix 2 all deform 1 xy variable v_displace v_rate remap v
```

All of the tilt styles change the xy, xz, yz tilt factors during a simulation. In LAMMPS, tilt factors (xy,xz,yz) for triclinic boxes are normally bounded by half the distance of the parallel box length. See the discussion of the *flip* keyword below, to allow this bound to be exceeded, if desired.

For example, if xlo = 2 and xhi = 12, then the x box length is 10 and the xy tilt factor must be between -5 and 5. Similarly, both xz and yz must be between -(xhi-xlo)/2 and +(yhi-ylo)/2. Note that this is not a limitation, since if the maximum tilt factor is 5 (as in this example), then configurations with tilt = . . . , -15, -5, 5, 15, 25, . . . are all equivalent.

To obey this constraint and allow for large shear deformations to be applied via the *xy*, *xz*, or *yz* parameters, the following algorithm is used. If *prd* is the associated parallel box length (10 in the example above), then if the tilt factor exceeds the accepted range of -5 to 5 during the simulation, then the box is flipped to the other limit (an equivalent box) and the simulation continues. Thus for this example, if the initial xy tilt factor was 0.0 and "xy final 100.0" was specified, then during the simulation the xy tilt factor would increase from 0.0 to 5.0, the box would be flipped so that the tilt factor becomes -5.0, the tilt factor would increase from -5.0 to 5.0, the box would be flipped again, etc. The flip occurs 10 times and the final tilt factor at the end of the simulation would be 0.0. During each flip event, atoms are remapped into the new box in the appropriate manner.

The one exception to this rule is if the first dimension in the tilt factor (x for xy) is non-periodic. In that case, the limits on the tilt factor are not enforced, since flipping the box in that dimension does not change the atom positions due to non-periodicity. In this mode, if you tilt the system to extreme angles, the simulation will simply become inefficient due to the highly skewed simulation box.

---

Each time the box size or shape is changed, the *remap* keyword determines whether atom positions are remapped to the new box. If *remap* is set to *x* (the default), atoms in the fix group are remapped; otherwise they are not. Note that their velocities are not changed, just their positions are altered. If *remap* is set to *v*, then any atom in the fix group that crosses a periodic boundary will have a delta added to its velocity equal to the difference in velocities between the lo and hi boundaries. Note that this velocity difference can include tilt components, e.g. a delta in the x velocity when an atom crosses the y periodic boundary. If *remap* is set to *none*, then neither of these remappings take place.

Conceptually, setting *remap* to *x* forces the atoms to deform via an affine transformation that exactly matches the box deformation. This setting is typically appropriate for solids. Note that though the atoms are effectively "moving" with the box over time, it is not due to their having a velocity that tracks the box change, but only due to the remapping. By contrast, setting *remap* to *v* is typically appropriate for fluids, where you want the atoms to respond to the change in box size/shape on their own and acquire a velocity that matches the box change, so that their motion will naturally track the box without explicit remapping of their coordinates.

**Note**

When non-equilibrium MD (NEMD) simulations are performed using this fix, the option "remap v" should normally be used. This is because *fix nvt/sllod* with the "peculiar no" option (the default) adjusts the atom positions and velocities to induce a velocity profile that matches the changing box size/shape. Thus atom coordinates should NOT be remapped by fix deform, but velocities SHOULD be when atoms cross periodic boundaries, since that is consistent with maintaining the velocity profile already created by fix nvt/sllod. If fix nvt/sllod uses "peculiar yes" then fix deform should use "remap none". LAMMPS will warn you if the *remap* setting is not consistent with fix nvt/sllod.

**Note**

For non-equilibrium MD (NEMD) simulations using "remap v" it is usually desirable that the fluid (or flowing material, e.g. granular particles) stream with a velocity profile consistent with the deforming box. As mentioned above, using a thermostat such as *fix nvt/sllod* or *fix lavgevin* (with a bias provided by *compute temp/deform*), will typically accomplish that. If you do not use a thermostat, then there is no driving force pushing the atoms to flow in a manner consistent with the deforming box. E.g. for a shearing system the box deformation velocity may vary

from 0 at the bottom to 10 at the top of the box. But the stream velocity profile of the atoms may vary from -5 at the bottom to +5 at the top. You can monitor these effects using the *fix ave/chunk*, *compute temp/deform*, and *compute temp/profile* commands. One way to induce atoms to stream consistent with the box deformation is to give them an initial velocity profile, via the *velocity ramp* command, that matches the box deformation rate. This also typically helps the system come to equilibrium more quickly, even if a thermostat is used. In some cases, a convenient way to induce the correct streaming velocity is with the *kick* = yes option of *fix nvt/sllod*, which superimposes the initial velocity profile such that the velocity added to a particle at the corner of the box will be the same as the velocity of that corner. This ensures the flow is consistent with the box deformation and crossing of periodic boundaries is minimized, but is not compatible with all deformation styles. See *fix nvt/sllod* for details.

**Note**

If a *fix rigid* is defined for rigid bodies, and *remap* is set to *x*, then the center-of-mass coordinates of rigid bodies will be remapped to the changing simulation box. This will be done regardless of whether atoms in the rigid bodies are in the fix deform group or not. The velocity of the centers of mass are not remapped even if *remap* is set to *v*, since *fix nvt/sllod* does not currently do anything special for rigid particles. If you wish to perform a NEMD simulation of rigid particles, you can either thermostat them independently or include a background fluid and thermostat the fluid via *fix nvt/sllod*.

The *flip* keyword allows the tilt factors for a triclinic box to exceed half the distance of the parallel box length, as discussed above. If the *flip* value is set to *yes*, the bound is enforced by flipping the box when it is exceeded. If the *flip* value is set to *no*, the tilt will continue to change without flipping. Note that if you apply large deformations, this means the box shape can tilt dramatically LAMMPS will run less efficiently, due to the large volume of communication needed to acquire ghost atoms around a processor's irregular-shaped subdomain. For extreme values of tilt, LAMMPS may also lose atoms and generate an error.

The *units* keyword determines the meaning of the distance units used to define various arguments. A *box* value selects standard distance units as defined by the *units* command, e.g. Angstroms for units = real or metal. A *lattice* value means the distance units are in lattice spacings. The *lattice* command must have been previously used to define the lattice spacing. Note that the units choice also affects the *vel* style parameters since it is defined in terms of distance/time. Also note that the units keyword does not affect the *variable* style. You should use the *xlat*, *ylat*, *zlat* keywords of the *thermo_style* command if you want to include lattice spacings in a variable formula.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 2.46.4 Restart, fix_modify, output, run start/stop, minimize info

This fix will restore the initial box settings from *binary restart files*, which allows the fix to be properly continue deformation, when using the start/stop options of the *run* command. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*.

This fix can perform deformation over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.46.5 Restrictions

You cannot apply x, y, or z deformations to a dimension that is shrink-wrapped via the *boundary* command.

You cannot apply xy, yz, or xz deformations to a second dimension (y in xy) that is shrink-wrapped via the *boundary* command.

### 2.46.6 Related commands

*fix deform/pressure*, *change_box*

### 2.46.7 Default

The option defaults are remap = x, flip = yes, and units = lattice.

## 2.47 fix deform/pressure command

### 2.47.1 Syntax

```
fix ID group-ID deform/pressure N parameter style args ... keyword value ...
```

- ID, group-ID are documented in *fix* command
- deform/pressure = style name of this fix command
- N = perform box deformation every this many timesteps
- one or more parameter/args sequences may be appended

```
parameter = x or y or z or xy or xz or yz or box
  x, y, z args = style value(s)
    style = final or delta or scale or vel or erate or trate or volume or wiggle or␣
↪variable or pressure or pressure/mean
      final values = lo hi
        lo hi = box boundaries at end of run (distance units)
      delta values = dlo dhi
        dlo dhi = change in box boundaries at end of run (distance units)
      scale values = factor
        factor = multiplicative factor for change in box length at end of run
      vel value = V
        V = change box length at this velocity (distance/time units),
            effectively an engineering strain rate
      erate value = R
        R = engineering strain rate (1/time units)
      trate value = R
        R = true strain rate (1/time units)
      volume value = none = adjust this dim to preserve volume of system
      wiggle values = A Tp
        A = amplitude of oscillation (distance units)
        Tp = period of oscillation (time units)
      variable values = v_name1 v_name2
        v_name1 = variable with name1 for box length change as function of time
```

```
      v_name2 = variable with name2 for change rate as function of time
    pressure values = target gain
      target = target pressure (pressure units)
      gain = proportional gain constant (1/(time * pressure) or 1/time units)
    pressure/mean values = target gain
      target = target pressure (pressure units)
      gain = proportional gain constant (1/(time * pressure) or 1/time units)

  xy, xz, yz args = style value
    style = final or delta or vel or erate or erate/rescale or trate or wiggle or␣
↪variable or pressure
    final value = tilt
      tilt = tilt factor at end of run (distance units)
    delta value = dtilt
      dtilt = change in tilt factor at end of run (distance units)
    vel value = V
      V = change tilt factor at this velocity (distance/time units),
          effectively an engineering shear strain rate
    erate value = R
      R = engineering shear strain rate (1/time units)
    erate/rescale value = R
      R = engineering shear strain rate (1/time units)
    trate value = R
      R = true shear strain rate (1/time units)
    wiggle values = A Tp
      A = amplitude of oscillation (distance units)
      Tp = period of oscillation (time units)
    variable values = v_name1 v_name2
      v_name1 = variable with name1 for tilt change as function of time
      v_name2 = variable with name2 for change rate as function of time
    pressure values = target gain
      target = target pressure (pressure units)
      gain = proportional gain constant (1/(time * pressure) or 1/time units)

  box = style value
    style = volume or pressure
      volume value = none = isotropically adjust system to preserve volume of system
      pressure values = target gain
        target = target mean pressure (pressure units)
        gain = proportional gain constant (1/(time * pressure) or 1/time units)
```

- zero or more keyword/value pairs may be appended
- keyword = *remap* or *flip* or *units* or *couple* or *vol/balance/p* or *max/rate* or *normalize/pressure*

```
remap value = x or v or none
  x = remap coords of atoms in group into deforming box
  v = remap velocities of atoms in group when they cross periodic boundaries
  none = no remapping of x or v
flip value = yes or no
  allow or disallow box flips when it becomes highly skewed
units value = lattice or box
  lattice = distances are defined in lattice units
  box = distances are defined in simulation box units
couple value = none or xyz or xy or yz or xz
```

```
    couple pressure values of various dimensions
  vol/balance/p value = yes or no
    Modifies the behavior of the volume option to try and balance pressures
  max/rate value = rate
    rate = maximum strain rate for pressure control
  normalize/pressure value = yes or no
    Modifies pressure controls such that the deviation in pressure is normalized by␣
 ↪the target pressure
```

### 2.47.2 Examples

```
fix 1 all deform/pressure 1 x pressure 2.0 0.1 normalize/pressure yes max/rate 0.001
fix 1 all deform/pressure 1 x trate 0.1 y volume z volume vol/balance/p yes
fix 1 all deform/pressure 1 x trate 0.1 y pressure/mean 0.0 1.0 z pressure/mean 0.0 1.0
```

### 2.47.3 Description

Added in version 17Apr2024.

This fix is an extension of the *fix deform* command, which allows all of its options to be used as well as new pressure-based controls implemented by this command.

All arguments described on the *fix deform* doc page also apply to this fix unless otherwise noted below. The rest of this page explains the arguments specific to this fix only. Note that a simulation can define only a single deformation command: fix deform or fix deform/pressure.

**Inconsistent trajectories due to image flags**

When running long simulations while shearing the box or using a high shearing rate, it is possible that the image flags used for storing unwrapped atom positions will "wrap around". When LAMMPS is compiled with the default settings, case image flags are limited to a range of $-512 \leq i \leq 511$, which will overflow when atoms starting at zero image flag value have passed through a periodic box dimension more than 512 times.

Changing the *size of LAMMPS integer types* to the "bigbig" setting can make this overflow much less likely, since it increases the image flag value range to $-1,048,576 \leq i \leq 1\,048\,575$

---

For the $x$, $y$, and $z$ parameters, this is the meaning of the styles and values provided by this fix.

The *pressure* style adjusts a dimension's box length to control the corresponding component of the pressure tensor. This option attempts to maintain a specified target pressure using a linear controller where the box length $L$ evolves according to the equation

$$\frac{dL(t)}{dt} = L(t)k(P_t - P)$$

where $k$ is a proportional gain constant, $P_t$ is the target pressure, and $P$ is the current pressure along that dimension. This approach is similar to the method used to control the pressure by *fix press/berendsen*. The target pressure accepts either a constant numeric value or a LAMMPS *variable*. Notably, this variable can be a function of time or other components of the pressure tensor. By default, $k$ has units of 1/(time * pressure) although this will change if the *normalize/pressure* option is set as *discussed below*. There is no proven method to choosing an appropriate value of $k$ as it will depend on the specific details of a simulation. Testing different values is recommended.

By default, there is no limit on the resulting strain rate in any dimension. A maximum limit can be applied using the *max/rate* option. Akin to *fix npt and nph*, pressures in different dimensions can be coupled using the *couple* option.

This means the instantaneous pressure along coupled dimensions are averaged and the box strains identically along the coupled dimensions.

The *pressure/mean* style changes a dimension's box length to maintain a constant mean pressure defined as the trace of the pressure tensor. This option has identical arguments to the *pressure* style and a similar functional equation, except the current and target pressures refer to the mean trace of the pressure tensor. All options for the *pressure* style also apply to the *pressure/mean* style except for the *couple* option.

Note that while this style can be identical to coupled *pressure* styles, it is generally not the same. For instance in 2D, a coupled *pressure* style in the *x* and *y* dimensions would be equivalent to using the *pressure/mean* style with identical settings in each dimension. However, it would not be the same if settings (e.g. gain constants) were used in the *x* and *y* dimensions or if the *pressure/mean* command was only applied along one dimension.

---

For the *xy*, *xz*, and *yz* parameters, this is the meaning of the styles and values provided by this fix. Note that changing the tilt factors of a triclinic box does not change its volume.

The *pressure* style adjusts a tilt factor to control the corresponding off-diagonal component of the pressure tensor. This option attempts to maintain a specified target value using a linear controller where the tilt factor T evolves according to the equation

$$\frac{dT(t)}{dt} = L(t)k(P - P_t)$$

where $k$ is a proportional gain constant, $P_t$ is the target pressure, $P$ is the current pressure, and $L$ is the perpendicular box length. The target pressure accepts either a constant numeric value or a LAMMPS *variable*. Notably, this variable can be a function of time or other components of the pressure tensor. By default, $k$ has units of 1/(time * pressure) although this will change if the *normalize/pessure* option is set as *discussed below*. There is no proven method to choosing an appropriate value of $k$ as it will depend on the specific details of a simulation and testing different values is recommended. One can also apply a maximum limit to the magnitude of the applied strain using the *max/rate* option.

---

The *box* parameter provides an additional control over the *x*, *y*, and *z* box lengths by isotropically dilating or contracting the box to either maintain a fixed mean pressure or volume. This isotropic scaling is applied after the box is deformed by the above *x*, *y*, *z*, *xy*, *xz*, and *yz* styles, acting as a second deformation step. This parameter will change the overall strain rate in the *x*, *y*, or *z* dimensions. This parameter can only be used in combination with the *x*, *y*, or *z* commands: *vel*, *erate*, *trate*, *pressure*, or *wiggle*. This is the meaning of its styles and values.

The *volume* style isotropically scales box lengths to maintain a constant box volume in response to deformation from other parameters. This style may be useful in scenarios where one wants to apply a constant deviatoric pressure using *pressure* styles in the *x*, *y*, and *z* dimensions ( deforming the shape of the box), while maintaining a constant volume.

The *pressure* style isotropically scales box lengths in an attempt to maintain a target mean pressure (the trace of the pressure tensor) of the system. This is accomplished by isotropically scaling all box lengths $L$ by an additional factor of $k(P_t - P_m)$ where $k$ is the proportional gain constant, $P_t$ is the target pressure, and $P_m$ is the current mean pressure. This style may be useful in scenarios where one wants to apply a constant deviatoric strain rate using various strain-based styles (e.g. *trate*) along the *x*, *y*, and *z* dimensions (deforming the shape of the box), while maintaining a mean pressure.

---

The optional keywords provided by this fix are described below.

The *normalize/pressure* keyword changes how box dimensions evolve when using the *pressure* or *pressure/mean* deformation styles. If the *deform/normalize* value is set to *yes*, then the deviation from the target pressure is normalized by the absolute value of the target pressure such that the proportional gain constant scales a percentage error and has units of 1/time. If the target pressure is ever zero, this will produce an error unless the *max/rate* keyword is defined, described below, which will cap the divergence.

The *max/rate* keyword sets an upper threshold, *rate*, that limits the maximum magnitude of the instantaneous strain rate applied in any dimension. This keyword only applies to the *pressure* and *pressure/mean* options. If a pressure-controlled rate is used for both *box* and either *x*, *y*, or *z*, then this threshold will apply separately to each individual controller such that the cumulative strain rate on a box dimension may be up to twice the value of *rate*.

The *couple* keyword allows two or three of the diagonal components of the pressure tensor to be "coupled" together for the *pressure* option. The value specified with the keyword determines which are coupled. For example, *xz* means the *Pxx* and *Pzz* components of the stress tensor are coupled. *Xyz* means all 3 diagonal components are coupled. Coupling means two things: the instantaneous stress will be computed as an average of the corresponding diagonal components, and the coupled box dimensions will be changed together in lockstep, meaning coupled dimensions will be dilated or contracted by the same percentage every timestep. If a *pressure* style is defined for more than one coupled dimension, the target pressures and gain constants must be identical. Alternatively, if a *pressure* style is only defined for one of the coupled dimensions, its settings are copied to other dimensions with undefined styles. *Couple xyz* can be used for a 2d simulation; the *z* dimension is simply ignored.

The *vol/balance/p* keyword modifies the behavior of the *volume* style when applied to two of the *x*, *y*, and *z* dimensions. Instead of straining the two dimensions in lockstep, the two dimensions are allowed to separately dilate or contract in a manner to maintain a constant volume while simultaneously trying to keep the pressure along each dimension equal using a method described in *(Huang2014)*.

---

If any pressure controls are used, this fix computes a temperature and pressure each timestep. To do this, the fix creates its own computes of style "temp" and "pressure", as if these commands had been issued:

```
compute fix-ID_temp group-ID temp
compute fix-ID_press group-ID pressure fix-ID_temp
```

See the *compute temp* and *compute pressure* commands for details. Note that the IDs of the new computes are the fix-ID + underscore + "temp" or fix_ID + underscore + "press", and the group for the new computes is the same as the fix group.

Note that these are NOT the computes used by thermodynamic output (see the *thermo_style* command) with ID = *thermo_temp* and *thermo_press*. This means you can change the attributes of this fix's temperature or pressure via the *compute_modify* command or print this temperature or pressure during thermodynamic output via the *thermo_style custom* command using the appropriate compute-ID. It also means that changing attributes of *thermo_temp* or *thermo_press* will have no effect on this fix.

### 2.47.4 Restart, fix_modify, output, run start/stop, minimize info

This fix will restore the initial box settings from *binary restart files*, which allows the fix to be properly continue deformation, when using the start/stop options of the *run* command. No global or per-atom quantities are stored by this fix for access by various *output commands*.

If any pressure controls are used, the *fix_modify temp* and *press* options are supported by this fix, unlike in *fix deform*. You can use them to assign a *compute* you have defined to this fix which will be used in its temperature and pressure calculations. If you do this, note that the kinetic energy derived from the compute temperature should be consistent with the virial term computed using all atoms for the pressure. LAMMPS will warn you if you choose to compute temperature on a subset of atoms.

This fix can perform deformation over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.47.5 Restrictions

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

You cannot apply x, y, or z deformations to a dimension that is shrink-wrapped via the *boundary* command.

You cannot apply xy, yz, or xz deformations to a second dimension (y in xy) that is shrink-wrapped via the *boundary* command.

### 2.47.6 Related commands

*fix deform*, *change_box*

### 2.47.7 Default

The option defaults are normalize/pressure = no.

---

**(Huang2014)** X. Huang, "Exploring critical-state behavior using DEM", Doctoral dissertation, Imperial College. (2014). https://doi.org/10.25560/25316

## 2.48 fix deposit command

### 2.48.1 Syntax

```
fix ID group-ID deposit N type M seed keyword values ...
```

- ID, group-ID are documented in *fix* command
- deposit = style name of this fix command
- N = # of atoms or molecules to insert
- type = atom type (1-Ntypes or type label) to assign to inserted atoms (offset for molecule insertion)
- M = insert a single atom or molecule every M steps
- seed = random # seed (positive integer)
- one or more keyword/value pairs may be appended to args
- keyword = *region* or *var* or *set* or *id* or *global* or *local* or *near* or *gaussian* or *attempt* or *rate* or *vx* or *vy* or *vz* or *target* or *mol* or *molfrac* or *rigid* or *shake* or *orient* or *units*

```
region value = region-ID
  region-ID = ID of region to use as insertion volume
var value = name = variable name to evaluate for test of atom creation
set values = dim name
  dim = x or y or z
  name = name of variable to set with x, y, or z atom position
id value = max or next
  max = atom ID for new atom(s) is max ID of all current atoms plus one
  next = atom ID for new atom(s) increments by one for every deposition
global values = lo hi
  lo,hi = put new atom/molecule a distance lo-hi above all other atoms (distance␣
↪units)
local values = lo hi delta
```

```
  lo,hi = put new atom/molecule a distance lo-hi above any nearby atom beneath it␣
↪(distance units)
  delta = lateral distance within which a neighbor is considered "nearby" (distance␣
↪units)
near value = R
  R = only insert atom/molecule if further than R from existing particles (distance␣
↪units)
gaussian values = xmid ymid zmid sigma
  xmid,ymid,zmid = center of the gaussian distribution (distance units)
  sigma = width of gaussian distribution (distance units)
attempt value = Q
  Q = attempt a single insertion up to Q times
rate value = V
  V = z velocity (y in 2d) at which insertion volume moves (velocity units)
vx values = vxlo vxhi
  vxlo,vxhi = range of x velocities for inserted atom/molecule (velocity units)
vy values = vylo vyhi
  vylo,vyhi = range of y velocities for inserted atom/molecule (velocity units)
vz values = vzlo vzhi
  vzlo,vzhi = range of z velocities for inserted atom/molecule (velocity units)
target values = tx ty tz
  tx,ty,tz = location of target point (distance units)
mol value = template-ID
  template-ID = ID of molecule template specified in a separate molecule command
molfrac values = f1 f2 ... fN
  f1 to fN = relative probability of creating each of N molecules in template-ID
rigid value = fix-ID
  fix-ID = ID of fix rigid/small command
shake value = fix-ID
  fix-ID = ID of fix shake command
orient values = rx ry rz
  rx,ry,rz = vector to randomly rotate an inserted molecule around
units value = lattice or box
  lattice = the geometry is defined in lattice units
  box = the geometry is defined in simulation box units
```

### 2.48.2 Examples

```
fix 3 all deposit 1000 2 100 29494 region myblock local 1.0 1.0 1.0 units box
fix 2 newatoms deposit 10000 1 500 12345 region disk near 2.0 vz -1.0 -0.8
fix 4 sputter deposit 1000 2 500 12235 region sphere vz -1.0 -1.0 target 5.0 5.0 0.0␣
↪units lattice
fix 5 insert deposit 200 2 100 777 region disk gaussian 5.0 5.0 9.0 1.0 units box

labelmap atom 1 Au
fix 4 sputter deposit 1000 Au 500 12235 region sphere vz -1.0 -1.0 target 5.0 5.0 0.0␣
↪units lattice
```

### 2.48.3 Description

Insert a single atom or molecule into the simulation domain every M timesteps until N atoms or molecules have been inserted. This is useful for simulating deposition onto a surface. For the remainder of this doc page, a single inserted atom or molecule is referred to as a "particle".

If inserted particles are individual atoms, they are assigned the specified atom type. If they are molecules, the type of each atom in the inserted molecule is specified in the file read by the *molecule* command, and those values are added to the specified atom type. E.g. if the file specifies atom types 1,2,3, and those are the atom types you want for inserted molecules, then specify *type* = 0. If you specify *type* = 2, the in the inserted molecule will have atom types 3,4,5.

All atoms in the inserted particle are assigned to two groups: the default group "all" and the group specified in the fix deposit command (which can also be "all").

If you are computing temperature values which include inserted particles, you will want to use the *compute_modify dynamic/dof yes* option, which ensures the current number of atoms is used as a normalizing factor each time the temperature is computed.

Care must be taken that inserted particles are not too near existing atoms, using the options described below. When inserting particles above a surface in a non-periodic box (see the *boundary* command), the possibility of a particle escaping the surface and flying upward should be considered, since the particle may be lost or the box size may grow infinitely large. A *fix wall/reflect* command can be used to prevent this behavior. Note that if a shrink-wrap boundary is used, it is OK to insert the new particle outside the box, however the box will immediately be expanded to include the new particle. When simulating a sputtering experiment it is probably more realistic to ignore those atoms using the *thermo_modify* command with the *lost ignore* option and a fixed *boundary*.

The fix deposit command must use the *region* keyword to define an insertion volume. The specified region must have been previously defined with a *region* command. It must be defined with side = *in*.

**Note**

LAMMPS checks that the specified region is wholly inside the simulation box. It can do this correctly for orthonormal simulation boxes. However for *triclinic boxes*, it only tests against the larger orthonormal box that bounds the tilted simulation box. If the specified region includes volume outside the tilted box, then an insertion will likely fail, leading to a "lost atoms" error. Thus for triclinic boxes you should ensure the specified region is wholly inside the simulation box.

The locations of inserted particles are taken from uniform distributed random numbers, unless the *gaussian* keyword is used. Then the individual coordinates are taken from a gaussian distribution of width *sigma* centered on *xmid,ymid,zmid*.

Individual atoms are inserted, unless the *mol* keyword is used. It specifies a *template-ID* previously defined using the *molecule* command, which reads files that define one or more molecules. The coordinates, atom types, charges, etc, as well as any bond/angle/etc and special neighbor information for the molecule can be specified in the molecule file. See the *molecule* command for details. The only settings required to be in each file are the coordinates and types of atoms in the molecule.

If the molecule template contains more than one molecule, the relative probability of depositing each molecule can be specified by the *molfrac* keyword. N relative probabilities, each from 0.0 to 1.0, are specified, where N is the number of molecules in the template. Each time a molecule is deposited, a random number is used to sample from the list of relative probabilities. The N values must sum to 1.0.

If you wish to insert molecules via the *mol* keyword, that will be treated as rigid bodies, use the *rigid* keyword, specifying as its value the ID of a separate *fix rigid/small* command which also appears in your input script.

**Note**

If you wish the new rigid molecules (and other rigid molecules) to be thermostatted correctly via *fix rigid/small/nvt* or *fix rigid/small/npt*, then you need to use the *fix_modify dynamic/dof yes* command for the rigid fix. This is to inform that fix that the molecule count will vary dynamically.

If you wish to insert molecules via the *mol* keyword, that will have their bonds or angles constrained via SHAKE, use the *shake* keyword, specifying as its value the ID of a separate *fix shake* command which also appears in your input script.

Each timestep a particle is inserted, the coordinates for its atoms are chosen as follows. For insertion of individual atoms, the "position" referred to in the following description is the coordinate of the atom. For insertion of molecule, the "position" is the geometric center of the molecule; see the *molecule* doc page for details. A random rotation of the molecule around its center point is performed, which determines the coordinates all the individual atoms.

A random position within the region insertion volume is generated. If neither the *global* or *local* keyword is used, the random position is the trial position. If the *global* keyword is used, the random x,y values are used, but the z position of the new particle is set above the highest current atom in the simulation by a distance randomly chosen between lo/hi. (For a 2d simulation, this is done for the y position.) If the *local* keyword is used, the z position is set a distance between lo/hi above the highest current atom in the simulation that is "nearby" the chosen x,y position. In this context, "nearby" means the lateral distance (in x,y) between the new and old particles is less than the *delta* setting.

Once a trial x,y,z position has been selected, the insertion is only performed if both the *near* and *var* keywords are satisfied (see below). If either the *near* or the *var* keyword is not satisfied, a new random position within the insertion volume is chosen and another trial is made. Up to Q attempts are made. If one or more particle insertions are not successful, LAMMPS prints a warning message.

The *near* keyword ensures that no current atom in the simulation is within a distance R of any atom in the new particle, including the effect of periodic boundary conditions if applicable. Note that the default value for R is 0.0, which will allow atoms to strongly overlap if you are inserting where other atoms are present. This distance test is performed independently for each atom in an inserted molecule, based on the randomly rotated configuration of the molecule.

**Note**

If you are inserting finite size particles or a molecule or rigid body consisting of finite-size particles, then you should typically set R larger than the distance at which any inserted particle may overlap with either a previously inserted particle or an existing particle. LAMMPS will issue a warning if R is smaller than this value, based on the radii of existing and inserted particles.

Added in version 21Nov2023.

The *var* and *set* keywords can be used together to provide a criterion for accepting or rejecting the addition of an individual atom, based on its coordinates. The *name* specified for the *var* keyword is the name of an *equal-style variable* that should evaluate to a zero or non-zero value based on one or two or three variables that will store the $x$, $y$, or $z$ coordinates of an atom (one variable per coordinate). If used, these other variables must be *internal-style variables* specified by the *set* keyword. They must be internal-style variables, because this command resets their values directly. The internal-style variables do not need to be defined in the input script (though they can be); if one (or more) is not defined, then the *set* option creates an *internal-style variable* with the specified name.

When an atom is about to be created, its $(x, y, z)$ coordinates become the values for any *set* variable that is defined. The *var* variable is then evaluated. If the returned value is 0.0, the atom is not created. If it is non-zero, the atom is created. For an example of how to use the set/var keywords in a similar context, see the *create_atoms* command.

The *rate* option moves the insertion volume in the z direction (3d) or y direction (2d). This enables particles to be inserted from a successively higher height over time. Note that this parameter is ignored if the *global* or *local* keywords are used, since those options choose a z-coordinate for insertion independently.

The vx, vy, and vz components of velocity for the inserted particle are set by sampling a uniform distribution between the bounds set by the values specified for the *vx*, *vy*, and *vz* keywords. Note that normally, new particles should be a assigned a negative vertical velocity so that they move towards the surface. For molecules, the same velocity is given to every particle (no rotation or bond vibration).

If the *target* option is used, the velocity vector of the inserted particle is changed so that it points from the insertion

position towards the specified target point. The magnitude of the velocity is unchanged. This can be useful, for example, for simulating a sputtering process. E.g. the target point can be far away, so that all incident particles strike the surface as if they are in an incident beam of particles at a prescribed angle.

The *orient* keyword is only used when molecules are deposited. By default, each molecule is inserted at a random orientation. If this keyword is specified, then (rx,ry,rz) is used as an orientation vector, and each inserted molecule is rotated around that vector with a random value from zero to 2*PI. For a 2d simulation, rx = ry = 0.0 is required, since rotations can only be performed around the z axis.

The *id* keyword determines how atom IDs and molecule IDs are assigned to newly deposited particles. Molecule IDs are only assigned if molecules are being inserted. For the *max* setting, the atom and molecule IDs of all current atoms are checked. Atoms in the new particle are assigned IDs starting with the current maximum plus one. If a molecule is inserted it is assigned an ID = current maximum plus one. This means that if particles leave the system, the new IDs may replace the lost ones. For the *next* setting, the maximum ID of any atom and molecule is stored at the time the fix is defined. Each time a new particle is added, this value is incremented to assign IDs to the new atom(s) or molecule. Thus atom and molecule IDs for deposited particles will be consecutive even if particles leave the system over time.

The *units* keyword determines the meaning of the distance units used for the other deposition parameters. A *box* value selects standard distance units as defined by the *units* command, e.g. Angstroms for units = real or metal. A *lattice* value means the distance units are in lattice spacings. The *lattice* command must have been previously used to define the lattice spacing. Note that the units choice affects all the keyword values that have units of distance or velocity.

**Note**

If you are monitoring the temperature of a system where the atom count is changing due to adding particles, you typically should use the *compute_modify dynamic/dof yes* command for the temperature compute you are using.

### 2.48.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the state of the deposition to *binary restart files*. This includes information about how many particles have been deposited, the random number generator seed, the next timestep for deposition, etc. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

**Note**

For this to work correctly, the timestep must **not** be changed after reading the restart with *reset_timestep*. The fix will try to detect it and stop with an error.

None of the *fix_modify* options are relevant to this fix. This fix computes a global scalar, which can be accessed by various output commands. The scalar is the cumulative number of insertions. The scalar value calculated by this fix is “intensive”. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.48.5 Restrictions

The specified insertion region cannot be a “dynamic” region, as defined by the *region* command.

### 2.48.6 Related commands

*fix pour*, *region*

### 2.48.7 Default

Insertions are performed for individual atoms, i.e. no *mol* setting is defined. If the *mol* keyword is used, the default for *molfrac* is an equal probabilities for all molecules in the template. Additional option defaults are id = max, delta = 0.0, near = 0.0, attempt = 10, rate = 0.0, vx = 0.0 0.0, vy = 0.0 0.0, vz = 0.0 0.0, and units = lattice.

## 2.49 fix dpd/energy command

Accelerator Variants: *dpd/energy/kk*

### 2.49.1 Syntax

```
fix ID group-ID dpd/energy
```

- ID, group-ID are documented in *fix* command
- dpd/energy = style name of this fix command

### 2.49.2 Examples

```
fix 1 all dpd/energy
```

### 2.49.3 Description

Perform constant energy dissipative particle dynamics (DPD-E) integration. This fix updates the internal energies for particles in the group at each timestep. It must be used in conjunction with a deterministic integrator (e.g. *fix nve*) that updates the particle positions and velocities.

For fix *dpd/energy*, the particle internal temperature is related to the particle internal energy through a mesoparticle equation of state. An additional fix must be specified that defines the equation of state for each particle, e.g. *fix eos/cv*.

This fix must be used with the *pair_style dpd/fdt/energy* command.

Note that numerous variants of DPD can be specified by choosing an appropriate combination of the integrator and *pair_style dpd/fdt/energy* command. DPD under isoenergetic conditions can be specified by using fix *dpd/energy*, fix *nve* and pair_style *dpd/fdt/energy*. DPD under isoenthalpic conditions can be specified by using fix *dpd/energy*, fix *nph* and pair_style *dpd/fdt/energy*. Examples of each DPD variant are provided in the examples/PACKAGES/dpd-react directory.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.49.4 Restrictions

This command is part of the DPD-REACT package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix must be used with an additional fix that specifies time integration, e.g. *fix nve*.

The fix *dpd/energy* requires the *dpd atom_style* to be used in order to properly account for the particle internal energies and temperature.

The fix *dpd/energy* must be used with an additional fix that specifies the mesoparticle equation of state for each particle.

### 2.49.5 Related commands

*fix nve fix eos/cv*

### 2.49.6 Default

none

---

**(Lisal)** M. Lisal, J.K. Brennan, J. Bonet Avalos, J. Chem. Phys., 135, 204105 (2011).

**(Larentzos)** J.P. Larentzos, J.K. Brennan, J.D. Moore, and W.D. Mattson, ARL-TR-6863, U.S. Army Research Laboratory, Aberdeen Proving Ground, MD (2014).

## 2.50 fix edpd/source command

## 2.51 fix tdpd/source command

### 2.51.1 Syntax

```
fix ID group-ID edpd/source keyword values ...
fix ID group-ID tdpd/source cc_index keyword values ...
```

- ID, group-ID are documented in *fix* command
- edpd/source or tdpd/source = style name of this fix command
- index (only specified for tdpd/source) = index of chemical species (1 to Nspecies)
- keyword = *sphere* or *cuboid* or *region*

```
sphere args = cx cy cz radius source
  cx,cy,cz = x,y,z center of spherical domain (distance units)
  radius = radius of a spherical domain (distance units)
  source = heat source or concentration source (flux units, see below)
cuboid values = cx cy cz dLx dLy dLz source
  cx,cy,cz = x,y,z center of a cuboid domain (distance units)
  dLx,dLy,dLz = x,y,z side length of a cuboid domain (distance units)
  source = heat source or concentration source (flux units, see below)
region values = region-ID source
  region = ID of region for heat or concentration source
  source = heat source or concentration source (flux units, see below)
```

### 2.51.2 Examples

```
fix 1 all edpd/source sphere 0.0 0.0 0.0 5.0 0.01
fix 1 all edpd/source cuboid 0.0 0.0 0.0 20.0 10.0 10.0 -0.01
fix 1 all tdpd/source 1 sphere 5.0 0.0 0.0 5.0 0.01
fix 1 all tdpd/source 2 cuboid 0.0 0.0 0.0 20.0 10.0 10.0 0.01
fix 1 all tdpd/source 1 region lower -0.01
```

### 2.51.3 Description

Fix *edpd/source* adds a heat source as an external heat flux to each atom in a spherical or cuboid domain, where the *source* is in units of energy/time. Fix *tdpd/source* adds an external concentration source of the chemical species specified by *index* as an external concentration flux for each atom in a spherical or cuboid domain, where the *source* is in units of mole/volume/time.

This command can be used to give an additional heat/concentration source term to atoms in a simulation, such as for a simulation of a heat conduction with a source term (see Fig.12 in *(Li2014)*) or diffusion with a source term (see Fig.1 in *(Li2015)*), as an analog of a periodic Poiseuille flow problem.

Deprecated since version 15Jun2023: The *sphere* and *cuboid* keywords will be removed in a future version of LAMMPS. The same functionality and more can be achieved with a region.

If the *sphere* keyword is used, the *cx, cy, cz, radius* values define a spherical domain to apply the source flux to.

If the *cuboid* keyword is used, the *cx, cy, cz, dLx, dLy, dLz* define a cuboid domain to apply the source flux to.

If the *region* keyword is used, the *region-ID* selects which *region* to apply the source flux to.

---

### 2.51.4 Restart, fix_modify, output, run start/stop, minimize info

No information of these fixes is written to *binary restart files*. None of the *fix_modify* options are relevant to these fixes. No global or per-atom quantities are stored by these fixes for access by various *output commands*. No parameter of these fixes can be used with the *start/stop* keywords of the *run* command. These fixes are not invoked during *energy minimization*.

### 2.51.5 Restrictions

These fixes are part of the DPD-MESO package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Fix *edpd/source* must be used with the *pair_style edpd* command. Fix *tdpd/source* must be used with the *pair_style tdpd* command.

### 2.51.6 Related commands

*pair_style edpd*, *pair_style tdpd*, *compute edpd/temp/atom*, *compute tdpd/cc/atom*

### 2.51.7 Default

none

---

**(Li2014)** Z. Li, Y.-H. Tang, H. Lei, B. Caswell and G.E. Karniadakis, "Energy-conserving dissipative particle dynamics with temperature-dependent properties", J. Comput. Phys., 265: 113-127 (2014). DOI: 10.1016/j.jcp.2014.02.003

**(Li2015)** Z. Li, A. Yazdani, A. Tartakovsky and G.E. Karniadakis, "Transport dissipative particle dynamics model for mesoscopic advection-diffusion-reaction problems", J. Chem. Phys., 143: 014101 (2015). DOI: 10.1063/1.4923254

# 2.52 fix drag command

Accelerator Variant: *drag/kk*

## 2.52.1 Syntax

```
fix ID group-ID drag x y z fmag delta
```

- ID, group-ID are documented in *fix* command
- drag = style name of this fix command
- x,y,z = coord to drag atoms towards
- fmag = magnitude of force to apply to each atom (force units)
- delta = cutoff distance inside of which force is not applied (distance units)

## 2.52.2 Examples

```
fix center small-molecule drag 0.0 10.0 0.0 5.0 2.0
```

## 2.52.3 Description

Apply a force to each atom in a group to drag it towards the point (x,y,z). The magnitude of the force is specified by fmag. If an atom is closer than a distance delta to the point, then the force is not applied.

Any of the x,y,z values can be specified as NULL which means do not include that dimension in the distance calculation or force application.

This command can be used to steer one or more atoms to a new location in the simulation.

## 2.52.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify* *respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is adding its forces. Default is the outermost level.

This fix computes a global 3-vector of forces, which can be accessed by various *output commands*. This is the total force on the group of atoms by the drag force. The vector values calculated by this fix are "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.52.5 Restrictions

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.52.6 Related commands

*fix spring*, *fix spring/self* , *fix spring/rg*, *fix smd*

### 2.52.7 Default

none

## 2.53 fix drude command

### 2.53.1 Syntax

```
fix ID group-ID drude flag1 flag2 ... flagN
```

- ID, group-ID are documented in *fix* command
- drude = style name of this fix command
- flag1 flag2 ... flagN = Drude flag for each atom type (1 to N) in the system

### 2.53.2 Examples

```
fix 1 all drude 1 1 0 1 0 2 2 2
fix 1 all drude C C N C N D D D
```

Example input scripts available: examples/PACKAGES/drude

### 2.53.3 Description

Assign each atom type in the system to be one of 3 kinds of atoms within the Drude polarization model. This fix is designed to be used with the *thermalized Drude oscillator model*. Polarizable models in LAMMPS are described on the *Howto polarizable* doc page.

The three possible types can be designated with an integer (0,1,2) or capital letter (N,C,D):

- 0 or N = non-polarizable atom (not part of Drude model)
- 1 or C = Drude core
- 2 or D = Drude electron

### 2.53.4 Restrictions

This fix should be invoked before any other commands that implement the Drude oscillator model, such as *fix langevin/drude*, *fix tgnvt/drude*, *fix drude/transform*, *compute temp/drude*, *pair_style thole*.

### 2.53.5 Related commands

*fix langevin/drude*, *fix tgnvt/drude*, *fix drude/transform*, *compute temp/drude*, *pair_style thole*

### 2.53.6 Default

none

## 2.54 fix drude/transform/direct command

## 2.55 fix drude/transform/inverse command

### 2.55.1 Syntax

```
fix ID group-ID style keyword value ...
```

- ID, group-ID are documented in *fix* command
- style = *drude/transform/direct* or *drude/transform/inverse*

### 2.55.2 Examples

```
fix 3 all drude/transform/direct
fix 1 all drude/transform/inverse
```

Example input scripts available: examples/PACKAGES/drude

### 2.55.3 Description

Transform the coordinates of Drude oscillators from real to reduced and back for thermalizing the Drude oscillators as described in *(Lamoureux)* using a Nose-Hoover thermostat. This fix is designed to be used with the *thermalized Drude oscillator model*. Polarizable models in LAMMPS are described on the *Howto polarizable* doc page.

Drude oscillators are a pair of atoms representing a single polarizable atom. Ideally, the mass of Drude particles would vanish and their positions would be determined self-consistently by iterative minimization of the energy, the cores' positions being fixed. It is however more efficient and it yields comparable results, if the Drude oscillators (the motion of the Drude particle relative to the core) are thermalized at a low temperature. In that case, the Drude particles need a small mass.

The thermostats act on the reduced degrees of freedom, which are defined by the following equations. Note that in these equations upper case denotes atomic or center of mass values and lower case denotes Drude particle or dipole values. Primes denote the transformed (reduced) values, while bare letters denote the original values.

Masses:

$$M' = M + m$$

$$m' = \frac{M\,m}{M'}$$

Positions:

$$X' = \frac{M\,X + m\,x}{M'}$$

$$x' = x - X$$

Velocities:

$$V' = \frac{M\,V + m\,v}{M'}$$

$$v' = v - V$$

Forces:

$$F' = F + f$$

$$f' = \frac{M\,f - m\,F}{M'}$$

This transform conserves the total kinetic energy

$$\frac{1}{2}\,(M\,V^2 \;+ m\,v^2) = \frac{1}{2}\,(M'\,V'^2 \;+ m'\,v'^2)$$

and the virial defined with absolute positions

$$X\,F + x\,f = X'\,F' + x'\,f'$$

---

This fix requires each atom know whether it is a Drude particle or not. You must therefore use the *fix drude* command to specify the Drude status of each atom type.

**Note**

only the Drude core atoms need to be in the group specified for this fix. A Drude electron will be transformed together with its core even if it is not itself in the group. It is safe to include Drude electrons or non-polarizable atoms in the group. The non-polarizable atoms will simply not be transformed.

---

This fix does NOT perform time integration. It only transform masses, coordinates, velocities and forces. Thus you must use separate time integration fixes, like *fix nve* or *fix npt* to actually update the velocities and positions of atoms. In order to thermalize the reduced degrees of freedom at different temperatures, two Nose-Hoover thermostats must be defined, acting on two distinct groups.

**Note**

The *fix drude/transform/direct* command must appear before any Nose-Hoover thermostatting fixes. The *fix drude/transform/inverse* command must appear after any Nose-Hoover thermostatting fixes.

Example:

```
fix fDIRECT all drude/transform/direct
fix fNVT gCORES nvt temp 300.0 300.0 100
fix fNVT gDRUDES nvt temp 1.0 1.0 100
fix fINVERSE all drude/transform/inverse
compute TDRUDE all temp/drude
thermo_style custom step cpu etotal ke pe ebond ecoul elong press vol temp c_TDRUDE[1] c_
↪TDRUDE[2]
```

In this example, *gCORES* is the group of the atom cores and *gDRUDES* is the group of the Drude particles (electrons). The centers of mass of the Drude oscillators will be thermostatted at 300.0 and the internal degrees of freedom will be thermostatted at 1.0. The temperatures of cores and Drude particles, in center-of-mass and relative coordinates, are calculated using *compute temp/drude*

In addition, if you want to use a barostat to simulate a system at constant pressure, only one of the Nose-Hoover fixes must be *npt*, the other one should be *nvt*. You must add a *compute temp/com* and a *fix_modify* command so that the temperature of the *npt* fix be just that of its group (the Drude cores) but the pressure be the overall pressure *thermo_press*.

Example:

```
compute cTEMP_CORE gCORES temp/com
fix fDIRECT all drude/transform/direct
fix fNPT gCORES npt temp 298.0 298.0 100 iso 1.0 1.0 500
fix_modify fNPT temp cTEMP_CORE press thermo_press
fix fNVT gDRUDES nvt temp 5.0 5.0 100
fix fINVERSE all drude/transform/inverse
```

In this example, *gCORES* is the group of the atom cores and *gDRUDES* is the group of the Drude particles. The centers of mass of the Drude oscillators will be thermostatted at 298.0 and the internal degrees of freedom will be thermostatted at 5.0. The whole system will be barostatted at 1.0.

In order to avoid the flying ice cube problem (irreversible transfer of linear momentum to the center of mass of the system), you may need to add a *fix momentum* command:

```
fix fMOMENTUM all momentum 100 linear 1 1 1
```

---

### 2.55.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

### 2.55.5 Restrictions

none

### 2.55.6 Related commands

*fix drude*, *fix langevin/drude*, *compute temp/drude*, *pair_style thole*

### 2.55.7 Default

none

---

**(Lamoureux)** Lamoureux and Roux, J Chem Phys, 119, 3025-3039 (2003).

## 2.56 fix dt/reset command

Accelerator Variants: *dt/reset/kk*

### 2.56.1 Syntax

```
fix ID group-ID dt/reset N Tmin Tmax Xmax keyword values ...
```

- ID, group-ID are documented in *fix* command
- dt/reset = style name of this fix command
- N = re-compute dt every N timesteps

- Tmin = minimum dt allowed which can be NULL (time units)
- Tmax = maximum dt allowed which can be NULL (time units)
- Xmax = maximum distance for an atom to move in one timestep (distance units)
- zero or more keyword/value pairs may be appended
- keyword = *emax* or *units*

```
emax value = Emax
  Emax = maximum kinetic energy change for an atom in one timestep (energy units)
units value = lattice or box
  lattice = Xmax is defined in lattice units
  box = Xmax is defined in simulation box units
```

### 2.56.2 Examples

```
fix 5 all dt/reset 10 1.0e-5 0.01 0.1
fix 5 all dt/reset 10 0.01 2.0 0.2 units box
fix 5 all dt/reset 5 NULL 0.001 0.5 emax 30 units box
```

### 2.56.3 Description

Reset the timestep size every N steps during a run, so that no atom moves further than the specified *Xmax* distance, based on current atom velocities and forces. Optionally an additional criterion is imposed by the *emax* keyword, so that no atom's kinetic energy changes by more than the specified *Emax*.

This can be useful when starting from a configuration with overlapping atoms, where forces will be large. Or it can be useful when running an impact simulation where one or more high-energy atoms collide with a solid, causing a damage cascade.

This fix overrides the timestep size setting made by the *timestep* command. The new timestep size *dt* is computed in the following manner.

For each atom, the timestep is computed that would cause it to displace *Xmax* on the next integration step, as a function of its current velocity and force. Since performing this calculation exactly would require the solution to a quartic equation, a cheaper estimate is generated. The estimate is conservative in that the atom's displacement is guaranteed not to exceed *Xmax*, though it may be smaller.

In addition if the *emax* keyword is used, the specified *Emax* value is enforced as a limit on how much an atom's kinetic energy can change. If the timestep required is even smaller than for the *Xmax* displacement, then the smaller timestep is used.

Given this putative timestep for each atom, the minimum timestep value across all atoms is computed. Then the *Tmin* and *Tmax* bounds are applied, if specified. If one (or both) is specified as NULL, it is not applied.

When the *run style* is *respa*, this fix resets the outer loop (largest) timestep, which is the same timestep that the *timestep* command sets.

Note that the cumulative simulation time (in time units), which accounts for changes in the timestep size as a simulation proceeds, can be accessed by the *thermo_style time* keyword.

Also note that the *dump_modify every/time* option allows dump files to be written at intervals specified by simulation time, rather than by timesteps. Simulation time is in time units; see the *units* doc page for details.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages*

page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 2.56.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar stores the last timestep on which the timestep was reset to a new value.

The scalar value calculated by this fix is "intensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.56.5 Restrictions

none

### 2.56.6 Related commands

*timestep*, *dump_modify every/time*

### 2.56.7 Default

The option defaults are units = lattice, and no emax kinetic energy limit.

## 2.57 fix efield command

## 2.58 fix efield/tip4p command

Accelerator Variant: *efield/kk*

### 2.58.1 Syntax

```
fix ID group-ID style ex ey ez keyword value ...
```

- ID, group-ID are documented in *fix* command
- style = *efield* or *efield/tip4p*
- ex,ey,ez = E-field component values (electric field units)
- any of ex,ey,ez can be a variable (see below)
- zero or more keyword/value pairs may be appended to args
- keyword = *region* or *energy* or *potential*

```
region value = region-ID
  region-ID = ID of region atoms must be in to have added force
energy value = v_name
  v_name = variable with name that calculates the potential energy of each atom in␣
↪the added E-field
potential value = v_name
  v_name = variable with name that calculates the electric potential of each atom␣
↪in the added E-field
```

### 2.58.2 Examples

```
fix kick external-field efield 1.0 0.0 0.0
fix kick external-field efield 0.0 0.0 v_oscillate
fix kick external-field efield/tip4p 1.0 0.0 0.0
```

### 2.58.3 Description

Add a force $\vec{F} = q\vec{E}$ to each charged atom in the group due to an external electric field being applied to the system. If the system contains point-dipoles, also add a torque $\vec{T} = \vec{p} \times \vec{E}$ on the dipoles due to the external electric field. This fix does not compute the dipole force $\vec{F} = (\vec{p} \cdot \nabla)\vec{E}$, and the *fix efield/lepton* command should be used instead.

Added in version 28Mar2023.

When the *efield/tip4p* style is used, the E-field will be applied to the position of the virtual charge site M of a TIP4P molecule instead of the oxygen position as it is defined by a corresponding *TIP4P pair style*. The forces on the M site due to the external field are projected on the oxygen and hydrogen atoms of the TIP4P molecules.

For charges, any of the 3 quantities defining the E-field components can be specified as an equal-style or atom-style *variable*, namely *ex*, *ey*, *ez*. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated each timestep, and its value used to determine the E-field component.

For point-dipoles, equal-style variables can be used, but atom-style variables are not currently supported, since they imply a spatial gradient in the electric field which means additional terms with gradients of the field are required for the force and torque on dipoles. The *fix efield/lepton* command should be used instead.

Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent E-field.

Atom-style variables can specify the same formulas as equal-style variables but can also include per-atom values, such as atom coordinates. Thus it is easy to specify a spatially-dependent E-field with optional time-dependence as well.

If the *region* keyword is used, the atom must also be in the specified geometric *region* in order to have force added to it.

---

Adding a force or torque to atoms implies a change in their potential energy as they move or rotate due to the applied E-field.

For dynamics via the "run" command, this energy can be optionally added to the system's potential energy for thermodynamic output (see below). For energy minimization via the "minimize" command, this energy must be added to the system's potential energy to formulate a self-consistent minimization problem (see below).

The *energy* keyword is not allowed if the added field is a constant vector (ex,ey,ez), with all components defined as numeric constants and not as variables. This is because LAMMPS can compute the energy for each charged particle directly as

$$U_{efield} = -\vec{x} \cdot q\vec{E} = -q(x \cdot E_x + y \cdot E_y + z \cdot Ez),$$

so that $-\nabla U_{efield} = \vec{F}$. Similarly for point-dipole particles the energy can be computed as

$$U_{efield} = -\vec{\mu} \cdot \vec{E} = -\mu_x \cdot E_x + \mu_y \cdot E_y + \mu_z \cdot E_z$$

The *energy* keyword is optional if the added force is defined with one or more variables, and if you are performing dynamics via the *run* command. If the keyword is not used, LAMMPS will set the energy to 0.0, which is typically fine for dynamics.

The *energy* keyword (or *potential* keyword, described below) is required if the added force is defined with one or more variables, and you are performing energy minimization via the "minimize" command for charged particles. It is not required for point-dipoles, but a warning is issued since the minimizer in LAMMPS does not rotate dipoles, so you should not expect to be able to minimize the orientation of dipoles in an applied electric field.

The *energy* keyword specifies the name of an atom-style *variable* which is used to compute the energy of each atom as function of its position. Like variables used for *ex*, *ey*, *ez*, the energy variable is specified as "v_name", where "name" is the variable name.

Note that when the *energy* keyword is used during an energy minimization, you must ensure that the formula defined for the atom-style *variable* is consistent with the force variable formulas, i.e. that -Grad(E) = F. For example, if the force due to the electric field were a spring-like F = kx, then the energy formula should be E = -0.5kx^2. If you don't do this correctly, the minimization will not converge properly.

Added in version 15Jun2023.

The *potential* keyword can be used as an alternative to the *energy* keyword to specify the name of an atom-style variable, which is used to compute the added electric potential to each atom as a function of its position. The variable should have units of electric field multiplied by distance (that is, in *units real*, the potential should be in volts). As with the *energy* keyword, the variable name is specified as "v_name". The energy added by this fix is then calculated as the electric potential multiplied by charge.

The *potential* keyword is mainly intended for correct charge equilibration in simulations with *fix qeq/reaxff*, since with variable charges the electric potential can be known beforehand but the energy cannot. A small additional benefit is that the *energy* keyword requires an additional conversion to energy units which the *potential* keyword avoids. Thus, when the *potential* keyword is specified, the *energy* keyword must not be used. As with *energy*, the *potential* keyword is not allowed if the added field is a constant vector. The *potential* keyword is not supported by *fix efield/tip4p*.

---

### 2.58.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify energy* option is supported by this fix to add the potential energy inferred by the added force due to the electric field to the global potential energy of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify energy no*. Note that this energy is a fictitious quantity but is needed so that the *minimize* command can include the forces added by this fix in a consistent manner. I.e. there is a decrease in potential energy when atoms move in the direction of the added force due to the electric field.

The *fix_modify virial* option is supported by this fix to add the contribution due to the added forces on atoms to both the global pressure and per-atom stress of the system via the *compute pressure* and *compute stress/atom* commands. The former can be accessed by *thermodynamic output*. The default setting for this fix is *fix_modify virial no*.

The *fix_modify respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix adding its forces. Default is the outermost level.

This fix computes a global scalar and a global 3-vector of forces, which can be accessed by various *output commands*. The scalar is the potential energy discussed above. The vector is the total force added to the group of atoms. The scalar and vector values calculated by this fix are "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The forces due to this fix are imposed during an energy minimization, invoked by the *minimize* command. You should not specify force components with a variable that has time-dependence for use with a minimizer, since the minimizer increments the timestep as the iteration count during the minimization.

**Note**

If you want the fictitious potential energy associated with the added forces to be included in the total potential energy of the system (the quantity being minimized), you MUST enable the *fix_modify energy* option for this fix.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.58.5 Restrictions

Fix style *efield/tip4p* is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Fix style *efield/tip4p* can only be used with tip4p pair styles.

### 2.58.6 Related commands

*fix addforce*, *fix efield/lepton*

### 2.58.7 Default

none

## 2.59 fix efield/lepton command

### 2.59.1 Syntax

```
fix ID group-ID efield/lepton V ...
```

- ID, group-ID are documented in the *fix* command
- style = *efield/lepton*
- V = electric potential (electric field * distance units)
- V must be a Lepton expression (see below)
- zero or more keyword/value pairs may be appended to args
- keyword = *region* or *step*

```
  region value = region-ID
    region-ID = ID of region atoms must be in to have effect
  step value = h
    h = step size for numerical differentiation (distance units)
```

### 2.59.2 Examples

```
fix ex all efield/lepton "-E*x; E=1"
fix dexx all efield/lepton "-0.5*x^2" step 1
fix yukawa all efield/lepton "A*exp(-B*r)/r; r=abs(sqrt(x^2+y^2+z^2)); A=1; B=1" step 1e-
↪6
fix infp all efield/lepton "-abs(x)" step 1

variable th equal 2*PI*ramp(0,1)
fix erot all efield/lepton "-(x*cos(v_th)+y*sin(v_th))"
```

### 2.59.3 Description

Added in version 4Feb2025.

Add an electric potential $V$ that applies to a group of charged atoms a force $\vec{F} = q\vec{E}$, and to dipoles a force $\vec{F} = (\vec{p} \cdot \nabla)\vec{E}$ and torque $\vec{T} = \vec{p} \times \vec{E}$, where $\vec{E} = -\nabla V$. The fix also evaluates the electrostatic energy ($U_q = qV$ and $U_p = -\vec{p} \cdot \vec{E}$) due to this potential when the *fix_modify energy yes* command is specified (see below).

**Note**

This command should be used instead of *fix efield* if you want to impose a non-uniform electric field on a system with dipoles since the latter does not include the dipole force term. If you only have charges or if the electric field gradient is negligible, *fix efield* should be used since it is faster.

The Lepton library, that the *efield/lepton* fix style interfaces with, evaluates the expression string at run time to compute the energy, forces, and torques. It creates an analytical representation of $V$ and $\vec{E}$, while the gradient force is computed using a central difference scheme

$$\vec{F} = \frac{|\vec{p}|}{2h}\left[\vec{E}(\vec{x} + h\hat{p}) - \vec{E}(\vec{x} - h\hat{p})\right].$$

The Lepton expression must be either enclosed in quotes or must not contain any whitespace so that LAMMPS recognizes it as a single keyword. More on valid Lepton expressions below. The final Lepton expression must be a function of only $x, y, z$, which refer to the current *unwrapped* coordinates of the atoms to ensure continuity. Special care must be taken when using this fix with periodic boundary conditions or box-changing commands.

---

### 2.59.4 Lepton expression syntax and features

Lepton supports the following operators in expressions:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| + | Add | - | Subtract | * | Multiply | / | Divide | ^ | Power |

The following mathematical functions are available:

| | | | |
|---|---|---|---|
| sqrt(x) | Square root | exp(x) | Exponential |
| log(x) | Natural logarithm | sin(x) | Sine (angle in radians) |
| cos(x) | Cosine (angle in radians) | sec(x) | Secant (angle in radians) |
| csc(x) | Cosecant (angle in radians) | tan(x) | Tangent (angle in radians) |
| cot(x) | Cotangent (angle in radians) | asin(x) | Inverse sine (in radians) |
| acos(x) | Inverse cosine (in radians) | atan(x) | Inverse tangent (in radians) |
| sinh(x) | Hyperbolic sine | cosh(x) | Hyperbolic cosine |
| tanh(x) | Hyperbolic tangent | erf(x) | Error function |
| erfc(x) | Complementary Error function | abs(x) | Absolute value |
| min(x,y) | Minimum of two values | max(x,y) | Maximum of two values |
| delta(x) | delta(x) is 1 for $x = 0$, otherwise 0 | step(x) | step(x) is 0 for $x < 0$, otherwise 1 |

Numbers may be given in either decimal or exponential form. All of the following are valid numbers: *5*, *-3.1*, *1e6*, and *3.12e-2*.

As an extension to the standard Lepton syntax, it is also possible to use LAMMPS *variables* in the format "v_name". Before evaluating the expression, "v_name" will be replaced with the value of the variable "name". This is compatible with all kinds of scalar variables, but not with vectors, arrays, local, or per-atom variables. If necessary, a custom scalar variable needs to be defined that can access the desired (single) item from a non-scalar variable. As an example, the following lines will instruct LAMMPS to ramp the force constant for a harmonic bond from 100.0 to 200.0 during the next run:

```
variable fconst equal ramp(100.0, 200)
bond_style lepton
bond_coeff 1 1.5 "v_fconst * (r^2)"
```

An expression may be followed by definitions for intermediate values that appear in the expression. A semicolon ";" is used as a delimiter between value definitions. For example, the expression:

```
a^2+a*b+b^2; a=a1+a2; b=b1+b2
```

is exactly equivalent to

```
(a1+a2)^2+(a1+a2)*(b1+b2)+(b1+b2)^2
```

The definition of an intermediate value may itself involve other intermediate values. Whitespace and quotation characters ('' and "") are ignored. All uses of a value must appear *before* that value's definition. For efficiency reasons, the expression string is parsed, optimized, and then stored in an internal, pre-parsed representation for evaluation.

Evaluating a Lepton expression is typically between 2.5 and 5 times slower than the corresponding compiled and optimized C++ code. If additional speed or GPU acceleration (via GPU or KOKKOS) is required, the interaction can be represented as a table. Suitable table files can be created either internally using the *pair_write* or *bond_write* command or through the Python scripts in the *tools/tabulate* folder.

---

If the *region* keyword is used, the atom must also be in the specified geometric *region* in order to be affected by the potential.

The *step* keyword is required when *atom_style dipole* is used and the electric field is non-uniform.

---

### 2.59.5 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify energy* option is supported by this fix to add the potential energy defined above to the global potential energy of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify energy no*.

The *fix_modify virial* option is supported by this fix to add the contribution due to the added ***forces*** on charges and dipoles to both the global pressure and per-atom stress of the system via the *compute pressure* and *compute stress/atom* commands. The former can be accessed by *thermodynamic output*. The default setting for this fix is *fix_modify virial no*.

The *fix_modify respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix adding its forces. Default is the outermost level.

This fix computes a global scalar and a global 3-vector of forces, which can be accessed by various *output commands*. The scalar is the potential energy discussed above. The vector is the total force added to the group of atoms. The scalar and vector values calculated by this fix are "extensive".

This fix cannot be used with the *start/stop* keywords of the *run* command.

The forces due to this fix are imposed during an energy minimization, invoked by the *minimize* command. You should not specify force components with a variable that has time-dependence for use with a minimizer, since the minimizer increments the timestep as the iteration count during the minimization.

**Note**

If you want the electric potential energy to be included in the total potential energy of the system (the quantity being minimized), you MUST enable the *fix_modify energy* option for this fix.

---

### 2.59.6 Restrictions

Fix style *efield/lepton* is part of the LEPTON package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.59.7 Related commands

*fix efield*

### 2.59.8 Default

none

## 2.60 fix ehex command

### 2.60.1 Syntax

```
fix ID group-ID ehex nevery F keyword value
```

- ID, group-ID are documented in *fix* command
- ehex = style name of this fix command
- nevery = add/subtract heat every this many timesteps
- F = energy flux into the reservoir (energy/time units)

- zero or more keyword/value pairs may be appended to args
- keyword = *region* or *constrain* or *com* or *hex*

```
region value = region-ID
  region-ID = ID of region (reservoir) atoms must be in for added thermostatting␣
↪force
constrain value = none
  apply the constraint algorithm (SHAKE or RATTLE) again at the end of the timestep
com value = none
  rescale all sites of a constrained cluster of atom if its COM is in the reservoir
hex value = none
  omit the coordinate correction to recover the HEX algorithm
```

## 2.60.2 Examples

```
# Lennard-Jones, from examples/in.ehex.lj

fix fnve all nve
# specify regions rhot and rcold
...
fix fhot all ehex 1 0.15 region rhot
fix fcold all ehex 1 -0.15 region rcold

# SPC/E water, from examples/in.ehex.spce
fix fnve all nve
# specify regions rhot and rcold
...
fix fhot all ehex 1 0.075 region rhot constrain com
fix fcold all ehex 1 -0.075 region rcold constrain com
fix frattle all rattle 1e-10 400 0 b 1 a 1
```

## 2.60.3 Description

This fix implements the asymmetric version of the enhanced heat exchange algorithm *(Wirnsberger)*. The eHEX algorithm is an extension of the heat exchange algorithm *(Ikeshoji)* and adds an additional coordinate integration to account for higher-order truncation terms in the operator splitting. The original HEX algorithm (implemented as *fix heat*) is known to exhibit a slight energy drift limiting the accessible simulation times to a few nanoseconds. This issue is greatly improved by the new algorithm decreasing the energy drift by at least a factor of a hundred (LJ and SPC/E water) with little computational overhead.

In both algorithms (non-translational) kinetic energy is constantly swapped between regions (reservoirs) to impose a heat flux onto the system. The equations of motion are therefore modified if a particle $i$ is located inside a reservoir $\Gamma_k$ where $k > 0$. We use $\Gamma_0$ to label those parts of the simulation box which are not thermostatted.) The input parameter *region-ID* of this fix corresponds to $k$. The energy swap is modelled by introducing an additional thermostatting force to the equations of motion, such that the time evolution of coordinates and momenta of particle $i$ becomes *(Wirnsberger)*

$$\begin{aligned}\dot{\mathbf{r}}_i &= \mathbf{v}_i,\\ \dot{\mathbf{v}}_i &= \frac{\mathbf{f}_i}{m_i} + \frac{\mathbf{g}_i}{m_i}.\end{aligned}$$

The thermostatting force is given by

$$\mathbf{g}_i = \begin{cases} \frac{m_i}{2}\frac{F_{\Gamma_{k(\mathbf{r}_i)}}}{K_{\Gamma_{k(\mathbf{r}_i)}}}\left(\mathbf{v}_i - \mathbf{v}_{\Gamma_{k(\mathbf{r}_i)}}\right) & k(\mathbf{r}_i) > 0 \text{ (inside a reservoir)},\\ 0 & \text{otherwise},\end{cases}$$

where $m_i$ is the mass and $k(\mathbf{r}_i)$ maps the particle position to the respective reservoir. The quantity $F_{\Gamma_{k(\mathbf{r}_i)}}$ corresponds to the input parameter $F$, which is the energy flux into the reservoir. Furthermore, $K_{\Gamma_{k(\mathbf{r}_i)}}$ and $v_{\Gamma_{k(\mathbf{r}_i)}}$ denote the non-translational kinetic energy and the center of mass velocity of that reservoir. The thermostatting force does not affect the center of mass velocities of the individual reservoirs and the entire simulation box. A derivation of the equations and details on the numerical implementation with velocity Verlet in LAMMPS can be found in reference "(Wirnsberger)"#_Wirnsberger.

**Note**

This fix only integrates the thermostatting force and must be combined with another integrator, such as *fix nve*, to solve the full equations of motion.

This fix is different from a thermostat such as *fix nvt* or *fix temp/rescale* in that energy is added/subtracted continually. Thus if there is no other mechanism in place to counterbalance this effect, the entire system will heat or cool continuously.

**Note**

If heat is subtracted from the system too aggressively so that the group's kinetic energy would go to zero, then LAMMPS will halt with an error message. Increasing the value of *nevery* means that heat is added/subtracted less frequently but in larger portions. The resulting temperature profile will therefore be the same.

This fix will default to *fix_heat* (HEX algorithm) if the keyword *hex* is specified.

---

**Compatibility with SHAKE and RATTLE (rigid molecules)**:

This fix is compatible with *fix shake* and *fix rattle*. If either of these constraining algorithms is specified in the input script and the keyword *constrain* is set, the bond distances will be corrected a second time at the end of the integration step. It is recommended to specify the keyword *com* in addition to the keyword *constrain*. With this option all sites of a constrained cluster are rescaled, if its center of mass is located inside the region. Rescaling all sites of a cluster by the same factor does not introduce any velocity components along fixed bonds. No rescaling takes place if the center of mass lies outside the region.

**Note**

You can only use the keyword *com* along with *constrain*.

To achieve the highest accuracy it is recommended to use *fix rattle* with the keywords *constrain* and *com* as shown in the second example. Only if RATTLE is employed, the velocity constraints will be satisfied.

**Note**

Even if RATTLE is used and the keywords *com* and *constrain* are both set, the coordinate constraints will not necessarily be satisfied up to the target precision. The velocity constraints are satisfied as long as all sites of a cluster are rescaled (keyword *com*) and the cluster does not span adjacent reservoirs. The current implementation of the eHEX algorithm introduces a small error in the bond distances, which goes to zero with order three in the timestep. For example, in a simulation of SPC/E water with a timestep of 2 fs the maximum relative error in the bond distances was found to be on the order of $10^{-7}$ for relatively large temperature gradients. A higher precision can be achieved by decreasing the timestep.

### 2.60.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.60.5 Restrictions

This fix is part of the RIGID package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.60.6 Related commands

*fix heat*, *fix thermal/conductivity*, *compute temp*, *compute temp/region*

### 2.60.7 Default

none

---

**(Ikeshoji)** Ikeshoji and Hafskjold, Molecular Physics, 81, 251-261 (1994).

**(Wirnsberger)** Wirnsberger, Frenkel, and Dellago, J Chem Phys, 143, 124104 (2015).

## 2.61 fix electrode/conp command

Accelerator Variant: *electrode/conp/intel*

## 2.62 fix electrode/conq command

Accelerator Variant: *electrode/conq/intel*

## 2.63 fix electrode/thermo command

Accelerator Variant: *electrode/thermo/intel*

### 2.63.1 Syntax

```
fix ID group-ID style args keyword value ...
```

- ID, group-ID are documented in *fix* command
- style = *electrode/conp* or *electrode/conq* or *electrode/thermo*
- args = arguments used by a particular style

  *electrode/conp* args = potential eta
  *electrode/conq* args = charge eta
  *electrode/thermo* args = potential eta *temp* values
  potential = electrode potential
  charge = electrode charge

```
        eta = reciprocal width of electrode charge smearing (can be NULL if eta␣
    ↪keyword is used)
        temp values = T_v tau_v rng_v
            T_v = temperature of thermo-potentiostat
            tau_v = time constant of thermo-potentiostat
            rng_v = integer used to initialize random number generator
```

- zero or more keyword/value pairs may be appended
- keyword = *algo* or *symm* or *couple* or *etypes* or *ffield* or *write_mat* or *write_inv* or *read_mat* or *read_inv* or *qtotal* or *eta*

```
algo values = mat_inv or mat_cg tol or cg tol
    specify the algorithm used to compute the electrode charges
symm value = on or off
    turn on/off charge neutrality constraint for the electrodes
couple values = group-ID val
    group-ID = group of atoms treated as additional electrode
    val = electric potential or charge on this electrode
etypes value = on or off
    turn on/off type-based optimized neighbor lists (electrode and electrolyte types may␣
    ↪not overlap)
ffield value = on or off
    turn on/off finite-field implementation
write_mat value = filename
    filename = file to which to write elastance matrix
write_inv value = filename
    filename = file to which to write inverted matrix
read_mat value = filename
    filename = file from which to read elastance matrix
read_inv value = filename
    filename = file from which to read inverted matrix
qtotal value = number or v_ equal-style variable
    add overall potential so that all electrode charges add up to qtotal
eta value = d_propname
    d_propname = a custom double vector defined via fix property/atom
```

### 2.63.2 Examples

```
fix fxconp bot electrode/conp -1.0 1.805 couple top 1.0 couple ref 0.0 write_inv inv.csv␣
↪symm on
fix fxconp electrodes electrode/conq 0.0 1.805 algo cg 1e-5
fix fxconp bot electrode/thermo -1.0 1.805 temp 298 100 couple top 1.0
```

### 2.63.3 Description

The *electrode* fixes implement the constant potential method (CPM) (*Siepmann*, *Reed*), and modern variants, to accurately model electrified, conductive electrodes. This is primarily useful for studying electrode-electrolyte interfaces, especially at high potential differences or ionicities, with non-planar electrodes such as nanostructures or nanopores, and to study dynamic phenomena such as charging or discharging time scales or conductivity or ionic diffusivities.

Each *electrode* fix allows users to set additional electrostatic relationships between the specified groups which model useful electrostatic configurations:

- *electrode/conp* sets potentials or potential differences between electrodes

  - (resulting in changing electrode total charges)
- *electrode/conq* sets the total charge on each electrode
  - (resulting in changing electrode potentials)
- *electrode/thermo* sets a thermopotentiostat *(Deissenbeck)* between two electrodes
  - (resulting in changing charges and potentials with appropriate average potential difference and thermal variance)

The first group-ID provided to each fix specifies the first electrode group, and more group(s) are added using the *couple* keyword for each additional group. While *electrode/thermo* only accepts two groups, *electrode/conp* and *electrode/conq* accept any number of groups, up to LAMMPS's internal restrictions (see Restrictions below). Electrode groups must not overlap, i.e. the fix will issue an error if any particle is detected to belong to at least two electrode groups.

CPM involves updating charges on groups of electrode particles, per time step, so that the system's total energy is minimized with respect to those charges. From basic electrostatics, this is equivalent to making each group conductive, or imposing an equal electrostatic potential on every particle in the same group (hence the name CPM). The charges are usually modelled as a Gaussian distribution to make the charge-charge interaction matrix invertible (*Gingrich*). The keyword *eta* specifies the distribution's width in units of inverse length.

Added in version 22Dec2022.

Three algorithms are available to minimize the energy, varying in how matrices are pre-calculated before a run to provide computational speedup. These algorithms can be selected using the keyword *algo*:

- *algo mat_inv* pre-calculates the capacitance matrix and obtains the charge configuration in one matrix-vector calculation per time step
- *algo mat_cg* pre-calculates the elastance matrix (inverse of capacitance matrix) and obtains the charge configuration using a conjugate gradient solver in multiple matrix-vector calculations per time step
- *algo cg* does not perform any pre-calculation and obtains the charge configuration using a conjugate gradient solver and multiple calculations of the electric potential per time step.

For both *cg* methods, the command must specify the conjugate gradient tolerance. *fix electrode/thermo* currently only supports the *mat_inv* algorithm.

The keyword *symm* can be set *on* (or *off*) to turn on (or turn off) the capacitance matrix constraint that sets total electrode charge to be zero. This has slightly different effects for each *fix electrode* variant. For *fix electrode/conp*, with *symm off*, the potentials specified are absolute potentials, but the charge configurations satisfying them may add up to an overall non-zero, varying charge for the electrodes (and thus the simulation box). With *symm on*, the total charge over all electrode groups is constrained to zero, and potential differences rather than absolute potentials are the physically relevant quantities.

For *fix electrode/conq*, with *symm off*, overall neutrality is explicitly obeyed or violated by the user input (which is not checked!). With *symm on*, overall neutrality is ensured by ignoring the user-input charge for the last listed electrode (instead, its charge will always be minus the total sum of all other electrode charges). For *fix electrode/thermo*, overall neutrality is always automatically imposed for any setting of *symm*, but *symm on* allows finite-field mode (*ffield on*, described below) for faster simulations.

For all three fixes, any potential (or charge for *conq*) can be specified as an equal-style variable prefixed with "v_". For example, the following code will ramp the potential difference between electrodes from 0.0V to 2.0V over the course of the simulation:

```
fix fxconp bot electrode/conp 0.0 1.805 couple top v_v symm on
variable v equal ramp(0.0, 2.0)
```

Note that these fixes only parse their supplied variable name when starting a run, and so these fixes will accept equal-style variables defined *after* the fix definition, including variables dependent on the fix's own output. This is useful,

for example, in the fix's internal finite-field commands (see below). For an advanced example of this see the in.conq2 input file in the directory `examples/PACKAGES/electrode/graph-il`.

This fix necessitates the use of a long range solver that calculates and provides the matrix of electrode-electrode interactions and a vector of electrode-electrolyte interactions. The Kspace styles *ewald/electrode*, *pppm/electrode* and *pppm/electrode/intel* are created specifically for this task *(Ahrens-Iwers)*.

For systems with non-periodic boundaries in one or two directions dipole corrections are available with the *kspace_modify*. For ewald/electrode a two-dimensional Ewald summation *(Hu)* can be used by setting "slab ew2d":

```
kspace_modify slab <slab_factor>
kspace_modify wire <wire_factor>
kspace_modify slab ew2d
```

Two implementations for the calculation of the elastance matrix are available with pppm and can be selected using the *amat onestep/twostep* keyword. *onestep* is the default; *twostep* can be faster for large electrodes and a moderate mesh size but requires more memory.

```
kspace_modify amat onestep/twostep
```

For all versions of the fix, the keyword-value *ffield on* enables the finite-field mode (*Dufils*, *Tee*), which uses an electric field across a periodic cell instead of non-periodic boundary conditions to impose a potential difference between the two electrodes bounding the cell. The fix (with name *fix-ID*) detects which of the two electrodes is "on top" (has the larger maximum $z$-coordinate among all particles). Assuming the first electrode group is on top, it then issues the following commands internally:

```
variable fix-ID_ffield_zfield equal (f_fix-ID[2]-f_fix-ID[1])/lz
efield fix-ID_efield all efield 0.0 0.0 v_fix-ID_ffield_zfield
```

which implements the required electric field as the potential difference divided by cell length. The internal commands use variable so that the electric field will correctly vary with changing potentials in the correct way (for example with equal-style potential difference or with *fix electrode/conq*). This keyword requires two electrodes and will issue an error with any other number of electrodes. This keyword requires electroneutrality to be imposed (*symm on*) and will issue an error otherwise.

Changed in version 22Dec2022.

For all versions of the fix, the keyword-value *etypes on* enables type-based optimized neighbor lists. With this feature enabled, LAMMPS provides the fix with an occasional neighbor list restricted to electrode-electrode interactions for calculating the electrode matrix, and a perpetual neighbor list restricted to electrode-electrolyte interactions for calculating the electrode potentials, using particle types to list only desired interactions, and typically resulting in 5–10% less computational time. Without this feature the fix will simply use the active pair style's neighbor list. This feature cannot be enabled if any electrode particle has the same type as any electrolyte particle (which would be unusual in a typical simulation) and the fix will issue an error in that case.

Added in version 17Apr2024.

The keyword *qtotal* causes *fix electrode/conp* and *fix electrode/thermo* to add an overall potential to all electrodes so that the total charge on the electrodes is a specified amount (which may be an equal-style variable). For example, if a user wanted to simulate a solution of excess cations such that the total electrolyte charge is +2, setting *qtotal -2* would cause the total electrode charge to be -2, so that the simulation box remains overall electroneutral. Since *fix electrode/conq* constrains the total charges of individual electrodes, and since *symm on* constrains the total charge of all electrodes to be zero, either option is incompatible with the *qtotal* keyword (even if *qtotal* is set to zero).

Added in version 17Apr2024.

The keyword *eta* takes the name of a custom double vector defined via fix property/atom. The values will be used instead of the standard eta value. The property/atom fix must be for vector of double values and use the *ghost on* option.

### 2.63.4 Restart, fix_modify, output, run start/stop, minimize info

This fix currently does not write any information to restart files.

The *fix_modify tf* option enables the Thomas-Fermi metallicity model (*Scalfi*) and allows parameters to be set for each atom type.

```
fix_modify ID tf type length voronoi
```

If this option is used, these two parameters must be set for all atom types of the electrode:

- *tf* is the Thomas-Fermi length $l_{TF}$
- *voronoi* is the Voronoi volume per atom in units of length cubed

Different types may have different *tf* and *voronoi* values. The following self-energy term is then added for all electrode atoms:

$$A_{ii} += \frac{1}{4\pi\varepsilon_0} \times \frac{4\pi l_{TF}^2}{\text{Voronoivolume}}$$

The *fix_modify timer* option turns on (off) additional timer outputs in the log file, for code developers to track optimization.

```
fix_modify ID timer on/off
```

---

These fixes compute a global (extensive) scalar, a global (intensive) vector, and a global array, which can be accessed by various *output commands*.

The global scalar outputs the energy added to the system by this fix, which is the negative of the total charge on each electrode multiplied by that electrode's potential.

The global vector outputs the potential on each electrode (and thus has $N$ entries if the fix manages $N$ electrode groups), in *units* of electric field multiplied by distance (thus volts for *real* and *metal* units). The electrode groups' ordering follows the order in which they were input in the fix command using *couple*. The global vector output is useful for *fix electrode/conq* and *fix electrode/thermo*, where potential is dynamically updated based on electrolyte configuration instead of being directly set.

The global array has $N$ rows and $2N+1$ columns, where the fix manages $N$ electrode groups managed by the fix. For the $I$-th row of the array, the elements are:

- array[I][1] = total charge that group $I$ would have had *if it were at 0 V applied potential*
- array[I][2 to $N$ + 1] = the $N$ entries of the $I$-th row of the electrode capacitance matrix (definition follows)
- array[I][$N$ + 2 to $2N$ + 1] = the $N$ entries of the $I$-th row of the electrode elastance matrix (the inverse of the electrode capacitance matrix)

The $N \times N$ electrode capacitance matrix, denoted $\mathbf{C}$ in the following equation, summarizes how the total charge induced on each electrode ($\mathbf{Q}$ as an $N$-vector) is related to the potential on each electrode, $\mathbf{V}$, and the charge-at-0V $\mathbf{Q}_{0V}$ (which is influenced by the local electrolyte structure):

$$\mathbf{Q} = \mathbf{Q}_{0V} + \mathbf{C} \cdot \mathbf{V}$$

The charge-at-0V, electrode capacitance and elastance matrices are internally used to calculate the potentials required to induce the specified total electrode charges in *fix electrode/conq* and *fix electrode/thermo*. With the *symm on* option, the electrode capacitance matrix would be singular, and thus its last row is replaced with $N$ copies of its top-left entry ($\mathbf{C}_{11}$) for invertibility.

The global array output is mainly useful for quickly determining the 'vacuum capacitance' of the system (capacitance with only electrodes, no electrolyte), and can also be used for advanced simulations setting the potential as some function of the charge-at-0V (such as the `in.conq2` example mentioned above).

Please cite *(Ahrens-Iwers2022)* in any publication that uses this implementation. Please cite also the publication on the combination of the CPM with PPPM if you use *pppm/electrode (Ahrens-Iwers)*.

---

### 2.63.5 Restrictions

For algorithms that use a matrix for the electrode-electrode interactions, positions of electrode particles have to be immobilized at all times.

With *ffield off* (i.e. the default), the box geometry is expected to be $z$-non-periodic (i.e. *boundary p p f*), and this fix will issue an error if the box is $z$-periodic. With *ffield on*, the box geometry is expected to be $z$-periodic, and this fix will issue an error if the box is $z$-non-periodic.

The parallelization for the fix works best if electrode atoms are evenly distributed across processors. For a system with two electrodes at the bottom and top of the cell this can be achieved with *processors * * 2*, or with the line

```
if "$(extract_setting(world_size) % 2) == 0" then "processors * * 2"
```

which avoids an error if the script is run on an odd number of processors (such as on just one processor for testing).

The fix creates an additional group named *[fix-ID]_group* which is the union of all electrode groups supplied to LAMMPS. This additional group counts towards LAMMPS's limitation on the total number of groups (currently 32), which may not allow scripts that use that many groups to run with this fix.

The matrix-based algorithms (*algo mat_inv* and *algo mat_cg*) currently store an interaction matrix (either elastance or capacitance) of $N$ by $N$ doubles for each MPI process. This memory requirement may be prohibitive for large electrode groups. The fix will issue a warning if it expects to use more than 0.5 GiB of memory.

### 2.63.6 Default

The default keyword-option settings are *algo mat_inv*, *symm off*, *etypes off* and *ffield off*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

**(Siepmann)** Siepmann and Sprik, J. Chem. Phys. 102, 511 (1995).

**(Reed)** Reed *et al.*, J. Chem. Phys. 126, 084704 (2007).

**(Deissenbeck)** Deissenbeck *et al.*, Phys. Rev. Letters 126, 136803 (2021).

**(Gingrich)** Gingrich, *MSc thesis* <https://gingrich.chem.northwestern.edu/papers/ThesiswCorrections.pdf>` (2010).

**(Ahrens-Iwers)** Ahrens-Iwers and Meissner, J. Chem. Phys. 155, 104104 (2021).

**(Hu)** Hu, J. Chem. Theory Comput. 10, 5254 (2014).

**(Dufils)** Dufils *et al.*, Phys. Rev. Letters 123, 195501 (2019).

**(Tee)** Tee and Searles, J. Chem. Phys. 156, 184101 (2022).

**(Scalfi)** Scalfi *et al.*, J. Chem. Phys., 153, 174704 (2020).

**(Ahrens-Iwers2022)** Ahrens-Iwers *et al.*, J. Chem. Phys. 157, 084801 (2022).

## 2.64 fix electron/stopping command

Accelerator Variants: *electron/stopping/kk*

## 2.65 fix electron/stopping/fit command

### 2.65.1 Syntax

```
fix ID group-ID style args
```

- ID, group-ID are documented in *fix* command
- style = *electron/stopping* or *electron/stopping/fit*

```
electron/stopping args = Ecut file keyword value ...
  Ecut  = minimum kinetic energy for electronic stopping (energy units)
  file  = name of the file containing the electronic stopping power table

electron/stopping/fit args = Ecut c1 c2 ...
  Ecut  = minimum kinetic energy for electronic stopping (energy units)
  c1 c2 = linear and quadratic coefficients for the fitted quadratic polynomial
```

- zero or more keyword/value pairs may be appended to args for style = *electron/stopping*

```
keyword = region or minneigh
   region value = region-ID
      region-ID = region whose atoms will be affected by this fix
   minneigh value = minneigh
      minneigh = minimum number of neighbors an atom to have stopping applied
```

### 2.65.2 Examples

```
fix el all electron/stopping 10.0 elstop-table.txt
fix el all electron/stopping 10.0 elstop-table.txt minneigh 3
fix el mygroup electron/stopping 1.0 elstop-table.txt region bulk
fix 1 all electron/stopping/fit 4.63 3.3e-3 4.0e-8
fix 1 all electron/stopping/fit 3.49 1.8e-3 9.0e-8 7.57 4.2e-3 5.0e-8
```

### 2.65.3 Description

This fix implements inelastic energy loss for fast projectiles in solids. It applies a friction force to fast moving atoms to slow them down due to *electronic stopping* (energy lost via electronic collisions per unit of distance). This fix should be used for simulation of irradiation damage or ion implantation, where the ions can lose noticeable amounts of energy from electron excitations. If the electronic stopping power is not considered, the simulated range of the ions can be severely overestimated (*Nordlund98*, *Nordlund95*).

The electronic stopping is implemented by applying a friction force to each atom as:

$$\vec{F}_i = \vec{F}_i^0 - \frac{\vec{v}_i}{\|\vec{v}_i\|} \cdot S_e$$

where $\vec{F}_i$ is the resulting total force on the atom. $\vec{F}_i^0$ is the original force applied to the atom, $\vec{v}_i$ is its velocity and $S_e$ is the stopping power of the ion.

**Note**

In addition to electronic stopping, atomic cascades and irradiation simulations require the use of an adaptive timestep (see *fix dt/reset*) and the repulsive ZBL potential (see *ZBL* potential) or similar. Without these settings the interaction between the ion and the target atoms will be faulty. It is also common to use in such simulations a thermostat (*fix_nvt*) in the borders of the simulation cell.

**Note**

This fix removes energy from fast projectiles without depositing it as a heat to the simulation cell. Such implementation might lead to the unphysical results when the amount of energy deposited to the electronic system is large, e.g. simulations of Swift Heavy Ions (energy per nucleon of 100 keV/amu or higher) or multiple projectiles. You could compensate energy loss by coupling bulk atoms with some thermostat or control heat transfer between electronic and atomic subsystems with the two-temperature model (*fix_ttm*).

At low velocities the electronic stopping is negligible. The electronic friction is not applied to atoms whose kinetic energy is smaller than *Ecut*, or smaller than the lowest energy value given in the table in *file*. Electronic stopping should be applied only when a projectile reaches bulk material. This fix scans neighbor list and excludes atoms with fewer than *minneigh* neighbors (by default one). If the pair potential cutoff is large, minneigh should be increased, though not above the number of nearest neighbors in bulk material. An alternative is to disable the check for neighbors by setting *minneigh* to zero and using the *region* keyword. This is necessary when running simulations of cluster bombardment.

If the *region* keyword is used, the atom must also be in the specified geometric *region* in order to have electronic stopping applied to it. This is useful if the position of the bulk material is fixed. By default the electronic stopping is applied everywhere in the simulation cell.

---

The energy ranges and stopping powers are read from the file *file*. Lines starting with # and empty lines are ignored. Otherwise each line must contain exactly **N+1** numbers, where **N** is the number of atom types in the simulation.

The first column is the energy for which the stopping powers on that line apply. The energies must be sorted from the smallest to the largest. The other columns are the stopping powers $S_e$ for each atom type, in ascending order, in force *units*. The stopping powers for intermediate energy values are calculated with linear interpolation between 2 nearest points.

For example:

```
# This is a comment
#       atom-1    atom-2
# eV    eV/Ang    eV/Ang  # units metal
 10        0         0
250       60        80
750      100       150
```

If an atom which would have electronic stopping applied to it has a kinetic energy higher than the largest energy given in *file*, LAMMPS will exit with an error message.

The stopping power depends on the energy of the ion and the target material. The electronic stopping table can be obtained from scientific publications, experimental databases or by using *SRIM* software. Other programs such as *CasP* can calculate the energy deposited as a function of the impact parameter of the ion; these results can be used to derive the stopping power.

---

Style *electron/stopping/fit* calculates the electronic stopping power and cumulative energy lost to the electron gas via a quadratic functional and applies a drag force to the classical equations-of-motion for all atoms moving above some minimum cutoff velocity (i.e., kinetic energy). These coefficients can be determined by fitting a quadratic polynomial to electronic stopping data predicted by, for example, SRIM or TD-DFT. Multiple 'Ecut c1 c2' values can be provided for multi-species simulations in the order of the atom types. There is an examples/PACKAGES/electron_stopping/ directory, which illustrates uses of this command. Details of this implementation are further described in *Stewart2018* and *Lee2020*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

**Note**

The region keyword is supported by Kokkos, but a Kokkos-enabled region must be used. See the region *region* command for more information.

---

### 2.65.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify* options are not supported.

This fix computes a global scalar, which can be accessed by various *output commands*. The scalar is the total energy loss from electronic stopping applied by this fix since the start of the latest run. It is considered "intensive".

The *start/stop* keywords of the *run* command have no effect on this fix.

### 2.65.5 Restrictions

This pair style is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* doc page for more info.

### 2.65.6 Default

The default is no limitation by region, and minneigh = 1.

---

**(electronic stopping)** Wikipedia - Electronic Stopping Power: https://en.wikipedia.org/wiki/Stopping_power_%28particle_radiation%29

**(Nordlund98)** Nordlund, Kai, et al. Physical Review B 57.13 (1998): 7556.

**(Nordlund95)** Nordlund, Kai. Computational materials science 3.4 (1995): 448-456.

**(SRIM)** SRIM webpage: http://www.srim.org/

**(CasP)** CasP webpage: https://lief.if.ufrgs.br/pub/CasP/

**(Stewart2018)** J.A. Stewart, et al. (2018) Journal of Applied Physics, 123(16), 165902.

**(Lee2020)** C.W. Lee, et al. (2020) Physical Review B, 102(2), 024107.

## 2.66 fix enforce2d command

Accelerator Variants: *enforce2d/kk*

### 2.66.1 Syntax

```
fix ID group-ID enforce2d
```

- ID, group-ID are documented in *fix* command
- enforce2d = style name of this fix command

### 2.66.2 Examples

```
fix 5 all enforce2d
```

### 2.66.3 Description

Zero out the z-dimension velocity and force on each atom in the group. This is useful when running a 2d simulation to ensure that atoms do not move from their initial z coordinate.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.66.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The forces due to this fix are imposed during an energy minimization, invoked by the *minimize* command.

### 2.66.5 Restrictions

none

### 2.66.6 Related commands

none

### 2.66.7 Default

none

## 2.67 fix eos/cv command

### 2.67.1 Syntax

```
fix ID group-ID eos/cv cv
```

- ID, group-ID are documented in *fix* command
- eos/cv = style name of this fix command
- cv = constant-volume heat capacity (energy/temperature units)

### 2.67.2 Examples

```
fix 1 all eos/cv 0.01
```

### 2.67.3 Description

Fix *eos/cv* applies a mesoparticle equation of state to relate the particle internal energy ($u_i$) to the particle internal temperature ($\theta_i$). The *eos/cv* mesoparticle equation of state requires the constant-volume heat capacity, and is defined as follows:

$$u_i = u_i^{mech} + u_i^{cond} = C_V \theta_i$$

where $C_V$ is the constant-volume heat capacity, $u^{cond}$ is the internal conductive energy, and $u^{mech}$ is the internal mechanical energy. Note that alternative definitions of the mesoparticle equation of state are possible.

---

### 2.67.4 Restrictions

This command is part of the DPD-REACT package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This command also requires use of the *atom_style dpd* command.

### 2.67.5 Related commands

*fix shardlow*, *pair dpd/fdt*

### 2.67.6 Default

none

---

**(Larentzos)** J.P. Larentzos, J.K. Brennan, J.D. Moore, and W.D. Mattson, "LAMMPS Implementation of Constant Energy Dissipative Particle Dynamics (DPD-E)", ARL-TR-6863, U.S. Army Research Laboratory, Aberdeen Proving Ground, MD (2014).

## 2.68 fix eos/table command

### 2.68.1 Syntax

```
fix ID group-ID eos/table style file N keyword
```

- ID, group-ID are documented in *fix* command
- eos/table = style name of this fix command
- style = *linear* = method of interpolation
- file = filename containing the tabulated equation of state
- N = use N values in *linear* tables
- keyword = name of table keyword corresponding to table file

### 2.68.2 Examples

```
fix 1 all eos/table linear eos.table 100000 KEYWORD
```

### 2.68.3 Description

Fix *eos/table* applies a tabulated mesoparticle equation of state to relate the particle internal energy (u_i) to the particle internal temperature (dpdTheta_i).

Fix *eos/table* creates interpolation tables of length *N* from internal energy values listed in a file as a function of internal temperature.

The interpolation tables are created by fitting cubic splines to the file values and interpolating energy values at each of *N* internal temperatures, and vice versa. During a simulation, these tables are used to interpolate internal energy or temperature values as needed. The interpolation is done with the *linear* style.

For the *linear* style, the internal temperature is used to find 2 surrounding table values from which an internal energy is computed by linear interpolation, and vice versa.

The filename specifies a file containing tabulated internal temperature and internal energy values. The keyword specifies a section of the file. The format of this file is described below.

---

The format of a tabulated file is as follows (without the parenthesized comments):

```
# EOS TABLE                (one or more comment or blank lines)

KEYWORD                    (keyword is first text on line)
N 500                      (N  parameter)
                           (blank)
1   1.00 0.000             (index, internal temperature, internal energy)
2   1.02 0.001
...
500 10.0 0.500
```

A section begins with a non-blank line whose first character is not a "#"; blank lines or lines starting with "#" can be used as comments between sections. The first line begins with a keyword which identifies the section. The line can contain additional text, but the initial text must match the argument specified in the fix command.

The next line lists the number of table entries. The parameter "N" is required and its value is the number of table entries that follow. Note that this may be different than the *N* specified in the *fix eos/table* command. Let Ntable = *N* in the fix command, and Nfile = "N" in the tabulated file. What LAMMPS does is a preliminary interpolation by creating splines using the Nfile tabulated values as nodal points. It uses these to interpolate as needed to generate energy and temperature values at Ntable different points. The resulting tables of length Ntable are then used as described above, when computing energy and temperature relationships. This means that if you want the interpolation tables of length Ntable to match exactly what is in the tabulated file (with effectively no preliminary interpolation), you should set Ntable = Nfile.

Following a blank line, the next N lines list the tabulated values. On each line, the first value is the index from 1 to N, the second value is the internal temperature (in temperature units), the third value is the internal energy (in energy units).

Note that the internal temperature and internal energy values must increase from one line to the next.

Note that one file can contain many sections, each with a tabulated potential. LAMMPS reads the file section by section until it finds one that matches the specified keyword.

---

### 2.68.4 Restrictions

This command is part of the DPD-REACT package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This command also requires use of the *atom_style dpd* command.

The equation of state must be a monotonically increasing function.

An error will occur if the internal temperature or internal energies are not within the table cutoffs.

### 2.68.5 Related commands

*fix shardlow*, *pair dpd/fdt*

### 2.68.6 Default

none

## 2.69 fix eos/table/rx command

Accelerator Variants: *eos/table/rx/kk*

### 2.69.1 Syntax

```
fix ID group-ID eos/table/rx style file1 N keyword ...
```

- ID, group-ID are documented in *fix* command
- eos/table/rx = style name of this fix command
- style = *linear* = method of interpolation
- file1 = filename containing the tabulated equation of state
- N = use N values in *linear* tables
- keyword = name of table keyword corresponding to table file
- file2 = filename containing the heats of formation of each species (optional)
- deltaHf = heat of formation for a single species in energy units (optional)
- energyCorr = energy correction in energy units (optional)
- tempCorrCoeff = temperature correction coefficient (optional)

### 2.69.2 Examples

```
fix 1 all eos/table/rx linear eos.table 10000 KEYWORD thermo.table
fix 1 all eos/table/rx linear eos.table 10000 KEYWORD 1.5
fix 1 all eos/table/rx linear eos.table 10000 KEYWORD 1.5 0.025 0.0
```

### 2.69.3 Description

Fix *eos/table/rx* applies a tabulated mesoparticle equation of state to relate the concentration-dependent particle internal energy ($u_i$) to the particle internal temperature ($\theta_i$).

The concentration-dependent particle internal energy ($u_i$) is computed according to the following relation:

$$U_i = \sum_{j=1}^{m} c_{i,j}(u_j + \Delta H_{f,j}) + \frac{3k_BT}{2} + Nk_BT$$

where $m$ is the number of species, $c_{i,j}$ is the concentration of species $j$ in particle $i$, $u_j$ is the internal energy of species j, $\Delta H_{f,j}$ is the heat of formation of species $j$, N is the number of molecules represented by the coarse-grained particle, $k_B$ is the Boltzmann constant, and $T$ is the temperature of the system. Additionally, it is possible to modify the concentration-dependent particle internal energy relation by adding an energy correction, temperature-dependent correction, and/or a molecule-dependent correction. An energy correction can be specified as a constant (in energy units). A temperature correction can be specified by multiplying a temperature correction coefficient by the internal temperature. A molecular correction can be specified by by multiplying a molecule correction coefficient by the average number of product gas particles in the coarse-grain particle.

Fix *eos/table/rx* creates interpolation tables of length $N$ from $m$ internal energy values of each species $u_j$ listed in a file as a function of internal temperature. During a simulation, these tables are used to interpolate internal energy or temperature values as needed. The interpolation is done with the *linear* style. For the *linear* style, the internal temperature is used to find 2 surrounding table values from which an internal energy is computed by linear interpolation. A secant solver is used to determine the internal temperature from the internal energy.

The first filename specifies a file containing tabulated internal temperature and $m$ internal energy values for each species $u_j$. The keyword specifies a section of the file. The format of this file is described below.

The second filename specifies a file containing heat of formation $\Delta H_{f,j}$ for each species.

In cases where the coarse-grain particle represents a single molecular species (i.e., no reactions occur and fix *rx* is not present in the input file), fix *eos/table/rx* can be applied in a similar manner to fix *eos/table* within a non-reactive DPD simulation. In this case, the heat of formation filename is replaced with the heat of formation value for the single species. Additionally, the energy correction and temperature correction coefficients may also be specified as fix arguments.

---

The format of a tabulated file is as follows (without the parenthesized comments):

```
# EOS TABLE                (one or more comment or blank lines)

KEYWORD                    (keyword is first text on line)
N 500 h2 no2 n2 ... no     (N  parameter species1 species2 ... speciesN)
                           (blank)
1   1.00 0.000 ... 0.0000  (index, internal temperature, internal energy of species 1, ..
↪., internal energy of species m)
2   1.02 0.001 ... 0.0002
...
500 10.0 0.500 ... 1.0000
```

A section begins with a non-blank line whose first character is not a "#"; blank lines or lines starting with "#" can be used as comments between sections. The first line begins with a keyword which identifies the section. The line can contain additional text, but the initial text must match the argument specified in the fix command.

The next line lists the number of table entries and the species names that correspond with all the species listed in the reaction equations through the *fix rx* command. The parameter "N" is required and its value is the number of table entries that follow. Let Nfile = "N" in the tabulated file. What LAMMPS does is a preliminary interpolation by creating splines using the Nfile tabulated values as nodal points.

Following a blank line, the next N lines list the tabulated values. On each line, the first value is the index from 1 to N, the second value is the internal temperature (in temperature units), the third value until the *m+3* value are the internal energies of the m species (in energy units).

Note that all internal temperature and internal energy values must increase from one line to the next.

Note that one file can contain many sections, each with a tabulated potential. LAMMPS reads the file section by section until it finds one that matches the specified keyword.

---

The format of a heat of formation file is as follows (without the parenthesized comments):

```
# HEAT OF FORMATION TABLE  (one or more comment or blank lines)

                           (blank)
h2      0.00               (species name, heat of formation)
no2     0.34
n2      0.00
...
no      0.93
```

Note that the species can be listed in any order. The tag that is used as the species name must correspond with the tags used to define the reactions with the *fix rx* command.

Alternatively, corrections to the EOS can be included by specifying three additional columns that correspond to the energy correction, the temperature correction coefficient and molecule correction coefficient. In this case, the format of the file is as follows:

```
# HEAT OF FORMATION TABLE     (one or more comment or blank lines)

                              (blank)
h2      0.00 1.23 0.025  0.0  (species name, heat of formation, energy correction,␣
↪temperature correction coefficient, molecule correction coefficient)
no2     0.34 0.00 0.000 -1.76
n2      0.00 0.00 0.000 -1.76
...
no      0.93 0.00 0.000 -1.76
```

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.69.4 Restrictions

This command is part of the DPD-REACT package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This command also requires use of the *atom_style dpd* command.

The equation of state must be a monotonically increasing function.

An error will occur if the internal temperature or internal energies are not within the table cutoffs.

### 2.69.5 Related commands

*fix rx*, *pair dpd/fdt*

### 2.69.6 Default

none

## 2.70 fix evaporate command

### 2.70.1 Syntax

```
fix ID group-ID evaporate N M region-ID seed
```

- ID, group-ID are documented in *fix* command
- evaporate = style name of this fix command
- N = delete atoms every this many timesteps

- M = number of atoms to delete each time
- region-ID = ID of region within which to perform deletions
- seed = random number seed to use for choosing atoms to delete
- zero or more keyword/value pairs may be appended

```
keyword = molecule
  molecule value = no or yes
```

### 2.70.2 Examples

```
fix 1 solvent evaporate 1000 10 surface 49892
fix 1 solvent evaporate 1000 10 surface 38277 molecule yes
```

### 2.70.3 Description

Remove M atoms from the simulation every N steps. This can be used, for example, to model evaporation of solvent particles or molecules (i.e. drying) of a system. Every N steps, the number of atoms in the fix group and within the specified region are counted. M of these are chosen at random and deleted. If there are less than M eligible particles, then all of them are deleted.

If the setting for the *molecule* keyword is *no*, then only single atoms are deleted. In this case, you should ensure you do not delete only a portion of a molecule (only some of its atoms), or LAMMPS will soon generate an error when it tries to find those atoms. LAMMPS will warn you if any of the atoms eligible for deletion have a non-zero molecule ID, but does not check for this at the time of deletion.

If the setting for the *molecule* keyword is *yes*, then when an atom is chosen for deletion, the entire molecule it is part of is deleted. The count of deleted atoms is incremented by the number of atoms in the molecule, which may make it exceed *M*. If the molecule ID of the chosen atom is 0, then it is assumed to not be part of a molecule, and just the single atom is deleted.

As an example, if you wish to delete 10 water molecules every *N* steps, you should set *M* to 30. If only the water's oxygen atoms were in the fix group, then two hydrogen atoms would be deleted when an oxygen atom is selected for deletion, whether the hydrogen atoms are inside the evaporation region or not.

Note that neighbor lists are re-built on timesteps that atoms are removed. Thus you should not remove atoms too frequently or you will incur overhead due to the cost of building neighbor lists.

**Note**

If you are monitoring the temperature of a system where the atom count is changing due to evaporation, you typically should use the *compute_modify dynamic/dof yes* command for the temperature compute you are using.

### 2.70.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix computes a global scalar, which can be accessed by various *output commands*. The scalar is the cumulative number of deleted atoms. The scalar value calculated by this fix is "intensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.70.5 Restrictions

None

### 2.70.6 Related commands

*fix deposit*

### 2.70.7 Default

The option defaults are molecule = no.

## 2.71 fix external command

Accelerator Variants: *external/kk*

### 2.71.1 Syntax

```
fix ID group-ID external mode args
```

- ID, group-ID are documented in *fix* command
- external = style name of this fix command
- mode = *pf/callback* or *pf/array*

```
pf/callback args = Ncall Napply
  Ncall = make callback every Ncall steps
  Napply = apply callback forces every Napply steps
pf/array args = Napply
  Napply = apply array forces every Napply steps
```

### 2.71.2 Examples

```
fix 1 all external pf/callback 1 1
fix 1 all external pf/callback 100 1
fix 1 all external pf/array 10
```

### 2.71.3 Description

This fix allows external programs that are running LAMMPS through its *library interface* to modify certain LAMMPS properties on specific timesteps, similar to the way other fixes do. The external driver can be a *C/C++ or Fortran program* or a *Python script*.

---

If mode is *pf/callback* then the fix will make a callback every *Ncall* timesteps or minimization iterations to the external program. The external program computes forces on atoms by setting values in an array owned by the fix. The fix then adds these forces to each atom in the group, once every *Napply* steps, similar to the way the *fix addforce* command works. Note that if *Ncall* > *Napply*, the force values produced by one callback will persist, and be used multiple times to update atom forces.

The callback function "foo" is invoked by the fix as:

```
foo(void *ptr, bigint timestep, int nlocal, tagint *ids, double **x, double **fexternal);
```

The arguments are as follows:

- *ptr* = pointer provided by and simply passed back to external driver
- *timestep* = current LAMMPS timestep
- *nlocal* = # of atoms on this processor
- *ids* = list of atom IDs on this processor
- *x* = coordinates of atoms on this processor
- *fexternal* = forces to add to atoms on this processor

Note that *timestep* is a "bigint" which is defined in src/lmptype.h, typically as a 64-bit integer. And *ids* is a pointer to type "tagint" which is typically a 32-bit integer unless LAMMPS is compiled with -DLAMMPS_BIGBIG. For more info please see the *build settings* section of the manual. Finally, *fexternal* are the forces returned by the driver program.

The best way to set up the callback function is to use the C-language library interface function *lammps_set_fix_external_callback()*.

---

If mode is *pf/array* then the fix simply stores force values in an array. The fix adds these forces to each atom in the group, once every *Napply* steps, similar to the way the *fix addforce* command works.

The name of the public force array provided by the FixExternal class is

```
double **fexternal;
```

It is allocated by the FixExternal class as an (N,3) array where N is the number of atoms owned by a processor. The 3 corresponds to the fx, fy, fz components of force.

It is up to the external program to set the values in this array to the desired quantities, as often as desired. For example, the driver program might perform an MD run in stages of 1000 timesteps each. In between calls to the LAMMPS *run* command, it could retrieve atom coordinates from LAMMPS, compute forces, set values in fexternal, etc.

---

To use this fix during energy minimization, the energy corresponding to the added forces must also be set so as to be consistent with the added forces. Otherwise the minimization will not converge correctly. Correspondingly, the global virial needs to be updated to be use this fix with variable cell calculations (e.g. *fix box/relax* or *fix npt*).

This can be done from the external driver by calling these public methods of the FixExternal class:

```
void set_energy_global(double eng);
void set_virial_global(double *virial);
```

where *eng* is the potential energy, and *virial* an array of the 6 stress tensor components. Eng is an extensive quantity, meaning it should be the sum over per-atom energies of all affected atoms. It should also be provided in *energy units* consistent with the simulation. See the details below for how to ensure this energy setting is used appropriately in a minimization.

Additional public methods that the caller can use to update system properties are:

```
void set_energy_peratom(double *eng);
void set_virial_peratom(double **virial);
void set_vector_length(int n);
void set_vector(int idx, double val);
```

These enable setting per-atom energy and per-atom stress contributions, the length and individual values of a global vector of properties that the caller code may want to communicate to LAMMPS (e.g. for use in *fix ave/time* or in *equal-style variables* or for *custom thermo output*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.71.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify energy* option is supported by this fix to add the potential energy set by the external driver to both the global potential energy and peratom potential energies of the system as part of *thermodynamic output* or output by the *compute pe/atom* command. The default setting for this fix is *fix_modify energy yes*. Note that this energy may be a fictitious quantity but it is needed so that the *minimize* command can include the forces added by this fix in a consistent manner. I.e. there is a decrease in potential energy when atoms move in the direction of the added force.

The *fix_modify virial* option is supported by this fix to add the contribution computed by the external program to both the global pressure and per-atom stress of the system via the *compute pressure* and *compute stress/atom* commands. The former can be accessed by *thermodynamic output*. The default setting for this fix is *fix_modify virial yes*.

This fix computes a global scalar, a global vector, and a per-atom array which can be accessed by various *output commands*. The scalar is the potential energy discussed above. The scalar stored by this fix is "extensive". The global vector has a custom length and needs to be set by the external program using the `lammps_fix_external_set_vector()` and `lammps_fix_external_set_vector_length()` calls of the LAMMPS library interface or the equivalent call of the Python or Fortran modules. The per-atom array has 3 values for each atom and is the applied external force.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The forces due to this fix are imposed during an energy minimization, invoked by the *minimize* command.

**Note**

If you want the fictitious potential energy associated with the added forces to be included in the total potential energy of the system (the quantity being minimized), you MUST not disable the *fix_modify energy* option for this fix.

### 2.71.5 Restrictions

none

### 2.71.6 Related commands

none

### 2.71.7 Default

none

## 2.72 fix ffl command

### 2.72.1 Syntax

```
fix ID id-group ffl tau Tstart Tstop seed [flip-type]
```

- ID, group-ID are documented in *fix* command
- ffl = style name of this fix command
- tau = thermostat parameter (positive real)
- Tstart, Tstop = temperature ramp during the run
- seed = random number seed to use for generating noise (positive integer)
- one more value may be appended

```
flip-type  = determines the flipping type, can be chosen between rescale - no_flip -
↪ hard - soft, if no flip type is given, rescale will be chosen by default
```

### 2.72.2 Examples

```
fix 3 boundary ffl 10 300 300 31415
fix 1 all ffl 100 500 500 9265 soft
```

### 2.72.3 Description

Apply a Fast-Forward Langevin Equation (FFL) thermostat as described in *(Hijazi)*. Contrary to *fix langevin*, this fix performs both thermostatting and evolution of the Hamiltonian equations of motion, so it should not be used together with *fix nve* – at least not on the same atom groups.

The time-evolution of a single particle undergoing Langevin dynamics is described by the equations

$$\frac{dq}{dt} = \frac{p}{m},$$

$$\frac{dp}{dt} = -\gamma p + W + F,$$

where $F$ is the physical force, $\gamma$ is the friction coefficient, and $W$ is a Gaussian random force.

The friction coefficient is the inverse of the thermostat parameter : $\gamma = 1/\tau$, with $\tau$ the thermostat parameter *tau*. The thermostat parameter is given in the time units, $\gamma$ is in inverse time units.

Equilibrium sampling a temperature T is obtained by specifying the target value as the *Tstart* and *Tstop* arguments, so that the internal constants depending on the temperature are computed automatically.

The random number *seed* must be a positive integer. A Marsaglia random number generator is used. Each processor uses the input seed to generate its own unique seed and its own stream of random numbers. Thus the dynamics of the system will not be identical on two runs on different numbers of processors.

The flipping type *flip-type* can be chosen between 4 types described in *(Hijazi)*. The flipping operation occurs during the thermostatting step and it flips the momenta of the atoms. If no_flip is chosen, no flip will be executed and the integration will be the same as a standard Langevin thermostat *(Bussi)*. The other flipping types are : rescale - hard - soft.

### 2.72.4 Restart, fix_modify, output, run start/stop, minimize info

The instantaneous values of the extended variables are written to *binary restart files*. Because the state of the random number generator is not saved in restart files, this means you cannot do "exact" restarts with this fix, where the simulation continues on the same as if no restart had taken place. However, in a statistical sense, a restarted simulation should produce the same behavior. Note however that you should use a different seed each time you restart, otherwise the same sequence of random numbers will be used each time, which might lead to stochastic synchronization and subtle artifacts in the sampling.

The cumulative energy change in the system imposed by this fix is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* doc page for details.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the same cumulative energy change due to this fix described in the previous paragraph. The scalar value calculated by this fix is "extensive".

This fix can ramp its target temperature over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.72.5 Restrictions

In order to perform constant-pressure simulations please use *fix press/berendsen*, rather than *fix npt*, to avoid duplicate integration of the equations of motion.

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.72.6 Related commands

*fix nvt*, *fix temp/rescale*, *fix viscous*, *fix nvt*, *pair_style dpd/tstat*, *fix gld*, *fix gle*

---

**(Hijazi)** M. Hijazi, D. M. Wilkins, M. Ceriotti, J. Chem. Phys. 148, 184109 (2018)

**(Bussi)** G. Bussi, M. Parrinello, Phs. Rev. E 75, 056707 (2007)

## 2.73 fix filter/corotate command

### 2.73.1 Syntax

```
fix ID group-ID filter/corotate keyword value ...
```

- ID, group-ID are documented in *fix* command
- one or more constraint/value pairs are appended
- constraint = *b* or *a* or *t* or *m*

```
b values = one or more bond types
a values = one or more angle types
t values = one or more atom types
m value = one or more mass values
```

### 2.73.2 Examples

```
timestep 8
run_style respa 3 2 8 bond 1 pair 2 kspace 3
fix cor all filter/corotate m 1.0

fix cor all filter/corotate b 4 19 a 3 5 2
```

### 2.73.3 Description

This fix implements a corotational filter for a mollified impulse method. In biomolecular simulations, it allows the usage of larger timesteps for long-range electrostatic interactions. For details, see *(Fath)*.

When using *run_style respa* for a biomolecular simulation with high-frequency covalent bonds, the outer time-step is restricted to below ~ 4fs due to resonance problems. This fix filters the outer stage of the respa and thus a larger (outer) time-step can be used. Since in large biomolecular simulations the computation of the long-range electrostatic contributions poses a major bottleneck, this can significantly accelerate the simulation.

The filter computes a cluster decomposition of the molecular structure following the criteria indicated by the options a, b, t and m. This process is similar to the approach in *fix shake*, however, the clusters are not kept constrained. Instead, the position is slightly modified only for the computation of long-range forces. A good cluster decomposition constitutes in building clusters which contain the fastest covalent bonds inside clusters.

If the clusters are chosen suitably, the *run_style respa* is stable for outer timesteps of at least 8fs.

---

### 2.73.4 Restart, fix_modify, output, run start/stop, minimize info

No information about these fixes is written to *binary restart files*. None of the *fix_modify* options are relevant to these fixes. No global or per-atom quantities are stored by these fixes for access by various *output commands*. No parameter of these fixes can be used with the *start/stop* keywords of the *run* command. These fixes are not invoked during *energy minimization*.

### 2.73.5 Restrictions

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Currently, it does not support *molecule templates*.

### 2.73.6 Related commands

### 2.73.7 Default

none

---

**(Fath)** Fath, Hochbruck, Singh, J Comp Phys, 333, 180-198 (2017).

## 2.74 fix flow/gauss command

### 2.74.1 Syntax

```
fix ID group-ID flow/gauss xflag yflag zflag keyword
```

- ID, group-ID are documented in *fix* command
- flow/gauss = style name of this fix command
- xflag,yflag,zflag = 0 or 1

```
0 = do not conserve current in this dimension
1 = conserve current in this dimension
```

- zero or more keyword/value pairs may be appended
- keyword = *energy*

```
energy value = no or yes
  no = do not compute work done by this fix
  yes = compute work done by this fix
```

### 2.74.2 Examples

```
fix GD fluid flow/gauss 1 0 0
fix GD fluid flow/gauss 1 1 1 energy yes
```

### 2.74.3 Description

This fix implements the Gaussian dynamics (GD) method to simulate a system at constant mass flux *(Strong)*. GD is a nonequilibrium molecular dynamics simulation method that can be used to study fluid flows through pores, pipes, and channels. In its original implementation GD was used to compute the pressure required to achieve a fixed mass flux through an opening. The flux can be conserved in any combination of the directions, x, y, or z, using xflag,yflag,zflag. This fix does not initialize a net flux through a system, it only conserves the center-of-mass momentum that is present when the fix is declared in the input script. Use the *velocity* command to generate an initial center-of-mass momentum.

GD applies an external fluctuating gravitational field that acts as a driving force to keep the system away from equilibrium. To maintain steady state, a profile-unbiased thermostat must be implemented to dissipate the heat that is added by the driving force. *Compute temp/profile* can be used to implement a profile-unbiased thermostat.

A common use of this fix is to compute a pressure drop across a pipe, pore, or membrane. The pressure profile can be computed in LAMMPS with *compute stress/atom* and *fix ave/chunk*. Note that the simple *compute stress/atom* method is only accurate away from inhomogeneities in the fluid, such as fixed wall atoms. Further, the computed pressure profile must be corrected for the acceleration applied by GD before computing a pressure drop or comparing it to other methods, such as the pump method *(Zhu)*. The pressure correction is discussed and described in *(Strong)*.

For a complete example including the considerations discussed above, see the examples/PACKAGES/flow_gauss directory.

**Note**

Only the flux of the atoms in group-ID will be conserved. If the velocities of the group-ID atoms are coupled to the velocities of other atoms in the simulation, the flux will not be conserved. For example, in a simulation with fluid atoms and harmonically constrained wall atoms, if a single thermostat is applied to group *all*, the fluid atom velocities will be coupled to the wall atom velocities, and the flux will not be conserved. This issue can be avoided by thermostatting the fluid and wall groups separately.

Adding an acceleration to atoms does work on the system. This added energy can be optionally subtracted from the potential energy for the thermodynamic output (see below) to check that the timestep is small enough to conserve energy. Since the applied acceleration is fluctuating in time, the work cannot be computed from a potential. As a result, computing the work is slightly more computationally expensive than usual, so it is not performed by default. To

invoke the work calculation, use the *energy* keyword. The *fix_modify energy* option also invokes the work calculation, and overrides an *energy no* setting here. If neither *energy yes* or *fix_modify energy yes* are set, the global scalar computed by the fix will return zero.

**Note**

In order to check energy conservation, any other fixes that do work on the system must have *fix_modify energy yes* set as well. This includes thermostat fixes and any constraints that hold the positions of wall atoms fixed, such as *fix spring/self*.

If this fix is used in a simulation with the *rRESPA* integrator, the applied acceleration must be computed and applied at the same rRESPA level as the interactions between the flowing fluid and the obstacle. The rRESPA level at which the acceleration is applied can be changed using the *fix_modify respa* option discussed below. If the flowing fluid and the obstacle interact through multiple interactions that are computed at different rRESPA levels, then there must be a separate flow/gauss fix for each level. For example, if the flowing fluid and obstacle interact through pairwise and long-range Coulomb interactions, which are computed at rRESPA levels 3 and 4, respectively, then there must be two separate flow/gauss fixes, one that specifies *fix_modify respa 3* and one with *fix_modify respa 4*.

---

### 2.74.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify energy* option is supported by this fix to add the potential energy added by the fix to the global potential energy of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify energy no*.

The *fix_modify respa* option is supported by this fix. This allows the user to set at which level of the *rRESPA* integrator the fix computes and adds the external acceleration. Default is the outermost level.

This fix computes a global scalar and a global 3-vector of forces, which can be accessed by various *output commands*. The scalar is the negative of the work done on the system, see the discussion above. It is only calculated if the *energy* keyword is enabled or *fix_modify energy yes* is set.

The vector is the total force that this fix applied to the group of atoms on the current timestep. The scalar and vector values calculated by this fix are "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

This fix is not invoked during *energy minimization*.

### 2.74.5 Restrictions

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.74.6 Related commands

*fix addforce*, *compute temp/profile*, *velocity*

### 2.74.7 Default

The option default for the *energy* keyword is energy = no.

---

**(Strong)** Strong and Eaves, J. Phys. Chem. B 121, 189 (2017).

**(Evans)** Evans and Morriss, Phys. Rev. Lett. 56, 2172 (1986).

**(Zhu)** Zhu, Tajkhorshid, and Schulten, Biophys. J. 83, 154 (2002).

## 2.75 fix freeze command

Accelerator Variants: *freeze/kk*

### 2.75.1 Syntax

```
fix ID group-ID freeze
```

- ID, group-ID are documented in *fix* command
- freeze = style name of this fix command

### 2.75.2 Examples

```
fix 2 bottom freeze
```

### 2.75.3 Description

Zero out the force and torque on a granular particle. This is useful for preventing certain particles from moving in a simulation. The *granular pair styles* also detect if this fix has been defined and compute interactions between frozen and non-frozen particles appropriately, as if the frozen particle has infinite mass. A similar functionality for normal (point) particles can be obtained using *fix setforce*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.75.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix computes a global 3-vector of forces, which can be accessed by various *output commands*. This is the total force on the group of atoms before the forces on individual atoms are changed by the fix. The vector values calculated by this fix are “extensive”.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.75.5 Restrictions

This fix is part of the GRANULAR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

There can only be a single freeze fix defined. This is because other the *granular pair styles* treat frozen particles differently and need to be able to reference a single group to which this fix is applied.

### 2.75.6 Related commands

*atom_style sphere*, *fix setforce*

### 2.75.7 Default

none

## 2.76 fix gcmc command

### 2.76.1 Syntax

```
fix ID group-ID gcmc N X M type seed T mu displace keyword values ...
```

- ID, group-ID are documented in *fix* command
- gcmc = style name of this fix command
- N = invoke this fix every N steps
- X = average number of GCMC exchanges to attempt every N steps
- M = average number of MC moves to attempt every N steps
- type = atom type (1-Ntypes or type label) for inserted atoms (must be 0 if mol keyword used)
- seed = random # seed (positive integer)
- T = temperature of the ideal gas reservoir (temperature units)
- mu = chemical potential of the ideal gas reservoir (energy units)
- displace = maximum Monte Carlo translation distance (length units)
- zero or more keyword/value pairs may be appended to args

```
keyword = mol or mcmoves or rigid or shake or region or maxangle or pressure or␣
↪fugacity_coeff or full_energy or charge or group or grouptype or intra_energy or␣
↪tfac_insert or overlap_cutoff or max or min
  mol value = template-ID
    template-ID = ID of molecule template specified in a separate molecule command
  mcmoves values = Patomtrans Pmoltrans Pmolrotate
    Patomtrans = proportion of atom translation MC moves
    Pmoltrans = proportion of molecule translation MC moves
    Pmolrotate = proportion of molecule rotation MC moves
  rigid value = fix-ID
    fix-ID = ID of fix rigid/small command
  shake value = fix-ID
    fix-ID = ID of fix shake command
  region value = region-ID
    region-ID = ID of region where GCMC exchanges and MC moves are allowed
  maxangle value = maximum molecular rotation angle (degrees)
```

```
  pressure value = pressure of the gas reservoir (pressure units)
  fugacity_coeff value = fugacity coefficient of the gas reservoir (unitless)
  full_energy = compute the entire system energy when performing GCMC exchanges and␣
↪MC moves
  charge value = charge of inserted atoms (charge units)
  group value = group-ID
    group-ID = group-ID for inserted atoms (string)
  grouptype values = type group-ID
    type = atom type (1-Ntypes or type label)
    group-ID = group-ID for inserted atoms (string)
  intra_energy value = intramolecular energy (energy units)
  tfac_insert value = scale up/down temperature of inserted atoms (unitless)
  overlap_cutoff value = maximum pair distance for overlap rejection (distance␣
↪units)
  max value = Maximum number of atoms allowed in the fix group (and region)
  min value = Minimum number of atoms allowed in the fix group (and region)
```

### 2.76.2 Examples

```
fix 2 gas gcmc 10 1000 1000 2 29494 298.0 -0.5 0.01
fix 3 water gcmc 10 100 100 0 3456543 3.0 -2.5 0.1 mol my_one_water maxangle 180 full_
↪energy
fix 4 my_gas gcmc 1 10 10 1 123456543 300.0 -12.5 1.0 region disk

labelmap atom 1 Li
fix 2 ion gcmc 10 1000 1000 Li 29494 298.0 -0.5 0.01
```

### 2.76.3 Description

This fix performs grand canonical Monte Carlo (GCMC) exchanges of atoms or molecules with an imaginary ideal gas reservoir at the specified $T$ and chemical potential ($\mu$) as discussed in *(Frenkel)*. It also attempts Monte Carlo (MC) moves (translations and molecule rotations) within the simulation cell or region. If used with the *fix nvt* command, simulations in the grand canonical ensemble (muVT, constant chemical potential, constant volume, and constant temperature) can be performed. Specific uses include computing isotherms in microporous materials, or computing vapor-liquid coexistence curves.

Every $N$ timesteps the fix attempts both GCMC exchanges (insertions or deletions) and MC moves of gas atoms or molecules. On those timesteps, the average number of attempted GCMC exchanges is $X$, while the average number of attempted MC moves is $M$. For GCMC exchanges of either molecular or atomic gasses, these exchanges can be either deletions or insertions, with equal probability.

**Note**

To run an MC-only simulation (no MD), you should define no time-integration fix, set the *thermo* command to 1, set $N$ to 1, and set $X$+$M$ small enough to see the MC evolution of the system. But if $X$+$M$ is too small, the overhead at the start and stop of MC moves each timestep will slow down the simulation.

The possible choices for MC moves are translation of an atom, translation of a molecule, and rotation of a molecule. The relative amounts of each are determined by the optional *mcmoves* keyword (see below). The default behavior is as follows. If the *mol* keyword is used, only molecule translations and molecule rotations are performed with equal probability. Conversely, if the *mol* keyword is not used, only atom translations are performed. $M$ should typically be chosen to be approximately equal to the expected number of gas atoms or molecules of the given type within the simulation cell or region, which will result in roughly one MC move per atom or molecule per MC cycle.

All inserted particles are always added to two groups: the default group "all" and the fix group specified in the fix command. In addition, particles are also added to any groups specified by the *group* and *grouptype* keywords. If inserted particles are individual atoms, they are assigned the atom type given by the type argument. If they are molecules, the type argument has no effect and must be set to zero. Instead, the type of each atom in the inserted molecule is specified in the file read by the *molecule* command.

**Note**

Care should be taken to apply *fix gcmc* to a group that contains *only* those atoms and molecules that you wish to manipulate using Monte Carlo. Hence it is generally not a good idea to specify the default group "all" in the fix command, although it is allowed.

This fix cannot be used to perform GCMC insertions of gas atoms or molecules other than the exchanged type, but GCMC deletions, and MC translations, and rotations can be performed on any atom/molecule in the fix group. All atoms in the simulation cell can be moved using regular time integration translations, e.g. via *fix nvt*, resulting in a hybrid GCMC+MD simulation. A smaller-than-usual timestep size may be needed when running such a hybrid simulation, especially if the inserted molecules are not well equilibrated.

This command may optionally use the *region* keyword to define an exchange and move volume. The specified region must have been previously defined with a *region* command. It must be defined with side = *in*. Insertion attempts occur only within the specified region. For non-rectangular regions, random trial points are generated within the rectangular bounding box until a point is found that lies inside the region. If no valid point is generated after 1000 trials, no insertion is performed, but it is counted as an attempted insertion. Move and deletion attempt candidates are selected from gas atoms or molecules within the region. If there are no candidates, no move or deletion is performed, but it is counted as an attempt move or deletion. If an attempted move places the atom or molecule center-of-mass outside the specified region, a new attempted move is generated. This process is repeated until the atom or molecule center-of-mass is inside the specified region.

If used with *fix nvt*, the temperature of the imaginary reservoir, $T$, should be set to be equivalent to the target temperature used in fix nvt. Otherwise, the imaginary reservoir will not be in thermal equilibrium with the simulation cell. Also, it is important that the temperature used by *fix nvt* is dynamically updated, which can be achieved as follows:

```
compute mdtemp mdatoms temp
compute_modify mdtemp dynamic/dof yes
fix mdnvt mdatoms nvt temp 300.0 300.0 10.0
fix_modify mdnvt temp mdtemp
```

Note that neighbor lists are re-built every timestep that this fix is invoked, so you should not set $N$ to be too small. However, periodic rebuilds are necessary in order to avoid dangerous rebuilds and missed interactions. Specifically, avoid performing so many MC translations per timestep that atoms can move beyond the neighbor list skin distance. See the *neighbor* command for details.

When an atom or molecule is to be inserted, its coordinates are chosen at a random position within the current simulation cell or region, and new atom velocities are randomly chosen from the specified temperature distribution given by $T$. The effective temperature for new atom velocities can be increased or decreased using the optional keyword *tfac_insert* (see below). Relative coordinates for atoms in a molecule are taken from the template molecule provided by the user. The center of mass of the molecule is placed at the insertion point. The orientation of the molecule is chosen at random by rotating about this point.

Individual atoms are inserted, unless the *mol* keyword is used. It specifies a *template-ID* previously defined using the *molecule* command, which reads a file that defines the molecule. The coordinates, atom types, charges, etc., as well as any bonding and special neighbor information for the molecule can be specified in the molecule file. See the *molecule* command for details. The only settings required to be in this file are the coordinates and types of atoms in the molecule.

When not using the *mol* keyword, you should ensure you do not delete atoms that are bonded to other atoms, or LAMMPS will soon generate an error when it tries to find bonded neighbors. LAMMPS will warn you if any of the

atoms eligible for deletion have a non-zero molecule ID, but does not check for this at the time of deletion.

If you wish to insert molecules using the *mol* keyword that will be treated as rigid bodies, use the *rigid* keyword, specifying as its value the ID of a separate *fix rigid/small* command which also appears in your input script.

**Note**

If you wish the new rigid molecules (and other rigid molecules) to be thermostatted correctly via *fix rigid/small/nvt* or *fix rigid/small/npt*, then you need to use the *fix_modify dynamic/dof yes* command for the rigid fix. This is to inform that fix that the molecule count will vary dynamically.

If you wish to insert molecules via the *mol* keyword, that will have their bonds or angles constrained via SHAKE, use the *shake* keyword, specifying as its value the ID of a separate *fix shake* command which also appears in your input script.

Optionally, users may specify the relative amounts of different MC moves using the *mcmoves* keyword. The values *Patomtrans*, *Pmoltrans*, *Pmolrotate* specify the average proportion of atom translations, molecule translations, and molecule rotations, respectively. The values must be non-negative integers or real numbers, with at least one non-zero value. For example, (10,30,0) would result in 25% of the MC moves being atomic translations, 75% molecular translations, and no molecular rotations.

Optionally, users may specify the maximum rotation angle for molecular rotations using the *maxangle* keyword and specifying the angle in degrees. Rotations are performed by generating a random point on the unit sphere and a random rotation angle on the range [0,maxangle). The molecule is then rotated by that angle about an axis passing through the molecule center of mass. The axis is parallel to the unit vector defined by the point on the unit sphere. The same procedure is used for randomly rotating molecules when they are inserted, except that the maximum angle is 360 degrees.

Note that *fix gcmc* does not use configurational bias MC or any other kind of sampling of intramolecular degrees of freedom. Inserted molecules can have different orientations, but they will all have the same intramolecular configuration, which was specified in the molecule command input.

For atomic gasses, inserted atoms have the specified atom type, but deleted atoms are any atoms that have been inserted or that already belong to the fix group. For molecular gasses, exchanged molecules use the same atom types as in the template molecule supplied by the user. In both cases, exchanged atoms/molecules are assigned to two groups: the default group "all" and the fix group (which can also be "all").

The chemical potential is a user-specified input parameter defined as:

$$\mu = \mu^{id} + \mu^{ex}$$

The second term $\mu^{ex}$ is the excess chemical potential due to energetic interactions and is formally zero for the fictitious gas reservoir but is non-zero for interacting systems. So, while the chemical potential of the reservoir and the simulation cell are equal, $\mu^{ex}$ is not, and as a result, the densities of the two are generally quite different. The first term $\mu^{id}$ is the ideal gas contribution to the chemical potential. $\mu^{id}$ can be related to the density or pressure of the fictitious gas reservoir by:

$$\begin{aligned}\mu^{id} &= kT \ln \rho\Lambda^3 \\ &= kT \ln \frac{\phi P \Lambda^3}{k_B T}\end{aligned}$$

where $k_B$ is the Boltzmann constant, $T$ is the user-specified temperature, $\rho$ is the number density, $P$ is the pressure, and $\phi$ is the fugacity coefficient. The constant $\Lambda$ is required for dimensional consistency. For all unit styles except *lj* it is defined as the thermal de Broglie wavelength

$$\Lambda = \sqrt{\frac{h^2}{2\pi m k_B T}}$$

where $h$ is Planck's constant, and $m$ is the mass of the exchanged atom or molecule. For unit style *lj*, $\Lambda$ is simply set to unity. Note that prior to March 2017, $\Lambda$ for unit style *lj* was calculated using the above formula with $h$ set to the rather specific value of 0.18292026. Chemical potential under the old definition can be converted to an equivalent value under the new definition by subtracting $3kT \ln(\Lambda_{old})$.

As an alternative to specifying mu directly, the ideal gas reservoir can be defined by its pressure $P$ using the *pressure* keyword, in which case the user-specified chemical potential is ignored. The user may also specify the fugacity coefficient $\phi$ using the *fugacity_coeff* keyword, which defaults to unity.

The *full_energy* option means that the fix calculates the total potential energy of the entire simulated system, instead of just the energy of the part that is changed. The total system energy before and after the proposed GCMC exchange or MC move is then used in the Metropolis criterion to determine whether or not to accept the proposed change. By default, this option is off, in which case only partial energies are computed to determine the energy difference due to the proposed change.

The *full_energy* option is needed for systems with complicated potential energy calculations, including the following:

- long-range electrostatics (kspace)
- many-body pair styles
- hybrid pair styles
- eam pair styles
- tail corrections
- need to include potential energy contributions from other fixes

In these cases, LAMMPS will automatically apply the *full_energy* keyword and issue a warning message.

When the *mol* keyword is used, the *full_energy* option also includes the intramolecular energy of inserted and deleted molecules, whereas this energy is not included when *full_energy* is not used. If this is not desired, the *intra_energy* keyword can be used to define an amount of energy that is subtracted from the final energy when a molecule is inserted, and subtracted from the initial energy when a molecule is deleted. For molecules that have a non-zero intramolecular energy, this will ensure roughly the same behavior whether or not the *full_energy* option is used.

Inserted atoms and molecules are assigned random velocities based on the specified temperature $T$. Because the relative velocity of all atoms in the molecule is zero, this may result in inserted molecules that are systematically too cold. In addition, the intramolecular potential energy of the inserted molecule may cause the kinetic energy of the molecule to quickly increase or decrease after insertion. The *tfac_insert* keyword allows the user to counteract these effects by changing the temperature used to assign velocities to inserted atoms and molecules by a constant factor. For a particular application, some experimentation may be required to find a value of *tfac_insert* that results in inserted molecules that equilibrate quickly to the correct temperature.

Some fixes have an associated potential energy. Examples of such fixes include: *efield*, *gravity*, *addforce*, *langevin*, *restrain*, *temp/berendsen*, *temp/rescale*, and *wall fixes*. For that energy to be included in the total potential energy of the system (the quantity used when performing GCMC exchange and MC moves), you MUST enable the *fix_modify energy* option for that fix. The doc pages for individual *fix* commands specify if this should be done.

Use the *charge* option to insert atoms with a user-specified point charge. Note that doing so will cause the system to become non-neutral. LAMMPS issues a warning when using long-range electrostatics (kspace) with non-neutral systems. See the *compute group/group* documentation for more details about simulating non-neutral systems with kspace on.

Use of this fix typically will cause the number of atoms to fluctuate, therefore, you will want to use the *compute_modify dynamic/dof* command to ensure that the current number of atoms is used as a normalizing factor each time temperature is computed. A simple example of this is:

```
compute_modify thermo_temp dynamic/dof yes
```

A more complicated example is listed earlier on this page in the context of NVT dynamics.

**Note**

If the density of the cell is initially very small or zero, and increases to a much larger density after a period of equilibration, then certain quantities that are only calculated once at the start (kspace parameters) may no longer be accurate. The solution is to start a new simulation after the equilibrium density has been reached.

With some pair_styles, such as *Buckingham*, *Born-Mayer-Huggins* and *ReaxFF*, two atoms placed close to each other may have an arbitrary large, negative potential energy due to the functional form of the potential. While these unphysical configurations are inaccessible to typical dynamical trajectories, they can be generated by Monte Carlo moves. The *overlap_cutoff* keyword suppresses these moves by effectively assigning an infinite positive energy to all new configurations that place any pair of atoms closer than the specified overlap cutoff distance.

The *max* and *min* keywords allow for the restriction of the number of atoms in the fix group (and region in case the *region* keyword is used). They automatically reject all insertion or deletion moves that would take the system beyond the set boundaries. Should the system already be beyond the boundary, only moves that bring the system closer to the bounds may be accepted.

The *group* keyword adds all inserted atoms to the *group* of the group-ID value. The *grouptype* keyword adds all inserted atoms of the specified type to the *group* of the group-ID value.

### 2.76.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the state of the fix to *binary restart files*. This includes information about the random number generator seed, the next timestep for MC exchanges, the number of MC step attempts and successes etc. This ensures that the sequence of Monte Carlo moves generated after reading from a restart file are exactly the same as those that are generated in the original run. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

**Note**

For this to work correctly, the timestep must **not** be changed after reading the restart with *reset_timestep*. The fix will try to detect it and stop with an error.

None of the *fix_modify* options are relevant to this fix.

This fix computes a global vector of length 8, which can be accessed by various *output commands*. The vector values are the following global cumulative quantities:

1. translation attempts
2. translation successes
3. insertion attempts
4. insertion successes
5. deletion attempts
6. deletion successes
7. rotation attempts
8. rotation successes

The vector values calculated by this fix are “intensive”.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.76.5 Restrictions

This fix is part of the MC package. It is only enabled if LAMMPS was built with that package. See the *Build package* doc page for more info.

This fix style requires an *atom style* with per atom type masses.

Do not set *neigh_modify once yes* or else this fix will never be called. Reneighboring is **required**.

This fix style is only usable for 3D simulations.

This fix can be run in parallel, but aspects of the GCMC part will not scale well in parallel. Currently, molecule translations and rotations are not supported with more than one MPI process. It is still possible to do parallel molecule exchange without translation and rotation moves by setting MC moves to zero and/or by using the *mcmoves* keyword with *Pmoltrans* = *Pmolrotate* = 0 .

When using *fix gcmc* in combination with *fix shake* or *fix rigid*, only GCMC exchange moves are supported, so the argument *M* must be zero.

When using *fix gcmc* in combination with *fix rigid*, deletion of the last remaining molecule is not allowed for technical reasons, and so the molecule count will never drop below 1, regardless of the specified chemical potential.

Note that very lengthy simulations involving insertions/deletions of billions of gas molecules may run out of atom or molecule IDs and trigger an error, so it is better to run multiple shorter-duration simulations. The *reset_atoms* command can be used to "compress" the atom and molecule IDs between runs. Likewise, very large molecules have not been tested and may turn out to be problematic.

Use of multiple *fix gcmc* commands in the same input script can be problematic if using a template molecule. The issue is that the user-referenced template molecule in the second *fix gcmc* command may no longer exist since it might have been deleted by the first *fix gcmc* command. An existing template molecule will need to be referenced by the user for each subsequent *fix gcmc* command.

### 2.76.6 Related commands

*fix gemc*, *fix widom*, *fix atom/swap*, *fix nvt*, *neighbor*, *fix deposit*, *fix evaporate*, *delete_atoms*

### 2.76.7 Defaults

The option defaults are mol = no, maxangle = 10, overlap_cutoff = 0.0, fugacity_coeff = 1.0, intra_energy = 0.0, tfac_insert = 1.0. (Patomtrans, Pmoltrans, Pmolrotate) = (1, 0, 0) for mol = no and (0, 1, 1) for mol = yes. full_energy = no, except for the situations where full_energy is required, as listed above.

---

**(Frenkel)** Frenkel and Smit, Understanding Molecular Simulation, Academic Press, London, 2002.

## 2.77 fix gemc command

### 2.77.1 Syntax

```
fix ID group-ID gemc N M X V T displace maxvol seed
```

- ID, group-ID are documented in *fix* command
- gemc = style name of this fix command

- N = invoke this fix every N steps
- M = average number of MC moves to attempt every N steps
- X = average number of GEMC exchanges to attempt every N steps
- V = average number of GEMC volume changes to attempt every N steps
- T = temperature of the Gibbs ensemble (temperature units)
- displace = maximum Monte Carlo translation distance (length units)
- maxdlogvolratio = maximum change in ln(vol1/vol2) (unitless)
- seed = random # seed (positive integer)
- zero keyword/value pairs may be appended to args

### 2.77.2 Examples

```
fix 2 all gemc 100 10 20 30 1.1 0.5 100.0 29494
```

### 2.77.3 Description

Added in version 4Jul2026.

This fix performs Gibbs ensemble Monte Carlo (GEMC) exchanges of atoms and volume between two simulation cells at specified *T*. It also attempts Monte Carlo (MC) moves (atom translations) within the simulation cell. This is usually used to establish thermodynamic equilibrium between bulk vapor and liquid phases, as discussed in *(Frenkel)*. If used with the *fix nvt* command, hybrid MD/MC simulations in the Gibbs ensemble (equal pressure, equal chemical potential, constant total volume, and constant temperature) can be performed. Specific uses include computing vapor-liquid coexistence curves.

Every *N* timesteps the fix attempts GEMC atom exchanges, GEMC volume changes, and MC moves of atoms. On those timesteps, the average number of attempted GEMC atom exchanges is *X*, the average number of volume changes is *V*, and the average number of attempted MC moves is *M*.

This fix requires that LAMMPS be run with two partitions that instantiate the two simulation cells. This requires using the *-partition command-line switch*. For example, on a workstation with 12 cores, `-partition 2x6` could be used. For better performance, it is recommended that the two partitions be initialized at two different densities. This allows assigning only one core to the partition running the vapor phase and all the other cores to the partition running the liquid phase. This can be achieved in a single script by using the *partition* command to initiate the two partitions at different densities. On a workstation with 12 processor cores, the command line option `-partition 1 8` can be used to assign 1 core to the first (vapor) partition and 8 cores to the second (liquid) partition.

If used with *fix nvt*, the temperature of the Gibbs ensemble, *T*, should be set to be equivalent to the target temperature used in fix nvt. Otherwise, the imaginary reservoir will not be in thermal equilibrium with the simulation cell. Also, it is important that the temperature used by *fix nvt* is dynamically updated, which can be achieved as follows:

```
compute mdtemp mdatoms temp
compute_modify mdtemp dynamic/dof yes
fix mdnvt mdatoms nvt temp 300.0 300.0 10.0
fix_modify mdnvt temp mdtemp
```

Note that neighbor lists are re-built every timestep that this fix is invoked, so you should not set *N* to be too small. However, periodic rebuilds are necessary in order to avoid dangerous rebuilds and missed interactions. Specifically, avoid performing so many MC translations per timestep that atoms can move beyond the neighbor list skin distance. See the *neighbor* command for details.

When an atom is inserted in either partition, its coordinates are chosen at a random position within the current simulation cell, and new atom velocities are randomly chosen from the specified temperature distribution given by *T*.

Some fixes have an associated potential energy. Examples of such fixes include: *efield*, *gravity*, *addforce*, *langevin*, *restrain*, *temp/berendsen*, *temp/rescale*, and *wall fixes*. For that energy to be included in the total potential energy of the system (the quantity used when performing GEMC atom exchange, GEMC volume exchange and MC moves), you MUST enable the *fix_modify* *energy* option for that fix. The doc pages for individual *fix* commands specify if this should be done.

Use of this fix typically will cause the number of atoms in each cell to fluctuate, therefore, you will want to use the *compute_modify dynamic/dof* command to ensure that the current number of atoms is used as a normalizing factor each time temperature is computed. A simple example of this is:

```
compute_modify thermo_temp dynamic/dof yes
```

A more complicated example is listed earlier on this page in the context of NVT dynamics.

**Note**

If the density of the cell is initially very small or zero, and increases to a much larger density after a period of equilibration, then certain quantities that are only calculated once at the start (kspace parameters) may no longer be accurate. The solution is to start a new simulation after the equilibrium density has been reached.

### 2.77.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the state of the fix to *binary restart files*. This includes information about the random number generator seed, the next timestep for MC exchanges, the number of MC step attempts and successes etc. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

**Note**

For this to work correctly, the timestep must **not** be changed after reading the restart with *reset_timestep*. The fix will try to detect it and stop with an error.

None of the *fix_modify* options are relevant to this fix.

This fix computes a global vector of length 8, which can be accessed by various *output commands*. The vector values are the following global cumulative quantities:

1. translation attempts
2. translation successes
3. exchange attempts
4. exchange successes
5. volume change attempts
6. volume change successes

The vector values calculated by this fix are “intensive”.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.77.5 Restrictions

This fix is part of the MC package. It is only enabled if LAMMPS was built with that package. See the *Build package* doc page for more info.

Do not set *neigh_modify once yes* or else this fix will never be called. Reneighboring is **required**.

*Fix gemc* currently **only** supports MC moves and exchanges on individual atoms.

Use of multiple *fix gemc* commands in the same input script can be problematic.

### 2.77.6 Related commands

*fix gcmc*, *fix widom*, *fix atom/swap*

### 2.77.7 Defaults

None

---

**(Frenkel)** Frenkel and Smit, Understanding Molecular Simulation, Academic Press, London, 2002.

## 2.78 fix gjf command

### 2.78.1 Syntax

```
fix ID group-ID gjf Tstart Tstop damp seed keyword values ...
```

- ID, group-ID are documented in *fix* command
- gjf = style name of this fix command
- Tstart,Tstop = desired temperature at start/end of run (temperature units)
- Tstart can be a variable (see below)
- damp = damping parameter (time units)
- seed = random number seed to use for white noise (positive integer)
- zero or more keyword/value pairs may be appended
- keyword = *vel* or *method*

```
vel value = vfull or vhalf
  vfull = use on-site velocity
  vhalf = use half-step velocity
method value = 1-8
  1-8 = choose one of the many GJ formulations
  7   = requires input of additional scalar between 0 and 1
```

### 2.78.2 Examples

```
fix 3 boundary gjf 10.0 10.0 1.0 699483
fix 1 all gjf 10.0 100.0 100.0 48279 vel vfull method 4
fix 2 all gjf 10.0 10.0 1.0 26488 method 7 0.95
```

### 2.78.3 Description

Added in version 12Jun2025.

Apply a Langevin thermostat as described in *(Gronbech-Jensen-2020)* to a group of atoms which models an interaction with a background implicit solvent. As described in the papers cited below, the GJ methods provide exact diffusion, drift, and Boltzmann sampling for linear systems for any time step within the stability limit. The purpose of this set of methods is therefore to significantly improve statistical accuracy at longer time steps compared to other thermostats.

The current implementation provides the user with the option to output the velocity in one of two forms: *vfull* or *vhalf*. The option *vhalf* outputs the 2GJ half-step velocity given in *Gronbech Jensen/Gronbech-Jensen*; for linear systems, this velocity is shown to not have any statistical errors for any stable time step. The option *vfull* outputs the on-site velocity given in *Gronbech-Jensen/Farago*; this velocity is shown to be systematically lower than the target temperature by a small amount, which grows quadratically with the timestep. An overview of statistically correct Boltzmann and Maxwell-Boltzmann sampling of true on-site and true half-step velocities is given in *Gronbech-Jensen-2020*.

This fix allows the use of several GJ methods as listed in *Gronbech-Jensen-2020*. The GJ-VII method is described in *Finkelstein* and GJ-VIII is described in *Gronbech-Jensen-2024*. The implementation follows the splitting form provided in Eqs. (24) and (25) in *Gronbech-Jensen-2024*, including the application of Gaussian noise values, per the description in *Gronbech-Jensen-2023*.

**Note**

Unlike the *fix langevin* command which performs force modifications only, this fix performs thermostatting and time integration. Thus you no longer need a separate time integration fix, like *fix nve*.

See the *Howto thermostat* page for a discussion of different ways to compute temperature and perform thermostatting.

The desired temperature at each timestep is a ramped value during the run from *Tstart* to *Tstop*.

*Tstart* can be specified as an equal-style or atom-style *variable*. In this case, the *Tstop* setting is ignored. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated each timestep, and its value used to determine the target temperature.

Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent temperature.

Atom-style variables can specify the same formulas as equal-style variables but can also include per-atom values, such as atom coordinates. Thus it is easy to specify a spatially-dependent temperature with optional time-dependence as well.

Like other fixes that perform thermostatting, this fix can be used with *compute commands* that remove a "bias" from the atom velocities. E.g. to apply the thermostat only to atoms within a spatial *region*, or to remove the center-of-mass velocity from a group of atoms, or to remove the x-component of velocity from the calculation.

This is not done by default, but only if the *fix_modify* command is used to assign a temperature compute to this fix that includes such a bias term. See the doc pages for individual *compute temp commands* to determine which ones include a bias.

The *damp* parameter is specified in time units and determines how rapidly the temperature is relaxed. For example, a value of 100.0 means to relax the temperature in a timespan of (roughly) 100 time units ($\tau$ or fs or ps - see the *units* command). The damp factor can be thought of as inversely related to the viscosity of the solvent. I.e. a small relaxation time implies a high-viscosity solvent and vice versa. See the discussion about $\gamma$ and viscosity in the documentation for the *fix viscous* command for more details.

The random # *seed* must be a positive integer. A Marsaglia random number generator is used. Each processor uses the input seed to generate its own unique seed and its own stream of random numbers. Thus the dynamics of the system will not be identical on two runs on different numbers of processors.

The keyword/value option pairs are used in the following ways.

The keyword *vel* determines which velocity is used to determine quantities of interest in the simulation.

The keyword *method* selects one of the eight GJ-methods implemented in LAMMPS.

### 2.78.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. Because the state of the random number generator is not saved in restart files, this means you cannot do "exact" restarts with this fix, where the simulation continues on the same as if no restart had taken place. However, in a statistical sense, a restarted simulation should produce the same behavior. Additionally, the GJ methods implement noise exclusively within each time step (unlike the BBK thermostat of the fix-langevin). The restart is done with either vfull or vhalf velocity output for as long as the choice of vfull/vhalf is the same for the simulation as it is in the restart file.

The *fix_modify* *temp* option is supported by this fix. You can use it to assign a temperature *compute* you have defined to this fix which will be used in its thermostatting procedure, as described above. For consistency, the group used by this fix and by the compute should be the same.

This fix can ramp its target temperature over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.78.5 Restrictions

This fix is not compatible with run_style respa. It is not compatible with accelerated packages such as KOKKOS.

### 2.78.6 Related commands

*fix langevin*, *fix nvt*

### 2.78.7 Default

The option defaults are vel = vhalf, method = 1.

**(Gronbech-Jensen-2020)** Gronbech-Jensen, Mol Phys 118, e1662506 (2020).

**(Gronbech Jensen/Gronbech-Jensen)** Gronbech Jensen and Gronbech-Jensen, Mol Phys, 117, 2511 (2019)

**(Gronbech-Jensen/Farago)** Gronbech-Jensen and Farago, Mol Phys, 111, 983 (2013).

**(Finkelstein)** Finkelstein, Cheng, Fiorin, Seibold, Gronbech-Jensen, J. Chem. Phys., 155, 18 (2021)

**(Gronbech-Jensen-2024)** Gronbech-Jensen, J. Stat. Phys. 191, 137 (2024).

**(Gronbech-Jensen-2023)** Gronbech-Jensen, J. Stat. Phys. 190, 96 (2023).

## 2.79 fix gld command

### 2.79.1 Syntax

```
fix ID group-ID gld Tstart Tstop N_k seed series c_1 tau_1 ... c_N_k tau_N_k keyword␣
↪values ...
```

- ID, group-ID are documented in *fix* command
- gld = style name of this fix command
- Tstart,Tstop = desired temperature at start/end of run (temperature units)
- N_k = number of terms in the Prony series representation of the memory kernel
- seed = random number seed to use for white noise (positive integer)
- series = *pprony* is presently the only available option
- c_k = the weight of the kth term in the Prony series (mass per time units)
- tau_k = the time constant of the kth term in the Prony series (time units)
- zero or more keyword/value pairs may be appended

```
keyword = frozen or zero
  frozen value = no or yes
    no = initialize extended variables using values drawn from equilibrium␣
↪distribution at Tstart
    yes = initialize extended variables to zero (i.e., from equilibrium␣
↪distribution at zero temperature)
  zero value = no or yes
    no = do not set total random force to zero
    yes = set total random force to zero
```

### 2.79.2 Examples

```
fix 1 all gld 1.0 1.0 2 82885 pprony 0.5 1.0 1.0 2.0 frozen yes zero yes
fix 3 rouse gld 7.355 7.355 4 48823 pprony 107.1 0.02415 186.0 0.04294 428.6 0.09661␣
↪1714 0.38643
```

### 2.79.3 Description

Applies Generalized Langevin Dynamics to a group of atoms, as described in *(Baczewski)*. This is intended to model the effect of an implicit solvent with a temporally non-local dissipative force and a colored Gaussian random force, consistent with the Fluctuation-Dissipation Theorem. The functional form of the memory kernel associated with the temporally non-local force is constrained to be a Prony series.

**Note**

While this fix bears many similarities to *fix langevin*, it has one significant difference. Namely, *fix gld* performs time integration, whereas *fix langevin* does NOT. To this end, the specification of another fix to perform time integration, such as *fix nve*, is NOT necessary.

With this fix active, the force on the $j$th atom is given as

$$\mathbf{F}_j(t) = \mathbf{F}_j^C(t) - \int_0^t \Gamma_j(t-s)\mathbf{v}_j(s)\,\mathrm{d}s + \mathbf{F}_j^R(t)$$

$$\Gamma_j(t-s) = \sum_{k=1}^{N_k} \frac{c_k}{\tau_k} e^{-(t-s)/\tau_k}$$

$$\left\langle \mathbf{F}_j^R(t), \mathbf{F}_j^R(s) \right\rangle = \mathrm{k_B T}\, \Gamma_j(t-s)$$

Here, the first term is representative of all conservative (pairwise, bonded, etc) forces external to this fix, the second is the temporally non-local dissipative force given as a Prony series, and the third is the colored Gaussian random force.

The Prony series form of the memory kernel is chosen to enable an extended variable formalism, with a number of exemplary mathematical features discussed in *(Baczewski)*. In particular, $3N_k$ extended variables are added to each atom, which effect the action of the memory kernel without having to explicitly evaluate the integral over time in the second term of the force. This also has the benefit of requiring the generation of uncorrelated random forces, rather than correlated random forces as specified in the third term of the force.

Presently, the Prony series coefficients are limited to being greater than or equal to zero, and the time constants are limited to being greater than zero. To this end, the value of series MUST be set to *pprony*, for now. Future updates will allow for negative coefficients and other representations of the memory kernel. It is with these updates in mind that the series option was included.

The units of the Prony series coefficients are chosen to be mass per time to ensure that the numerical integration scheme stably approaches the Newtonian and Langevin limits. Details of these limits, and the associated numerical concerns are discussed in *(Baczewski)*.

The desired temperature at each timestep is ramped from *Tstart* to *Tstop* over the course of the next run.

The random # *seed* must be a positive integer. A Marsaglia random number generator is used. Each processor uses the input seed to generate its own unique seed and its own stream of random numbers. Thus the dynamics of the system will not be identical on two runs on different numbers of processors.

---

The keyword/value option pairs are used in the following ways.

The keyword *frozen* can be used to specify how the extended variables associated with the GLD memory kernel are initialized. Specifying no (the default), the initial values are drawn at random from an equilibrium distribution at *Tstart*, consistent with the Fluctuation-Dissipation Theorem. Specifying yes, initializes the extended variables to zero.

The keyword *zero* can be used to eliminate drift due to the thermostat. Because the random forces on different atoms are independent, they do not sum exactly to zero. As a result, this fix applies a small random force to the entire system, and the center-of-mass of the system undergoes a slow random walk. If the keyword *zero* is set to *yes*, the total random force is set exactly to zero by subtracting off an equal part of it from each atom in the group. As a result, the center-of-mass of a system with zero initial momentum will not drift over time.

---

### 2.79.4 Restart, fix_modify, output, run start/stop, minimize info

The instantaneous values of the extended variables are written to *binary restart files*. Because the state of the random number generator is not saved in restart files, this means you cannot do "exact" restarts with this fix, where the simulation continues on the same as if no restart had taken place. However, in a statistical sense, a restarted simulation should produce the same behavior.

None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*.

This fix can ramp its target temperature over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.79.5 Restrictions

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.79.6 Related commands

*fix langevin*, *fix viscous*, *pair_style dpd/tstat*

### 2.79.7 Default

The option defaults are frozen = no, zero = no.

---

**(Baczewski)** A.D. Baczewski and S.D. Bond, J. Chem. Phys. 139, 044107 (2013).

## 2.80 fix gle command

### 2.80.1 Syntax

```
fix ID id-group gle Ns Tstart Tstop seed Amatrix [noneq Cmatrix] [every stride]
```

- ID, group-ID are documented in *fix* command
- gle = style name of this fix command
- Ns = number of additional fictitious momenta
- Tstart, Tstop = temperature ramp during the run
- Amatrix = file to read the drift matrix A from
- seed = random number seed to use for generating noise (positive integer)
- zero or more keyword/value pairs may be appended

```
keyword = noneq or every
  noneq Cmatrix  = file to read the non-equilibrium covariance matrix from
  every stride   = apply the GLE once every time steps. Reduces the accuracy
      of the integration of the GLE, but has *no effect* on the accuracy of␣
↪equilibrium
      sampling. It might change sampling properties when used together with noneq.
```

### 2.80.2 Examples

```
fix 3 boundary gle 6 300 300 31415 smart.A
fix 1 all gle 6 300 300 31415 qt-300k.A noneq qt-300k.C
```

### 2.80.3 Description

Apply a Generalized Langevin Equation (GLE) thermostat as described in *(Ceriotti)*. The formalism allows one to obtain a number of different effects ranging from efficient sampling of all vibrational modes in the system to inexpensive (approximate) modelling of nuclear quantum effects. Contrary to *fix langevin*, this fix performs both thermostatting and evolution of the Hamiltonian equations of motion, so it should not be used together with *fix nve* – at least not on the same atom groups.

Each degree of freedom in the thermostatted group is supplemented with Ns additional degrees of freedom s, and the equations of motion become

```
dq/dt=p/m
d(p,s)/dt=(F,0) - A(p,s) + B dW/dt
```

where F is the physical force, A is the drift matrix (that generalizes the friction in Langevin dynamics), B is the diffusion term and dW/dt un-correlated Gaussian random forces. The A matrix couples the physical (q,p) dynamics with that of the additional degrees of freedom, and makes it possible to obtain effectively a history-dependent noise and friction kernel.

The drift matrix should be given as an external file *Afile*, as a (Ns+1 x Ns+1) matrix in inverse time units. Matrices that are optimal for a given application and the system of choice can be obtained from *(GLE4MD)*.

Equilibrium sampling a temperature T is obtained by specifying the target value as the *Tstart* and *Tstop* arguments, so that the diffusion matrix that gives canonical sampling for a given A is computed automatically. However, the GLE framework also allow for non-equilibrium sampling, that can be used for instance to model inexpensively zero-point energy effects *(Ceriotti2)*. This is achieved specifying the *noneq* keyword followed by the name of the file that contains the static covariance matrix for the non-equilibrium dynamics. Please note, that the covariance matrix is expected to be given in **temperature units**.

Since integrating GLE dynamics can be costly when used together with simple potentials, one can use the *every* optional keyword to apply the Langevin terms only once every several MD steps, in a multiple time-step fashion. This should be used with care when doing non-equilibrium sampling, but should have no effect on equilibrium averages when using canonical sampling.

The random number *seed* must be a positive integer. A Marsaglia random number generator is used. Each processor uses the input seed to generate its own unique seed and its own stream of random numbers. Thus the dynamics of the system will not be identical on two runs on different numbers of processors.

Note also that the Generalized Langevin Dynamics scheme that is implemented by the *fix gld* scheme is closely related to the present one. In fact, it should be always possible to cast the Prony series form of the memory kernel used by GLD into an appropriate input matrix for *fix gle*. While the GLE scheme is more general, the form used by *fix gld* can be more directly related to the representation of an implicit solvent environment.

### 2.80.4 Restart, fix_modify, output, run start/stop, minimize info

The instantaneous values of the extended variables are written to *binary restart files*. Because the state of the random number generator is not saved in restart files, this means you cannot do "exact" restarts with this fix, where the simulation continues on the same as if no restart had taken place. However, in a statistical sense, a restarted simulation should produce the same behavior. Note however that you should use a different seed each time you restart, otherwise the same sequence of random numbers will be used each time, which might lead to stochastic synchronization and subtle artifacts in the sampling.

The cumulative energy change in the system imposed by this fix is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* doc page for details.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the same cumulative energy change due to this fix described in the previous paragraph. The scalar value calculated by this fix is "extensive".

This fix can ramp its target temperature over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.80.5 Restrictions

The GLE thermostat in its current implementation should not be used with rigid bodies, SHAKE or RATTLE. It is expected that all the thermostatted degrees of freedom are fully flexible, and the sampled ensemble will not be correct otherwise.

In order to perform constant-pressure simulations please use *fix press/berendsen*, rather than *fix npt*, to avoid duplicate integration of the equations of motion.

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.80.6 Related commands

*fix nvt*, *fix temp/rescale*, *fix viscous*, *fix nvt*, *pair_style dpd/tstat*, *fix gld*

---

**(Ceriotti)** Ceriotti, Bussi and Parrinello, J Chem Theory Comput 6, 1170-80 (2010)

**(GLE4MD)** https://gle4md.org/

**(Ceriotti2)** Ceriotti, Bussi and Parrinello, Phys Rev Lett 103, 030603 (2009)

## 2.81 fix graphics/arrows command

### 2.81.1 Syntax

```
fix ID group-ID graphics/arrows Nevery mode keyword args ...
```

- ID, group-ID are documented in *fix* command
- graphics/arrows = style name of this fix command
- Nevery = update graphics information every this many time steps
- mode = one of the following modes *dipole* or *force* or *velocity* or *variable* or *chunk*

```
dipole args = scale radius
  scale = scale factor for the dipole moment to determine the arrow length
  radius = radius for arrows (length units)
force args = scale radius
  scale = scale factor for the force vector to determine the arrow length
  radius = radius for arrows (length units)
velocity args = scale radius
  scale = scale factor for the velocity vector to determine the arrow length
  radius = radius for arrows (length units)
variable args = xval yval zval radius
  xval = x value for arrow vector (may be a variable)
```

```
    yval = y value for arrow vector (may be a variable)
    zval = z value for arrow vector (may be a variable)
    radius = radius for arrows (length units)
  chunk args = chunk-ID pos-ID vec-ID scale radius
    chunk-ID = ID of compute chunk/atom command
    pos-ID = ID of a per-chunk compute that computes the positions for the arrows
    vec-ID = ID of a per-chunk compute that computes the arrow vectors
    scale = scale factor for the per-chunk vector to determine the arrow length
    radius = radius for arrows (length units)
```

- zero or more keyword/value pairs may be appended
- keyword = *autoscale*

```
autoscale value = automatically scale arrows so they have an average length of␣
↪"value"
```

### 2.81.2 Examples

```
fix vec all graphics/arrows 10 velocity 20.0 0.066 autoscale 0.5
fix vec all graphics/arrows 100 variable v_xnorm v_znorm 0.0 0.066
fix vec all graphics/arrows 100 chunk molchunk com dip 1.0 0.05
```

### 2.81.3 Description

Added in version 11Feb2026.

This fix allows to add arrows to images rendered with *dump image* using the *fix* keyword to represent vector properties with arrows for either all atoms in the fix group or for *chunks*.

The *group-ID* sets the group ID of the atoms selected to have the selected property represented. This may be a dynamic group.

The *Nevery* keyword determines how often the arrows graphics data is updated. This should be the same value as the corresponding *N* parameter of the *dump* image command. LAMMPS will stop with an error message if the settings for this fix and the dump command are not compatible.

There are five keywords available that determine what is shown: *dipole* will show the per-atom dipole vector, *force* the per-atom force, *velocity* the per-atom velocity, *variable* a custom vector constructed from three constants or atom- or equal-style variables. With the *chunk* keyword the arrows shown will represent per-chunk vector data.

The *xval*, *yval*, and *zval*, arguments to the *variable* mode define a custom vector that can be composed of numbers or *atom- or equal-style variables*. If any of these values is a variable, it should be specified as *v_name*, where "name" is the variable name. In this case, the variable will be evaluated each timestep, and its value used to define the arrow for each atom. Since variables can reference *computes*, *fixes*, *custom per-atom properties*, and other variables, this can be used to construct arrows for almost any per-atom property available in LAMMPS.

The *chunk-ID* is the ID of a *compute chunk/atom* command. In LAMMPS, chunks are collections of atoms and there are per-chunk computes that compute properties for them. See the *compute chunk/atom* and *Howto chunk* pages for details of how chunks can be defined and examples of how they can be used to measure properties of a system.

The *pos-ID* is the ID of a per-chunk *compute command*. Most commonly this will be either *compute com/chunk* for "mobile" chunks or compute *compute property/chunk* for binning based chunks. The *vec-ID* is the ID of a per-chunk *compute command*. Either per-chunk compute must return a global array with at least 3 columns and *only* the first three columns are used for the arrows. For computes that compute a tensor only the trace of the tensor is used. Currently the following computes are compatible:

- *angmom/chunk*

- *com/chunk*
- *dipole/chunk*
- *dipole/tip4p/chunk*
- *gyration/chunk* (with optional *tensor* keyword)
- *gyration/shape/chunk*
- *inertia/chunk*
- *msd/chunk*
- *omega/chunk*
- *property/chunk* (with arguments *coord1 coord2 coord3*)
- *reduce/chunk* (with three or more properties)
- *torque/chunk*
- *vacf/chunk*
- *vcm/chunk*

The *scale* quantity determines the length of the arrows. It should be chosen so that when multiplied with the per-atom vector quantity the result is of the same order of magnitude as atom positions, so that the vectors can be seen well.

The *radius* quantity determines the width of the arrows.

The optional *autoscale* keyword allows to dynamically determine the *scale* quantity so that the average length of the arrows is set to the value of the keyword's argument. The computed scale factor can be accessed by various *output commands* as a global scalar (see below).

---

### 2.81.4 Dump image info

Added in version 11Feb2026.

Fix graphics/arrows is designed to be used with the *fix* keyword of *dump image*. The fix will add arrows based on the atoms in the fix group or based on chunks to *dump image* so that they are included in the rendered image.

The color of the arrows is by default that of the atoms when using color styles "type" or "element". With color style "const" the default value of "white" can be changed using *dump_modify fcolor*. The transparency is by default fully opaque and can be changed with *dump_modify ftrans*.

The *fflag1* setting of *dump image fix* allows to adjust the length of the arrows. Since that value is already set or computed by the fix, *fflag1* should be set to 0.0.

The *fflag2* setting allows you to adjust the radius of the rendered arrows. Since the radius of the arrows is an input parameter for this fix, it is recommended to set this flag to 0.0.

### 2.81.5 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

None of the *fix_modify* options apply to this fix.

This fix computes a global scalar representing the current scale factor for displaying the arrows, which can be accessed by various *output commands*. This is the *autoscale* keyword argument value divided by the average length of the selected vector property. If the *autoscale* keyword is not used, it is the scale value set by the *fix graphics/arrows* command or 1.0. The scalar value calculated by this fix is "intensive".

### 2.81.6 Restrictions

This fix is part of the GRAPHICS package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The *dipole* mode requires the use of *atom style dipole* or a hybrid atom style that includes it.

### 2.81.7 Related commands

*fix graphics/chunk*, *fix graphics/labels*, *fix graphics/isosurface*, *fix graphics/lines*, *fix graphics/objects*, *fix graphics/periodic*, *fix graphics/replica*, *compute hbond/local*, *dump image*

### 2.81.8 Default

autoscale is off by default

## 2.82 fix graphics/chunk command

### 2.82.1 Syntax

```
fix ID group-ID graphics/chunk Nevery chunkID keyword args ...
```

- ID, group-ID are documented in *fix* command
- graphics/chunk = style name of this fix command
- Nevery = update graphics information every this many time steps
- chunkID = ID of *compute chunk/atom* command
- zero or more keyword/args pairs may be appended
- keyword = *alpha* or *clip* or *maxreplace* or *shading* or *radius* or *mindist* or *region*

```
alpha value = override multiplier for alpha shape extraction (distance units)
clip value = truncate point cloud to box boundaries if yes, otherwise use all points
maxreplace value = set the largest cluster size up to which atoms are replaced by␣
↪icosahedra
shading value = smooth or flat
   smooth = compute per-vertex normals for smooth shading (default)
   flat = use face normals for flat shading
radius value = override per-atom or per-type radius if > 0.0 (distance units,␣
↪default 0.0)
mindist value = override automatic distance cutoff for slitting clusters (distance␣
↪units)
region value = region-ID
  region-ID = ID of region that atoms must be in to be visualized
```

### 2.82.2 Examples

```
compute cc1 all chunk/atom molecule
fix hull all graphics/chunk 100 cc1
fix hull all graphics/chunk 100 cc1 radius 1.0 shading smooth region upper
fix hull all graphics/chunk 100 cc1 shading flat alpha 10.0 maxreplace 50
```

### 2.82.3 Description

Added in version 4Jul2026.

This fix generates graphics objects from chunks of atoms defined by the *compute chunk/atom* command. For each chunk a point cloud is created from the atom positions. By default, for clusters with up to 100 atoms, each atom is replaced by the the positions of an icosahedron scaled to the radius of the atom. For larger clusters the point cloud only uses atom positions that shifted away from the center of the cluster by the atom radius. The threshold value can be set by the *maxreplace* keyword. A triangulated surface is created from that point cloud using a 3-D Delaunay triangulation combined with alpha shape extraction. This allows the resulting surface to follow the shape of the chunks. The resulting list of graphics objects is passed to *dump image* for rendering via the *fix* keyword.

The positions used for the generation of the graphics are based on coordinates for local and ghost atoms where then redundant generated triangles are not drawn. When a cluster straddles a periodic boundary it should be drawn in parts on both sides of the boundary.

If available, the per-atom radius (e.g. for simulations using *atom style sphere*) is used, otherwise - if available - half of the value of the Lennard-Jones *sigma* parameter for the atom type is used. If neither is available, half of the lattice spacing in x-direction is used as estimate for atom radii.

The *group-ID* selects the atoms included in the hull computation. Only atoms that belong to the specified group **and** are assigned to a chunk are considered.

The *Nevery* keyword determines how often the list of the graphics objects is recomputed. It should match the dump frequency of the corresponding *dump image* command.

The color of the graphics objects depends on the coloring scheme selected in *dump image* command. With the *type* or *element* coloring scheme the color is based on atom type as described below, with the *const* coloring scheme a uniform color is used instead. This color can be set with the *fcolor* keyword of the *dump modify* command. When using atom type based colors the vertices of the surface are colored using the atom type of the closest atom and the color between vertices is interpolated.

If the optional *region* keyword is used, only atoms in the specified geometric *region* are used for constructing the hull.

The optional *radius* keyword allows to override the radius value used to determine the size of the represented graphics by scaling the octahedron positions that represents each atom for computing the surface.

The optional *alpha* keyword allows to adjust the alpha shape extraction algorithm which determines how closely the generated triangulation follows the shape of chunks of atoms. It should be at least about 3x the average distance of closest neighbors. For larger values, the generated shape will become smother and more like a conventional convex hull. A value of 0.0 (the default) triggers an estimation of a suitable value from the average nearest neighbor distance.

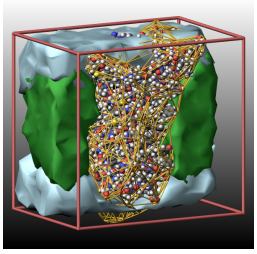
The optional *clip* keyword allows to adjust the behavior for chunks that straddle periodic boundaries. With the default value of *yes* all points are that are outside the simulation box are not used in the triangulation. When *clip* is set to *no* all points of ghost atoms that are outside the simulation box will be included in the triangulation. Typically, a value of *yes* would be used with large chunks and *no* might be preferred for small chunks.

The optional *maxreplace* keyword allows to define up to which chunk size atoms positions are replaced by those of an icosahedron to produce smoother surfaces. For larger chunks, this step has few advantages and can slow down the triangulation significantly.

The optional *mindist* keyword allows to override the heuristics used to split larger chunks when parts of it are scattered through the simulation box. The default value is 4x the largest radius in the system.

The optional *shading* keyword selects how triangle normals are determined for rendering surfaces. The *smooth* setting (the default) computes averaged per-vertex normals so that adjacent triangles appear curved and blend smoothly (except for sharp edges). The *flat* uses the face normal for all three corners of each triangle, giving the surface a faceted appearance.

### 2.82.4 Usage example



The commands below demonstrate a more complex usage of the *graphics/chunk* fix by using *fix property/atom* to assign a custom property to atoms which is used to define the chunks. These are commands that are added to the `in.rhodo` input in the `bench` folder. The full input is available in the `examples/GRAPHICS` folder.

```
# get a slice of the box in x-direction
region slice block $(xlo) $(xlo+0.6*lx) INF INF INF INF

# assign custom integer values to atoms, so we can
# group them into chunks by collections of atom types
fix   0 all property/atom i_chunk ghost yes
set type *     i_chunk 0  # default -> not part of a chunk
set type 1*35  i_chunk 1  # protein
set type 54*68 i_chunk 1  # chromophore
set type 4     i_chunk 2  # solvent (water plus Na Cl)
set type 33    i_chunk 2  # - "" -
set type 52*53 i_chunk 2  # - "" -
set type 36*51 i_chunk 3  # lipids

# define groups for individual subsets
group protein  type 1:3 5:32 34 35 54:68
group water    type 4 33 52 53
group lipid    type 36:51
group other    subtract all protein

# access the custom property and define chunks with it
compute ichunk all property/atom i_chunk
compute hull   all chunk/atom c_ichunk

# use three different fixes so we can use different settings
fix hull1 protein graphics/chunk 100 hull alpha 10.0
# use a box slice to restrict the triangulation to part of the box
# use clip option to cleanly truncate at the periodic box boundary
fix hull2 water   graphics/chunk 100 hull region slice alpha 6.0 clip yes
fix hull3 lipid   graphics/chunk 100 hull region slice alpha 6.0 clip yes

# compute rotation angle for slow rotation around z axis
# this needs 20000 steps for a full 360 degrees.
variable rot equal ((step/1000*18+180)%360)-180

# the fix vvvvvv  group selects atoms to be show, but does not affect fix graphics
dump viz protein image 100 myimage-*.png element type size 600 600 zoom 1.5 &
      shiny 0.2 fsaa yes bond atom 0.20 view 80 v_rot box yes 0.025 &
      fix hull1 const 2 0.4 fix hull2 const 1 0 fix hull3 const 1 0
```

(continues on next page)

(continued from previous page)

```
#         ^^^ wireframe ^^^         ^^^ smooth triangle surfaces ^^^

# set elements for atom colors and define diameters for space filling spheres
dump_modify viz pad 9 boxcolor indianred backcolor black backcolor2 white &
  element H H H H H H H H H C C C C C C C C C C C C C C N N N N N N N O O O O S S H H H H␣
↪H C C C C C C N O O O P Cl Na H H H N C C C C C C C C C C C &
adiam 1*9 1.92 adiam 10*22 2.72 adiam 23*29 2.48 adiam 30*33 2.432 adiam 34*35 2.88 &
adiam 36*40 1.92 adiam 41*46 2.72 adiam 47 2.48 adiam 48*50 2.432 adiam 51 2.88 &
adiam 52 3.632 adiam 53 2.176 adiam 54*56 1.92 adiam 57 2.48 adiam 58*68 2.72 &
 fcolor hull1 goldenrod fcolor hull2 lightblue fcolor hull3 forestgreen
#  ^^^^ assign colors for triangulated graphics ^^^^^

run             20000
```

---

### 2.82.5 Dump image info

Fix graphics/chunk is designed to be used with the *fix* keyword of *dump image*. The fix constructs a list of graphics objects based on the size and geometry of the chunks in the fix group and passes the information to the image rendering facility.

The *fflag1* setting of *dump image fix* determines whether the surface is rendered as connected rounded triangles (1) or as a wireframe mesh of cylinders (2). If using a wireframe mesh, the *fflag2* setting determines the diameter of the cylinders.

---

### 2.82.6 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

None of the *fix_modify* options apply to this fix.

### 2.82.7 Restrictions

This fix is part of the GRAPHICS package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix is not compatible with 2d simulations.

When running in parallel, the chunks and corresponding graphics objects are currently computed separately for each subdomain, so that the graphics will be different for all chunks that are distributed across sub-domains depending on the number of processors used.

### 2.82.8 Related commands

*compute chunk/atom*, *fix graphics/arrows*, *fix graphics/isosurface*, *fix graphics/labels*, *fix graphics/lines*, *fix graphics/objects*, *fix graphics/periodic*, *fix graphics/replica*, *dump image*

### 2.82.9 Defaults

radius = 0.0 (= auto-detect), alpha = 0.0 (= auto-detect), shading = smooth, maxreplace = 100, clip = yes, mindist = (automatic: 4x largest radius in system)

# 2.83 fix graphics/isosurface command

## 2.83.1 Syntax

```
fix ID group-ID graphics/isosurface Nevery isovalue radius keyword args ...
```

- ID, group-ID are documented in *fix* command
- graphics/isosurface = style name of this fix command
- Nevery = update graphics information every this many time steps
- isovalue = isovalue for the particle property isosurface selection
- radius = radius describing the spread of the atoms to the density grid (distance units)
- one or more keyword/args pairs may be appended
- keyword = *quality* or *property* or *filename* or *binary* or *pad*

```
quality keyword = isosurface grid resolution setting
   keyword = one of min, low, med, high, or max
property value = per-atom property used to create the isosurface grid
   value = none, mass,  c_ID, c_ID[i], f_ID, f_ID[i], v_name
      none = 1.0 for all atoms
      mass = mass of the atoms
      c_ID = per-atom vector calculated by a compute with ID
      c_ID[I] = Ith column of per-atom array calculated by a compute with ID
      f_ID = per-atom vector calculated by a fix with ID
      f_ID[I] = Ith column of per-atom array calculated by a fix with ID
      v_name = per-atom vector calculated by an atom-style variable with name
filename name = name pattern for output of a sequence of STL format mesh files␣
↪(must contain a * character to be replaced by the timestep number)
binary logical = select whether to output a binary STL file (default is text mode)
pad number = pad the timestep in the output file name with zeroes to have this many␣
↪digits (default is 0)
```

## 2.83.2 Examples

```
fix sf1 water graphics/isosurface 200 0.1 2.5 quality high property mass
fix stl water graphics/isosurface 200 0.01 1.5 filename water-isosurface-*.stl pad 5
```

## 2.83.3 Description

Added in version 11Feb2026.

This fix allows to add an isosurface graphics object representing the triangulated isosurface at a given isovalue on a grid to images rendered with *dump image* using the *fix* keyword and optionally to output the computed mesh as a series of STL format files for external processing.

The *group-ID* sets the group ID of the atoms selected to be represented by the isosurface. This may be a dynamic group.

The *Nevery* keyword determines how often the isosurface graphics data is updated. This should be the same value as the corresponding *N* parameter of the *dump* image command. LAMMPS will stop with an error message if the settings for this fix and the dump command are not compatible.

The isosurface objects will be colored by the atom type that is closest to each isosurface grid cell when the *type* coloring scheme is used in the *dump image fix* command. The color is that of the atom type's element color instead with the

*element* coloring scheme, or just a globally set constant color for the whole isosurface with the *const* coloring scheme. That color can be set with the *fcolor* keyword of the *dump modify* command. For rounded triangles, the color is interpolated across the triangle if there are different colors assigned to the different corners of the triangle.

The isosurface's transparency setting is fully opaque by default and can be changed with the *ftrans* keyword of the *dump modify* command.

The *isovalue* argument sets the isovalue used to compute the isosurface. The optimum value depends on the property on that is being used and the information that is supposed to be conveyed. It usually requires some experimentation in combination with varying the *radius* setting.

The *radius* argument sets the width of the gaussian distribution function used to distribute the per-particle data across the grid. Its value controls the smoothness of the isosurface and - as mentioned above - may need some experimentation in combination with the choice of *isovalue* to achieve the desired output.

The *quality* keyword can have any of these words as argument: "min", "low", "med", "high", or "max", and selects the grid resolution used for the isosurface. The actual grid dimensions depend on the geometry of the simulation cell.

The optional *property* keyword controls what property is used to set the values at the grid points for the isosurface. The default setting of *none* just uses a value of 1.0, resulting in the data grid representing a smoothed out number density. Other possible arguments are *mass* (for representing the smoothed out mass density) or a references to a *compute*, a *fix*, or a reference to an atom-style *variable*. The compute or fix must produce a per-atom vector or array, not a global or local quantity. In case the property is a per-atom array, the column must be selected.

The optional *filename* keyword controls whether the computed triangle mesh is exported to an STL format file for use with external visualization programs or 3d-printers. The filename must contain a star character (*) which will be replaced by the timestep number. There is a new file created for every timestep.

If LAMMPS has been compiled with the *corresponding setting* and if the filename ends with ".gz" or some other *supported compression format suffix*, the STL file is written in compressed format. A compressed STL file can be $5-10\times$ smaller than the text version, but may need to be uncompressed before it can be read into a graphics program.

The optional *binary* keyword controls whether the STL format output file is in ASCII text mode (the default when the keyword is not used or when using "no" or "off" as argument) or in binary mode. Binary STL files are about $4-5\times$ smaller than the ASCII text version, and can be written and read *much* faster. Not all programs that handle STL files can read binary files and thus they may be converted to ASCII format. LAMMPS includes the *stl_bin2text* program for that purpose.

---

### 2.83.4 Dump image info

Added in version 11Feb2026.

Fix graphics/isosurface is designed to be used with the *fix* keyword of *dump image*. The fix will construct an isosurface based on the atom positions and the selected property of the atoms in the fix group and pass the graphics geometry information about it to *dump image* so that it is included in the rendered image.

The *fflag1* setting of *dump image fix* determines whether the isosurface will be rendered as a set of connected rounded triangles (1) or as a mesh of cylinders (2).

If using a mesh of cylinders, the *fflag2* setting determines the diameter of the cylinders, otherwise it is ignored.

The *quality* settings of "min" and "low" work best with the cylinder mesh setting while the other quality settings are more suitable for a triangle mesh.

### 2.83.5 Example for using STL output in VMD

Below is an example input commands showcasing the use of the *graphics/isosurface* fix and exporting STL files. They are added to a simulation of *a bulk SPC/E water system* with 1350 water molecules.

```
region center sphere 10.0 10.0 10.0 10.0 units box
group sphere dynamic all region center

compute prop all property/atom mass
fix surf sphere graphics/isosurface 10 2.0 2.0 quality high property c_prop filename␣
↪sphere-*.stl pad 5

dump viz sphere image 10 sphere-lammps-*.png type type size 600 600 zoom 1.6 shiny 0.4␣
↪fsaa yes &
    view 70 -20 box no 0.025 fsaa yes bond atom 0.5 fix surf const 1 0.2
dump_modify viz pad 5 backcolor2 gray adiam 1 2.432 adiam 2 1.92 &
     acolor 1 firebrick acolor 2 silver fcolor surf forestgreen ftrans surf 0.25

dump xyz sphere xyz 10 sphere.xyz
dump_modify xyz element O H
```

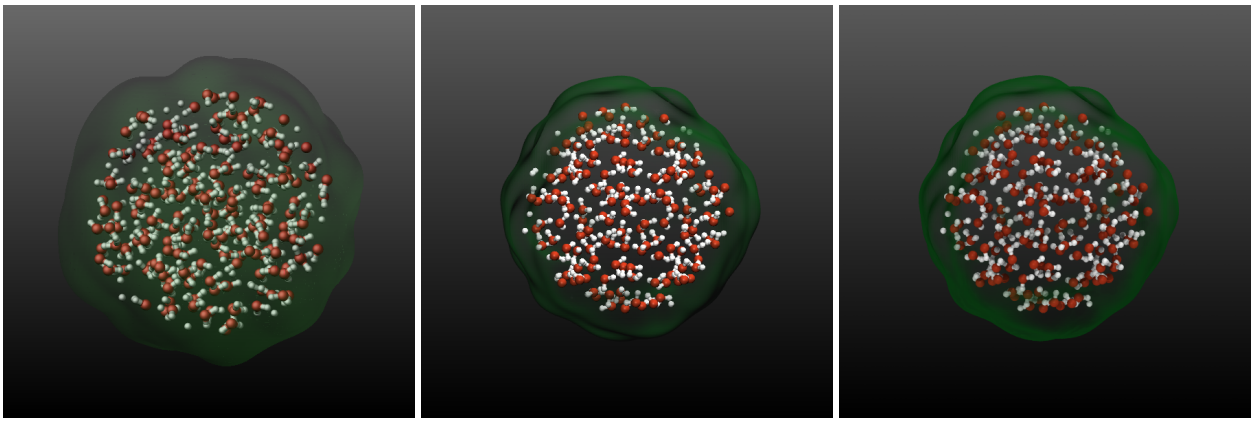
With the following script (use `vmd -e -eofexit vizsphere.vmd` to run the script) the first frame of the trajectory of those selected atoms and the corresponding STL file are then loaded into VMD via VMD/Tcl script commands and then rendered with both OpenGL (which is what you see on the screen) and then also with the Tachyon ray tracing program included with VMD. The following images compare the LAMMPS output with the VMD OpenGL output and the Tachyon ray tracer (from left to right).

```
display projection   Orthographic
display depthcue   off
display backgroundgradient on
display shadows on
display ambientocclusion on
display aoambient 0.800000
display aodirect 0.300000
display resize 600 600

mol new sphere.xyz type xyz first 0 last 0 step 1 autobonds 1 waitfor all
mol delrep 0 top
mol representation VDW 0.300000 12.000000
mol color Name
mol selection {all}
mol material AOShiny
mol addrep top
mol representation DynamicBonds 1.000000 0.200000 12.000000
mol color Name
mol selection {all}
mol material AOShiny
mol addrep top
graphics top delete all
graphics top color green
graphics top material BlownGlass
mol addfile sphere-00000.stl type stl waitfor all

render snapshot vmdscene.tga convert %s sphere-opengl.png
```

(continues on next page)

(continued from previous page)

```
render TachyonInternal vmdscene.tga convert %s sphere-raytrace.png
rm vmdscene.tga
```



### 2.83.6 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

None of the *fix_modify* options apply to this fix.

### 2.83.7 Restrictions

This fix is part of the GRAPHICS package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.83.8 Related commands

*fix graphics/arrows*, *fix graphics/chunk*, *fix graphics/labels*, *fix graphics/lines*, *fix graphics/objects*, *fix graphics/periodic*, *fix graphics/replica*, *dump image*

### 2.83.9 Defaults

quality = low, property = none, binary = no, pad = 0, filename = none

## 2.84 fix graphics/labels command

### 2.84.1 Syntax

```
fix ID group-ID graphics/labels Nevery mode keyword args ...
```

- ID, group-ID are documented in *fix* command
- graphics/labels = style name of this fix command
- Nevery = update graphics information every this many time steps
- zero or more keyword/args groups may be appended
- keyword = *image* or *text* or *colorscale*

```
image filename x y z keyword args = display image in visualization
   filename = name of the image file
   x, y, z  = position where the center of the image is located in the visualization
```

```
   any of x, y, or z can be a variable (see below)
   one or more keyword/arg pairs may be appended
   keyword = scale or transcolor
     scale value = the image is scaled by this value (default 1.0), can be a␣
↪variable (see below)
     transcolor arg = select color for transparency: auto or none or r/g/b
       auto = uses the color in the lower left corner of the image for transparency
       none = disables transparency
       r/g/b = provide three integers in the range 0 to 255 to select transparancy␣
↪color in RGB color space

text labeltext x y z keyword args = display text in visualization
   labeltext = text for the label, must be quoted if it contains whitespace
   x, y, z  = position where the center of the text is located in the visualization
   any of x, y, or z can be a variable (see below)

   keyword = fontcolor or framecolor or backcolor or transcolor or size or␣
↪horizontal or vertical
     fontcolor arg = select color for text: white (default) or black or r/g/b
       white = uses white
       black = uses black
       r/g/b = provide three integers in the range 0 to 255
     framecolor arg = select color for frame around text: silver (default) or␣
↪darkgray or white or black or r/g/b
       silver = uses a very light gray
       darkgray = uses a very dark gray
       white = uses white
       black = uses black
       r/g/b = provide three integers in the range 0 to 255
     backcolor arg = select color for background of the text: silver (default) or␣
↪darkgray or white or black r/g/b
       silver = uses a very light gray
       darkgray = uses a very dark gray
       white = uses white
       black = uses black
       r/g/b = provide three integers in the range 0 to 255
     transcolor arg = select color for transparency: silver (default) or darkgray␣
↪or white or black or none or r/g/b
       silver = uses a very light gray
       darkgray = uses a very dark gray
       white = uses white
       black = uses black
       none = disables transparency
       r/g/b = provide three integers in the range 0 to 255
     size value = set the size of the characters (default 24), can be a variable␣
↪(see below)
     horizontal = create horizontal text label
     vertical = create vertical text label

colorscale dump-ID titletext x y z keyword args = display a colormap label in␣
↪visualization
   labeltext = text for the legend of the colormap label, must be quoted if it␣
↪contains whitespace
   x, y, z  = position where the center of the colormap label is located in the␣
```

```
↪visualization
  any of x, y, or z can be a variable (see below)

  keyword = fontcolor or framecolor or backcolor or transcolor or size or length␣
↪or tics or map or horizontal or vertical
    fontcolor arg = select color for text: white (default) or black or r/g/b
      white = uses white
      black = uses black
      r/g/b = provide three integers in the range 0 to 255
    framecolor arg = select color for frame around text: silver (default) or␣
↪darkgray or white or black or r/g/b
      silver = uses a very light gray
      darkgray = uses a very dark gray
      white = uses white
      black = uses black
      r/g/b = provide three integers in the range 0 to 255
    backcolor arg = select color for background of the text: silver (default) or␣
↪darkgray or white or black r/g/b
      silver = uses a very light gray
      darkgray = uses a very dark gray
      white = uses white
      black = uses black
      r/g/b = provide three integers in the range 0 to 255
    transcolor arg = select color for transparency: silver (default) or darkgray␣
↪or white or black or none or r/g/b
      silver = uses a very light gray
      darkgray = uses a very dark gray
      white = uses white
      black = uses black
      none = disables transparency
      r/g/b = provide three integers in the range 0 to 255
    size value = set the size of the characters (default 24), can be a variable␣
↪(see below)
    length value = approximate minimal length of the colorscale label
    tics value = number of tics drawn between the colors of the colorscale label
    map value = which colormap of the dump to represent: atom (default) or grid or␣
↪bond
    horizontal = create horizontal text label
    vertical = create vertical text label
```

### 2.84.2 Examples

```
fix pix all graphics/labels 100 image teapot.png 5.0 -1.0 -2.0 transcolor auto
fix pot all graphics/labels 100 image teapot.ppm 1.0 v_ypos v_zpos scale v_prog␣
↪transcolor 19/92/192
fix lbl all graphics/labels 1000 text "LAMMPS graphics demo" 5.0 -1.0 -2.0 backcolor␣
↪darkgray framecolor black
fix info all graphics/labels 1000 text "Step: $(step)  Angle: ${rot}" 5.0 -1.0 -2.0 size␣
↪32
fix obj all graphics/labels 200 colorscale viz "Atom Velocity" 20.0 6.5 13.0 size 32␣
↪length 1000 &
                                transcolor none framecolor white backcolor darkgray␣
↪tics 12
```

(continues on next page)

(continued from previous page)

```
fix bnd all graphics/labels 200 colorscale viz "Bond Strain" 20.0 6.5 -13.0 map bond␣
↪size 32 length 1000
```

### 2.84.3 Description

Added in version 11Feb2026.

This fix allows to add either images or text as "labels" to *dump image* created images by using the *fix* keyword. This can be useful to augment images with additional graphics or text directly and without having to post-process the images. The positions can be either interpreted as coordinates in the simulation box or as coordinates in the coordinate system of the image. The selection is made by setting the *fflag1* keyword in the *dump image fix* command (see the "Dump image info" section below). When the positioning uses the coordinate system of the simulation the distance of the graphics objects from the camera is determined from the given z-coordinate and atoms or other graphics objects in the "scene" can be located in front of or behind any *image*, *text* or *colorscale* label. The label is *always* parallel to the image plane.

When the image coordinate system is used, the labels are *always* on top, and if two labels are overlapping, the label that is added to the image *first* will be on top of the other. That order cannot be changed within the same fix, but you can use multiple fix commands and then the order of the *fix* keywords in the *dump image"* command line determines the order and thus which label is drawn on top of the other.

The *group-id* is ignored by this fix.

The *Nevery* keyword determines how often the graphics data is updated. This should be the same value as the corresponding *N* parameter of the *dump* image command. LAMMPS will stop with an error message if the settings for this fix and the dump command are not compatible.

The *image* keyword reads an image file and adds it to the visualization centered around the provided position and optionally scaled by the provided scale factor. The filename suffix determines whether LAMMPS will try to read a file in JPEG, PNG, TGA, or PPM format. If the suffix is ".jpg" or ".jpeg", then LAMMPS attempts to read the image in JPEG format, if the suffix is ".png", then LAMMPS attempts to read the image in PNG format, and if the suffix is ".tga" then LAMMPS will read the file in TGA format. Otherwise LAMMPS will try to read the image in ppm (aka netpbm) format. Not all variants of those file formats are compatible with image reader code in LAMMPS. If LAMMPS encounters an incompatible or unrecognizable file format or a corrupted file, it will stop with an error.

If LAMMPS detects during a run that the file has been changed, it will re-read it. This allows for instance to create a plot using internal LAMMPS data or from processing an output file during the simulation with the matplotlib python module using a *python* command and *fix python/invoke* and then embed the resulting image into the dump image output. See below for a minimal example for such a setup.

When using the *image* keyword, the name of the image file and its position in the "scene" are required arguments. Optional keyword / value pairs may be added:

> The *scale* value determines if the image is scaled before it is added to the *dump image* output. LAMMPS currently employs a bilinear scaling algorithm.
>
> The *transcolor* value selects a color for transparency. All pixels in the image with that color will be skipped when the image is rendered. The color is specified as an R/G/B triple with values ranging from 0 to 255 for each channel. There are also two special arguments: *auto* will pick the color of the pixel in the lower left corner as transparency color and *none* will disable all transparency processing (this is the default).

The *text* keyword will process a provided text into a pixmap and adds it to the visualization centered around the provided position in a similar fashion as with the *image* keyword. The requirements for the text argument are the same as in the *fix print* command: it must be a single argument, so text with whitespace must be quoted; and the text may contain equal style or immediate variables using the `${name}` or `$(expression)` format. The variables are evaluated and expanded at every *Nevery* time step.

When using the *text* keyword, the text and its position in the "scene" are required arguments. Optional keyword / value pairs may be added:

The *size* value determines the size of the letters in the text in pixels (approximately) and values between 4 and 512 are accepted. The default value is 24.

There are four color settings: *fontcolor* or *framecolor* or *backcolor* or *transcolor*. The color can be specified for all of those either as an R/G/B triple with values ranging from 0 to 255 for each channel (e.g. yellow would be "255/255/0"). There are also a few shortcuts for common choices: *silver*, *darkgray*, *white*, *black*. The default *backcolor* value is *silver*.

- *fontcolor* selects the color for the text, default is *white*
- *backcolor* selects the color for the background, default is *silver*
- *framecolor* selects the color for the frame around the background, default is *silver*.
- *transcolor* value selects a color for transparency, default is *silver*. If this color is the same as any of the other color settings, those pixels are not drawn. Thus with the default settings, the text will be rendered in white without background or frame. The *none* setting for *transcolor* disables transparency processing.

When rendering text with transparent background it is recommended to select a similar color but slightly darker or brighter color as background. This will reduce unwanted color effects at the edges due to anti-aliasing.

The *horizontal* keyword selects creating a horizontal text label (this is the default setting). The *vertical* keyword selects creating a vertical text label instead.

The *colorscale* keyword will create a colormap legend indicating the mapping of values to colors in the *dump image* instance with the given dump-ID and adds it to the visualization centered around the provided position in a similar fashion as with the *image* or *text* keywords. The requirements for the text argument are the same as in the *fix print* command: it must be a single argument, so text with whitespace must be quoted; and the text may contain equal style or immediate variables using the `${name}` or `$(expression)` format. The variables are evaluated and expanded at every *Nevery* time step. The text is shown in the center of and above the colormap. To the left from the text is the lower boundary value and to the right the upper boundary value. The colors are created by a linear interpolation between the lower and upper boundary value and writing out pixels in the corresponding color. The fix will receive the actual values from the dump with the given *dump-ID*.

**Dynamic color maps**

When using a dynamic color map with "min" or "max" as the upper or lower range values of the map, the dump will execute only *after* the fix, and thus the upper and lower boundary values will be those from the *previous* step where the dump created an image. will be determined every time the fix is executed and the numbers updated accordingly. Thus when adding a *colorscale* label with this fix it is generally recommended to use a map with a fixed range. This is especially true when creating movies as a fixed range prevents the color scale label to shrink or grow due to the different width of characters.

When using the *colorscale* keyword, the dump-ID, text and its position in the "scene" are required arguments. Optional keyword / value pairs may be added:

The *size* value determines the size of the letters in the text in pixels (approximately) and values between 4 and 512 are accepted. The default value is 24. The size (height and width) of the colorbar follows the size of the text.

The *length* value allows to set a minimal length of the colorscale label. For technical reasons, this is not exactly enforced, but rather a rough approximation that is used to determine the amount of padding in the text.

The *tics* value determines how many "tics" or lines separating the colors are drawn. This can simplify determining which value a specific color corresponds to.

Added in version 4Jul2026.

The *map* value selects which of the colormaps of the *dump image* instance the legend represents. A dump image has separate colormaps for coloring atoms (set with *dump_modify amap*), grid cells (*gmap*), and bonds (*bmap*). The *atom* setting (the default) selects the atom colormap, *grid* the grid cell colormap, and *bond* the bond colormap. If the selected colormap is not actually used to color anything in the corresponding dump image, a warning is printed and the legend will show that colormap's (unused) default range.

There are four color settings: *fontcolor* or *framecolor* or *backcolor* or *transcolor*. The color can be specified for all of those either as an R/G/B triple with values ranging from 0 to 255 for each channel (e.g. yellow would be "255/255/0"). There are also a few shortcuts for common choices: *silver*, *darkgray*, *white*, *black*. The default *backcolor* value is *silver*.

- *fontcolor* selects the color for the text, the border of the colorbar and the tics. default is *white*
- *backcolor* selects the color for the background, default is *silver*
- *framecolor* selects the color for the frame around the background, default is *silver*.
- *transcolor* value selects a color for transparency, default is *silver*. If this color is the same as any of the other color settings, those pixels are not drawn. Thus with the default settings, the text will be rendered in white without background or frame. The *none* setting for *transcolor* disables transparency processing.

When rendering text with transparent background it is recommended to select a similar color but slightly darker or brighter color as background. This will reduce unwanted color effects at the edges due to anti-aliasing.

The *horizontal* keyword selects creating a horizontal colorscale label (this is the default setting). The *vertical* keyword selects creating a vertical text label instead.

There may be multiple *image* or *text* or *colorscale* keywords with their arguments in a single fix *graphics/labels* command.

The arguments for the positions of an *image* or *text* and the *scale* factor of an *image* or the *size* of a *text* can be specified as equal-style *variables*, namely *x*, *y*, *z*, *scale*, or *size*. If any of these values is a variable, it should be specified as *v_name*, where *name* is the variable name. In this case, the variable will be evaluated each *nevery* timestep, and its value used to position and resize the image or text. Please see the documentation of the *fix graphics/objects* command for a more detailed discussion on using variables with graphics objects.

---

### 2.84.4 Dump image info

The fix graphics/labels command is designed to be used with the *fix* keyword of *dump image*. The fix adds images or text to the visualization.

The color style setting for the fix in the *dump image* has no effect on either image or text labels. The transparency is by default fully opaque and can be changed with *dump_modify ftrans*.

The *fflag1* setting of *dump image fix* determines how the coordinates for the location of the center of the image or the center of the text label are interpreted. Setting *fflag1* to 0 uses the simulation box coordinate system (x, y, and z) while setting *fflag1* to 1 uses the image coordinate system where (0,0) is the location of the lower left corner and (<image width>, <image height>) the upper right corner. In the latter case, the z-coordinate is ignored and the image or label is placed on top of everything.

The *fflag2* settings of *dump image fix* is ignored.

---

### 2.84.5 Including a "dynamic" image

The LAMMPS input commands below provide a demonstration for creating and updating a plot during a run and importing it into a visualization. This requires to compile LAMMPS with the *PYTHON* package and also compile and install the *LAMMPS Python module*.

The first *python* command loads the matplotlib and LAMMPS Python modules and configures *matplotlib* to use the non-interactive 'agg' back end for creating image files.

The second *python* command defines the `myplot()` Python function that is supposed to be called regularly during the run from *fix python/invoke*. This function has to accept two arguments, the LAMMPS object pointer and an integer as required by the fix. The LAMMPS object pointer can be utilized to query the running LAMMPS instance about internal data. In this example, we only retrieve the LAMMPS version and add it to the plot title. By default, plots have an opaque white background and black lines and text. In order to overlay the plot as a transparent image, all lines and text are set to use the color white, while backgrounds are set to use a very bright gray (to minimize anti-aliasing artifacts when deleting the background pixels).

The two fix commands invoke the python function and read and make the resulting PNG format image available to dump image. The plot is updated only for every 10th dumped image.

The final lines are *dump image* commands for integrating the generated plot into the visualization of the atom.

```
python source here """
import matplotlib
matplotlib.use('agg')
import matplotlib.pyplot as plt
from lammps import lammps
"""

python myplot input 2 SELF 0 format pi here """
def myplot(lmpptr, vflag):
   lmp = lammps(ptr=lmpptr)
   fig, ax = plt.subplots(facecolor=(0.9,0.9,0.9),edgecolor='white')
   ax.set_facecolor((0.9,0.9,0.9))
   ax.set_title('Demo Plot: LAMMPS version ' + str(lmp.version()) ,color='white')
   ax.set_xlabel('Time',color='white')
   ax.set_ylabel('Value',color='white')
   ax.tick_params(colors='white')
   for spine in ax.spines.values():
       spine.set_edgecolor('white')
   ax.plot([1,2,3,4,5,6],[2,3,2.5,3.1,2.8,3.0],color='white', linestyle='--')
   fname = 'myplot.png'
   plt.savefig(fname,dpi=180)
   plt.close()
"""

fix plot all python/invoke 1000 post_force myplot
fix label all graphics/labels 100 image myplot.png 20.0 9.0 10.0 transcolor auto scale 0.
↪5

dump viz all image 100 image-*.png type type size 800 800 view 80 10 box yes 0.02 &
             fsaa yes shiny 0.1 ssao yes 23154 0.8 zoom 1.4 fix label type 0 0
dump_modify viz pad 6 backcolor2 gray backcolor darkgray boxcolor silver
```

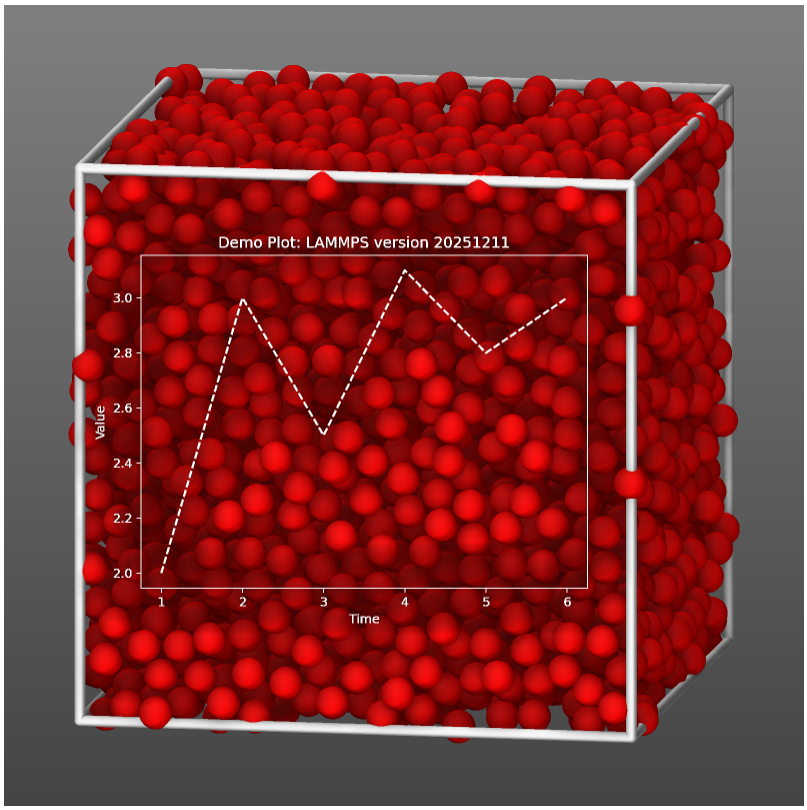



Fig. 1: Example image output for adding the above commands to the `melt` example.

### 2.84.6 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

None of the *fix_modify* options apply to this fix.

### 2.84.7 Restrictions

This fix is part of the GRAPHICS package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

To read JPEG or PNG format images, support for the corresponding graphics libraries must have been compiled and linked into LAMMPS. Please see the *instructions for building LAMMPS with the GRAPHICS package* for more information on how to do that.

### 2.84.8 Related commands

*fix print*, *fix graphics/arrows*, *fix graphics/chunk*, *fix graphics/isosurface*, *fix graphics/lines*, *fix graphics/objects*, *fix graphics/periodic*, *fix graphics/replica*, *dump image*

### 2.84.9 Default

transcolor = "none" for *image* and "silver" for *text*, scale = 1.0, fontcolor = white, backcolor = silver, framecolor = silver, size = 24, horizontal, tics = 0

## 2.85 fix graphics/lines command

### 2.85.1 Syntax

```
fix ID group-ID graphics/lines Nevery Nrepeat Nfreq Nlength
```

- ID, group-ID are documented in *fix* command
- graphics/lines = style name of this fix command
- Nevery = retrieve unwrapped position information every this many timesteps
- Nrepeat = number of times to use the position information for calculating averages
- Nfreq = store position averages every this many timesteps
- Nlength = length of averaged position history

### 2.85.2 Examples

```
fix  lines  ogroup  graphics/lines 10 50 500 15
```

### 2.85.3 Description

Added in version 11Feb2026.

This fix allows to add a trace of averaged atom positions in the fix group to images rendered with *dump image* using the *fix* keyword. This kind of position trace is sometimes referred to as "trajectory lines".

The trace is represented by a chain of connected cylinders where the endpoints are taken from the current atom positions and an internal history of averaged positions of the atoms. The averaging is performed by using *fix ave/atom* internally on unwrapped atom positions taken from *compute property/atom*. The *Nevery*, *Nrepeat*, and *Nfreq* values are passed on to the internal fix *ave/atom* instance for averaging. The averaged unwrapped positions are wrapped back into the

simulation box and stored internally using up to *Nlength* sets. For any additional sets of positions, the then oldest set in the history storage will be overwritten and thus limiting the length of the trace.

The *group-ID* sets the group ID of the atoms selected to have the selected property represented. This may *not* be a dynamic group.

The *Nfreq* value determines how often the graphics data is updated. This should be the same value as the corresponding *N* parameter of the *dump* image command. LAMMPS will stop with an error message if the settings for this fix and the dump command are not compatible.

---

### 2.85.4 Dump image info

Fix graphics/lines is designed to be used with the *fix* keyword of *dump image*. The fix will add a connected chain of cylinders based on the averaged position history of the atoms in the fix group to the rendered image.

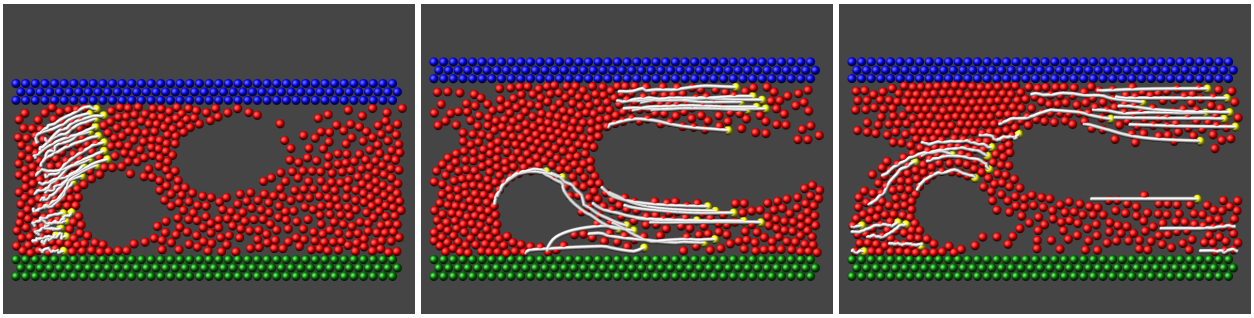
The color of the cylinders is by default that of the atoms when using color styles "type" or "element". With color style "const" the default value of "white" can be changed using *dump_modify fcolor*. The transparency is by default fully opaque and can be changed with *dump_modify ftrans*.

The *fflag1* setting of *dump image fix* determines whether the cylinder objects are capped with spheres: 0 means no caps, 1 means the lower end is capped, 2 means the upper end is capped, and 3 means both ends are capped.

The *fflag2* setting determines the diameter of the cylinders.

The following input can be added to the `examples/obstacle/in.obstacle` example input after adjusting its *create_box* and *mass* commands to support 4 atom types instead of 3.

```
# select atoms to trace and assign atom type 4 so they have a different color (yellow)
region   trace block 2 3 1.25 8.75 INF INF
group    trace region trace
set      group trace type 4

# create trajectory lines by averaging positions over 10 MD steps each 10 steps apart
fix      lines trace graphics/lines 10 10 100 25
# visualize the trace in silver with a diameter of 0.6 which is half of that of the atoms
dump     viz   all image 100 obstacle-*.png type type size 800 600 zoom 2.5 center s 0.
↪50 0.7 0 &
                                    shiny 0.5 fsaa yes box no 0.025 fix lines const␣
↪1 0.6
dump_modify  viz  pad 6 backcolor darkgray adiam * 1.2 fcolor lines silver
```



### 2.85.5 Restart, fix_modify, output, run start/stop, minimize info

This fix writes its current status to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the fix continues in an uninterrupted fashion. Upon restarting, the fix checks that the *Nevery*, *Nrepeat*, *Nfreq*, and *Nlength* settings are the same as when the restart file was written. Otherwise it stops with an error.

None of the *fix_modify* options apply to this fix.

This fix computes a global scalar representing the current length of the position history vector.

### 2.85.6 Restrictions

This fix is part of the GRAPHICS package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.85.7 Related commands

*fix graphics/arrows*, *fix graphics/chunk*, *fix graphics/isosurface*, *fix graphics/labels*, *fix graphics/lines*, *fix graphics/objects*, *fix graphics/periodic*, *fix ave/atom*, *dump image*

### 2.85.8 Defaults

none

## 2.86 fix graphics/objects command

### 2.86.1 Syntax

```
fix ID group-ID graphics/objects Nevery keyword args ...
```

- ID, group-ID are documented in *fix* command
- graphics/objects = style name of this fix command
- Nevery = update graphics information every this many time steps
- one or more keyword/args pairs may be appended
- keyword = *sphere* or *cylinder* or *arrow* or *cone* or *progbar*

```
sphere args = type x y z R
  type = an atom type value to select the color of the sphere
  x, y, z = position of the center of the sphere (distance units)
  R = sphere radius (distance units)
  any of x, y, z, and R can be a variable (see below)
cylinder args = type x1 y1 z1 x2 y2 z2 R
  type = an atom type value to select the color of the cylinder
  x1, y1, z1, x2, y2, z2 = positions of the centers at the two ends of the cylinder␣
↪(distance units)
  R = cylinder radius (distance units)
  any of x1, y1, z1, x2, y2, z2, and R can be a variable (see below)
arrow args = type x1 y1 z1 x2 y2 z2 R ratio
  type = an atom type value to select the color of the arrow
  x1, y1, z1, x2, y2, z2 = positions of the centers at the bottom (x1,y1,z1) and␣
↪the tip (x2,y2,z2) of the arrow (distance units)
  R = cylinder radius (distance units)
  ratio = tip to body ratio (unitless)
```

```
  any of x1, y1, z1, x2, y2, z2, and R can be a variable (see below)
cone args = type x1 y1 z1 x2 y2 z2 R1 R2 sides
  type = an atom type value to select the color of the cone
  x1, y1, z1, x2, y2, z2 = positions of the centers at the bottom (x1,y1,z1) and␣
↪the top (x2,y2,z2) of the cone (distance units)
  R1 = bottom radius (distance units)
  R2 = top radius (distance units)
  sides = bitmap value between 0 and 7 deciding whether bottom cap (1), top cap (2)␣
↪or side (4) is drawn (unitless)
  any of x1, y1, z1, x2, y2, z2, R1 and R2 can be a variable (see below)
progbar args = type1 type2 dim x y z length R ratio tics
  type1 = an atom type value to select the color of the progress bar body and the␣
↪tics
  type2 = an atom type value to select the color of the progress indicator
  dim = x or y or z, direction of the progress bar
  x, y, z = position of the progress bar center (distance units)
  length = length of progress bar (distance units)
  R = cylinder radius (distance units)
  ratio = progress status (unitless)
  tics = number of tics (unitless)
  only the progress ratio value can be a variable (see below)
```

## 2.86.2 Examples

```
fix 1 all graphics/objects 100 sphere 1 0.0 0.0 15.0 3.0 sphere 2 0.0 0.0 5.0 1.0
fix 1 all graphics/objects 1000 sphere 1 v_x v_y 0.0 v_radius cylinder 1 v_x v_y 0.0 v_x␣
↪v_y 10.0 3.0
fix 2 all graphics/objects 100 progbar 3 1 z 0.012 -0.012 0.0025 0.03 0.0003 v_prog 10
```

## 2.86.3 Description

Added in version 11Feb2026.

This fix allows to add arbitrary objects to images rendered with *dump image* using the *fix* keyword.

The *group-ID* is ignored by this fix.

The *Nevery* keyword determines how often the graphics object data is updated. This should be the same value as the corresponding *N* parameter of the *dump* image command. LAMMPS will stop with an error message if the settings for this fix and the dump command are not compatible.

Available graphics objects are (see above for exact command line syntax):

- *sphere* - a sphere defined by its center location and its radius
- *cylinder* - a cylinder defined by its two center endpoints and its radius
- *arrow* - a cylinder with a cone at one side (see note below)
- *cone* - a truncated cone with a flat circular cap at either side (see note below)
- *progbar* - progress bar along a selected axis and with optional tick marks

The *type* quantity determines the color of the object. It represents an *atom* type and the object will be colored the same as the corresponding atom type when the "type" or "element" color style is used in the *dump image fix* command. For the *progbar* object **two** atom type values must be specified. For color style "const" the color will be set globally to the same color for *all* objects of this fix instance, which can be changed using a *dump modify fcolor* command. The transparency is by default fully opaque and can be changed globally with *dump_modify ftrans*.

The *x*, *y*, and *z* parameters correspond to the position of the center of the object (*sphere* and *progbar*). *x1*, *y1*, and *z1* as well as *x2*, *y2*, and *z2* are instead representing the top and bottom position of a graphics object (*cylinder*, *arrow*, and *cone*). The *R* parameter determines the radius. For the *cone* object there is a bottom radius (*R1*) and top radius (*R2*).

The *cone* object has an additional setting that selects whether the circular cap at the bottom (value = 1), or the circular cap at the top (value = 2) or the side (value = 4) is drawn. The values are added and thus if the cone with both caps and the side should be drawn the required sides setting would be 7.

The *progbar* object has four additional parameters: *dim* sets the direction of the progress bar, "x", "y", or "z"; *length* sets the length of the entire object; *ratio* sets the ratio of progress and is expected to be between 0.0 and 1.0 (larger or smaller values will be reset to 1.0 or 0.0, respectively); and *tics* determines the number of tics shown on the progress bar, this must be a number between 0 and 20. Unlike for the other graphics objects, all settings except for *ratio* are fixed and cannot be a variable reference.

---

Many of the quantities defining a graphics object can be specified as an equal-style *variable*, namely *x*, *y*, *z*, or *R* for a *sphere* or *x1*, *y1*, *z1*, *x2*, *y2*, *z2*, or *R* for a *cylinder* or *x1*, *y1*, *z1*, *x2*, *y2*, *z2*, *R1*, or *R2* for a *cone*. If any of these values is a variable, it should be specified as *v_name*, where *name* is the variable name. In this case, the variable will be evaluated each *Nevery* timestep, and its value used to define the graphics object location, orientation, or size.

Note that equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify graphics object properties like position, orientation, radius or more that change as a function of time or span consecutive runs in a continuous fashion. For the latter, see the *start* and *stop* keywords of the *run* command and the *elaplong* keyword of *thermo_style custom* for details.

For example, if a sphere's x-position is specified as v_x, then this variable definition will keep its center at a relative position in the simulation box, 1/4 of the way from the left edge to the right edge, even if the box size changes:

```
variable x equal "xlo + 0.25*lx"
```

Similarly, either of these variable definitions will move the sphere from an initial position at 2.5 at a constant velocity of 5:

```
variable x equal "2.5 + 5*elaplong*dt"
variable x equal vdisplace(2.5,5)
```

If a sphere's radius is specified as v_r, then these variable definitions will grow the size of the sphere at a specified rate.

```
variable r0 equal 0.0
variable rate equal 1.0
variable r equal "v_r0 + step*dt*v_rate"
```

### 2.86.4 Dump image info

Added in version 11Feb2026.

Fix graphics/objects is designed to be used with the *fix* keyword of *dump image*. The fix will pass geometry information about the objects listed on the command line to *dump image* so that they are included in the rendered image.

The *fflag1* setting of *dump image fix* determines whether cylinder elements are capped with spheres: 0 means no caps, 1 means the lower end is capped, 2 means the upper end is capped, and 3 means both ends are capped. This applies to *cylinder* objects and the elements of the *progbar* object. For *arrow* objects the *fflag1* setting allows to adjust the length of the arrow. This allows for example to conveniently shrink the arrow by an atom diameter when the tip is located at the position of an atom and thus the tip would otherwise be (partially) obscured by the sphere representing the atom. If the arrow positions already take that into account, *fflag1* should be set to zero.

The *fflag2* setting allows you to adjust the radius of the rendered sphere, cylinder, arrow, or cone items comprising the objects. Since the radius of these objects is an input parameter for this fix, it is recommended to set this flag to 0.0.

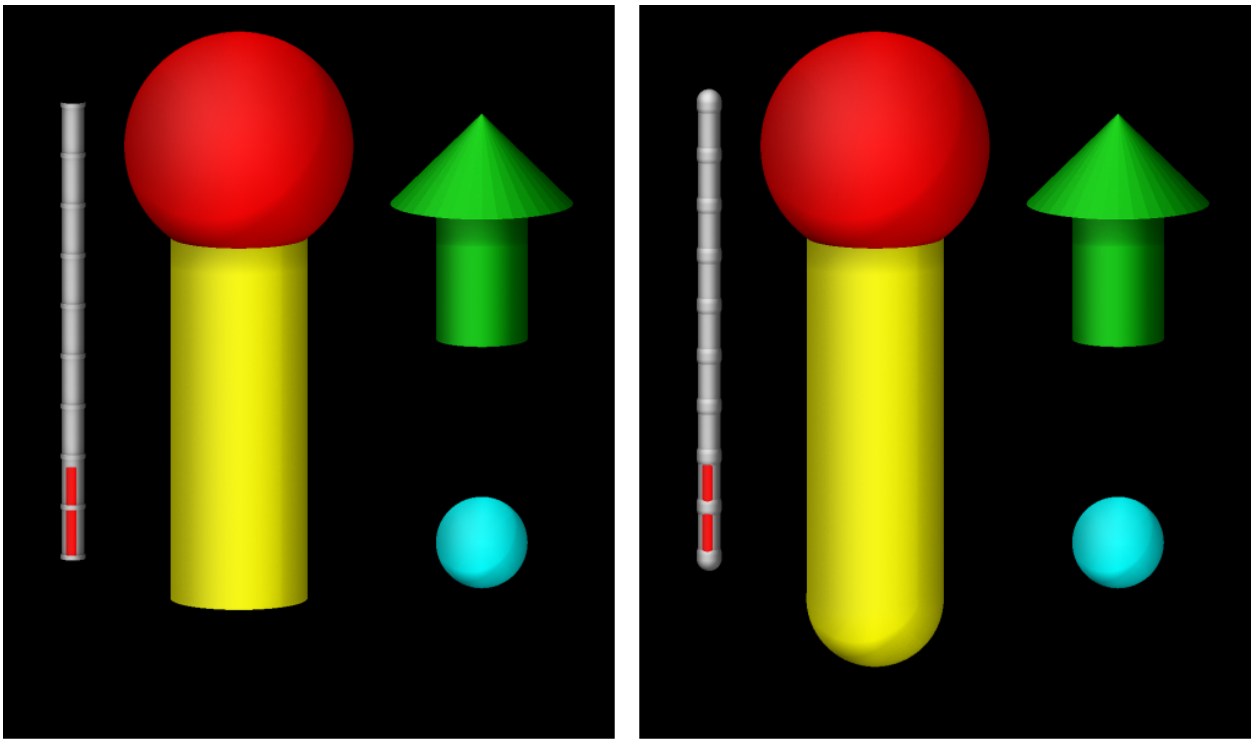


Fig. 2: Example of graphics objects rendered with *fix graphics/objects* with *fflag1* setting of 0 (left) and 3 (right)

These images were created with the following input file:

```
units           si
region      simulation_box block -0.01 0.01 -0.01 0.01 -0.01 0.01 units box
create_box 5 simulation_box
mass * 1

variable xpos equal 0.004*sin(PI*step/1000)
variable ypos equal 0.004*cos(PI*step/1000)
variable zpos equal 5.0*v_xpos
variable prog equal (step)/10000.0
fix gra all graphics/objects 50 sphere 5 v_xpos v_ypos -0.009 0.002 &
                         sphere 1 0.01 -0.005 0.01 0.005 &
                         progbar 3 1 z 0.012 -0.012 0.002 0.02 0.0005 v_prog 10 &
                         cylinder 4 0.01 -0.005 -0.01 0.01 -0.005 0.01 0.003  &
                         arrow 2 v_xpos v_ypos 0.0 v_xpos v_ypos 0.01 0.002 0.4

dump viz all image 100 myimage2-*.ppm type type size 500 600 zoom 1.24872 &
              shiny 0.2 fsaa yes ssao yes 4539 0.6 box no 0.01 axes no 0.5 0.025 &
              fix gra type 3 0 view 80 10 center s 0.4 0.3 0.4
dump_modify viz pad 9 backcolor black  acolor 3 gray

run 2000
```

### 2.86.5 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

None of the *fix_modify* options apply to this fix.

### 2.86.6 Restrictions

This fix is part of the GRAPHICS package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.86.7 Related commands

*fix graphics/arrows*, *fix graphics/chunk*, *fix graphics/isosurface*, *fix graphics/labels*, *fix graphics/lines*, *fix graphics/periodic*, *fix graphics/replica*, *dump image*

### 2.86.8 Default

none

## 2.87 fix graphics/periodic command

### 2.87.1 Syntax

```
fix ID group-ID graphics/periodic Nevery keyword args ...
```

- ID, group-ID are documented in *fix* command
- graphics/periodic = style name of this fix command
- Nevery = update graphics information every this many time steps
- zero or more keywords or keyword/value pairs may be appended
- keyword = *xlo* or *xhi* or *ylo* or *yhi* or *zlo* or *zhi* or *radius* or *atoms* or *bonds*

```
xlo, xhi, ylo, yhi, zlo, zhi = enable periodic images of atoms and bonds to either␣
↪side of the simulation box in the given direction
radius value = sets the atom radius
   value = either "auto" or a number (distance units)
atoms yes/no = enables or disables displaying periodic images of atoms
bonds yes/no = enables or disables displaying periodic images of bonds
```

### 2.87.2 Examples

```
fix vec all graphics/periodic 10 ylo zhi zlo yhi xlo
fix vec all graphics/periodic 1000 ylo zhi zlo yhi bonds no radius 0.5
```

### 2.87.3 Description

Added in version 11Feb2026.

This fix allows to add graphics of periodic images of atoms and bonds to *dump image* images using the *fix* keyword. This can be useful to visualize periodic systems.

The *group-ID* sets the group ID of the atoms selected to be displayed as periodic images. For bonds to be displayed, *both* atoms of the bond have to be inside the group.

The *Nevery* keyword determines how often the periodic graphics data is updated. This should be the same value as the corresponding *N* parameter of the *dump* image command. LAMMPS will stop with an error message if the settings for this fix and the dump command are not compatible.

The *xlo*, *xhi*, *ylo*, *yhi*, *zlo*, *zhi* keywords, if set, enable display of a periodic image of the system to the corresponding side in the corresponding direction of the principal simulations cell. If all keywords are used, there will be 26 additional copies of the system rendered.

The *radius* keyword determines the radius of the atoms. If a value of "auto" is used, the radius is inherited from the atom type.

---

### 2.87.4 Dump image info

Added in version 11Feb2026.

Fix graphics/periodic is designed to be used with the *fix* keyword of *dump image*. The fix adds graphics objects of periodic images of atoms and bonds in the fix group to *dump image* so that they are included in the rendered image.

The color of the atoms and bonds is by default the same as that of the atoms and bonds in the principal simulation cell when using color styles "type" or "element" with the fix command. With fix color style "const" the default value of "white" can be changed using *dump_modify fcolor*. The transparency is by default fully opaque and can be changed with *dump_modify ftrans*.

The *fflag1* setting of dump of *dump image fix* determines if the bonds are capped with spheres: a value of 0 means no caps, a value of 1 a cap at the lower end, a value of 2 a cap at the upper end, and a value of 3 caps at both ends. When also replicating atoms, a value other than 0 would be redundant, otherwise a value of 3 is probably the desired choice.

The *fflag2* settings of *dump image fix* allows to modify the bond diameter relative to the automatically chosen one. In most use cases a value of 0.0 is probably the desired choice.

### 2.87.5 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

None of the *fix_modify* options apply to this fix.

### 2.87.6 Restrictions

This fix is part of the GRAPHICS package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Currently only periodic images of atoms and bonds in each direction can be displayed.

Body particles or ellipsoids and similar are not fully supported; they are shown as spheres with this fix.

### 2.87.7 Related commands

*fix graphics/arrows*, *fix graphics/chunk*, *fix graphics/isosurface*, *fix graphics/labels*, *fix graphics/lines*, *fix graphics/objects*, *fix graphics/replica*, *dump image*

### 2.87.8 Default

radius = auto, atoms = yes, bonds = yes, if supported by atom style otherwise no, no periodic graphics

# 2.88 fix graphics/replica command

## 2.88.1 Syntax

```
fix ID group-ID graphics/replica Nevery type keyword args ...
```

- ID, group-ID are documented in *fix* command
- graphics/replica = style name of this fix command
- Nevery = update graphics information every this many time steps
- keyword = *display* or *average*

```
display radius = radius for the atoms or -1 to use the radius dump image uses for␣
↪the atom type
average radius = radius for the atoms or 0 to set the radius to that of the largest␣
↪distance from the center
```

## 2.88.2 Examples

```
fix sf1 water graphics/replica 200 display 1.0 average 0
```

## 2.88.3 Description

Added in version 4Jul2026.

This fix allows to add spheres to images rendered with *dump image* using the *fix* keyword to represent atoms from all replicas of a multi-replica simulation.

The *group-ID* sets the group ID of the atoms selected to be represented. This may be a dynamic group.

The *Nevery* keyword determines how often the replica graphics data is updated. This should be the same value as the corresponding *N* parameter of the *dump* image command. LAMMPS will stop with an error message if the settings for this fix and the dump command are not compatible.

There are two keywords available that determine what is shown: *display* and *average*. With *display* all atoms in the fix group from all replica will be displayed. With *average* only the average position of the atoms with the same atom-ID across all replica will be shown.

The *radius* quantity determines the radius of the atoms. A value > 0 sets an explicit radius; a value < 0 will use the same radius used by dump image for local atoms of the same atom type. For the keyword *average*, a *radius* sets the atom radius to the largest distance of an atom to the average position across all replica.

---

## 2.88.4 Dump image info

Added in version 4Jul2026.

Fix graphics/replica is designed to be used with the *fix* keyword of *dump image*. The fix will add spheres based on the atoms in the fix group across all replica to *dump image* so that they are included in the rendered image.

The *fflag1* setting of *dump image fix* are currently ignored.

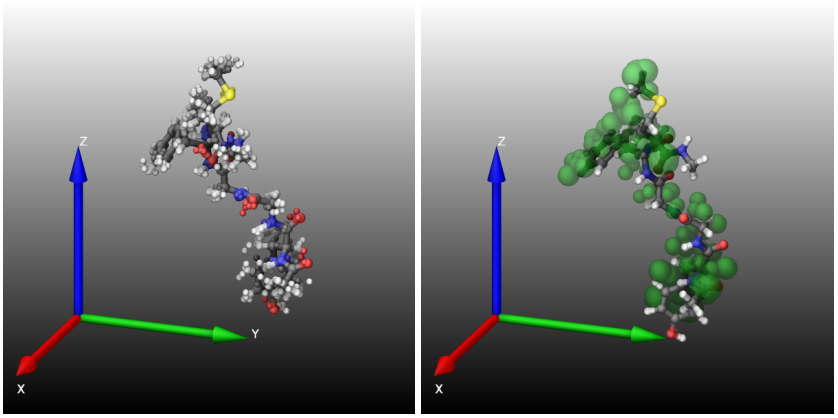
and *fflag2* setting of *dump image fix* is used as an adjustment to the radius of the rendered sphere. This can be used to grow or shrink the radius that is selected by *dump image* for the atom type.

---

### 2.88.5 Usage example

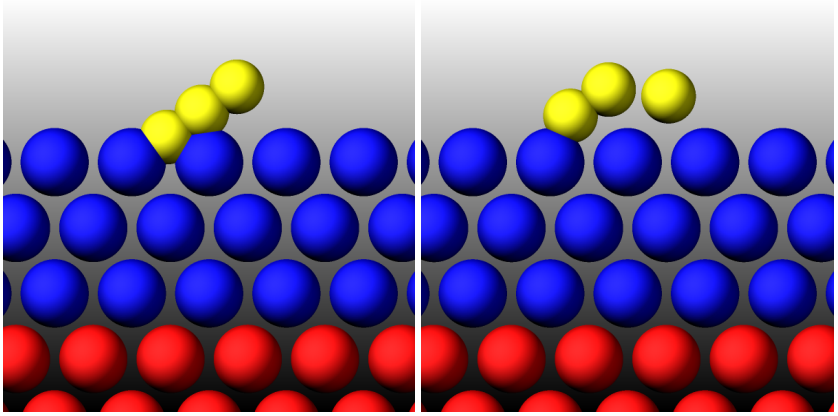
The following lines can be added to the `peptide` example input to run it in multi-partition mode with diverging trajectories and then visualize the atom positions from the different replica or the spread.

```
# reinitialize velocities differently on each partition
variable part uloop 16
velocity all create 275.0 $(12315235*v_part)

fix  sf1  peptide   graphics/replica 100 display 0.5
fix  sf2  peptide   graphics/replica 100 average 1.0

# must use dump image only on one partition
partition yes 1 dump viz peptide image 100 myimage-*.png element type size 600 600 zoom␣
↪1.77156 shiny 0.2  &
                ssao yes 23184 0.4  fsaa yes bond atom type view 70 20 box no 0.0 axes␣
↪yes 0.5 0.05 &
                fix sf2 const 0 0 &
                fix sf1 element 0 0 &

partition yes 1 dump_modify viz pad 6 backcolor black backcolor2 white &
             element C C O H N C C C O H H S O H &
             adiam 1*2 0.85 adiam 3 0.76 adiam 4 0.6 adiam 5 0.775 adiam 6*7 0.85 adiam␣
↪8 0.85 &
             adiam 9 0.76 adiam 10*11 0.6 adiam 12 0.9 adiam 13 0.76 adiam 14 0.6 &
             ftrans sf2 0.5 fcolor sf2 green
```



The following commands can be added to the `neb.hop1` example to show the initial (left image below) and final (right image below) state of the minimized transition path between adjacent energy minima. Each yellow sphere represents the same atom in different replicas.

```
fix  sf1  nebatoms  graphics/replica 10 display 0.8
partition yes 1 dump viz all image 10 myimage-*.png type type size 600 600 zoom 1.5 &
                 shiny 0.2 fsaa yes box no 0.0 axes yes 0.5 0.05 fix sf1 const 0 0
partition yes 1 dump_modify viz pad 6 backcolor black backcolor2 white fcolor sf1 yellow
```



### 2.88.6 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

None of the *fix_modify* options apply to this fix.

This fix is invoked during an energy minimization when using the *minimize* command and other commands the invoke minimizations like *neb*.

### 2.88.7 Restrictions

This fix is part of the GRAPHICS package. It is only only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.88.8 Related commands

*fix graphics/arrows*, *fix graphics/chunk*, *fix graphics/isosurface*, *fix graphics/labels*, *fix graphics/lines*, *fix graphics/objects*, *fix graphics/periodic*, *dump image*

### 2.88.9 Default

none

## 2.89 fix gravity command

Accelerator Variants: *gravity/omp*, *gravity/kk*

### 2.89.1 Syntax

```
fix ID group gravity magnitude style args
```

- ID, group are documented in *fix* command
- gravity = style name of this fix command
- magnitude = size of acceleration (force/mass units), magnitude can be a variable (see below)
- style = *chute* or *spherical* or *gradient* or *vector*

```
chute args = angle
  angle = angle in +x away from -z or -y axis in 3d/2d (in degrees)
  angle can be a variable (see below)
spherical args = phi theta
  phi = azimuthal angle from +x axis (in degrees)
  theta = angle from +z or +y axis in 3d/2d (in degrees)
  phi or theta can be a variable (see below)
vector args = x y z
  x y z = vector direction to apply the acceleration
  x or y or z can be a variable (see below)
```

### 2.89.2 Examples

```
fix 1 all gravity 1.0 chute 24.0
fix 1 all gravity v_increase chute 24.0
fix 1 all gravity 1.0 spherical 0.0 -180.0
fix 1 all gravity 10.0 spherical v_phi v_theta
fix 1 all gravity 100.0 vector 1 1 0
```

### 2.89.3 Description

Impose an additional acceleration on each particle in the group. This fix is typically used with granular systems to include a "gravity" term acting on the macroscopic particles. More generally, it can represent any kind of driving field, e.g. a pressure gradient inducing a Poiseuille flow in a fluid. Note that this fix operates differently than the *fix addforce* command. The addforce fix adds the same force to each atom, independent of its mass. This command imparts the same acceleration to each atom (force/mass).

The *magnitude* of the acceleration is specified in force/mass units. For granular systems (LJ units) this is typically 1.0. See the *units* command for details.

Style *chute* is typically used for simulations of chute flow where the specified *angle* is the chute angle, with flow occurring in the +x direction. For 3d systems, the tilt is away from the z axis; for 2d systems, the tilt is away from the y axis.

Style *spherical* allows an arbitrary 3d direction to be specified for the acceleration vector. *Phi* and *theta* are defined in the usual spherical coordinates. Thus for acceleration acting in the -z direction, *theta* would be 180.0 (or -180.0). *Theta* = 90.0 and *phi* = -90.0 would mean acceleration acts in the -y direction. For 2d systems, *phi* is ignored and *theta* is an angle in the xy plane where *theta* = 0.0 is the y-axis.

Style *vector* imposes an acceleration in the vector direction given by (x,y,z). Only the direction of the vector is important; it's length is ignored. For 2d systems, the $z$ component is ignored.

Any of the quantities *magnitude*, *angle*, *phi*, *theta*, $x$, $y$, $z$ which define the gravitational magnitude and direction, can be specified as an equal-style *variable*. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated each timestep, and its value used to determine the quantity. You should ensure that the variable calculates a result in the appropriate units, e.g. force/mass or degrees.

Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent gravitational field.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.89.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify energy* option is supported by this fix to add the gravitational potential energy of the system to the global potential energy of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify energy no*.

The *fix_modify respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is adding its forces. Default is the outermost level.

This fix computes a global scalar which can be accessed by various *output commands*. This scalar is the gravitational potential energy of the particles in the defined field, namely mass * (g dot x) for each particles, where x and mass are the particles position and mass, and g is the gravitational field. The scalar value calculated by this fix is "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.89.5 Restrictions

none

### 2.89.6 Related commands

*atom_style sphere*, *fix addforce*

### 2.89.7 Default

none

## 2.90 fix grem command

### 2.90.1 Syntax

```
fix ID group-ID grem lambda eta H0 thermostat-ID
```

- ID, group-ID are documented in *fix* command
- grem = style name of this fix command
- lambda = intercept parameter of linear effective temperature function
- eta = slope parameter of linear effective temperature function
- H0 = shift parameter of linear effective temperature function
- thermostat-ID = ID of Nose-Hoover thermostat or barostat used in simulation

### 2.90.2 Examples

```
fix             fxgREM all grem 400 -0.01 -30000 fxnpt
thermo_modify   press fxgREM_press

fix             fxgREM all grem 502 -0.15 -80000 fxnvt
```

### 2.90.3 Description

This fix implements the molecular dynamics version of the generalized replica exchange method (gREM) originally developed by *(Kim)*, which uses non-Boltzmann ensembles to sample over first order phase transitions. The is done by defining replicas with an enthalpy dependent effective temperature

$$T_{eff} = \lambda + \eta(H - H_0)$$

with $\eta$ negative and steep enough to only intersect the characteristic microcanonical temperature (Ts) of the system once, ensuring a unimodal enthalpy distribution in that replica. $\lambda$ is the intercept and effects the generalized ensemble similar to how temperature effects a Boltzmann ensemble. $H_0$ is a reference enthalpy, and is typically set as the lowest desired sampled enthalpy. Further explanation can be found in our recent papers *(Malolepsza)*.

This fix requires a Nose-Hoover thermostat fix reference passed to the grem as *thermostat-ID*. Two distinct temperatures exist in this generalized ensemble, the effective temperature defined above, and a kinetic temperature that controls the velocity distribution of particles as usual. Either constant volume or constant pressure algorithms can be used.

The fix enforces a generalized ensemble in a single replica only. Typically, this ideology is combined with replica exchange with replicas differing by $\lambda$ only for simplicity, but this is not required. A multi-replica simulation can be run within the LAMMPS environment using the *temper/grem* command. This utilizes LAMMPS partition mode and requires the number of available processors be on the order of the number of desired replicas. A 100-replica simulation would require at least 100 processors (1 per world at minimum). If many replicas are needed on a small number of processors, multi-replica runs can be run outside of LAMMPS. An example of this can be found in examples/PACKAGES/grem and has no limit on the number of replicas per processor. However, this is very inefficient and error prone and should be avoided if possible.

In general, defining the generalized ensembles is unique for every system. When starting a many-replica simulation without any knowledge of the underlying microcanonical temperature, there are several tricks we have utilized to optimize the process. Choosing a less-steep $\eta$ yields broader distributions, requiring fewer replicas to map the microcanonical temperature. While this likely struggles from the same sampling problems gREM was built to avoid, it provides quick insight to Ts. Initially using an evenly-spaced $\lambda$ distribution identifies regions where small changes in enthalpy lead to large temperature changes. Replicas are easily added where needed.

---

### 2.90.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *thermo_modify* *press* option is supported by this fix to add the rescaled kinetic pressure as part of *thermodynamic output*.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the effective temperature $T_{eff}$. The scalar value calculated by this fix is "intensive".

### 2.90.5 Restrictions

This fix is part of the REPLICA package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.90.6 Related commands

*temper/grem*, *fix nvt*, *fix npt*, *thermo_modify*

### 2.90.7 Default

none

---

**(Kim)** Kim, Keyes, Straub, J Chem. Phys, 132, 224107 (2010).

**(Malolepsza)** Malolepsza, Secor, Keyes, J Phys Chem B 119 (42), 13379-13384 (2015).

## 2.91 fix halt command

### 2.91.1 Syntax

```
fix ID group-ID halt N attribute operator avalue keyword value ...
```

- ID, group-ID are documented in *fix* command
- halt = style name of this fix command
- N = check halt condition every N steps
- attribute = *bondmax* or *tlimit* or v_name

  ```
  bondmax = length of longest bond in the system (in length units)
  tlimit = elapsed CPU time (in seconds)
  diskfree = free disk space (in MBytes)
  v_name = name of equal-style variable
  ```

- operator = "<" or "<=" or ">" or ">=" or "==" or "!=" or "|^"
- avalue = numeric value to compare attribute to
- zero or more keyword/value pairs may be appended
- keyword = *error* or *message* or *path* or *universe*

  ```
  error value = hard or soft or continue
  message value = yes or no
  path value = path to check for free space (may be in quotes)
  universe value = yes or no
  ```

### 2.91.2 Examples

```
fix 10 all halt 1 bondmax > 1.5
fix 10 all halt 10 v_myCheck != 0 error soft message no
fix 10 all halt 100 diskfree < 100000.0 path "dump storage/."
fix  2 all halt 100 v_curtime > ${maxtime} universe yes
```

### 2.91.3 Description

Check a condition every N steps during a simulation run. N must be >=1. If the condition is met, exit the run. In this context a "run" can be dynamics or minimization iterations, as specified by the *run* or *minimize* command.

The specified group-ID is ignored by this fix.

The specified *attribute* can be one of the options listed above, namely *bondmax*, *tlimit*, *diskfree*, or an *equal-style variable* referenced as *v_name*, where "name" is the name of a variable that has been defined previously in the input script.

The *bondmax* attribute will loop over all bonds in the system, compute their current lengths, and set *attribute* to the longest bond distance.

The *tlimit* attribute queries the elapsed CPU time (in seconds) since the current run began, and sets *attribute* to that value. This is an alternative way to limit the length of a simulation run, similar to the *timer* timeout command. There are two differences in using this method versus the timer command option. The first is that the clock starts at the beginning of the current run (not when the timer or fix command is specified), so that any setup time for the run is not included in the elapsed time. The second is that the timer invocation and syncing across all processors (via MPI_Allreduce) is not performed once every *N* steps by this command. Instead it is performed (typically) only a small number of times and the elapsed times are used to predict when the end-of-the-run will be. Both of these attributes can be useful when performing benchmark calculations for a desired length of time with minimal overhead. For example, if a run is performing 1000s of timesteps/sec, the overhead for syncing the timer frequently across a large number of processors may be non-negligible.

The *diskfree* attribute will check for available disk space (in MBytes) on supported operating systems. By default it will check the file system of the current working directory. This can be changed with the optional *path* keyword, which will take the path to a file or folder on the file system to be checked as argument. This path must be given with single or double quotes, if it contains blanks or other special characters (like $).

Equal-style variables evaluate to a numeric value. See the *variable* command for a description. They calculate formulas which can involve mathematical operations, atom properties, group properties, thermodynamic properties, global values calculated by a *compute* or *fix*, or references to other *variables*. Thus they are a very general means of computing some attribute of the current system. For example, the following "bondmax" variable will calculate the same quantity as the hstyle = bondmax option.

```
compute         bdist all bond/local dist
compute         bmax all reduce max c_bdist inputs local
variable        bondmax equal c_bmax
```

Thus these two versions of a fix halt command will do the same thing:

```
fix 10 all halt 1 bondmax > 1.5
fix 10 all halt 1 v_bondmax > 1.5
```

The version with "bondmax" will just run somewhat faster, due to less overhead in computing bond lengths and not storing them in a separate compute.

A variable can be used to implement a large variety of conditions, including to stop when a specific file exists. Example:

```
variable exit equal is_file(EXIT)
fix 10 all halt 100 v_exit != 0 error soft
```

Will stop the current run command when a file `EXIT` is created in the current working directory. The condition can be cleared by removing the file through the *shell* command.

The choice of operators listed above are the usual comparison operators. The XOR operation (exclusive or) is also included as "|^". In this context, XOR means that if either the attribute or avalue is 0.0 and the other is non-zero, then the result is "true". Otherwise it is "false".

The specified *avalue* must be a numeric value.

---

The optional *error* keyword determines how the current run is halted. If its value is *hard*, then LAMMPS will stop with an error message.

If its value is *soft*, LAMMPS will exit the current run, but continue to execute subsequent commands in the input script. However, additional *run* or *minimize* commands will be skipped. For example, this allows a script to output the current state of the system, e.g. via a *write_dump* or *write_restart* command. To re-enable regular runs after *fix halt* stopped a run, you need to issue a *timer timeout unlimited* command.

If its value is *continue*, the behavior is the same as for *soft*, except subsequent *run* or *minimize* commands are executed. This allows your script to remedy the condition that triggered the halt, if necessary. This is the equivalent of stopping with *error soft* and followed by *timer timeout unlimited* command. This can have undesired consequences, when a *run command* uses the *every* keyword, so using *error soft* and resetting the timer manually may be the preferred option.

You may wish use the *unfix* command on the *fix halt* ID before starting a subsequent run, so that the same condition is not immediately triggered again.

The optional *message* keyword determines whether a message is printed to the screen and logfile when the halt condition is triggered. If *message* is set to yes, a one line message with the values that triggered the halt is printed. If *message* is set to no, no message is printed; the run simply exits. The latter may be desirable for post-processing tools that extract thermodynamic information from log files.

Added in version 2Apr2025.

The optional *universe* keyword determines whether the halt request should be synchronized across the partitions of a *multi-partition run*. If *universe* is set to yes, fix halt will check if there is a specific message received from any of the other partitions requesting to stop the run on this partition as well. Consequently, if fix halt determines to halt the simulation, the fix will send messages to all other partitions so they stop their runs, too.

---

**Checking for 'inf' or 'nan'**

It is also possible to have fix halt stop when some properties become 'inf' or 'nan', provided this property is represented by a *thermo keyword*. Below is a minimal example with three loop iterations where the second iteration is stopped because its potential energy becomes infinity ('inf'). It will also work for 'nan', i.e. when replacing "pe" with "press". Both will be larger than the largest representable floating point number.

```
label newrun
variable i loop 3

clear
region box block 0 1 0 1 0 1
create_box 1 box

create_atoms 1 single 0.5 0.5 0.5
create_atoms 1 single 0.5 $(0.3+0.1*v_i) 0.5

mass 1 1.0
pair_style lj/cut 0.9
pair_coeff 1 1 1.0 1.0

variable pe equal pe
fix 1 all halt 10 v_pe > 1.7e308 error soft
thermo 10
run 30 post no

# fix halt soft stops a run by forcing a timeout
if $(is_timeout()) then "print 'run was terminated due to invalid energy'"
# reset timeout state
timer timeout off

next i
jump SELF newrun
```

### 2.91.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

### 2.91.5 Restrictions

The *diskfree* attribute is currently only supported on Linux, macOS, and *BSD.

### 2.91.6 Related commands

*variable*

### 2.91.7 Default

The option defaults are error = soft, message = yes, path = ".", and universe = no.

## 2.92 fix heat command

### 2.92.1 Syntax

```
fix ID group-ID heat N eflux
```

- ID, group-ID are documented in *fix* command
- heat = style name of this fix command
- N = add/subtract heat every this many timesteps
- eflux = rate of heat addition or subtraction (energy/time units)
- eflux can be a variable (see below)
- zero or more keyword/value pairs may be appended to args
- keyword = *region*

```
region value = region-ID
  region-ID = ID of region atoms must be in to have added force
```

### 2.92.2 Examples

```
fix 3 qin heat 1 1.0
fix 3 qin heat 10 v_flux
fix 4 qout heat 1 -1.0 region top
```

### 2.92.3 Description

Add non-translational kinetic energy (heat) to a group of atoms in a manner that conserves their aggregate momentum. Two of these fixes can be used to establish a temperature gradient across a simulation domain by adding heat (energy) to one group of atoms (hot reservoir) and subtracting heat from another (cold reservoir). E.g. a simulation sampling from the McDLT ensemble.

If the *region* keyword is used, the atom must be in both the group and the specified geometric *region* in order to have energy added or subtracted to it. If not specified, then the atoms in the group are affected wherever they may move to.

Heat addition/subtraction is performed every N timesteps.

The *eflux* parameter can be specified as a numeric constant or as an equal- or atom-style *variable*. If the value is a variable, it should be specified as v_name, where *name* is the variable name. In this case, the variable will be evaluated each timestep, and its current value(s) used to determine the flux.

If *eflux* is a numeric constant or equal-style variable which evaluates to a scalar value, then *eflux* determines the change in aggregate energy of the entire group of atoms per unit time, e.g. in eV/ps for *metal units*. In this case it is an "extensive" quantity, meaning its magnitude should be scaled with the number of atoms in the group. Note that since *eflux* also has per-time units (i.e. it is a flux), this means that a larger value of N will add/subtract a larger amount of energy each time the fix is invoked.

**Note**

The heat-exchange (HEX) algorithm implemented by this fix is known to exhibit a pronounced energy drift. An improved algorithm (eHEX) is available as a *fix ehex* command and might be preferable if energy conservation is important.

If *eflux* is specified as an atom-style variable (see below), then the variable computes one value per atom. In this case, each value is the energy flux for a single atom, again in units of energy per unit time. In this case, each value is an "intensive" quantity, which need not be scaled with the number of atoms in the group.

Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent flux.

Atom-style variables can specify the same formulas as equal-style variables but can also include per-atom values, such as atom coordinates. Thus it is easy to specify a spatially-dependent flux with optional time-dependence as well.

**Note**

If heat is subtracted from the system too aggressively so that the group's kinetic energy would go to zero, or any individual atom's kinetic energy would go to zero for the case where *eflux* is an atom-style variable, then LAMMPS will halt with an error message.

Fix heat is different from a thermostat such as *fix nvt* or *fix temp/rescale* in that energy is added/subtracted continually. Thus if there is not another mechanism in place to counterbalance this effect, the entire system will heat or cool continuously. You can use multiple heat fixes so that the net energy change is 0.0 or use *fix viscous* to drain energy from the system.

This fix does not change the coordinates of its atoms; it only scales their velocities. Thus you must still use an integration fix (e.g. *fix nve*) on the affected atoms. This fix should not normally be used on atoms that have their temperature controlled by another fix - e.g. *fix nvt* or *fix langevin* fix.

### 2.92.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix computes a global scalar which can be accessed by various *output commands*. This scalar is the most recent value by which velocities were scaled. The scalar value calculated by this fix is "intensive". If *eflux* is specified as an atom-style variable, this fix computes the average value by which the velocities were scaled for all of the atoms that had their velocities scaled.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.92.5 Restrictions

none

### 2.92.6 Related commands

*fix ehex*, *compute temp*, *compute temp/region*

### 2.92.7 Default

none

## 2.93 fix heat/flow command

### 2.93.1 Syntax

```
fix ID group-ID heat/flow style values ...
```

- ID, group-ID are documented in *fix* command
- heat/flow = style name of this fix command
- one style with corresponding value(s) needs to be listed

```
style = constant or type
  constant = cp
    cp = value of specifc heat (energy/(mass * temperature) units)
  type = cp1 ... cpN
    cpN = value of specifc heat for type N (energy/(mass * temperature) units)
```

### 2.93.2 Examples

```
fix 1 all heat/flow constant 1.0
fix 1 all heat/flow type 1.0 0.5
```

### 2.93.3 Description

Perform plain time integration to update temperature for atoms in the group each timestep. The specific heat of atoms can be defined using either the *constant* or *type* keywords. For style *constant*, the specific heat is a constant value *cp* for all atoms. For style *type*, *N* different values of the specific heat are defined, one for each of the *N* types of atoms.

---

### 2.93.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.93.5 Restrictions

This pair style is part of the GRANULAR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix requires that atoms store temperature and heat flow as defined by the *fix property/atom* command.

### 2.93.6 Related commands

*pair granular*, *fix add/heat*, *fix property/atom*

### 2.93.7 Default

none

## 2.94 fix hmc command

### 2.94.1 Syntax

```
fix ID group-ID hmc N seed T keyword values ...
```

- ID, group-ID are documented in *fix* command
- hmc = style name of this fix command
- N = invoke a Monte Carlo step every N steps
- seed = random # seed (positive integer)
- T = temperature for assigning velocities and acceptance criterion
- one or more keyword/value pairs may be appended

```
keyword = rigid or resample or mom
  rigid value = rigidID
     rigidID = ID of fix rigid/small command
  resample value = yes or no
  mom value = yes or no
```

### 2.94.2 Examples

```
fix 1 all hmc 10 123 500
fix hmc_water all hmc 100 123 298.15 rigid 1
fix 2 all hmc 10 12345 300 mom no resample yes
```

### 2.94.3 Description

Added in version 22Jul2025.

This fix implements the hybrid or Hamiltonian Monte Carlo (HMC) algorithm. The basic idea is to use molecular dynamics (MD) to generate trial MC "moves" which are then accepted or rejected via the Metropolis criterion. In this context, an MC "move" is the new configuration of particles after $N$ MD steps, i.e. all the particles in the system have moved to new positions. HMC generates a canonical distribution in configuration space. More details on the theory behind HMC can be found in the references, *(Mehlig)* and *(Mehlig)*.

The details of the HMC algorithm for a repeating series of $N$ MD steps are as follows:

(1) The configuration of the system is stored along with its current total energy. This includes all particle positions and velocities and other per-atom properties (e.g. dipole orientation vector for particles with dipole moments).

(2) The system is time integrated in the NVE ensemble for the specified $N$ MD steps and the new energy is calculated. The new configuration is the trial "move" to accept or reject.

(3) The energy change $\Delta H$ in the Hamiltonian of the system due to the "move" is calculated by the following equation:

$$\Delta H = H(q', p') - H(q, p)$$

The new configuration is then accepted/rejected according to the Metropolis criterion with probability:

$$p^{acc} = min(1, e^{\frac{-\Delta H}{k_B T}})$$

where $T$ is the specified temperature.

The idea of HMC is to use a timestep large enough that total energy is *not* conserved. The change in total energy (the Hamiltonian) is what the Metropolis criterion is based on, not the change in potential energy.

(4) If accepted, the new configuration becomes the starting point for the next trial MC "move". If *resample* is *yes* then the velocities are resampled at this point as well.

(5) If rejected, the old configuration (from $N$ steps ago) is restored and new momenta (velocities) are assigned to each particle in the fix group by randomly resampling from a normal distribution at the specified temperature $T$ using the following equation:

$$p_{x,y,z} = \mathbf{N}(0, 1)\sqrt{\frac{k_B T}{2m^2}}$$

The velocity-modified "old" configuration becomes the starting point for the next trial MC "move".

**Note**

HMC should be run with a larger timestep than would be used for traditional MD, which enables total energy fluctuations and generation of new conformations which MD would not normally generate as quickly. The timestep size may also affect the acceptance ratio as a larger timestep will lead to larger and more extreme MC moves which are less likely to be accepted. The timestep size must strike a balance between allowing the total energy to change and generating errors such as lost atoms due to atomic overlap. This means that during the MD portion of the algorithm, unphysical dynamics will take place, such as large temperature fluctuations and large forces between atoms. This is expected and is part of the HMC algorithm, as the MD step is not intended to produce a physically realistic trajectory, but rather to generate a new configuration of particles that can be accepted or rejected based on the Metropolis criterion.

**Note**

High acceptance ratios indicate that the MC algorithm is inefficient, as it is not generating new configurations of particles any faster than MD would on its own. In the limit of an acceptance ratio of 1.0, the algorithm is equivalent to MD (with momentum resampling every $N$ timesteps if *resample* = *yes*), and no benefit is gained from MC. A good rule of thumb is to aim for an acceptance ratio of 0.5 to 0.8, which can be monitored via the output of this fix. This can be achieved by adjusting the $N$ parameter and the timestep size. Increasing either of these values will increase the size of the total energy fluctuations, which can decrease acceptance ratio. Increasing $N$ will also increase the computation time for each MC step, as more MD steps are performed before each acceptance/rejection decision. As noted above, increasing the timestep too much can lead to LAMMPS errors due to lost atoms or bonds, so both of these parameters should be chosen carefully.

**Note**

This fix is designed to be used only for constant NVE simulations. No thermostat or barostat should be used, though LAMMPS does not check for this. A *fix nve* command must be defined to perform time integration for the MD portion of the algorithm. See the explanation of the *rigid* keyword below for an exception when rigid bodies are defined. Also note that only per-atom data is restored on MC move rejection, so anything which adds or remove particles, changes the box size, or has some external state not dependent on per-atom data will have undefined behavior.

---

The keyword/value options are as follows:

The *rigid* keyword enables use of HMC for systems containing a collection of small rigid bodies, with or without solvent (atomic fluid or non-rigid molecular fluid).

The *rigidID* value should be the ID of a *fix rigid/small* or *fix rigid/nve/small* command which defines the rigid bodies. Its integrator will be used during the MD timesteps. If there are additional particles in the system, e.g. solvent, they should be time-integrated by a *fix nve* command as explained above.

The *resample* keyword determines whether velocities are also resampled upon acceptance in step (4) above, in addition to step (5). If *resample* = *yes*, velocities are resampled upon acceptance. If *resample* = *no* (default), velocities are not resampled upon acceptance.

The *mom* keyword sets the linear momentum of the ensemble of particles each time velocities are reset in steps (4 or 5) above. If *mom* = *yes* (default), the linear momentum of the ensemble of velocities is zeroed. If *mom* = *no*, the linear momentum of the ensemble of velocities is not zeroed.

---

This fix creates several additional computes for monitoring the energy and virial of the system and storing/restoring the system state. This is done internally, as if these commands had been issued, where ID is the ID of this fix:

```
compute hmc_ke_ID all ke
compute hmc_pe_ID all pe
compute hmc_peatom_ID all pe/atom
compute hmc_press_ID all pressure NULL virial
compute hmc_pressatom_ID all stress/atom NULL virial
```

The output of these computes can be accessed by the input script, along with the other outputs described in the next section.

---

### 2.94.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix calculates a global scalar and global vector of length 5, which can be accessed by various *output commands*. The scalar is the fraction (0-1) of attempted MC moves which have been accepted. The vector stores the following quantities:

- 1 = cumulative number of accepted moves
- 2 = cumulative number of rejected moves
- 3 = change in potential energy for last trial move
- 4 = change in kinetic energy for last trial move
- 5 = change in total energy (kinetic + potential energy) for last trial move

These values are updated once every *N* timesteps. The scalar and cumulative counts are "intensive"; the three energies are "extensive" and are in energy *units*.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.94.5 Restrictions

This fix is part of the MC package and requires the RIGID package to be installed. It is only enabled if LAMMPS was built with both packages. See the *Build package* doc page for more info.

### 2.94.6 Related commands

*fix nvt*, *fix gcmc*, *fix tfmc*

### 2.94.7 Default

The option defaults are resample = no and mom = yes.

---

**(Mehlig1)** Mehlig, B., Heermann, D. W., & Forrest, B. M. (1992). Hybrid Monte Carlo method for condensed-matter systems. Physical Review B, 45(2), 679.

**(Mehlig2)** Mehlig, B., Heermann, D. W., & Forrest, B. M. (1992). Exact langevin algorithms. Molecular Physics, 76(6), 1347-1357.

## 2.95 fix hyper/global command

### 2.95.1 Syntax

```
fix ID group-ID hyper/global cutbond qfactor Vmax Tequil
```

- ID, group-ID are documented in *fix* command
- hyper/global = style name of this fix command
- cutbond = max distance at which a pair of atoms is considered bonded (distance units)
- qfactor = max strain at which bias potential goes to 0.0 (unitless)
- Vmax = height of bias potential (energy units)
- Tequil = equilibration temperature (temperature units)

### 2.95.2 Examples

```
fix 1 all hyper/global 1.0 0.3 0.8 300.0
```

### 2.95.3 Description

This fix is meant to be used with the *hyper* command to perform a bond-boost global hyperdynamics (GHD) simulation. The role of this fix is to a select a single pair of atoms in the system at each timestep to add a global bias potential to, which will alter the dynamics of the system in a manner that effectively accelerates time. This is in contrast to the *fix hyper/local* command, which can be user to perform a local hyperdynamics (LHD) simulation, by adding a local bias potential to multiple pairs of atoms at each timestep. GHD can time accelerate a small simulation with up to a few 100 atoms. For larger systems, LHD is needed to achieve good time acceleration.

For a system that undergoes rare transition events, where one or more atoms move over an energy barrier to a new potential energy basin, the effect of the bias potential is to induce more rapid transitions. This can lead to a dramatic speed-up in the rate at which events occurs, without altering their relative frequencies, thus leading to an overall increase in the elapsed real time of the simulation as compared to running for the same number of timesteps with normal MD. See the *hyper* page for a more general discussion of hyperdynamics and citations that explain both GHD and LHD.

The equations and logic used by this fix and described here to perform GHD follow the description given in *(Voter2013)*. The bond-boost form of a bias potential for HD is due to Miron and Fichthorn as described in *(Miron)*. In LAMMPS we use a simplified version of bond-boost GHD where a single bond in the system is biased at any one timestep.

Bonds are defined between each pair of atoms *ij*, whose $R_{ij}^0$ distance is less than *cutbond*, when the system is in a quenched state (minimum) energy. Note that these are not "bonds" in a covalent sense. A bond is simply any pair of atoms that meet the distance criterion. *Cutbond* is an argument to this fix; it is discussed below. A bond is only formed if one or both of the *ij* atoms are in the specified group.

The current strain of bond *ij* (when running dynamics) is defined as

$$E_{ij} = \frac{R_{ij} - R_{ij}^0}{R_{ij}^0}$$

where $R_{ij}$ is the current distance between atoms $i$ and $j$, and $R_{ij}^0$ is the equilibrium distance in the quenched state.

The bias energy $V_{ij}$ of any bond between atoms $i$ and $j$ is defined as

$$V_{ij} = V^{max} \cdot \left(1 - \left(\frac{E_{ij}}{q}\right)^2\right) \text{ for } |E_{ij}| < qfactor \text{ or } 0 \text{ otherwise}$$

where the prefactor $V^{max}$ and the cutoff *qfactor* are arguments to this fix; they are discussed below. This functional form is an inverse parabola centered at 0.0 with height $V^{max}$ and which goes to 0.0 at +/- qfactor.

Let $E^{max}$ be the maximum of $|E_{ij}|$ for all *ij* bonds in the system on a given timestep. On that step, $V_{ij}$ is added as a bias potential to only the single bond with strain $E^{max}$, call it $V_{ij}^{max}$. Note that $V_{ij}^{max}$ will be 0.0 if $E^{max}$ >= qfactor on that timestep. Also note that $V_{ij}^{max}$ is added to the normal interatomic potential that is computed between all atoms in the system at every step.

The derivative of $V_{ij}^{max}$ with respect to the position of each atom in the $E^{max}$ bond gives a bias force $F_{ij}^{max}$ acting on the bond as

$$F_{ij}^{max} = -\frac{dV_{ij}^{max}}{dE_{ij}} = \frac{2V^{max}E - ij}{\text{qfactor}^2} \text{ for } |E_{ij}| < \text{qfactor or } 0 \text{ otherwise}$$

which can be decomposed into an equal and opposite force acting on only the two *ij* atoms in the $E^{max}$ bond.

The time boost factor for the system is given each timestep I by

$$B_i = e^{\beta V_{ij}^{max}}$$

where $\beta = \frac{1}{kT_{equil}}$, and $T_{equil}$ is the temperature of the system and an argument to this fix. Note that $B_i$ >= 1 at every step.

**Note**

To run a GHD simulation, the input script must also use the *fix langevin* command to thermostat the atoms at the same $T_{equil}$ as specified by this fix, so that the system is running constant-temperature (NVT) dynamics. LAMMPS does not check that this is done.

The elapsed time $t_{hyper}$ for a GHD simulation running for $N$ timesteps is simply

$$t_{hyper} = \sum_{i=1,N} B - i \cdot dt$$

where *dt* is the timestep size defined by the *timestep* command. The effective time acceleration due to GHD is thus $t_{hyper}/N * dt$, where N*dt is elapsed time for a normal MD run of N timesteps.

Note that in GHD, the boost factor varies from timestep to timestep. Likewise, which bond has $E^{max}$ strain and thus which pair of atoms the bias potential is added to, will also vary from timestep to timestep. This is in contrast to local hyperdynamics (LHD) where the boost factor is an input parameter; see the *fix hyper/local* page for details.

---

Here is additional information on the input parameters for GHD.

The *cutbond* argument is the cutoff distance for defining bonds between pairs of nearby atoms. A pair of *ij* atoms in their equilibrium, minimum-energy configuration, which are separated by a distance $R_{ij} < cutbond$, are flagged as a bonded pair. Setting *cubond* to be ~25% larger than the nearest-neighbor distance in a crystalline lattice is a typical choice for solids, so that bonds exist only between nearest neighbor pairs.

The *qfactor* argument is the limiting strain at which the bias potential goes to 0.0. It is dimensionless, so a value of 0.3 means a bond distance can be up to 30% larger or 30% smaller than the equilibrium (quenched) $R_{ij}^0$ distance and the two atoms in the bond could still experience a non-zero bias force.

If *qfactor* is set too large, then transitions from one energy basin to another are affected because the bias potential is non-zero at the transition state (e.g. saddle point). If *qfactor* is set too small than little boost is achieved because the $E_{ij}$ strain of some bond in the system will (nearly) always exceed *qfactor*. A value of 0.3 for *qfactor* is typically reasonable.

The *Vmax* argument is the prefactor on the bias potential. Ideally, tt should be set to a value slightly less than the smallest barrier height for an event to occur. Otherwise the applied bias potential may be large enough (when added to the interatomic potential) to produce a local energy basin with a maxima in the center. This can produce artificial energy minima in the same basin that trap an atom. Or if *Vmax* is even larger, it may induce an atom(s) to rapidly transition to another energy basin. Both cases are "bad dynamics" which violate the assumptions of GHD that guarantee an accelerated time-accurate trajectory of the system.

Note that if *Vmax* is set too small, the GHD simulation will run correctly. There will just be fewer events because the hyper time ($t_{hyper}$ equation above) will be shorter.

**Note**

If you have no physical intuition as to the smallest barrier height in your system, a reasonable strategy to determine the largest *Vmax* you can use for a GHD model, is to run a sequence of simulations with smaller and smaller *Vmax* values, until the event rate does not change (as a function of hyper time).

The *Tequil* argument is the temperature at which the system is simulated; see the comment above about the *fix langevin* thermostatting. It is also part of the beta term in the exponential factor that determines how much boost is achieved as a function of the bias potential.

In general, the lower the value of *Tequil* and the higher the value of *Vmax*, the more time boost will be achievable by the GHD algorithm.

---

### 2.95.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify energy* option is supported by this fix to add the energy of the bias potential to the global potential energy of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify energy no*.

This fix computes a global scalar and global vector of length 12, which can be accessed by various *output commands*. The scalar is the magnitude of the bias potential (energy units) applied on the current timestep. The vector stores the following quantities:

1. boost factor on this step (unitless)
2. max strain $E_{ij}$ of any bond on this step (absolute value, unitless)
3. ID of first atom in the max-strain bond
4. ID of second atom in the max-strain bond
5. average # of bonds/atom on this step
6. fraction of timesteps where the biased bond has bias = 0.0 during this run
7. fraction of timesteps where the biased bond has negative strain during this run
8. max drift distance of any atom during this run (distance units)
9. max bond length during this run (distance units)
10. cumulative hyper time since fix was defined (time units)
11. cumulative count of event timesteps since fix was defined
12. cumulative count of atoms in events since fix was defined

The first 5 quantities are for the current timestep. Quantities 6-9 are for the current hyper run. They are reset each time a new hyper run is performed. Quantities 19-12 are cumulative across multiple runs (since the point in the input script the fix was defined).

For value 8, drift is the distance an atom moves between two quenched states when the second quench determines an event has occurred. Atoms involved in an event will typically move the greatest distance since others typically remain near their original quenched position.

For value 11, events are checked for by the *hyper* command once every *Nevent* timesteps. This value is the count of those timesteps on which one (or more) events was detected. It is NOT the number of distinct events, since more than one event may occur in the same *Nevent* time window.

For value 12, each time the *hyper* command checks for an event, it invokes a compute to flag zero or more atoms as participating in one or more events. E.g. atoms that have displaced more than some distance from the previous quench state. Value 11 is the cumulative count of the number of atoms participating in any of the events that were found.

The scalar and vector values calculated by this fix are all "intensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.95.5 Restrictions

This command can only be used if LAMMPS was built with the REPLICA package. See the *Build package* page for more info.

### 2.95.6 Related commands

*hyper*, *fix hyper/local*

### 2.95.7 Default

none

---

**(Voter2013)** S. Y. Kim, D. Perez, A. F. Voter, J Chem Phys, 139, 144110 (2013).

**(Miron)** R. A. Miron and K. A. Fichthorn, J Chem Phys, 119, 6210 (2003).

# 2.96 fix hyper/local command

## 2.96.1 Syntax

```
fix ID group-ID hyper/local cutbond qfactor Vmax Tequil Dcut alpha Btarget
```

- ID, group-ID are documented in *fix* command
- hyper/local = style name of this fix command
- cutbond = max distance at which a pair of atoms is considered bonded (distance units)
- qfactor = max strain at which bias potential goes to 0.0 (unitless)
- Vmax = estimated height of bias potential (energy units)
- Tequil = equilibration temperature (temperature units)
- Dcut = minimum distance between boosted bonds (distance units)
- alpha = boostostat relaxation time (time units)
- Btarget = desired time boost factor (unitless)
- zero or more keyword/value pairs may be appended
- keyword = *bound* or *reset* or *check/ghost* or *check/bias*

```
bound value = Bfrac
  Bfrac =  -1 or a value >= 0.0
reset value = Rfreq
  Rfreq = -1 or 0 or timestep value > 0
check/ghost values = none
check/bias values = Nevery error/warn/ignore
```

## 2.96.2 Examples

```
fix 1 all hyper/local 1.0 0.3 0.8 300.0
fix 1 all hyper/local 1.0 0.3 0.8 300.0 bound 0.1 reset 0
```

## 2.96.3 Description

This fix is meant to be used with the *hyper* command to perform a bond-boost local hyperdynamics (LHD) simulation. The role of this fix is to a select multiple pairs of atoms in the system at each timestep to add a local bias potential to, which will alter the dynamics of the system in a manner that effectively accelerates time. This is in contrast to the *fix hyper/global* command, which can be user to perform a global hyperdynamics (GHD) simulation, by adding a global bias potential to a single pair of atoms at each timestep. GHD can time accelerate a small simulation with up to a few 100 atoms. For larger systems, LHD is needed to achieve good time acceleration.

For a system that undergoes rare transition events, where one or more atoms move over an energy barrier to a new potential energy basin, the effect of the bias potential is to induce more rapid transitions. This can lead to a dramatic speed-up in the rate at which events occurs, without altering their relative frequencies, thus leading to an overall increase in the elapsed real time of the simulation as compared to running for the same number of timesteps with normal MD. See the *hyper* page for a more general discussion of hyperdynamics and citations that explain both GHD and LHD.

The equations and logic used by this fix and described here to perform LHD follow the description given in *(Voter2013)*. The bond-boost form of a bias potential for HD is due to Miron and Fichthorn as described in *(Miron)*.

To understand this description, you should first read the description of the GHD algorithm on the *fix hyper/global* doc page. This description of LHD builds on the GHD description.

The definition of bonds and $E_{ij}$ are the same for GHD and LHD. The formulas for $V_{ij}^{max}$ and $F_{ij}^{max}$ are also the same except for a prefactor $C_{ij}$, explained below.

The bias energy $V_{ij}$ applied to a bond $ij$ with maximum strain is

$$V_{ij}^{max} = C_{ij} \cdot V^{max} \cdot \left(1 - \left(\frac{E_{ij}}{q}\right)^2\right) \text{ for } |E_{ij}| < qfactor \text{ or 0 otherwise}$$

The derivative of $V_{ij}^{max}$ with respect to the position of each atom in the $ij$ bond gives a bias force $F_{ij}^{max}$ acting on the bond as

$$F_{ij}^{max} = -\frac{dV_{ij}^{max}}{dE_{ij}} = 2C_{ij}V^{max}\frac{E_{ij}}{qfactor^2} \text{ for } |E_{ij}| < qfactor \text{ or 0 otherwise}$$

which can be decomposed into an equal and opposite force acting on only the two atoms $i$ and $j$ in the $ij$ bond.

The key difference is that in GHD a bias energy and force is added (on a particular timestep) to only one bond (pair of atoms) in the system, which is the bond with maximum strain $E^{max}$.

In LHD, a bias energy and force can be added to multiple bonds separated by the specified *Dcut* distance or more. A bond $ij$ is biased if it is the maximum strain bond within its local "neighborhood", which is defined as the bond $ij$ plus any neighbor bonds within a distance *Dcut* from $ij$. The "distance" between bond $ij$ and bond $kl$ is the minimum distance between any of the $ik$, $il$, $jk$, and $jl$ pairs of atoms.

For a large system, multiple bonds will typically meet this requirement, and thus a bias potential $V_{ij}^{max}$ will be applied to many bonds on the same timestep.

In LHD, all bonds store a $C_{ij}$ prefactor which appears in the $V_{ij}^{max}$ and $F_{ij}^{max}$ equations above. Note that the $C_{ij}$ factor scales the strength of the bias energy and forces whenever bond $ij$ is the maximum strain bond in its neighborhood.

$C_{ij}$ is initialized to 1.0 when a bond between the $ij$ atoms is first defined. The specified *Btarget* factor is then used to adjust the $C_{ij}$ prefactors for each bond every timestep in the following manner.

An instantaneous boost factor $B_{ij}$ is computed each timestep for each bond, as

$$B_{ij} = e^{\beta V_{kl}^{max}}$$

where $V_{kl}^{max}$ is the bias energy of the maxstrain bond $kl$ within bond $ij$'s neighborhood, $\beta = \frac{1}{kT_{equil}}$, and $T_{equil}$ is the temperature of the system and an argument to this fix.

**Note**

To run an LHD simulation, the input script must also use the *fix langevin* command to thermostat the atoms at the same $T_{equil}$ as specified by this fix, so that the system is running constant-temperature (NVT) dynamics. LAMMPS does not check that this is done.

Note that if $ij$== $kl$, then bond $ij$ is a biased bond on that timestep, otherwise it is not. But regardless, the boost factor $B_{ij}$ can be thought of an estimate of time boost currently being applied within a local region centered on bond $ij$. For LHD, we want this to be the specified *Btarget* value everywhere in the simulation domain.

To accomplish this, if $B_{ij} < B_{target}$, the $C_{ij}$ prefactor for bond $ij$ is incremented on the current timestep by an amount proportional to the inverse of the specified $\alpha$ and the difference $(B_{ij} - B_{target})$. Conversely if $B_{ij} > B_{target}$, $C_{ij}$ is decremented by the same amount. This procedure is termed "boostostatting" in *(Voter2013)*. It drives all of the individual $C_{ij}$ to values such that when $V_{ij}^{max}$ is applied as a bias to bond $ij$, the resulting boost factor $B_{ij}$ will be close to $B_{target}$ on average. Thus the LHD time acceleration factor for the overall system is effectively *Btarget*.

Note that in LHD, the boost factor $B_{target}$ is specified by the user. This is in contrast to global hyperdynamics (GHD) where the boost factor varies each timestep and is computed as a function of $V_{max}$, $E_{max}$, and $T_{equil}$; see the *fix hyper/global* page for details.

Here is additional information on the input parameters for LHD.

Note that the *cutbond*, *qfactor*, and *Tequil* arguments have the same meaning as for GHD. The *Vmax* argument is slightly different. The *Dcut*, *alpha*, and *Btarget* parameters are unique to LHD.

The *cutbond* argument is the cutoff distance for defining bonds between pairs of nearby atoms. A pair of I,J atoms in their equilibrium, minimum-energy configuration, which are separated by a distance $R_{ij} < cutbond$, are flagged as a bonded pair. Setting *cubond* to be ~25% larger than the nearest-neighbor distance in a crystalline lattice is a typical choice for solids, so that bonds exist only between nearest neighbor pairs.

The *qfactor* argument is the limiting strain at which the bias potential goes to 0.0. It is dimensionless, so a value of 0.3 means a bond distance can be up to 30% larger or 30% smaller than the equilibrium (quenched) $R_{ij}^0$ distance and the two atoms in the bond could still experience a non-zero bias force.

If *qfactor* is set too large, then transitions from one energy basin to another are affected because the bias potential is non-zero at the transition state (e.g. saddle point). If *qfactor* is set too small than little boost can be achieved because the $E_{ij}$ strain of some bond in the system will (nearly) always exceed *qfactor*. A value of 0.3 for *qfactor* is typically a reasonable value.

The *Vmax* argument is a fixed prefactor on the bias potential. There is a also a dynamic prefactor $C_{ij}$, driven by the choice of *Btarget* as discussed above. The product of these should be a value less than the smallest barrier height for an event to occur. Otherwise the applied bias potential may be large enough (when added to the interatomic potential) to produce a local energy basin with a maxima in the center. This can produce artificial energy minima in the same basin that trap an atom. Or if $C_{ij} \cdot V^{max}$ is even larger, it may induce an atom(s) to rapidly transition to another energy basin. Both cases are "bad dynamics" which violate the assumptions of LHD that guarantee an accelerated time-accurate trajectory of the system.

**Note**

It may seem that $V^{max}$ can be set to any value, and $C_{ij}$ will compensate to reduce the overall prefactor if necessary. However the $C_{ij}$ are initialized to 1.0 and the boostostatting procedure typically operates slowly enough that there can be a time period of bad dynamics if $V^{max}$ is set too large. A better strategy is to set $V^{max}$ to the slightly smaller than the lowest barrier height for an event (the same as for GHD), so that the $C_{ij}$ remain near unity.

The *Tequil* argument is the temperature at which the system is simulated; see the comment above about the *fix langevin* thermostatting. It is also part of the beta term in the exponential factor that determines how much boost is achieved as a function of the bias potential. See the discussion of the *Btarget* argument below.

As discussed above, the *Dcut* argument is the distance required between two locally maxstrain bonds for them to both be selected as biased bonds on the same timestep. Computationally, the larger *Dcut* is, the more work (computation and communication) must be done each timestep within the LHD algorithm. And the fewer bonds can be simultaneously biased, which may mean the specified *Btarget* time acceleration cannot be achieved.

Physically *Dcut* should be a long enough distance that biasing two pairs of atoms that close together will not influence the dynamics of each pair. E.g. something like 2x the cutoff of the interatomic potential. In practice a *Dcut* value of ~10 Angstroms seems to work well for many solid-state systems.

**Note**

You should ensure that ghost atom communication is performed for a distance of at least *Dcut* + *cutevent* = the distance one or more atoms move (between quenched states) to be considered an "event". It is an argument to the "compute event/displace" command used to detect events. By default the ghost communication distance is set by

the pair_style cutoff, which will typically be < *Dcut*. The *comm_modify cutoff* command should be used to override the ghost cutoff explicitly, e.g.

```
comm_modify cutoff 12.0
```

Note that this fix does not know the *cutevent* parameter, but uses half the *cutbond* parameter as an estimate to warn if the ghost cutoff is not long enough.

As described above the *alpha* argument is a prefactor in the boostostat update equation for each bond's $C_{ij}$ prefactor. *Alpha* is specified in time units, similar to other thermostat or barostat damping parameters. It is roughly the physical time it will take the boostostat to adjust a $C_{ij}$ value from a too high (or too low) value to a correct one. An *alpha* setting of a few ps is typically good for solid-state systems. Note that the *alpha* argument here is the inverse of the alpha parameter discussed in *(Voter2013)*.

The *Btarget* argument is the desired time boost factor (a value > 1) that all the atoms in the system will experience. The elapsed time $t_{hyper}$ for an LHD simulation running for $N$ timesteps is simply

$$t_{hyper} = B_{target} \cdot N \cdot dt$$

where $dt$ is the timestep size defined by the *timestep* command. The effective time acceleration due to LHD is thus $\frac{t_{hyper}}{N \cdot dt} = B_{target}$, where $N \cdot dt$ is the elapsed time for a normal MD run of N timesteps.

You cannot choose an arbitrarily large setting for *Btarget*. The maximum value you should choose is

$$B_{target} = e^{\beta V_{small}}$$

where $V_{small}$ is the smallest event barrier height in your system, $\beta = \frac{1}{kT_{equil}}$, and $T_{equil}$ is the specified temperature of the system (both by this fix and the Langevin thermostat).

Note that if *Btarget* is set smaller than this, the LHD simulation will run correctly. There will just be fewer events because the hyper time ($t_{hyper}$ equation above) will be shorter.

**Note**

If you have no physical intuition as to the smallest barrier height in your system, a reasonable strategy to determine the largest *Btarget* you can use for an LHD model, is to run a sequence of simulations with smaller and smaller *Btarget* values, until the event rate does not change (as a function of hyper time).

---

Here is additional information on the optional keywords for this fix.

The *bound* keyword turns on min/max bounds for bias coefficients $C_{ij}$ for all bonds. $C_{ij}$ is a prefactor for each bond on the bias potential of maximum strength $V^{max}$. Depending on the choice of *alpha* and *Btarget* and *Vmax*, the boostostatting can cause individual $C_{ij}$ values to fluctuate. If the fluctuations are too large $C_{ij} \cdot V^{max}$ can exceed low barrier heights and induce bad event dynamics. Bounding the $C_{ij}$ values is a way to prevent this. If *Bfrac* is set to -1 or any negative value (the default) then no bounds are enforced on $C_{ij}$ values (except they must always be >= 0.0). A *Bfrac* setting >= 0.0 sets a lower bound of 1.0 - Bfrac and upper bound of 1.0 + Bfrac on each $C_{ij}$ value. Note that all $C_{ij}$ values are initialized to 1.0 when a bond is created for the first time. Thus *Bfrac* limits the bias potential height to *Vmax* +/- *Bfrac*Vmax*.

The *reset* keyword allow *Vmax* to be adjusted dynamically depending on the average value of all $C_{ij}$ prefactors. This can be useful if you are unsure what value of *Vmax* will match the *Btarget* boost for the system. The $C_{ij}$ values will then adjust in aggregate (up or down) so that $C_{ij} \cdot V^{max}$ produces a boost of *Btarget*, but this may conflict with the *bound* keyword settings. By using *bound* and *reset* together, $V^{max}$ itself can be reset, and desired bounds still applied to the $C_{ij}$ values.

A setting for *Rfreq* of -1 (the default) means *Vmax* never changes. A setting of 0 means $V^{max}$ is adjusted every time an event occurs and bond pairs are recalculated. A setting of N > 0 timesteps means $V^{max}$ is adjusted on the first time an event occurs on a timestep >= N steps after the previous adjustment. The adjustment to $V^{max}$ is computed as follows. The current average of all $C_{ij} \cdot V^{max}$ values is computed and the $V^{max}$ is reset to that value. All $C_{ij}$ values are changed to new prefactors such the new $C_{ij} \cdot V^{max}$ is the same as it was previously. If the *bound* keyword was used, those bounds are enforced on the new $C_{ij}$ values. Henceforth, new bonds are assigned a $C_{ij}$ = 1.0, which means their bias potential magnitude is the new $V^{max}$.

The *check/ghost* keyword turns on extra computation each timestep to compute statistics about ghost atoms used to determine which bonds to bias. The output of these stats are the vector values 14 and 15, described below. If this keyword is not enabled, the output of the stats will be zero.

The *check/bias* keyword turns on extra computation and communication to check if any biased bonds are closer than *Dcut* to each other, which should not be the case if LHD is operating correctly. Thus it is a debugging check. The *Nevery* setting determines how often the check is made. The *error*, *warn*, or *ignore* setting determines what is done if the count of too-close bonds is not zero. Either the code will exit, or issue a warning, or silently tally the count. The count can be output as vector value 17, as described below. If this keyword is not enabled, the output of that statistic will be 0.

Note that both of these computations are costly, hence they are only enabled by these keywords.

---

### 2.96.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify* *energy* option is supported by this fix to add the energy of the bias potential to the global potential energy of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify energy no*.

This fix computes a global scalar and global vector of length 28, which can be accessed by various *output commands*. The scalar is the magnitude of the bias potential (energy units) applied on the current timestep, summed over all biased bonds. The vector stores the following quantities:

1. average boost for all bonds on this step (unitless)
2. # of biased bonds on this step
3. max strain $E_{ij}$ of any bond on this step (absolute value, unitless)
4. value of $V^{max}$ on this step (energy units)
5. average bias coeff for all bonds on this step (unitless)
6. min bias coeff for all bonds on this step (unitless)
7. max bias coeff for all bonds on this step (unitless)
8. average # of bonds/atom on this step
9. average neighbor bonds/bond on this step within *Dcut*
10. average boost for all bonds during this run (unitless)
11. average # of biased bonds/step during this run
12. fraction of biased bonds with no bias during this run
13. fraction of biased bonds with negative strain during this run
14. max bond length during this run (distance units)
15. average bias coeff for all bonds during this run (unitless)
16. min bias coeff for any bond during this run (unitless)

17. max bias coeff for any bond during this run (unitless)
18. max drift distance of any bond atom during this run (distance units)
19. max distance from proc subbox of any ghost atom with maxstrain < qfactor during this run (distance units)
20. max distance outside my box of any ghost atom with any maxstrain during this run (distance units)
21. count of ghost atoms that could not be found on reneighbor steps during this run
22. count of bias overlaps (< *Dcut*) found during this run
23. cumulative hyper time since fix created (time units)
24. cumulative count of event timesteps since fix created
25. cumulative count of atoms in events since fix created
26. cumulative # of new bonds formed since fix created
27. average boost for biased bonds on this step (unitless)
28. # of bonds with absolute strain >= q on this step

Quantities 1-9 are for the current timestep. Quantities 10-22 are for the current hyper run. They are reset each time a new hyper run is performed. Quantities 23-26 are cumulative across multiple runs (since the point in the input script the fix was defined).

For value 10, each bond instantaneous boost factor is given by the equation for $B_{ij}$ above. The total system boost (average across all bonds) fluctuates, but should average to a value close to the specified $B_{target}$.

For value 12, the numerator is a count of all biased bonds on each timestep whose bias energy = 0.0 due to $E_{ij} >= qfactor$. The denominator is the count of all biased bonds on all timesteps.

For value 13, the numerator is a count of all biased bonds on each timestep with negative strain. The denominator is the count of all biased bonds on all timesteps.

Values 18-22 are mostly useful for debugging and diagnostic purposes.

For value 18, drift is the distance an atom moves between two quenched states when the second quench determines an event has occurred. Atoms involved in an event will typically move the greatest distance since others typically remain near their original quenched position.

For values 19-21, neighbor atoms in the full neighbor list with cutoff *Dcut* may be ghost atoms outside a processor's sub-box. Before the next event occurs they may move further than *Dcut* away from the sub-box boundary. Value 19 is the furthest (from the sub-box) any ghost atom in the neighbor list with maxstrain < *qfactor* was accessed during the run. Value 20 is the same except that the ghost atom's maxstrain may be >= *qfactor*, which may mean it is about to participate in an event. Value 21 is a count of how many ghost atoms could not be found on reneighbor steps, presumably because they moved too far away due to their participation in an event (which will likely be detected at the next quench).

Typical values for 19 and 20 should be slightly larger than *Dcut*, which accounts for ghost atoms initially at a *Dcut* distance moving thermally before the next event takes place.

Note that for values 19 and 20 to be computed, the optional keyword *check/ghost* must be specified. Otherwise these values will be zero. This is because computing them incurs overhead, so the values are only computed if requested.

Value 21 should be zero or small. As explained above a small count likely means some ghost atoms were participating in their own events and moved a longer distance. If the value is large, it likely means the communication cutoff for ghosts is too close to *Dcut* leading to many not-found ghost atoms before the next event. This may lead to a reduced number of bonds being selected for biasing, since the code assumes those atoms are part of highly strained bonds. As explained above, the *comm_modify cutoff* command can be used to set a longer cutoff.

For value 22, no two bonds should be biased if they are within a *Dcut* distance of each other. This value should be zero, indicating that no pair of biased bonds are closer than *Dcut* from each other.

Note that for value 22 to be computed, the optional keyword *check/bias* must be specified and it determines how often this check is performed. This is because performing the check incurs overhead, so if only computed as often as requested.

The result at the end of the run is the cumulative total from every timestep the check was made. Note that the value is a count of atoms in bonds which found other atoms in bonds too close, so it is almost always an over-count of the number of too-close bonds.

Value 23 is simply the specified *boost* factor times the number of timesteps times the timestep size.

For value 24, events are checked for by the *hyper* command once every *Nevent* timesteps. This value is the count of those timesteps on which one (or more) events was detected. It is NOT the number of distinct events, since more than one event may occur in the same *Nevent* time window.

For value 25, each time the *hyper* command checks for an event, it invokes a compute to flag zero or more atoms as participating in one or more events. E.g. atoms that have displaced more than some distance from the previous quench state. Value 25 is the cumulative count of the number of atoms participating in any of the events that were found.

Value 26 tallies the number of new bonds created by the bond reset operation. Bonds between a specific I,J pair of atoms may persist for the entire hyperdynamics simulation if neither I or J are involved in an event.

Value 27 computes the average boost for biased bonds only on this step.

Value 28 is the count of bonds with an absolute value of strain >= q on this step.

The scalar value and vector values are all "intensive".

This fix also computes a local vector of length the number of bonds currently in the system. The value for each bond is its $C_{ij}$ prefactor (bias coefficient). These values can be can be accessed by various *output commands*. A particularly useful one is the *fix ave/histo* command which can be used to histogram the Cij values to see if they are distributed reasonably close to 1.0, which indicates a good choice of $V^{max}$.

The local values calculated by this fix are unitless.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.96.5 Restrictions

This fix is part of the REPLICA package. It is only enabled if LAMMPS was built with that package. See the *Build package* doc page for more info.

### 2.96.6 Related commands

*hyper*, *fix hyper/global*

### 2.96.7 Default

The default settings for optimal keywords are bounds = -1 and reset = -1. The check/ghost and check/bias keywords are not enabled by default.

---

**(Voter2013)** S. Y. Kim, D. Perez, A. F. Voter, J Chem Phys, 139, 144110 (2013).

**(Miron)** R. A. Miron and K. A. Fichthorn, J Chem Phys, 119, 6210 (2003).

# 2.97 fix ilves command

## 2.97.1 Syntax

```
fix ID group-ID ilves tol iter N selectors ... [keyword args ...]
```

- ID, group-ID are documented in *fix* command
- tol = convergence tolerance on the relative bond-length error
- iter = maximum number of Newton iterations per time step
- N = print constraint statistics every this many time steps (0 = never)
- selectors = one or more of *b*, *a*, *t*, *m*, each followed by one or more numeric values

  *b* values = one or more bond types (may use type labels)
  *a* values = one or more angle types (may use type labels)
  *t* values = one or more atom types (may use type labels)
  *m* values = one or more atom masses

- zero or more keyword/value pairs may be appended
- keyword = *mode* or *linearangle* or *kbond* or *store*

  *mode* value = *converge* or *fixed*
    *converge* = iterate until the tolerance is met (default)
    *fixed* = always perform *iter* Newton iterations
  *linearangle* values = style threshold
    style = *error* or *skip* or *restrain*
      *error* = stop if a selected angle type is near-linear (default)
      *skip* = do not constrain near-linear angle types
      *restrain* = replace the near-linear A-C constraint with a stiff harmonic␣
  ↪restraint
    threshold = equilibrium angle (in degrees) at or above which an angle type is␣
  ↪near-linear
  *kbond* value = dynamics force constant of the harmonic-bond substitute used
    for the *linearangle restrain* mode (the energy-minimization substitute is a
    fixed factor stiffer); in the active unit system
  *store* value = *yes* or *no*
    *yes* exposes the per-atom constraint forces via a per-atom array

## 2.97.2 Examples

```
fix 1 all ilves 1.0e-6 25 0 b 4 6 8 10
fix 2 wat ilves 1.0e-8 25 1000 b 18 a 31
fix 3 all ilves 1.0e-6 25 0 b 1 mode fixed
fix 4 sol ilves 1.0e-8 25 0 t 1 m 1.008 store yes
fix 5 co2 ilves 1.0e-8 25 0 b 1 a 1 linearangle restrain 170 kbond 2000
```

## 2.97.3 Description

Added in version 4Jul2026.

Apply bond-length and angle constraints using the ILVES algorithm of *(Lopez-Villellas)*. ILVES enforces holonomic distance constraints with Newton's method on a sparse system of nonlinear equations. Unlike *fix shake*, ILVES handles arbitrarily large connected constraint clusters — for example all the C-C backbone bonds of a long polymer or protein chain — in a single solve.

This command is a LAMMPS port of the reference ILVES implementation that the algorithm authors integrated into GROMACS; the constraint solver itself (the parallel Schur-complement sparse direct solver and the constraint topology handling) is reused largely unchanged, while the interface to the LAMMPS data structures and the time integration follow *fix shake*.

### User interface

The user interface of *fix ilves* follows that of *fix shake*. The three arguments after the *ilves* keyword are the tolerance $\frac{|g_k|}{d_k^2}$ (where $g_k = \frac{1}{2}(|s_k|^2 - d_k^2)$ and $d_k$ is the target length of constraint $k$), the maximum number of Newton iterations per step, and the frequency of statistics output (0 turns it off).

Then one or more groups of selectors (`b`, `a`, `t`, `m` lists) pick which bonds and angles get constrained. This selection is *identical* to *fix shake*. A bond is constrained when *both* of its atoms are in the fix group AND at least one of the selectors matches:

- the bond type is in the *b* list, or
- either atom type is in the *t* list (i.e. all bonds connected to an atom of a listed type are constrained), or
- either atom mass is within a fudge factor of MASSDELTA (0.1 mass units, defined in `src/RIGID/fix_ilves.cpp`) of a value in the *m* list (i.e. all bonds connected to an atom of a listed mass are constrained).

The types may be given as type labels *only* if there is no atom, bond, or angle type label named *b*, *a*, *t*, or *m* defined in the simulation. If that is the case, type labels cannot be used as constraint type index with these two fixes, because the type labels would be incorrectly treated as a new type of constraint instead. Thus, LAMMPS will print a warning and type label handling is disabled and numeric types must be used.

An angle whose type is in the *a* list contributes a "virtual bond" constraint on the distance between its two outer atoms (A and C of an A-B-C angle), which together with the two constrained legs makes the angle rigid. The angle is constrained when all three of its atoms are in the fix group, its type is selected, and *both* flanking bonds (A-B and B-C) are themselves constrained. The A-C target distance is computed from the two bond equilibrium lengths and the angle equilibrium value via the law of cosines, identical to *fix shake*.

For each bond or angle that is selected, *fix ilves* sets the corresponding `bond_type` or `angle_type` to its negative value so that the configured *bond_style* / *angle_style* skips the (now rigid) interaction and thus avoids double-counting of bonded forces. This mirrors how *fix shake* handles the same problem and is reversed automatically when the fix is deleted.

Unlike *fix shake*, which only supports small isolated clusters, the legs and the A-C virtual bond of an angle become part of the same connected constraint cluster that the ILVES solver handles directly, so angles sharing atoms (or angles within larger constrained networks) need no special casing.

### Near-linear angles

As the equilibrium angle approaches 180 degrees the A-C virtual bond becomes rank-deficient: the A-C distance is nearly stationary with respect to the angle, so the constraint is ill-conditioned and the solve degrades and eventually fails. Angle types whose equilibrium angle is at or above the *linearangle* threshold (default 175 degrees) are therefore treated specially according to the *linearangle* style:

- *error* (default): stop with an error identifying the offending angle type(s).
- *skip*: do not constrain those angle types at all. Their two flanking bonds are still held rigid, but the angle itself is left to the configured *angle_style*, which is the recommended choice when the force field provides a (well-behaved) angle term for the near-linear angle.
- *restrain*: replace the rank-deficient A-C constraint with a stiff harmonic restraint on the A-C distance, $E = k\,(r_{AC} - d_{AC})^2$, with force constant $k$ set by the *kbond* keyword (default 1000 in the active unit system). Because the A-C distance is insensitive to the angle near 180 degrees, this is a soft control on the angle itself; *skip* gives tighter angle behavior when a force-field angle term is available.

Unlike a constrained angle, a *skip* or *restrain* angle keeps its bending degree of freedom: *skip* leaves it free, and the *restrain* substitute is a soft potential whose A-C coordinate still vibrates and carries kinetic energy. Only the two rigid legs remove degrees of freedom in these modes, and the reported temperature accounts for this automatically. For *restrain*, choose *kbond* soft enough that this vibration is resolved by the timestep; an excessively stiff value is neither stable nor properly thermostatted.

The restraint energy of the *restrain* substitute is available as the global scalar of this fix and, with *fix_modify energy yes*, is added to the potential energy and the thermodynamic output.

## Algorithm

At each time step, after the force computation, *fix ilves*:

1. predicts the unconstrained position $x_i' = x_i + \Delta t\, v_i + \frac{\Delta t^2}{m_i} f_i$ for every atom (identical to *fix shake*),
2. solves the constraint equations $g_k(x') = 0$ with Newton's method, using a sparse direct (Schur-complement) solver, and
3. converts the resulting Lagrange multipliers into constraint forces that are added to `atom->f`, so that the following integration step produces the constrained positions.

As with *fix shake*, the constraints are enforced on the positions only; the velocities are projected onto the constraint manifold once at the start of a run (the analogue of SHAKE's initial velocity correction) but not every step. `fix ilves` therefore has no special requirement on its placement relative to other time-integration fixes, unlike a full RATTLE integrator.

The constraint topology is distributed: each MPI rank builds its constraint list from its own local bond storage plus the ghost atoms in the standard communication shell. The Newton iteration is global — its convergence test is the maximum relative bond-length violation reduced over all ranks — and the cross-rank coupling is resolved by communicating the predicted positions to the ghosts and reverse-summing the per-atom corrections to their owners each iteration. The results are therefore independent of the number of MPI ranks (to the solver tolerance).

By default the Newton iteration runs until the global maximum relative bond-length violation falls below *tol* (or *iter* iterations are reached). With `mode fixed` the solver instead performs exactly *iter* iterations every step. This removes the per-iteration global reduction used by the convergence test and can improve parallel efficiency, at the cost of not guaranteeing the tolerance. Because ILVES converges quadratically, a small fixed count is usually sufficient; run first with the default `mode converge` and a nonzero statistics interval *N* to see how many iterations are actually needed (the maximum is reported with the statistics) before choosing a fixed count.

## Output info

When `N > 0`, every `N` time steps the fix prints constraint statistics in the same layout as *fix shake*, so the two can be parsed interchangeably. Each line gives the bond or angle type, the average value (bond length for `Bond:` lines, bend angle in degrees for `Angle:` lines), the spread (max - min) of that value across all constraints of the type, and the count. The header line additionally reports the largest number of Newton iterations used since the previous output, which is specific to ILVES and is useful for choosing a fixed iteration count (see `mode fixed` above).

With the `store yes` keyword, this fix exposes a *per-atom array* with 3 columns containing the constraint-force components `(fx, fy, fz)` added by the fix on the current time step, accessible as `f_<ID>[1]`, `f_<ID>[2]`, and `f_<ID>[3]`.

This fix computes a global scalar, the potential energy of the *linearangle restrain* substitute (zero unless that mode is active), accessible as `f_<ID>`.

The constraint forces contribute to the global pressure virial by default, as for *fix shake*; use *fix_modify virial no* to exclude their contribution. The contribution is computed from the converged constraint multipliers and the start-of-step bond vectors, so it is reproduced exactly after a *restart* and does not depend on whether the masses are defined per atom type or per atom. The *linearangle restrain* and minimization restraint energy is likewise included in the potential energy by default.

#### Choosing between *fix shake* and *fix ilves*

As a rule of thumb, use *fix shake* wherever you can without running into its limitation of constraining only small isolated clusters, and use *fix ilves* elsewhere. Both fixes constrain the same bonds and angles to the same rigid geometry with the same tolerance.

*fix shake* is limited to small, isolated clusters (a central atom bonded to at most three others, or two bonds and an angle) and the code is optimized for this use case by minimizing communication and solving the constraints for each cluster independently.

Using *fix ilves* instead of *fix shake* avoids the small cluster limitation and can, for example, handle chains of constrained bonds, but that requires a different kind of constraint solver – one that needs communication between MPI ranks and, in parallel, more iterations and thus is not as efficient.

If you have a system that has both cases, e.g. a peptide with a constrained backbone and a rigid water solvent, you can consider using both fixes at the same time. That is, use *fix ilves* for the peptide and *fix shake* for the water.

---

### 2.97.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*; the constraint topology is rebuilt from `atom->bond_atom` at the first reneighbor. Redeclare the same `fix ilves` command after *read_restart* to restore the constraints (the negated bond types are preserved in the restart file).

The *fix_modify virial* and *energy* options are supported and both are enabled by default: the constraint contribution to the global pressure is included in the virial (use *fix_modify virial no* to exclude it), and the restraint and minimization energy is included in the potential energy (use *fix_modify energy no* to exclude it).

During *energy minimization* there is no time integration, so the holonomic constraints cannot be enforced as during dynamics. Instead each constrained bond (and angle A-C virtual bond) is replaced by a stiff harmonic bond $E = k_{\mathrm{bond}}(r - d)^2$ whose energy and forces are added to the minimization, exactly as *fix shake* does.

The *kbond* keyword sets the dynamics force constant (used by the *linearangle restrain* substitute); the minimization substitute is automatically a fixed factor of 2000 stiffer, because minimization has no timestep stability limit while a minimization-stiff restraint would be unstable in dynamics. Both defaults are expressed as multiples of the Boltzmann constant so they scale with the unit system rather than being tied to one unit choice: if *kbond* is not given it defaults to $5 \times 10^5 k_B$ for dynamics (about 1000 in real units), so the minimization default works out to $10^9 k_B$, the value used by *fix shake*. A *kbond* value set explicitly is likewise the dynamics constant and is scaled by the same factor for minimization. Minimize first, then run, so the holonomic solver tightens the bonds exactly once dynamics begins.

### 2.97.5 Restrictions

This fix is part of the RIGID package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The molecular topology (bonds) must be defined; an *atom_style* such as *full*, *molecular*, or *bond* (or a hybrid atom style including them) is required, and an *atom map* must be enabled.

Only one `fix ilves` instance may be defined at a time. `fix ilves` and *fix shake* may be used at the same time, but the sets of constrained atoms *must not* overlap.

`fix ilves` supports *run_style respa*. As for *fix shake*, the constraints are enforced at every r-RESPA level using the level-dependent effective timestep.

All atoms of a constraint cluster must lie within the communication cutoff of each other on every rank. For small clusters (water, methyl, hydrogen-only constraints) this is satisfied automatically; for clusters that span large distances increase the cutoff with *comm_modify cutoff*. The fix stops with an error if a constraint partner is not available locally.

`fix ilves` does not support dynamic topologies. Fixes or commands that add, remove, or change constrained bonds during a run (for example *fix bond/create*, *fix bond/break*, or *fix bond/react*) must not be applied to the constrained atoms.

### 2.97.6 Related commands

*fix shake*, *fix rattle*, *fix restrain*, *fix rigid*

### 2.97.7 Default

The keyword defaults are *mode* = *converge*, *linearangle* = *error* 175, and *store* = *no*. If *kbond* is not set, the dynamics restrain substitute uses $5 \times 10^5 k_B$ (about 1000 in real units) and the minimization substitute is a factor of 2000 stiffer ($10^9 k_B$, as *fix shake*); a *kbond* value sets the dynamics constant and is scaled by the same factor for minimization.

---

**(Lopez-Villellas)** L. Lopez-Villellas, C. C. K. Mikkelsen, J. J. Galano-Frutos, S. Marco-Sola, J. Alastruey-Benede, P. Ibanez, P. Echenique, M. Moreto, M. C. De Rosa, and P. Garcia-Risueno, "ILVES: Accurate and Efficient Bond Length and Angle Constraints in Molecular Dynamics", J. Chem. Theory Comput. 21, 8711-8719 (2025), https://doi.org/10.1021/acs.jctc.5c01376

## 2.98 fix imd command

### 2.98.1 Syntax

```
fix ID group-ID imd trate port keyword values ...
```

- ID, group-ID are documented in *fix* command
- imd = style name of this fix command
- port = port number on which the fix listens for an IMD client
- keyword = *unwrap* or *fscale* or *trate* or *nowait*

```
unwrap arg = on or off
  off = coordinates are wrapped back into the principal unit cell (default)
  on = "unwrapped" coordinates using the image flags used
fscale arg = factor
  factor = floating point number to scale IMD forces (default: 1.0)
trate arg = transmission rate of coordinate data sets (default: 1)
nowait arg = on or off
  off = LAMMPS waits to be connected to an IMD client before continuing (default)
  on = LAMMPS listens for an IMD client, but continues with the run
version arg = 2 or 3
  2 = use IMD protocol version 2 (default)
  3 = use IMD protocol version 3.
```

The following keywords are only supported for IMD protocol version 3.

```
time arg = on or off
   off = simulation time is not transmitted (default)
   on = simulation time is transmitted.
box arg = on or off
   off = simulation box data is not transmitted (default)
   on = simulation box data is transmitted.
coordinates arg = on or off
```

```
      off = atomic coordinates are not transmitted (default)
      on = atomic coordinates are transmitted.
   velocities arg = on or off
      off = atomic velocities are not transmitted (default)
      on = atomic velocities are transmitted.
   forces arg = on or off
      off = atomic forces are not transmitted (default)
      on = atomic forces are transmitted.
```

### 2.98.2 Examples

```
fix vmd all imd 5678
fix comm all imd 8888 trate 5 unwrap on fscale 10.0
```

### 2.98.3 Description

This fix implements the "Interactive MD" (IMD) protocol which allows realtime visualization and manipulation of MD simulations through the IMD protocol, as initially implemented in VMD and NAMD. Specifically it allows LAMMPS to connect an IMD client, for example the VMD visualization program (currently only supports IMDv2) or the Python IMDClient (supports both IMDv2 and IMDv3), so that it can monitor the progress of the simulation and interactively apply forces to selected atoms.

If LAMMPS is compiled with the pre-processor flag *-DLAMMPS_ASYNC_IMD* then fix imd will use POSIX threads to spawn an IMD communication thread on MPI rank 0 in order to offload data exchange with the IMD client from the main execution thread and potentially lower the inferred latencies for slow communication links. This feature has only been tested under linux.

The source code for this fix includes code developed by the Theoretical and Computational Biophysics Group in the Beckman Institute for Advanced Science and Technology at the University of Illinois at Urbana-Champaign. We thank them for providing a software interface that allows codes like LAMMPS to hook to VMD.

Upon initialization of the fix, it will open a communication port on the node with MPI task 0 and wait for an incoming connection. As soon as an IMD client is connected, the simulation will continue and the fix will send the current coordinates of the fix's group to the IMD client at every trate MD step. When using r-RESPA, trate applies to the steps of the outmost RESPA level. During a run with an active IMD connection also the IMD client can request to apply forces to selected atoms of the fix group.

The port number selected must be an available network port number. On many machines, port numbers < 1024 are reserved for accounts with system manager privilege and specific applications. If multiple imd fixes would be active at the same time, each needs to use a different port number.

The *nowait* keyword controls the behavior of the fix when no IMD client is connected. With the default setting of *off*, LAMMPS will wait until a connection is made before continuing with the execution. Setting *nowait* to *on* will have the LAMMPS code be ready to connect to a client, but continue with the simulation. This can for example be used to monitor the progress of an ongoing calculation without the need to be permanently connected or having to download a trajectory file.

The *trate* keyword allows to select how often the coordinate data is sent to the IMD client. It can also be changed on request of the IMD client through an IMD protocol message. The *unwrap* keyword allows to send "unwrapped" coordinates to the IMD client that undo the wrapping back of coordinates into the principle unit cell, as done by default in LAMMPS. The *fscale* keyword allows to apply a scaling factor to forces transmitted by the IMD client. The IMD protocols stipulates that forces are transferred in kcal/mol/Angstrom under the assumption that coordinates are given in Angstrom. For LAMMPS runs with different units or as a measure to tweak the forces generated by the manipulation of the IMD client, this option allows to make adjustments.

Added in version 4Feb2025.

In IMDv3, the IMD protocol has been extended to allow for the transmission of simulation time, box dimensions, atomic coordinates, velocities, and forces. The *version* keyword allows to select the version of the protocol to be used. The *time*, *box*, *coordinates*, *velocities*, and *forces* keywords allow to select which data is transmitted to the IMD client. The default is to transmit all data.

To connect VMD to a listening LAMMPS simulation on the same machine with fix imd enabled, one needs to start VMD and load a coordinate or topology file that matches the fix group. When the VMD command prompts appears, one types the command:

```
imd connect localhost 5678
```

This assumes that *fix imd* was started with 5678 as a port number for the IMD protocol.

The steps to do interactive manipulation of a running simulation in VMD are the following:

In the Mouse menu of the VMD Main window, select "Mouse -> Force -> Atom". You may alternately select "Residue", or "Fragment" to apply forces to whole residues or fragments. Your mouse can now be used to apply forces to your simulation. Click on an atom, residue, or fragment and drag to apply a force. Click quickly without moving the mouse to turn the force off. You can also use a variety of 3D position trackers to apply forces to your simulation. Game controllers or haptic devices with force-feedback such as the Novint Falcon or Sensable PHANTOM allow you to feel the resistance due to inertia or interactions with neighbors that the atoms experience you are trying to move, as if they were real objects. See the VMD IMD Homepage for more details.

If IMD control messages are received, a line of text describing the message and its effect will be printed to the LAMMPS output screen, if screen output is active.

### 2.98.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global scalar or vector or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.98.5 Restrictions

This fix is part of the MISC package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

When used in combination with VMD, a topology or coordinate file has to be loaded, which matches (in number and ordering of atoms) the group the fix is applied to. The fix internally sorts atom IDs by ascending integer value; in VMD (and thus the IMD protocol) those will be assigned 0-based consecutive index numbers.

When using multiple active IMD connections at the same time, each fix instance needs to use a different port number.

### 2.98.6 Related commands

none

### 2.98.7 Default

none

# 2.99 fix indent command

## 2.99.1 Syntax

```
fix ID group-ID indent K gstyle args keyword value ...
```

- ID, group-ID are documented in *fix* command
- indent = style name of this fix command
- K = force constant for indenter surface (force/distance^2 units)
- gstyle = *sphere* or *cylinder* or *cone* or *plane*

```
sphere args = x y z R
  x, y, z = position of center of indenter (distance units)
  R = sphere radius of indenter (distance units)
  any of x, y, z, R can be a variable (see below)
cylinder args = dim c1 c2 R
  dim = x or y or z = axis of cylinder
  c1, c2 = coords of cylinder axis in other 2 dimensions (distance units)
  R = cylinder radius of indenter (distance units)
  any of c1,c2,R can be a variable (see below)
cone args = dim c1 c2 radlo radhi lo hi
  dim = x or y or z = axis of cone
  c1, c2 = coords of cone axis in other 2 dimensions (distance units)
  radlo,radhi = cone radii at lo and hi end (distance units)
  lo,hi = bounds of cone in dim (distance units)
  any of c1, c2, radlo, radhi, lo, hi can be a variable (see below)
plane args = dim pos side
  dim = x or y or z = plane perpendicular to this dimension
  pos = position of plane in dimension x, y, or z (distance units)
  pos can be a variable (see below)
  side = lo or hi
```

- zero or more keyword/value pairs may be appended
- keyword = *side* or *units*

```
side value = in or out
  in = the indenter acts on particles inside the sphere or cylinder or cone
  out = the indenter acts on particles outside the sphere or cylinder or cone
units value = lattice or box
  lattice = the geometry is defined in lattice units
  box = the geometry is defined in simulation box units
```

## 2.99.2 Examples

```
fix 1 all indent 10.0 sphere 0.0 0.0 15.0 3.0
fix 1 all indent 10.0 sphere v_x v_y 0.0 v_radius side in
fix 2 flow indent 10.0 cylinder z 0.0 0.0 10.0 units box
```

### 2.99.3 Description

Insert an indenter within a simulation box. The indenter repels all atoms in the group that touch it, so it can be used to push into a material or as an obstacle in a flow. Alternatively, it can be used as a constraining wall around a simulation; see the discussion of the *side* keyword below.

The *gstyle* keyword selects the geometry of the indenter and it can either have the value of *sphere*, *cylinder*, *cone*, or *plane*.

A spherical indenter (*gstyle* = *sphere*) exerts a force of magnitude

$$F(r) = -K\,(r - R)^2$$

on each atom where $K$ is the specified force constant, $r$ is the distance from the atom to the center of the indenter, and $R$ is the radius of the indenter. The force is repulsive and F(r) = 0 for $r > R$.

A cylindrical indenter (*gstyle* = *cylinder*) follows the same formula for the force as a sphere, except that $r$ is defined the distance from the atom to the center axis of the cylinder. The cylinder extends infinitely along its axis.

Added in version 17April2024.

A conical indenter (*gstyle* = *cone*) is similar to a cylindrical indenter except that it has a finite length (between *lo* and *hi*), and that two different radii (one at each end, *radlo* and *radhi*) can be defined.

Spherical, cylindrical, and conical indenters account for periodic boundaries in two ways. First, the center point of a spherical indenter (x,y,z) or axis of a cylindrical/conical indenter (c1,c2) is remapped back into the simulation box, if the box is periodic in a particular dimension. This occurs every timestep if the indenter geometry is specified with a variable (see below), e.g. it is moving over time. Second, the calculation of distance to the indenter center or axis accounts for periodic boundaries. Both of these mean that an indenter can effectively move through and straddle one or more periodic boundaries.

A planar indenter (*gstyle* = *plane*) behaves like an axis-aligned infinite-extent wall with the same force expression on atoms in the system as before, but where $R$ is the position of the plane and $r$-$R$ is the distance of an from the plane. If the *side* parameter of the plane is specified as *lo* then it will indent from the lo end of the simulation box, meaning that atoms with a coordinate less than the plane's current position will be pushed towards the hi end of the box and atoms with a coordinate higher than the plane's current position will feel no force. Vice versa if *side* is specified as *hi*.

Any of the 4 quantities defining a spherical indenter's geometry can be specified as an equal-style *variable*, namely $x$, $y$, $z$, or $R$. For a cylindrical indenter, any of the 3 quantities *c1*, *c2*, or $R$, can be a variable. For a conical indenter, any of the 6 quantities *c1*, *c2*, *radlo*, *radhi*, *lo*, or *hi* can be a variable. For a planar indenter, the single value *pos* can be a variable.

If any of these values is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated each timestep, and its value used to define the indenter geometry.

Note that equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify indenter properties that change as a function of time or span consecutive runs in a continuous fashion. For the latter, see the *start* and *stop* keywords of the *run* command and the *elaplong* keyword of *thermo_style custom* for details.

For example, if a spherical indenter's x-position is specified as v_x, then this variable definition will keep its center at a relative position in the simulation box, 1/4 of the way from the left edge to the right edge, even if the box size changes:

```
variable x equal "xlo + 0.25*lx"
```

Similarly, either of these variable definitions will move the indenter from an initial position at 2.5 at a constant velocity of 5:

```
variable x equal "2.5 + 5*elaplong*dt"
variable x equal vdisplace(2.5,5)
```

If a spherical indenter's radius is specified as v_r, then these variable definitions will grow the size of the indenter at a specified rate.

```
variable r0 equal 0.0
variable rate equal 1.0
variable r equal "v_r0 + step*dt*v_rate"
```

If the *side* keyword is specified as *out*, which is the default, then particles outside the indenter are pushed away from its outer surface, as described above. This only applies to spherical, cylindrical, and conical indenters. If the *side* keyword is specified as *in*, the action of the indenter is reversed. Particles inside the indenter are pushed away from its inner surface. In other words, the indenter is now a containing wall that traps the particles inside it. If the radius shrinks over time, it will squeeze the particles.

The *units* keyword determines the meaning of the distance units used to define the indenter geometry. A *box* value selects standard distance units as defined by the *units* command, e.g. Angstroms for units = real or metal. A *lattice* value means the distance units are in lattice spacings. The *lattice* command must have been previously used to define the lattice spacing. The (x,y,z) coords of the indenter position are scaled by the x,y,z lattice spacings respectively. The radius of a spherical or cylindrical indenter is scaled by the x lattice spacing.

Note that the units keyword only affects indenter geometry parameters specified directly with numbers, not those specified as variables. In the latter case, you should use the *xlat*, *ylat*, *zlat* keywords of the *thermo_style* command if you want to include lattice spacings in a variable formula.

The force constant *K* is not affected by the *units* keyword. It is always in force/distance^2 units where force and distance are defined by the *units* command. If you wish K to be scaled by the lattice spacing, you can define K with a variable whose formula contains *xlat*, *ylat*, *zlat* keywords of the *thermo_style* command, e.g.

```
variable k equal 100.0/xlat/xlat
fix 1 all indent $k sphere ...
```

### 2.99.4 Dump image info

Added in version 11Feb2026.

Fix indent supports the *fix* keyword of *dump image*. The fix will pass geometry information about the indenter to *dump image* so that the indenter object will be included in the rendered image. Please note, that for *2d systems*, a planar indenter rendered as a plane would be invisible and it is thus rendered as a cylinder.

The color of the indenter object is by default that of the first atom type when using color styles "type" or "element". With color style "const" the default value of "white" can be changed using *dump_modify fcolor*. The transparency is by default fully opaque and can be changed with *dump_modify ftrans*.

The *fflag1* setting of *dump image fix* has no impact on rendering a spherical indenter or a planar indenter in 3d systems. For a cylindrical indenter and a planar indenter in 2d systems it determines whether the cylinder is capped with a sphere at the ends: 0 means no caps, 1 means the lower end is capped, 2 means the upper end is capped, and 3 means both ends are capped.

The *fflag2* setting allows you to adjust the radius of the rendered object for spherical indenters, cylindrical indenters, and planar indenters in 2d systems. In many cases you want to use a negative value to reduce the radius of the rendered object so that it does not obscure atoms close to it. For a planar indenter in 2d systems, it should be set to a positive value or the indenter will not be visible since there is no radius parameter associated with it (unlike for spherical or cylindrical indenters) and thus its radius is set to zero internally.

### 2.99.5 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify energy* option is supported by this fix to add the energy of interaction between atoms and the indenter to the global potential energy of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify energy no*. The energy of each particle interacting with the indenter is K/3 (r - R)^3.

The *fix_modify respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is adding its forces. Default is the outermost level.

This fix computes a global scalar energy and a global 3-vector of forces (on the indenter), which can be accessed by various *output commands*. The scalar and vector values calculated by this fix are "extensive".

The forces due to this fix are imposed during an energy minimization, invoked by the *minimize* command. Note that if you define the indenter geometry with a variable using a time-dependent formula, LAMMPS uses the iteration count in the minimizer as the timestep. But it is almost certainly a bad idea to have the indenter change its position or size during a minimization. LAMMPS does not check if you have done this.

**Note**

If you want the atom/indenter interaction energy to be included in the total potential energy of the system (the quantity being minimized), you must enable the *fix_modify energy* option for this fix.

### 2.99.6 Restrictions

none

### 2.99.7 Related commands

none

### 2.99.8 Default

The option defaults are side = out and units = lattice.

## 2.100 fix ipi command

### 2.100.1 Syntax

```
fix ID group-ID ipi address port [unix] [reset]
```

- ID, group-ID are documented in *fix* command
- ipi = style name of this fix command
- address = internet address (FQDN or IP), or UNIX socket name
- port = port number (ignored for UNIX sockets)
- zero or more keywords may be appended
- keyword = *unix* or *reset*

  *unix* args = none = use a unix socket
  *reset* args = none = reset electrostatics at each call

### 2.100.2 Examples

```
fix 1 all ipi my.server.com 12345
fix 1 all ipi mysocket 666 unix reset
```

### 2.100.3 Description

This fix enables LAMMPS to be run as a client for the i-PI Python wrapper *(IPI)*. i-PI is a universal force engine, designed to perform advanced molecular simulations, with a special focus on path integral molecular dynamics (PIMD) simulation. The philosophy behind i-PI is to separate the evaluation of the energy and forces, which is delegated to the client, and the evolution of the dynamics, that is the responsibility of i-PI. This approach also simplifies combining energies computed from different codes, which can for instance be useful to mix first-principles calculations, empirical force fields or machine-learning potentials. The following publication *(IPI-CPC-2014)* discusses the overall implementation of i-PI, and focuses on path-integral techniques, while a later release *(IPI-CPC-2019)* introduces several additional features and simulation schemes.

The communication between i-PI and LAMMPS takes place using sockets, and is reduced to the bare minimum. All the parameters of the dynamics are specified in the input of i-PI, and all the parameters of the force field must be specified as LAMMPS inputs, preceding the *fix ipi* command.

The server address must be specified by the *address* argument, and can be either the IP address, the fully-qualified name of the server, or the name of a UNIX socket for local, faster communication. In the case of internet sockets, the *port* argument specifies the port number on which i-PI is listening, while the *unix* optional switch specifies that the socket is a UNIX socket.

Note that there is no check of data integrity, or that the atomic configurations make sense. It is assumed that the species in the i-PI input are listed in the same order as in the data file of LAMMPS. The initial configuration is ignored, as it will be substituted with the coordinates received from i-PI before forces are ever evaluated.

A note of caution when using potentials that contain long-range electrostatics, or that contain parameters that depend on box size: all of these options will be initialized based on the cell size in the LAMMPS-side initial configuration and kept constant during the run. This is required to e.g. obtain reproducible and conserved forces. If the cell varies too wildly, it may be advisable to re-initialize these interactions at each call. This behavior can be requested by setting the *reset* switch.

### 2.100.4 Obtaining i-PI

Here are the commands to set up a virtual environment and install i-PI into it with all its dependencies via the PyPI repository and the pip package manager.

```
python -m venv ipienv
source ipienv/bin/activate
pip install --upgrade pip
pip install ipi
```

### 2.100.5 Restart, fix_modify, output, run start/stop, minimize info

There is no restart information associated with this fix, since all the dynamical parameters are dealt with by i-PI.

### 2.100.6 Restrictions

Using this fix on anything other than all atoms requires particular care, since i-PI will know nothing on atoms that are not those whose coordinates are transferred. However, one could use this strategy to define an external potential acting on the atoms that are moved by i-PI.

Since the i-PI code uses atomic units internally, this fix needs to convert LAMMPS data to and from its *specified units* accordingly when communicating with i-PI. This is not possible for reduced units ("units lj") and thus *fix ipi* will stop with an error in this case.

This fix is part of the MISC package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info. Because of the use of UNIX domain sockets, this fix will only work in a UNIX environment.

### 2.100.7 Related commands

*fix nve*

---

**(IPI-CPC-2014)** Ceriotti, More and Manolopoulos, Comp Phys Comm 185, 1019-1026 (2014).

**(IPI-CPC-2019)** Kapil et al., Comp Phys Comm 236, 214-223 (2019).

**(IPI)** https://ipi-code.org

## 2.101 fix lambda/apip command

### 2.101.1 Syntax

```
fix ID group-ID lambda/apip thr_lo thr_hi keyword args ...
```

- ID, group-ID are documented in *fix* command
- lambda/apip = style name of this fix command
- thr_lo = value below which $\lambda_i^{\text{input}}$ results in a switching parameter of 1
- thr_hi = value above which $\lambda_i^{\text{input}}$ results in a switching parameter of 0
- zero or one keyword/args pairs may be appended
- keyword = *time_averaged_zone* or *min_delta_lambda* or *lambda_non_group* or *store_atomic_stats* or *dump_atomic_history* or *group_fast* or *group_precise* or *group_ignore_lambda_input*

```
time_averaged_zone args = cut_lo cut_hi history_len_lambda_input history_len_lambda
  cut_lo = distance at which the radial function decreases from 1
  cut_hi = distance from which on the radial function is 0
  history_len_lambda_input = number of time steps for which lambda_input is averaged
  history_len_lambda = number of time steps for which the switching parameter is␣
↪averaged
min_delta_lambda args = delta
  delta = value below which changes of the switching parameter are neglected (>= 0)
lambda_non_group args = lambda_ng
  lambda_ng = precise or fast or float
    precise = assign a constant switching parameter of 0 to atoms, that are not in␣
↪the group specified by group-ID
    fast = assign a constant switching parameter of 1 to atoms, that are not in the␣
↪group specified by group-ID
    float = assign this constant switching parameter to atoms, that are not in the␣
↪group specified by group-ID (0 <= float <= 1)
group_fast args = group-ID-fast
  group-ID-fast = the switching parameter of 1 is used instead of the one computed␣
↪by lambda_input for atoms in the group specified by group-ID-fast
group_precise args = group-ID-precise
```

```
  group-ID-precise = the switching parameter of 0 is used instead of the one␣
↪computed by lambda_input for atoms in the group specified by group-ID-precise
group_ignore_lambda_input args = group-ID-ignore-lambda-input
  group-ID-ignore-lambda-input = the switching parameter of lambda_ng is used␣
↪instead of the one computed by lambda_input for atoms in the group specified by␣
↪group-ID-ignore-lambda-input
store_atomic_stats args = none
dump_atomic_history args = none
```

### 2.101.2 Examples

```
fix 2 all lambda/apip 3.0 3.5 time_averaged_zone 4.0 12.0 110 110 min_delta_lambda 0.01
fix 2 mobile lambda/apip 3.0 3.5 time_averaged_zone 4.0 12.0 110 110 min_delta_lambda 0.
↪01 group_ignore_lambda_input immobile lambda_non_group fast
```

### 2.101.3 Description

The potential energy $E_i$ of an atom $i$ of an adaptive-precision potential according to *(Immel)* is given by

$$E_i = \lambda_i E_i^{(\mathrm{fast})} + (1 - \lambda_i) E_i^{(\mathrm{precise})},$$

whereas $E_i^{(\mathrm{fast})}$ is the potential energy of atom $i$ according to a fast interatomic potential like EAM, $E_i^{(\mathrm{precise})}$ is the potential energy according to a precise interatomic potential such as ACE and $\lambda_i \in [0, 1]$ is the switching parameter that decides which potential energy is used. This fix calculates the switching parameter $\lambda_i$ based on the input provided from *pair_style lambda/input/apip*.

The calculation of the switching parameter is described in detail in *(Immel)*. This fix calculates the switching parameter for all atoms in the *group* described by group-ID, while the value of *lambda_non_group* is used as switching parameter for all other atoms.

First, this fix calculates per atom $i$ the time averaged input $\lambda_{\mathrm{avg},i}^{\mathrm{input}}$ from $\lambda_i^{\mathrm{input}}$, whereas the number of averaged timesteps can be set via *time_averaged_zone*.

**Note**

$\lambda_i^{\mathrm{input}}$ is calculated by *pair_style lambda/input/apip*, which needs to be included in the input script as well.

The time averaged input $\lambda_{\mathrm{avg},i}^{\mathrm{input}}$ is then used to calculate the switching parameter

$$\lambda_{0,i}(t) = f^{(\mathrm{cut})}\left(\frac{\lambda_{\mathrm{avg},i}^{\mathrm{input}}(t) - \lambda_{\mathrm{lo}}^{\mathrm{input}}}{\lambda_{\mathrm{hi}}^{\mathrm{input}} - \lambda_{\mathrm{lo}}^{\mathrm{input}}}\right),$$

whereas the thresholds $\lambda_{\mathrm{hi}}^{\mathrm{input}}$ and $\lambda_{\mathrm{lo}}^{\mathrm{input}}$ are set by the values provided as *thr_lo* and *thr_hi* and $f^{(\mathrm{cut})}(x)$ is a cutoff function that is 1 for $x \leq 0$, decays from 1 to 0 for $x \in [0, 1]$, and is 0 for $x \geq 1$. If the *group_precise* argument is used, $\lambda_{0,i} = 0$ is used for all atoms $i$ assigned to the corresponding *group*. If the *group_fast* argument is used, $\lambda_{0,i} = 1$ is used for all atoms $i$ assigned to the corresponding *group*. If an atom is in the groups *group_fast* and *group_precise*, $\lambda_{0,i} = 0$ is used. If the *group_ignore_lambda_input* argument is used, $\lambda_i^{\mathrm{input}}$ is not computed for all atoms $i$ assigned to the corresponding *group*; instead, if the value is not already set by *group_fast* or *group_precise*, the value of *lambda_non_group* is used.

**Note**

The computation of $\lambda_i^{\text{input}}$ is not required for atoms that are in the groups *group_fast* and *group_precise*. Thus, one should use *group_ignore_lambda_input* and prevent the computation of $\lambda_i^{\text{input}}$ for all atoms, for which a constant input is used.

A spatial transition zone between the fast and the precise potential is introduced via

$$\lambda_{\text{min},i}(t) = \min\left(\left\{1 - (1 - \lambda_{0,j}(t))f^{(\text{cut})}\left(\frac{r_{ij}(t) - r_{\lambda,\text{lo}}}{r_{\lambda,\text{hi}} - r_{\lambda,\text{lo}}}\right) : j \in \Omega_{\lambda,i}\right\}\right),$$

whereas the thresholds $r_{\lambda,\text{lo}}$ and $r_{\lambda,\text{hi}}$ of the cutoff function are set via *time_averaged_zone* and $\Omega_{\lambda,i}$ is the set of neighboring atoms of atom $i$.

**Note**

$\lambda_{\text{min},i}$ is calculated by *pair_style lambda/zone/apip*, which needs to be included in the input script as well.

The switching parameter is smoothed by the calculation of the time average

$$\lambda_{\text{avg},i}(t) = \frac{1}{N_{\lambda,\text{avg}}} \sum_{n=1}^{N_{\lambda,\text{avg}}} \lambda_{\text{min},i}(t - n\Delta t),$$

whereas $\Delta t$ is the *timestep* and $N_{\lambda,\text{avg}}$ is the number of averaged timesteps, that can be set via *time_averaged_zone*.

Finally, numerical fluctuations of the switching parameter are suppressed by the usage of

$$\lambda_i(t) = \begin{cases} \lambda_{\text{avg},i}(t) & \text{for } |\lambda_{\text{avg},i}(t) - \lambda_i(t - \Delta t)| \geq \Delta\lambda_{\text{min}} \text{ or } \lambda_{\text{avg},i}(t) \in \{0, 1\}, \\ \lambda_i(t - \Delta t) & \text{otherwise}, \end{cases}$$

whereas the minimum change $\Delta\lambda_{\text{min}}$ is set by the *min_delta_lambda* argument.

**Note**

*group_fast* affects only $\lambda_{0,i}(t)$. The switching parameter of atoms in this *group* may change due to the calculation of the spatial switching zone. A switching parameter of 1 can be enforced by excluding the corresponding atoms from the *group* described by group-ID and using *lambda_non_group* 1 as argument.

---

A code example for the calculation of the switching parameter for an adaptive-precision potential is given in the following: The adaptive-precision potential is created by combining *pair_style eam/fs/apip* and *pair_style pace/precise/apip*. The input, from which the switching parameter is calculated, is provided by *pair lambda/input/csp/apip*. The switching parameter is calculated by this fix, whereas the spatial transition zone of the switching parameter is calculated by *pair_style lambda/zone/apip*.

```
pair_style hybrid/overlay eam/fs/apip pace/precise/apip lambda/input/csp/apip fcc cutoff␣
↪5.0 lambda/zone/apip 12.0
pair_coeff * * eam/fs/apip Cu.eam.fs Cu
pair_coeff * * pace/precise/apip Cu_precise.yace Cu
pair_coeff * * lambda/input/csp/apip
pair_coeff * * lambda/zone/apip
fix 2 all lambda/apip 3.0 3.5 time_averaged_zone 4.0 12.0 110 110 min_delta_lambda 0.01
```

---

### 2.101.4 Restart, fix_modify, output, run start/stop, minimize info

The saved history of the switching parameter $\lambda_i$ and the saved history of $\lambda_i^{\text{input}}$ are written to *binary restart files* to allow a smooth restart of a simulation. None of the *fix_modify* options are relevant to this fix.

If the *store_atomic_stats* argument is used, basic statistics is provided as per-atom array:

1. $\lambda_i^{\text{input}}(t)$
2. $\lambda_{\text{avg},i}^{\text{input}}(t)$
3. $\lambda_{0,i}(t)$
4. $\lambda_{\text{min},i}(t)$
5. $\lambda_i(t)$

If the *dump_atomic_history* argument is used, the whole saved history of $\lambda_i^{\text{input}}(t)$ is appended to the previously mentioned array per atom.

The per-atom vector can be accessed by various *output commands*.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

---

### 2.101.5 Restrictions

This fix is part of the APIP package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.101.6 Related commands

*pair_style lambda/zone/apip*, *pair_style lambda/input/apip*, *pair_style eam/apip*, *pair_style pace/apip*, *fix atom_weight/apip fix lambda_thermostat/apip*,

### 2.101.7 Default

*min_delta_lambda* = 0, *lambda_non_group* = 1, *cut_lo* = 4.0, *cut_hi* = 12.0, *history_len_lambda_input* = 100, *history_len_lambda* = 100, *store_atomic_stats* is not used, *dump_atomic_history* is not used, *group_fast* is not used, *group_precise* is not used, *group_ignore_lambda_input* is not used

---

**(Immel)** Immel, Drautz and Sutmann, J Chem Phys, 162, 114119 (2025)

## 2.102 fix lambda/la/csp/apip command

### 2.102.1 Syntax

```
fix ID group-ID lambda/la/csp/apip thr_lo thr_hi cut_lo cut_hi lattice keyword args ...
```

- ID, group-ID are documented in *fix* command
- lambda/la/csp/apip = style name of this fix command
- thr_lo = value below which the differentiable CSP results in a switching parameter of 1 (squared distance units)
- thr_hi = value above which the differentiable CSP results in a switching parameter of 0 (squared distance units)

- cut_lo = radius below which the weighting function for the local averaging is 1 (distance units)
- cut_hi = radius above which the weighting function for the local averaging is 0 (distance units)
- lattice = *fcc* or *bcc* or N = # of neighbors per atom to include in the CSP calculation
- zero or more keyword/args pairs may be appended
- keyword = *csp_cut* or *csp_mode* or *forces* or *lambda_non_group* or *store_peratom*

```
csp_cut args = float
    float = neighboring atoms outside of this cutoff radius may not be considered␣
↪for the CSP calculation
csp_mode args = dynamic or static
    dynamic = use the differentiable CSP to calculate the switching parameter
    static = use the non-differentiable CSP instead of the differentiable one
forces args = no or yes
    yes = compute the forces caused by the differentiation of the switching␣
↪parameter
    no = do not compute the forces caused by the differentiation of the switching␣
↪parameter
lambda_non_group args = precise or fast or float
    precise = assign a constant switching parameter of 0 to atoms, that are not in␣
↪the group specified by group-ID
    fast = assign a constant switching parameter of 1 to atoms, that are not in the␣
↪group specified by group-ID
    float = assign this constant switching parameter to atoms, that are not in the␣
↪group specified by group-ID (0 <= float <= 1)
store_peratom args = integer
    integer = provide per-atom output every this many timesteps
```

### 2.102.2 Examples

```
fix lambda_la all lambda/la/csp/apip 0.25 1.5 15.0 16.0 bcc
fix lambda_la mobile lambda/la/csp/apip 0.24 1.5 11.0 12.0 bcc lambda_non_group fast
```

### 2.102.3 Description

Added in version 30Mar2026.

The potential energy $E_i$ of an atom $i$ according to an adaptive-precision potential is given by *(Immel2025)*

$$E_i = \lambda_i E_i^{(\mathrm{fast})} + (1 - \lambda_i) E_i^{(\mathrm{precise})} ,$$

where $E_i^{(\mathrm{fast})}$ is the potential energy of atom $i$ according to a fast computable interatomic potential, $E_i^{(\mathrm{precise})}$ is the potential energy according to a precise interatomic potential and $\lambda_i \in [0, 1]$ is the switching parameter that decides which potential energy is used.

This fix calculates the switching parameter $\lambda_i$ based on local averaging of a descriptor according to *(Immel2026)* and parts of the conservatively calculated force as will be discussed later. The descriptor is averaged within the cutoff radius provided as *cut_hi*.

Per default, a differentiable version of the centro-symmetry parameter (CSP) is used as descriptor. This differentiable version is described in detail in *(Immel2026)*. The usage of a differentiable CSP results in a conservative potential, that conserves (in the absence of external forces) energy and momentum by design. The force $\boldsymbol{F}_i = -\nabla_i \sum_k E_k$ following from the adaptive-precision potential energy is given by

$$\boldsymbol{F}_i = \sum_k \left( -\lambda_k \nabla_i E_k^{(\mathrm{fast})} - (1 - \lambda_k) \nabla_i E_k^{(\mathrm{precise})} + (\nabla_i \lambda_k)(E_k^{(\mathrm{precise})} - E_k^{(\mathrm{fast})}) \right) .$$

This fix calculates the terms $(\nabla_i \lambda_k)(E_k^{(\text{precise})} - E_k^{(\text{fast})})$ based on potential energies that are provided by pair styles. This force-calculation is enabled by default, but prevented by *forces = no*. Thus, one can use this fix to only calculate switching parameters.

**Note**

The fast potential and the precise potential are combined via *pair_style hybrid/overlay* as shown in the code example below.

The original CSP *(Kelchner)* is used instead of the differentiable CSP with the option *csp_mode dynamic*.

**Warning**

Note that the original CSP is not differentiable and does not result in a conservative potential.

This fix calculates the switching parameter for all atoms in the *group* described by group-ID, while the value of *lambda_non_group* is used as switching parameter for all other atoms.

---

A code example for the usage of a conservative adaptive-precision interatomic potential is given in the following: This fix calculates the switching parameter. The *pair_style hybrid/overlay* is used to combine the two pair styles *pair_style eam/fs/apip* and *pair_style pace/precise/apip*, which calculate the potential energies $E_k^{(\text{precise})}$ and $E_k^{(\text{fast})}$. The conservative force is calculated by both the fix and the pair styles.

```
fix lambda_la all lambda/la/csp/apip 0.24 1.5 11.0 12.0 bcc
pair_style hybrid/overlay eam/fs/apip pace/apip
pair_coeff * * eam/fs/apip W.eam.fs W
pair_coeff * * pace/apip W.yace W
```

---

### 2.102.4 Restart, fix_modify, output, run start/stop, minimize info

The CSP neighbors of the differentiable CSP are saved and written to *binary restart files* to allow a smooth restart of a simulation. None of the *fix_modify* options are relevant to this fix.

This fix can compute the force caused by the differentiation of the switching parameters. Therefore, the *fix_modify virial* option is supported by this fix to add the contribution of these forces to both the global pressure and per-atom stress of the system via the *compute pressure* and *compute stress/atom* commands. The former can be accessed by *thermodynamic output*. The default setting for this fix is *fix_modify virial yes*.

If the (non-conservative) dynamic CSP-pairs are used, this fix returns the accumulated number of changed CSP-pairs as global scalar. The potential is conservative as long as this number of changed CSP-pairs is constant (every changed CSP pair may change the total energy of the system). The global scalar can be accessed by various *output commands*.

If *store_peratom* is used, 5 quantities are provided as per-atom vector: 1.-3. the force that is caused by the differentiation of the switching parameter, 4. the differentiable CSP and 5. the locally averaged differentiable CSP that was used to calculate the switching parameter. The per-atom vector can be accessed by various *output commands*.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

---

### 2.102.5 Restrictions

This fix is part of the APIP package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.102.6 Related commands

*pair_style eam/apip*, *pair_style pace/apip*

### 2.102.7 Default

*forces* = *yes*, *csp_cut* = 5.0, *csp_mode* = *static*, *lambda_non_group* = 1, *store_peratom* = $\infty$

---

**(Immel2025)** Immel, Drautz and Sutmann, J Chem Phys, 162, 114119 (2025)

**(Immel2026)** Immel, Drautz and Sutmann, arXiv:2512.07693

**(Kelchner)** Kelchner, Plimpton, Hamilton, Phys Rev B, 58, 11085 (1998)

## 2.103 fix lambda_thermostat/apip command

### 2.103.1 Syntax

```
fix ID group-ID lambda_thermostat/apip keyword values ...
```

- ID, group-ID are documented in *fix* command
- lambda_thermostat/apip = style name of this fix command
- zero or more keyword/value pairs may be appended
- keyword = *seed* or *store_atomic_forces* or *N_rescaling*

```
seed value = integer
  integer = integer that is used as seed for the random number generator (> 0)
store_atomic_forces value = nevery
  nevery = provide per-atom output every this many steps
N_rescaling value = groupsize
  groupsize = rescale this many neighboring atoms (> 1)
```

### 2.103.2 Examples

```
fix 2 all lambda_thermostat/apip
fix 2 all lambda_thermostat/apip N_rescaling 100
fix 2 all lambda_thermostat/apip seed 42
fix 2 all lambda_thermostat/apip seed 42 store_atomic_forces 1000
```

### 2.103.3 Description

This command applies the local thermostat described in *(Immel)* to conserve the energy when the switching parameters of an *adaptive-precision interatomic potential* (APIP) are updated while the gradient of the switching parameter is neglected in the force calculation.

**Warning**

The temperature change caused by this fix is only the means to the end of conserving the energy. Thus, this fix is not a classical thermostat, that ensures a given temperature in the system. All available thermostats are listed *here*.

The potential energy $E_i$ of an atom $i$ is given by the formula from *(Immel)*

$$E_i = \lambda_i E_i^{(\mathrm{fast})} + (1 - \lambda_i) E_i^{(\mathrm{precise})},$$

whereas $E_i^{(\mathrm{fast})}$ is the potential energy of atom $i$ according to a fast interatomic potential like EAM, $E_i^{(\mathrm{precise})}$ is the potential energy according to a precise interatomic potential such as ACE and $\lambda_i \in [0, 1]$ is the switching parameter that decides which potential energy is used. This potential energy and the corresponding forces are conservative when the switching parameter $\lambda_i$ is constant in time for all atoms $i$.

For a conservative force calculation and dynamic switching parameters, the atomic force on an atom is given by $F_i = -\nabla_i \sum_j E_j$ and includes the derivative of the switching parameter $\lambda_i$. The force contribution of this gradient of the switching function can cause large forces which are not similar to the forces of the fast or the precise interatomic potential as discussed in *(Immel)*. Thus, one can neglect the gradient of the switching parameter in the force calculation and compensate for the violation of energy conservation by the application of the local thermostat implemented in this fix. One can compute the violation of the energy conservation $\Delta H_i$ for all atoms $i$ as discussed in *(Immel)*. To locally correct this energy violation $\Delta H_i$, one can rescale the velocity of atom $i$ and of neighboring atoms. The rescaling is done relative to the center-of-mass velocity of the group and, thus, conserves the momentum.

**Note**

This local thermostat provides the NVE ensemble rather than the NVT ensemble as the energy $\Delta H_i$ determines the rescaling factor rather than a temperature.

Velocities $v$ are updated by the integrator according to $\Delta v_i = (F_i/m_i)\Delta t$, whereas $m$ denotes the mass of atom $i$ and $\Delta t$ is the time step. One can interpret the velocity difference $\Delta v$ caused by the rescaling as the application of an additional force which is given by $F_i^{\mathrm{lt}} = (v_i^{\mathrm{unscaled}} - v_i^{\mathrm{rescaled}}) m_i / \Delta t$ *(Immel)*. This additional force is computed when the *store_atomic_forces* option is used.

The local thermostat is not appropriate for simulations at a temperature of 0K.

**Note**

The maximum decrease of the kinetic energy is achieved with a rescaling factor of 0, i.e., the relative velocity of the group of rescaled atoms is set to zero. One cannot decrease the energy further. Thus, the local thermostat can fail, which is, however, reported by the returned vector.

### 2.103.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

If the *store_atomic_forces* option is used, this fix produces every *nevery* time steps a per-atom array that contains the theoretical force applied by the local thermostat in all three spatial dimensions in the first three components. $\Delta H_i$ is the fourth component of the per-atom array. The per-atom array can only be accessed on timesteps that are multiples of *nevery*.

Furthermore, this fix computes a global vector of length 6 with information about the rescaling:

1. number of atoms whose energy changed due to the last $\lambda$ update
2. contribution of the potential energy to the last computed $\Delta H$
3. contribution of the kinetic energy to the last computed $\Delta H$
4. sum over all atoms of the absolute energy change caused by the last rescaling step
5. energy change that could not be compensated accumulated over all timesteps
6. number of atoms whose energy change could not be compensated accumulated over all timesteps

The vector and the per-atom vector can be accessed by various *output commands*.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

---

### 2.103.5 Restrictions

This fix is part of the APIP package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.103.6 Related commands

*fix lambda/apip*, *pair_style lambda/zone/apip*, *pair_style lambda/input/apip*, *pair_style eam/apip*, *pair_style pace/apip*, *fix atom_weight/apip*

### 2.103.7 Default

seed = 42, N_rescaling = 200, *store_atomic_forces* is not used

---

**(Immel)** Immel, Drautz and Sutmann, J Chem Phys, 162, 114119 (2025)

## 2.104 fix langevin command

Accelerator Variants: *langevin/kk*

### 2.104.1 Syntax

```
fix ID group-ID langevin Tstart Tstop damp seed keyword values ...
```

- ID, group-ID are documented in *fix* command
- langevin = style name of this fix command
- Tstart,Tstop = desired temperature at start/end of run (temperature units)
- Tstart can be a variable (see below)
- damp = damping parameter (time units)
- seed = random number seed to use for white noise (positive integer)
- zero or more keyword/value pairs may be appended
- keyword = *angmom* or *omega* or *scale* or *tally* or *zero*

```
angmom value = no or factor
  no = do not thermostat rotational degrees of freedom via the angular momentum
  factor = do thermostat rotational degrees of freedom via the angular momentum and␣
→apply numeric scale factor as discussed below
omega value = no or yes
  no = do not thermostat rotational degrees of freedom via the angular velocity
  yes = do thermostat rotational degrees of freedom via the angular velocity
scale values = type ratio
  type = atom type (1-N)
  ratio = factor by which to scale the damping coefficient
tally value = no or yes
  no = do not tally the energy added/subtracted to atoms
  yes = do tally the energy added/subtracted to atoms
zero value = no or yes
  no = do not set total random force to zero
  yes = set total random force to zero
```

### 2.104.2 Examples

```
fix 3 boundary langevin 1.0 1.0 1000.0 699483
fix 1 all langevin 1.0 1.1 100.0 48279 scale 3 1.5
fix 1 all langevin 1.0 1.1 100.0 48279 angmom 3.333
```

### 2.104.3 Description

Apply a Langevin thermostat as described in *(Bruenger)* to a group of atoms which models an interaction with a background implicit solvent. Used with *fix nve*, this command performs Brownian dynamics (BD), since the total force on each atom will have the form:

$$
\begin{aligned}
F &= F_c + F_f + F_r \\
F_f &= -\frac{m}{\text{damp}} v \\
F_r &\propto \sqrt{\frac{k_B T m}{dt \ \text{damp}}}
\end{aligned}
$$

$F_c$ is the conservative force computed via the usual inter-particle interactions (*pair_style*, *bond_style*, etc). The $F_f$ and $F_r$ terms are added by this fix on a per-particle basis. See the *pair_style dpd/tstat* command for a thermostatting option that adds similar terms on a pairwise basis to pairs of interacting particles.

$F_f$ is a frictional drag or viscous damping term proportional to the particle's velocity. The proportionality constant for each atom is computed as $\frac{m}{\text{damp}}$, where $m$ is the mass of the particle and damp is the damping factor specified by the user.

$F_r$ is a force due to solvent atoms at a temperature $T$ randomly bumping into the particle. As derived from the fluctuation/dissipation theorem, its magnitude as shown above is proportional to $\sqrt{\frac{k_B T m}{dt \ \text{damp}}}$, where $k_B$ is the Boltzmann constant, $T$ is the desired temperature, $m$ is the mass of the particle, $dt$ is the timestep size, and damp is the damping factor. Random numbers are used to randomize the direction and magnitude of this force as described in *(Dunweg)*, where a uniform random number is used (instead of a Gaussian random number) for speed.

Note that unless you use the *omega* or *angmom* keywords, the thermostat effect of this fix is applied to only the translational degrees of freedom for the particles, which is an important consideration for finite-size particles, which have rotational degrees of freedom, are being thermostatted. The translational degrees of freedom can also have a bias velocity removed from them before thermostatting takes place; see the description below.

**Note**

Unlike the *fix nvt* command which performs Nose/Hoover thermostatting AND time integration, this fix does NOT perform time integration. It only modifies forces to effect thermostatting. Thus you must use a separate time integration fix, like *fix nve* to actually update the velocities and positions of atoms using the modified forces. Likewise, this fix should not normally be used on atoms that also have their temperature controlled by another fix - e.g. by *fix nvt* or *fix temp/rescale* commands.

See the *Howto thermostat* page for a discussion of different ways to compute temperature and perform thermostatting.

The desired temperature at each timestep is a ramped value during the run from *Tstart* to *Tstop*.

*Tstart* can be specified as an equal-style or atom-style *variable*. In this case, the *Tstop* setting is ignored. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated each timestep, and its value used to determine the target temperature.

Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent temperature.

Atom-style variables can specify the same formulas as equal-style variables but can also include per-atom values, such as atom coordinates. Thus it is easy to specify a spatially-dependent temperature with optional time-dependence as well.

Like other fixes that perform thermostatting, this fix can be used with *compute commands* that remove a "bias" from the atom velocities. E.g. to apply the thermostat only to atoms within a spatial *region*, or to remove the center-of-mass velocity from a group of atoms, or to remove the x-component of velocity from the calculation.

This is not done by default, but only if the *fix_modify* command is used to assign a temperature compute to this fix that includes such a bias term. See the doc pages for individual *compute temp commands* to determine which ones include a bias. In this case, the thermostat works in the following manner: bias is removed from each atom, thermostatting is performed on the remaining thermal degrees of freedom, and the bias is added back in.

The *damp* parameter is specified in time units and determines how rapidly the temperature is relaxed. For example, a value of 100.0 means to relax the temperature in a timespan of (roughly) 100 time units ($\tau$ or fs or ps - see the *units* command). The damp factor can be thought of as inversely related to the viscosity of the solvent. I.e. a small relaxation time implies a high-viscosity solvent and vice versa. See the discussion about $\gamma$ and viscosity in the documentation for the *fix viscous* command for more details.

The random # *seed* must be a positive integer. A Marsaglia random number generator is used. Each processor uses the input seed to generate its own unique seed and its own stream of random numbers. Thus the dynamics of the system will not be identical on two runs on different numbers of processors.

---

The keyword/value option pairs are used in the following ways.

The keyword *angmom* and *omega* keywords enable thermostatting of rotational degrees of freedom in addition to the usual translational degrees of freedom. This can only be done for finite-size particles.

A simulation using atom_style sphere defines an omega for finite-size spheres. A simulation using atom_style ellipsoid defines a finite size and shape for aspherical particles and an angular momentum. The Langevin formulas for thermostatting the rotational degrees of freedom are the same as those above, where force is replaced by torque, m is replaced by the moment of inertia I, and v is replaced by omega (which is derived from the angular momentum in the case of aspherical particles).

The rotational temperature of the particles can be monitored by the *compute temp/sphere* and *compute temp/asphere* commands with their rotate options.

For the *omega* keyword there is also a scale factor of $\frac{10.0}{3.0}$ that is applied as a multiplier on the $F_f$ (damping) term in the equation above and of $\sqrt{\frac{10.0}{3.0}}$ as a multiplier on the $F_r$ term. This does not affect the thermostatting behavior of the Langevin formalism but ensures that the randomized rotational diffusivity of spherical particles is correct.

For the *angmom* keyword a similar scale factor is needed which is $\frac{10.0}{3.0}$ for spherical particles, but is anisotropic for aspherical particles (e.g. ellipsoids). Currently LAMMPS only applies an isotropic scale factor, and you can choose its magnitude as the specified value of the *angmom* keyword. If your aspherical particles are (nearly) spherical than a value of $\frac{10.0}{3.0} = 3.\overline{3}$ is a good choice. If they are highly aspherical, a value of 1.0 is as good a choice as any, since the effects on rotational diffusivity of the particles will be incorrect regardless. Note that for any reasonable scale factor, the thermostatting effect of the *angmom* keyword on the rotational temperature of the aspherical particles should still be valid.

The keyword *scale* allows the damp factor to be scaled up or down by the specified factor for atoms of that type. This can be useful when different atom types have different sizes or masses. It can be used multiple times to adjust damp for several atom types. Note that specifying a ratio of 2 increases the relaxation time which is equivalent to the solvent's viscosity acting on particles with $\frac{1}{2}$ the diameter. This is the opposite effect of scale factors used by the *fix viscous* command, since the damp factor in fix *langevin* is inversely related to the $\gamma$ factor in fix *viscous*. Also note that the damping factor in fix *langevin* includes the particle mass in Ff, unlike fix *viscous*. Thus the mass and size of different atom types should be accounted for in the choice of ratio values.

The keyword *tally* enables the calculation of the cumulative energy added/subtracted to the atoms as they are thermostatted. Effectively it is the energy exchanged between the infinite thermal reservoir and the particles. As described below, this energy can then be printed out or added to the potential energy of the system to monitor energy conservation.

The keyword *zero* can be used to eliminate drift due to the thermostat. Because the random forces on different atoms are independent, they do not sum exactly to zero. As a result, this fix applies a small random force to the entire system, and the center-of-mass of the system undergoes a slow random walk. If the keyword *zero* is set to *yes*, the total random force is set exactly to zero by subtracting off an equal part of it from each atom in the group. As a result, the center-of-mass of a system with zero initial momentum will not drift over time.

Deprecated since version 22Jul2025.

The *gjf* keyword in fix langevin has been removed and the GJF functionality has been moved to its own fix style *fix gjf* . and it is strongly recommended to use that fix instead.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.104.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. Because the state of the random number generator is not saved in restart files, this means you cannot do "exact" restarts with this fix, where the simulation continues on the same as if no restart had taken place. However, in a statistical sense, a restarted simulation should produce the same behavior.

The *fix_modify temp* option is supported by this fix. You can use it to assign a temperature *compute* you have defined to this fix which will be used in its thermostatting procedure, as described above. For consistency, the group used by this fix and by the compute should be the same.

The cumulative energy change in the system imposed by this fix is included in the *thermodynamic output* keywords *ecouple* and *econserve*, but only if the *tally* keyword to set to *yes*. See the *thermo_style* page for details.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the same cumulative energy change due to this fix described in the previous paragraph. The scalar value calculated by this fix is "extensive". Note that calculation of this quantity also requires setting the *tally* keyword to *yes*.

This fix can ramp its target temperature over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.104.5 Restrictions

none.

### 2.104.6 Related commands

*fix nvt*, *fix temp/rescale*, *fix viscous*, *fix nvt*, *pair_style dpd/tstat*, *fix gjf* , *fix gle*, *fix gld*

### 2.104.7 Default

The option defaults are angmom = no, omega = no, scale = 1.0 for all types, tally = no, zero = no.

---

**(Bruenger)** Bruenger, Brooks, and Karplus, Chem. Phys. Lett. 105, 495 (1982). [Previously attributed to Schneider and Stoll, Phys. Rev. B 17, 1302 (1978). Implementation remains unchanged.]

**(Dunweg)** Dunweg and Paul, Int J of Modern Physics C, 2, 817-27 (1991).

## 2.105 fix langevin/drude command

### 2.105.1 Syntax

```
fix ID group-ID langevin/drude Tcom damp_com seed_com Tdrude damp_drude seed_drude␣
↪keyword values ...
```

- ID, group-ID are documented in *fix* command
- langevin/drude = style name of this fix command
- Tcom = desired temperature of the centers of mass (temperature units)
- damp_com = damping parameter for the thermostat on centers of mass (time units)
- seed_com = random number seed to use for white noise of the thermostat on centers of mass (positive integer)
- Tdrude = desired temperature of the Drude oscillators (temperature units)
- damp_drude = damping parameter for the thermostat on Drude oscillators (time units)
- seed_drude = random number seed to use for white noise of the thermostat on Drude oscillators (positive integer)
- zero or more keyword/value pairs may be appended
- keyword = *zero*

```
zero value = no or yes
  no = do not set total random force on centers of mass to zero
  yes = set total random force on centers of mass to zero
```

### 2.105.2 Examples

```
fix 3 all langevin/drude 300.0 100.0 19377 1.0 20.0 83451
fix 1 all langevin/drude 298.15 100.0 19377 5.0 10.0 83451 zero yes
```

Example input scripts available: examples/PACKAGES/drude

### 2.105.3 Description

Apply two Langevin thermostats as described in *(Jiang1)* for thermalizing the reduced degrees of freedom of Drude oscillators. This link describes how to use the *thermalized Drude oscillator model* in LAMMPS and polarizable models in LAMMPS are discussed on the *Howto polarizable* doc page.

Drude oscillators are a way to simulate polarizables atoms, by splitting them into a core and a Drude particle bound by a harmonic bond. The thermalization works by transforming the particles degrees of freedom by these equations. In these equations upper case denotes atomic or center of mass values and lower case denotes Drude particle or dipole values. Primes denote the transformed (reduced) values, while bare letters denote the original values.

Velocities:

$$V' = \frac{M\,V + m\,v}{M'}$$

$$v' = v - V$$

Masses:

$$M' = M + m$$

$$m' = \frac{M\,m}{M'}$$

The Langevin forces are computed as

$$F' = -\frac{M'}{\mathrm{damp_c om}}\,V' + F_r'$$

$$f' = -\frac{m'}{\mathrm{damp_d rude}}\,v' + f_r'$$

$F_r'$ is a random force proportional to $\sqrt{\frac{2\,k_B \mathrm{T com}\,m'}{\mathrm{d}t\ \mathrm{damp_c om}}}$. $f_r'$ is a random force proportional to $\sqrt{\frac{2\,k_B \mathrm{T drude}\,m'}{\mathrm{d}t\ \mathrm{damp_d rude}}}$. Then the real forces acting on the particles are computed from the inverse transform:

$$F = \frac{M}{M'}\,F' - f'$$

$$f = \frac{m}{M'}\,F' + f'$$

This fix also thermostats non-polarizable atoms in the group at temperature *Tcom*, as if they had a massless Drude partner. The Drude particles themselves need not be in the group. The center of mass and the dipole are thermostatted iff the core atom is in the group.

Note that the thermostat effect of this fix is applied to only the translational degrees of freedom of the particles, which is an important consideration if finite-size particles, which have rotational degrees of freedom, are being thermostatted. The translational degrees of freedom can also have a bias velocity removed from them before thermostatting takes place; see the description below.

**Note**

Like the *fix langevin* command, this fix does NOT perform time integration. It only modifies forces to effect thermostatting. Thus you must use a separate time integration fix, like *fix nve* or *fix nph* to actually update the velocities and positions of atoms using the modified forces. Likewise, this fix should not normally be used on atoms that also have their temperature controlled by another fix - e.g. by *fix nvt* or *fix temp/rescale* commands.

See the *Howto thermostat* page for a discussion of different ways to compute temperature and perform thermostatting.

---

This fix requires each atom know whether it is a Drude particle or not. You must therefore use the *fix drude* command to specify the Drude status of each atom type.

**Note**

only the Drude core atoms need to be in the group specified for this fix. A Drude electron will be transformed together with its cores even if it is not itself in the group. It is safe to include Drude electrons or non-polarizable atoms in the group. The non-polarizable atoms will simply be thermostatted as if they had a massless Drude partner (electron).

**Note**

Ghost atoms need to know their velocity for this fix to act correctly. You must use the *comm_modify* command to enable this, e.g.

```
comm_modify vel yes
```

---

*Tcom* is the target temperature of the centers of mass, which would be used to thermostat the non-polarizable atoms. *Tdrude* is the (normally low) target temperature of the core-Drude particle pairs (dipoles). *Tcom* and *Tdrude* can be specified as an equal-style *variable*. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated each timestep, and its value used to determine the target temperature.

Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent temperature.

Like other fixes that perform thermostatting, this fix can be used with *compute commands* that remove a "bias" from the atom velocities. E.g. to apply the thermostat only to atoms within a spatial *region*, or to remove the center-of-mass velocity from a group of atoms, or to remove the x-component of velocity from the calculation.

This is not done by default, but only if the *fix_modify* command is used to assign a temperature compute to this fix that includes such a bias term. See the doc pages for individual *compute temp commands* to determine which ones include a bias. In this case, the thermostat works in the following manner: bias is removed from each atom, thermostatting is performed on the remaining thermal degrees of freedom, and the bias is added back in.

Note: The temperature thermostatting the core-Drude particle pairs should be chosen low enough, so as to mimic as closely as possible the self-consistent minimization. It must however be high enough, so that the dipoles can follow the local electric field exerted by the neighboring atoms. The optimal value probably depends on the temperature of the centers of mass and on the mass of the Drude particles.

*damp_com* is the characteristic time for reaching thermal equilibrium of the centers of mass. For example, a value of 100.0 means to relax the temperature of the centers of mass in a timespan of (roughly) 100 time units (tau or fs or ps - see the *units* command). *damp_drude* is the characteristic time for reaching thermal equilibrium of the dipoles. It is typically a few timesteps.

The number *seed_com* and *seed_drude* are positive integers. They set the seeds of the Marsaglia random number generators used for generating the random forces on centers of mass and on the dipoles. Each processor uses the input seed to generate its own unique seed and its own stream of random numbers. Thus the dynamics of the system will not be identical on two runs on different numbers of processors.

The keyword *zero* can be used to eliminate drift due to the thermostat on centers of mass. Because the random forces on different centers of mass are independent, they do not sum exactly to zero. As a result, this fix applies a small random force to the entire system, and the momentum of the total center of mass of the system undergoes a slow random walk. If the keyword *zero* is set to *yes*, the total random force on the centers of mass is set exactly to zero by subtracting off an equal part of it from each center of mass in the group. As a result, the total center of mass of a system with zero initial momentum will not drift over time.

The actual temperatures of cores and Drude particles, in center-of-mass and relative coordinates, respectively, can be calculated using the *compute temp/drude* command.

---

Usage example for rigid bodies in the NPT ensemble:

```
comm_modify vel yes
fix TEMP all langevin/drude 300. 100. 1256 1. 20. 13977 zero yes
fix NPH ATOMS rigid/nph/small molecule iso 1. 1. 500.
fix NVE DRUDES nve
compute TDRUDE all temp/drude
thermo_style custom step cpu etotal ke pe ebond ecoul elong press vol temp c_TDRUDE[1] c_
↪TDRUDE[2]
```

Comments:

- Drude particles should not be in the rigid group, otherwise the Drude oscillators will be frozen and the system will lose its polarizability.
- *zero yes* avoids a drift of the center of mass of the system, but is a bit slower.
- Use two different random seeds to avoid unphysical correlations.
- Temperature is controlled by the fix *langevin/drude*, so the time-integration fixes do not thermostat. Don't forget to time-integrate both cores and Drude particles.
- Pressure is time-integrated only once by using *nve* for Drude particles and *nph* for atoms/cores (or vice versa). Do not use *nph* for both.
- The temperatures of cores and Drude particles are calculated by *compute temp/drude*
- Contrary to the alternative thermostatting using Nose-Hoover thermostat fix *npt* and *fix drude/transform*, the *fix_modify* command is not required here, because the fix *nph* computes the global pressure even if its group is *ATOMS*. This is what we want. If we thermostatted *ATOMS* using *npt*, the pressure should be the global one, but the temperature should be only that of the cores. That's why the command *fix_modify* should be called in that case.

---

### 2.105.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. Because the state of the random number generator is not saved in restart files, this means you cannot do "exact" restarts with this fix, where the simulation continues on the same as if no restart had taken place. However, in a statistical sense, a restarted simulation should produce the same behavior.

The *fix_modify* *temp* option is supported by this fix. You can use it to assign a temperature *compute* you have defined to this fix which will be used in its thermostatting procedure, as described above. For consistency, the group used by the compute should include the group of this fix and the Drude particles.

This fix is not invoked during *energy minimization*.

### 2.105.5 Restrictions

none

### 2.105.6 Related commands

*fix langevin*, *fix drude*, *fix drude/transform*, *compute temp/drude*, *pair_style thole*

### 2.105.7 Default

The option defaults are zero = no.

---

**(Jiang1)** Jiang, Hardy, Phillips, MacKerell, Schulten, and Roux, J Phys Chem Lett, 2, 87-92 (2011).

## 2.106 fix langevin/eff command

### 2.106.1 Syntax

```
fix ID group-ID langevin/eff Tstart Tstop damp seed keyword values ...
```

- ID, group-ID are documented in *fix* command
- langevin/eff = style name of this fix command
- Tstart,Tstop = desired temperature at start/end of run (temperature units)
- damp = damping parameter (time units)
- seed = random number seed to use for white noise (positive integer)
- zero or more keyword/value pairs may be appended

```
keyword = scale or tally or zero
  scale values = type ratio
    type = atom type (1-N)
    ratio = factor by which to scale the damping coefficient
  tally values = no or yes
    no = do not tally the energy added/subtracted to atoms
    yes = do tally the energy added/subtracted to atoms

zero value = no or yes
  no = do not set total random force to zero
  yes = set total random force to zero
```

### 2.106.2 Examples

```
fix 3 boundary langevin/eff 1.0 1.0 10.0 699483
fix 1 all langevin/eff 1.0 1.1 10.0 48279 scale 3 1.5
```

### 2.106.3 Description

Apply a Langevin thermostat as described in *(Schneider)* to a group of nuclei and electrons in the *electron force field* model. Used with *fix nve/eff*, this command performs Brownian dynamics (BD), since the total force on each atom will have the form:

$$
\begin{aligned}
F &= F_c + F_f + F_r \\
F_f &= -\frac{m}{\text{damp}} v \\
F_r &\propto \sqrt{\frac{k_B T m}{dt\ \text{damp}}}
\end{aligned}
$$

$F_c$ is the conservative force computed via the usual inter-particle interactions (*pair_style*). The $F_f$ and $F_r$ terms are added by this fix on a per-particle basis.

The operation of this fix is exactly like that described by the *fix langevin* command, except that the thermostatting is also applied to the radial electron velocity for electron particles.

### 2.106.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. Because the state of the random number generator is not saved in restart files, this means you cannot do "exact" restarts with this fix, where the simulation continues on the same as if no restart had taken place. However, in a statistical sense, a restarted simulation should produce the same behavior.

The *fix_modify* *temp* option is supported by this fix. You can use it to assign a temperature *compute* you have defined to this fix which will be used in its thermostatting procedure, as described above. For consistency, the group used by this fix and by the compute should be the same.

The cumulative energy change in the system imposed by this fix is included in the *thermodynamic output* keywords *ecouple* and *econserve*, but only if the *tally* keyword to set to *yes*. See the *thermo_style* page for details.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the same cumulative energy change due to this fix described in the previous paragraph. The scalar value calculated by this fix is "extensive". Note that calculation of this quantity also requires setting the *tally* keyword to *yes*.

This fix can ramp its target temperature over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.106.5 Restrictions

none

This fix is part of the EFF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.106.6 Related commands

*fix langevin*

### 2.106.7 Default

The option defaults are scale = 1.0 for all types and tally = no.

---

**(Dunweg)** Dunweg and Paul, Int J of Modern Physics C, 2, 817-27 (1991).

**(Schneider)** Schneider and Stoll, Phys Rev B, 17, 1302 (1978).

## 2.107 fix langevin/spin command

### 2.107.1 Syntax

```
fix ID group-ID langevin/spin T Tdamp seed
```

- ID, group-ID are documented in *fix* command
- langevin/spin = style name of this fix command
- T = desired temperature of the bath (temperature units, K in metal units)
- Tdamp = transverse magnetic damping parameter (adim)
- seed = random number seed to use for white noise (positive integer)

### 2.107.2 Examples

```
fix 2 all langevin/spin 300.0 0.01 21
```

### 2.107.3 Description

Apply a Langevin thermostat as described in *(Mayergoyz)* to the magnetic spins associated to the atoms. Used with *fix nve/spin*, this command performs Brownian dynamics (BD). A random torque and a transverse dissipation are applied to each spin i according to the following stochastic differential equation:

$$\frac{d\vec{s}_i}{dt} = \frac{1}{\left(1+\lambda^2\right)}\left(\left(\vec{\omega}_i + \vec{\eta}\right) \times \vec{s}_i + \lambda\, \vec{s}_i \times \left(\vec{\omega}_i \times \vec{s}_i\right)\right)$$

with $\lambda$ the transverse damping, and $\eta$ a random vector. This equation is referred to as the stochastic Landau-Lifshitz (sLL) equation.

The components of $\eta$ are drawn from a Gaussian probability law. Their amplitude is defined as a proportion of the temperature of the external thermostat T (in K in metal units).

More details about this implementation are reported in *(Tranchida)*.

Note: due to the form of the sLL equation, this fix has to be defined just before the nve/spin fix (and after all other magnetic fixes). As an example:

```
fix 1 all precession/spin zeeman 0.01 0.0 0.0 1.0
fix 2 all langevin/spin 300.0 0.01 21
fix 3 all nve/spin lattice moving
```

is correct, but defining a force/spin command after the langevin/spin command would give an error message.

Note: The random # *seed* must be a positive integer. A Marsaglia random number generator is used. Each processor uses the input seed to generate its own unique seed and its own stream of random numbers. Thus the dynamics of the system will not be identical on two runs on different numbers of processors.

---

### 2.107.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. Because the state of the random number generator is not saved in restart files, this means you cannot do "exact" restarts with this fix, where the simulation continues on the same as if no restart had taken place. However, in a statistical sense, a restarted simulation should produce the same behavior.

This fix is not invoked during *energy minimization*.

### 2.107.5 Restrictions

The *langevin/spin* fix is part of the SPIN package. This style is only enabled if LAMMPS was built with this package. See the *Build package* page for more info.

The numerical integration has to be performed with *fix nve/spin* when *fix langevin/spin* is enabled.

This fix has to be the last defined magnetic fix before the time integration fix (e.g. *fix nve/spin*).

### 2.107.6 Related commands

*fix nve/spin*, *fix precession/spin*

### 2.107.7 Default

none

---

**(Mayergoyz)** I.D. Mayergoyz, G. Bertotti, C. Serpico (2009). Elsevier (2009)

**(Tranchida)** Tranchida, Plimpton, Thibaudeau and Thompson, Journal of Computational Physics, 372, 406-425, (2018).

## 2.108 fix lb/fluid command

### 2.108.1 Syntax

```
fix ID group-ID lb/fluid nevery viscosity density keyword values ...
```

- ID, group-ID are documented in *fix* command
- lb/fluid = style name of this fix command
- nevery = update the lattice-Boltzmann fluid every this many timesteps (should normally be 1)
- viscosity = the fluid viscosity (units of mass/(time*length)).
- density = the fluid density.
- zero or more keyword/value pairs may be appended
- keyword = *dx* or *dm* or *noise* or *stencil* or *read_restart* or *write_restart* or *zwall_velocity* or *pressurebcx* or *bodyforce* or *D3Q19* or *dumpxdmf* or *linearInit* or *dof* or *scaleGamma* or *a0* or *npits* or *wp* or *sw*

```
dx values = dx_LB = the lattice spacing.
dm values = dm_LB = the lattice-Boltzmann mass unit.
noise values = Temperature seed
    Temperature = fluid temperature.
    seed = random number generator seed (positive integer)
stencil values = 2 (trilinear stencil, the default), 3 (3-point immersed boundary␣
↪stencil), or 4 (4-point Keys' interpolation stencil)
read_restart values = restart file = name of the restart file to use to restart a␣
↪fluid run.
write_restart values = N = write a restart file every N MD timesteps.
zwall_velocity values = velocity_bottom velocity_top = velocities along the␣
↪y-direction of the bottom and top walls (located at z=zmin and z=zmax).
pressurebcx values = pgradav = imposes a pressure jump at the (periodic) x-boundary␣
↪of pgradav*Lx*1000.
bodyforce values = bodyforcex bodyforcey bodyforcez = the x,y and z components of a␣
↪constant body force added to the fluid.
D3Q19 values = none (used to switch from the default D3Q15, 15 velocity lattice, to␣
↪the D3Q19, 19 velocity lattice).
dumpxdmf values = N file timeI
    N = output the force and torque every N timesteps
    file = output file name
    timeI = 1 (use simulation time to index xdmf file), 0 (use output frame number␣
↪to index xdmf file)
linearInit values = none = initialize density and velocity using linear␣
↪interpolation (default is uniform density, no velocities)
dof values = dof = specify the number of degrees of freedom for temperature␣
↪calculation
scaleGamma values = type gammaFactor
    type = atom type (1-N)
    gammaFactor = factor to scale the setGamma gamma value by, for the specified␣
↪atom type.
a0 values = a_0_real = the square of the speed of sound in the fluid.
npits values = npits h_p l_p l_pp l_e
    npits = number of pit regions
    h_p = z-height of pit regions (floor to bottom of slit)
    l_p = x-length of pit regions
    l_pp = x-length of slit regions between consecutive pits
    l_e = x-length of slit regions at ends
wp values = w_p = y-width of slit regions (defaults to full width if not present or␣
↪if sw active)
sw values = none  (turns on y-sidewalls (in xz plane) if npits option active)
```

### 2.108.2 Examples

```
fix 1 all lb/fluid 1 1.0 0.0009982071 dx 1.2 dm 0.001
fix 1 all lb/fluid 1 1.0 0.0009982071 dx 1.2 dm 0.001 noise 300.0 2761
fix 1 all lb/fluid 1 1.0 1.0 dx 4.0 dm 10.0 dumpxdmf 500 fflow 0 pressurebcx 0.01 npits␣
↪2 20 40 5 0 wp 30
```

## 2.108.3 Description

Changed in version 24Mar2022.

Implement a lattice-Boltzmann fluid on a uniform mesh covering the LAMMPS simulation domain. Note that this fix was updated in 2022 and is not backward compatible with the previous version. If you need the previous version, please download an older version of LAMMPS. The MD particles described by *group-ID* apply a velocity dependent force to the fluid.

The lattice-Boltzmann algorithm solves for the fluid motion governed by the Navier Stokes equations,

$$\begin{aligned}\partial_t \rho + \partial_\beta \left(\rho u_\beta\right) =& 0 \\ \partial_t \left(\rho u_\alpha\right) + \partial_\beta \left(\rho u_\alpha u_\beta\right) =& \partial_\beta \sigma_{\alpha\beta} + F_\alpha + \partial_\beta \left(\eta_{\alpha\beta\gamma\nu}\partial_\gamma u_\nu\right)\end{aligned}$$

with,

$$\eta_{\alpha\beta\gamma\nu} = \eta\left[\delta_{\alpha\gamma}\delta_{\beta\nu} + \delta_{\alpha\nu}\delta_{\beta\gamma} - \frac{2}{3}\delta_{\alpha\beta}\delta_{\gamma\nu}\right] + \Lambda\delta_{\alpha\beta}\delta_{\gamma\nu}$$

where $\rho$ is the fluid density, $u$ is the local fluid velocity, $\sigma$ is the stress tensor, $F$ is a local external force, and $\eta$ and $\Lambda$ are the shear and bulk viscosities respectively. Here, we have implemented

$$\sigma_{\alpha\beta} = -P_{\alpha\beta} = -\rho a_0 \delta_{\alpha\beta}$$

with $a_0$ set to $\frac{1}{3}\frac{dx}{dt}^2$ by default. You should not normally need to change this default.

The algorithm involves tracking the time evolution of a set of partial distribution functions which evolve according to a velocity discretized version of the Boltzmann equation,

$$\left(\partial_t + e_{i\alpha}\partial_\alpha\right) f_i = -\frac{1}{\tau}\left(f_i - f_i^{eq}\right) + W_i$$

where the first term on the right hand side represents a single time relaxation towards the equilibrium distribution function, and $\tau$ is a parameter physically related to the viscosity. On a technical note, we have implemented a 15 velocity model (D3Q15) as default; however, the user can switch to a 19 velocity model (D3Q19) through the use of the *D3Q19* keyword. Physical variables are then defined in terms of moments of the distribution functions,

$$\begin{aligned}\rho =& \sum_i f_i \\ \rho u_\alpha =& \sum_i f_i e_{i\alpha}\end{aligned}$$

Full details of the lattice-Boltzmann algorithm used can be found in *Denniston et al.*.

The fluid is coupled to the MD particles described by *group-ID* through a velocity dependent force. The contribution to the fluid force on a given lattice mesh site j due to MD particle $\alpha$ is calculated as:

$$\mathbf{F}_{j\alpha} = \gamma\left(\mathbf{v}_n - \mathbf{u}_f\right)\zeta_{j\alpha}$$

where $\mathbf{v}_n$ is the velocity of the MD particle, $\mathbf{u}_f$ is the fluid velocity interpolated to the particle location, and $\gamma$ is the force coupling constant. This force, as with most forces in LAMMPS, and hence the velocities, are calculated at the half-time step. $\zeta$ is a weight assigned to the grid point, obtained by distributing the particle to the nearest lattice sites.

The force coupling constant, $\gamma$, is calculated according to

$$\gamma = \frac{2m_u m_v}{m_u + m_v}\left(\frac{1}{\Delta t}\right)$$

Here, $m_v$ is the mass of the MD particle, $m_u$ is a representative fluid mass at the particle location, and $\Delta t$ is the time step. The fluid mass $m_u$ that the MD particle interacts with is calculated internally. This coupling is chosen to constrain the particle and associated fluid velocity to match at the end of the time step. As with other constraints, such as *shake*, this constraint can remove degrees of freedom from the simulation which are accounted for internally in the algorithm.

**Note**

While this fix applies the force of the particles on the fluid, it does not apply the force of the fluid to the particles. There is only one option to include this hydrodynamic force on the particles, and that is through the use of the *lb/viscous* fix. This fix adds the hydrodynamic force to the total force acting on the particles, after which any of the built-in LAMMPS integrators can be used to integrate the particle motion. If the *lb/viscous* fix is NOT used to add the hydrodynamic force to the total force acting on the particles, this physically corresponds to a situation in which an infinitely massive particle is moving through the fluid (since collisions between the particle and the fluid do not act to change the particle's velocity). In this case, setting *scaleGamma* to -1 for the corresponding particle type will explicitly take this limit (of infinite particle mass) in computing the force coupling for the fluid force.

---

Physical parameters describing the fluid are specified through *viscosity* and *density*. These parameters should all be given in terms of the mass, distance, and time units chosen for the main LAMMPS run, as they are scaled by the LB timestep, lattice spacing, and mass unit, inside the fix.

The *dx* keyword allows the user to specify a value for the LB grid spacing and the *dm* keyword allows the user to specify the LB mass unit. Inside the fix, parameters are scaled by the lattice-Boltzmann timestep, $dt_{LB}$, grid spacing, $dx_{LB}$, and mass unit, $dm_{LB}$. $dt_{LB}$ is set equal to nevery $\cdot$ $dt_{MD}$, where $dt_{MD}$ is the MD timestep. By default, $dm_{LB}$ is set equal to 1.0, and $dx_{LB}$ is chosen so that $\frac{\tau}{dt} = \frac{3\eta dt}{\rho dx^2}$ is approximately equal to 1.

**Note**

Care must be taken when choosing both a value for $dx_{LB}$, and a simulation domain size. This fix uses the same subdivision of the simulation domain among processors as the main LAMMPS program. In order to uniformly cover the simulation domain with lattice sites, the lengths of the individual LAMMPS subdomains must all be evenly divisible by $dx_{LB}$. If the simulation domain size is cubic, with equal lengths in all dimensions, and the default value for $dx_{LB}$ is used, this will automatically be satisfied.

If the *noise* keyword is used, followed by a positive temperature value, and a positive integer random number seed, the thermal LB algorithm of *Adhikari et al.* is used.

If the keyword *stencil* is used, the value sets the number of interpolation points used in each direction. For this, the user has the choice between a trilinear stencil (*stencil* 2), which provides a support of 8 lattice sites, or the 3-point immersed boundary method stencil (*stencil* 3), which provides a support of 27 lattice sites, or the 4-point Keys' interpolation stencil (stencil 4), which provides a support of 64 lattice sites. The trilinear stencil is the default as it is better suited for simulation of objects close to walls or other objects, due to its smaller support. The 3-point stencil provides smoother motion of the lattice and is suitable for particles not likely to be to close to walls or other objects.

If the keyword *write_restart* is used, followed by a positive integer, N, a binary restart file is printed every N LB timesteps. This restart file only contains information about the fluid. Therefore, a LAMMPS restart file should also be written in order to print out full details of the simulation.

**Note**

When a large number of lattice grid points are used, the restart files may become quite large.

In order to restart the fluid portion of the simulation, the keyword *read_restart* is specified, followed by the name of the binary lb_fluid restart file to be used.

If the *zwall_velocity* keyword is used y-velocities are assigned to the lower and upper walls. This keyword requires the presence of walls in the z-direction. This is set by assigning fixed boundary conditions in the z-direction. If fixed boundary conditions are present in the z-direction, and this keyword is not used, the walls are assumed to be stationary.

If the *pressurebcx* keyword is used, a pressure jump (implemented by a step jump in density) is imposed at the (periodic) x-boundary. The value set specifies what would be the resulting equilibrium average pressure gradient in the x-direction if the system had a constant cross-section (i.e. resistance to flow). It is converted to a pressure jump by multiplication by the system size in the x-direction. As this value should normally be quite small, it is also assumed to be scaled by 1000.

If the *bodyforce* keyword is used, a constant body force is added to the fluid, defined by it's x, y and z components.

If the keyword *D3Q19* is used, the 19 velocity (D3Q19) lattice is used by the lattice-Boltzmann algorithm. By default, the 15 velocity (D3Q15) lattice is used.

If the *dumpxdmf* keyword is used, followed by a positive integer, N, and a file name, the fluid densities and velocities at each lattice site are output to an xdmf file every N timesteps. This is a binary file format that can be read by visualization packages such as Paraview . The xdmf file format contains a time index for each frame dump and the value timeI = 1 uses simulation time while 0 uses the output frame number to index xdmf file. The later can be useful if the *dump vtk* command is used to output the particle positions at the same timesteps and you want to visualize both the fluid and particle data together in Paraview .

The *scaleGamma* keyword allows the user to scale the $\gamma$ value by a factor, gammaFactor, for a given atom type. Setting *scaleGamma* to -1 for the corresponding particle type will explicitly take the limit of infinite particle mass in computing the force coupling for the fluid force (see note above).

If the *a0* keyword is used, the value specified is used for the square of the speed of sound in the fluid. If this keyword is not present, the speed of sound squared is set equal to $\frac{1}{3}\left(\frac{dx_{LB}}{dt_{LB}}\right)^2$. Setting $a0 > (\frac{dx_{LB}}{dt_{LB}})^2$ is not allowed, as this may lead to instabilities. As the speed of sound should usually be much larger than any fluid velocity of interest, its value does not normally have a significant impact on the results. As such, it is usually best to use the default for this option.

The *npits* keyword (followed by integer arguments: npits, h_p, l_p, l_pp, l_e) sets the fluid domain to the pits geometry. These arguments should only be used if you actually want something more complex than a rectangular/cubic geometry. The npits value sets the number of pits regions (arranged along x). The remaining arguments are sizes measured in multiples of dx_lb: h_p is the z-height of the pit regions, l_p is the x-length of the pit regions, l_pp is the length of the region between consecutive pits (referred to as a "slit" region), and l_e is the x-length of the slit regions at each end of the channel. The pit geometry must fill the system in the x-direction but can be longer, in which case it is truncated (which enables asymmetric entrance/exit end sections). The additional *wp* keyword allows the width (in y-direction) of the pit to be specified (the default is full width) and the *sw* keyword indicates that there should be sidewalls in the y-direction (default is periodic in y-direction). These parameters are illustrated below:

```
Sideview (in xz plane) of pit geometry:
_____________________________________________________________________________
  slit                          slit                            slit     ^
                                                                         |
<---le---><---------lp-------><---lpp---><-------lp--------><---le---> hs = (Nbz-1) - hp
                                                                         |
__________                   __________                    __________ v
          |                 |          |                  |          ^       z
          |                 |          |                  |          |       |
          |       pit       |          |       pit        |          hp      +-x
          |                 |          |                  |          |
          |_________________|          |__________________|          v

Endview (in yz plane) of pit geometry (no sw so wp is active):
______________________
                      ^
                      |
                      hs
                      |
```

(continues on next page)

(continued from previous page)

```
_____________________ v
    |           |     ^
    |           |     |          z
    |<---wp--->|     hp          |
    |           |     |          +-y
    |___________|     v
```

For further details, as well as descriptions and results of several test runs, see *Denniston et al.*. Please include a citation to this paper if the lb_fluid fix is used in work contributing to published research.

### 2.108.4 Restart, fix_modify, output, run start/stop, minimize info

Due to the large size of the fluid data, this fix writes it's own binary restart files, if requested, independent of the main LAMMPS *binary restart files*; no information about *lb_fluid* is written to the main LAMMPS *binary restart files*.

None of the *fix_modify* options are relevant to this fix.

The fix computes a global scalar which can be accessed by various *output commands*. The scalar is the current temperature of the group of particles described by *group-ID* along with the fluid constrained to move with them. The temperature is computed via the kinetic energy of the group and fluid constrained to move with them and the total number of degrees of freedom (calculated internally). If the particles are not integrated independently (such as via *fix NVE*) but have additional constraints imposed on them (such as via integration using *fix rigid*) the degrees of freedom removed from these additional constraints will not be properly accounted for. In this case, the user can specify the total degrees of freedom independently using the *dof* keyword.

The fix also computes a global array of values which can be accessed by various *output commands*. There are 5 entries in the array. The first entry is the temperature of the fluid, the second entry is the total mass of the fluid plus particles, the third through fifth entries give the x, y, and z total momentum of the fluid plus particles.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.108.5 Restrictions

This fix is part of the LATBOLTZ package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix can only be used with an orthogonal simulation domain.

The boundary conditions for the fluid are specified independently to the particles. However, these should normally be specified consistently via the main LAMMPS *boundary* command (p p p, p p f, and p f f are the only consistent possibilities). Shrink-wrapped boundary conditions are not permitted with this fix.

This fix must be used before any of *fix lb/viscous* and *fix lb/momentum* as the fluid needs to be initialized before any of these routines try to access its properties. In addition, in order for the hydrodynamic forces to be added to the particles, this fix must be used in conjunction with the *lb/viscous* fix.

This fix needs to be used in conjunction with a standard LAMMPS integrator such as *fix NVE* or *fix rigid*.

### 2.108.6 Related commands

*fix lb/viscous*, *fix lb/momentum*

### 2.108.7 Default

$dx$ is chosen such that $\frac{\tau}{dt_{LB}} = \frac{3\eta dt_{LB}}{\rho dx_{LB}^2}$ is approximately equal to 1. $dm$ is set equal to 1.0. $a0$ is set equal to $\frac{1}{3}\left(\frac{dx_{LB}}{dt_{LB}}\right)^2$. The trilinear stencil is used as the default interpolation method. The D3Q15 lattice is used for the lattice-Boltzmann algorithm.

---

**(Denniston et al.)** Denniston, C., Afrasiabian, N., Cole-Andre, M.G., Mackay, F. E., Ollila, S.T.T., and Whitehead, T., LAMMPS lb/fluid fix version 2: Improved Hydrodynamic Forces Implemented into LAMMPS through a lattice-Boltzmann fluid, Computer Physics Communications 275 (2022) 108318 .

**(Mackay and Denniston)** Mackay, F. E., and Denniston, C., Coupling MD particles to a lattice-Boltzmann fluid through the use of conservative forces, J. Comput. Phys. 237 (2013) 289-298.

**(Adhikari et al.)** Adhikari, R., Stratford, K., Cates, M. E., and Wagner, A. J., Fluctuating lattice Boltzmann, Europhys. Lett. 71 (2005) 473-479.

## 2.109 fix lb/momentum command

### 2.109.1 Syntax

```
fix ID group-ID lb/momentum nevery keyword values ...
```

- ID, group-ID are documented in the *fix* command
- lb/momentum = style name of this fix command
- nevery = adjust the momentum every this many timesteps
- zero or more keyword/value pairs may be appended
- keyword = *linear*

  *linear* values = xflag yflag zflag
    xflag,yflag,zflag = 0/1 to exclude/include each dimension.

### 2.109.2 Examples

```
fix 1 sphere lb/momentum
fix 1 all lb/momentum linear 1 1 0
```

### 2.109.3 Description

This fix is based on the *fix momentum* command, and was created to be used in place of that command, when a lattice-Boltzmann fluid is present.

Zero the total linear momentum of the system, including both the atoms specified by group-ID and the lattice-Boltzmann fluid every nevery timesteps. If there are no atoms specified by group-ID only the fluid momentum is affected. This is accomplished by adjusting the particle velocities and the fluid velocities at each lattice site.

> **Note**
>
> This fix only considers the linear momentum of the system.

By default, the subtraction is performed for each dimension. This can be changed by specifying the keyword *linear*, along with a set of three flags set to 0/1 in order to exclude/ include the corresponding dimension.

### 2.109.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.109.5 Restrictions

Can only be used if a lattice-Boltzmann fluid has been created via the *fix lb/fluid* command, and must come after this command.

This fix is part of the LATBOLTZ package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.109.6 Related commands

*fix momentum*, *fix lb/fluid*

### 2.109.7 Default

Zeros the total system linear momentum in each dimension.

## 2.110 fix lb/viscous command

### 2.110.1 Syntax

```
fix ID group-ID lb/viscous
```

- ID, group-ID are documented in *fix* command
- lb/viscous = style name of this fix command

### 2.110.2 Examples

```
fix 1 flow lb/viscous
```

### 2.110.3 Description

This fix is similar to the *fix viscous* command, and is to be used in place of that command when a lattice-Boltzmann fluid is present using the *fix lb/fluid*. This should be used in conjunction with one of the built-in LAMMPS integrators, such as *fix NVE* or *fix rigid*.

This fix adds a viscous force to each atom to cause it move with the same velocity as the fluid (an equal and opposite force is applied to the fluid via *fix lb/fluid*). When *fix lb/fluid* is called with the noise option, the atoms will also experience random forces which will thermalize them to the same temperature as the fluid. In this way, the combination of this fix with *fix lb/fluid* and a LAMMPS integrator like *fix NVE* is analogous to *fix langevin* except here the fluid is explicit. The temperature of the particles can be monitored via the scalar output of *fix lb/fluid*.

---

For details of this fix, as well as descriptions and results of several test runs, see *Denniston et al.*. Please include a citation to this paper if this fix is used in work contributing to published research.

---

### 2.110.4 Restart, fix_modify, output, run start/stop, minimize info

As described in the *fix viscous* documentation:

"No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The forces due to this fix are imposed during an energy minimization, invoked by the *minimize* command. This fix should only be used with damped dynamics minimizers that allow for non-conservative forces. See the *min_style* command for details."

### 2.110.5 Restrictions

This fix is part of the LATBOLTZ package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Can only be used if a lattice-Boltzmann fluid has been created via the *fix lb/fluid* command, and must come after this command.

### 2.110.6 Related commands

*fix lb/fluid*

### 2.110.7 Default

none

---

**(Denniston et al.)** Denniston, C., Afrasiabian, N., Cole-Andre, M.G., Mackay, F. E., Ollila, S.T.T., and Whitehead, T., LAMMPS lb/fluid fix version 2: Improved Hydrodynamic Forces Implemented into LAMMPS through a lattice-Boltzmann fluid, Computer Physics Communications 275 (2022) 108318 .

## 2.111 fix lineforce command

Accelerator Variant: *lineforce/kk*

### 2.111.1 Syntax

```
fix ID group-ID lineforce x y z
```

- ID, group-ID are documented in *fix* command
- lineforce = style name of this fix command
- x y z = direction of line as a 3-vector

### 2.111.2 Examples

```
fix hold boundary lineforce 0.0 1.0 1.0
```

### 2.111.3 Description

Adjust the forces on each atom in the group so that only the component of force along the linear direction specified by the vector (x,y,z) remains. This is done by subtracting out components of force in the plane perpendicular to the line.

If the initial velocity of the atom is 0.0 (or along the line), then it should continue to move along the line thereafter.

### 2.111.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The forces due to this fix are imposed during an energy minimization, invoked by the *minimize* command.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.111.5 Restrictions

none

### 2.111.6 Related commands

*fix planeforce*

### 2.111.7 Default

none

## 2.112 fix manifoldforce command

### 2.112.1 Syntax

```
fix ID group-ID manifoldforce manifold manifold-args ...
```

- ID, group-ID are documented in *fix* command
- manifold = name of the manifold
- manifold-args = parameters for the manifold

### 2.112.2 Examples

```
fix constrain all manifoldforce sphere 5.0
```

### 2.112.3 Description

This fix subtracts each time step from the force the component along the normal of the specified *manifold*. This can be used in combination with *minimize* to remove overlap between particles while keeping them (roughly) constrained to the given manifold, e.g. to set up a run with *fix nve/manifold/rattle*. I have found that only *hftn* and *quickmin* with a very small time step perform adequately though.

---

### 2.112.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is invoked during *energy minimization*.

---

### 2.112.5 Restrictions

This fix is part of the MANIFOLD package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Only use this with *min_style hftn* or *min_style quickmin*. If not, the constraints will not be satisfied very well at all. A warning is generated if the *min_style* is incompatible but no error.

---

### 2.112.6 Related commands

*fix nve/manifold/rattle*, *fix nvt/manifold/rattle*

## 2.113 fix mdi/qm command

### 2.113.1 Syntax

```
fix ID group-ID mdi/qm keyword value(s) keyword value(s) ...
```

- ID, group-ID are documented in *fix* command
- mdi/qm = style name of this fix command
- zero or more keyword/value pairs may be appended
- keyword = *virial* or *add* or *every* or *connect* or *elements* or *mc*

```
virial args = yes or no
  yes = request virial tensor from server code
  no = do not request virial tensor from server code
add args = yes or no
  yes = add returned value from server code to LAMMPS quantities
  no = do not add returned values to LAMMPS quantities
every args = Nevery
  Nevery = request values from server code once every Nevery steps
connect args = yes or no
  yes = perform a one-time connection to the MDI engine code
  no = do not perform the connection operation
elements args = N_1 N_2 ... N_ntypes
```

```
  N_1,N_2,...N_ntypes = chemical symbol for each of ntypes LAMMPS atom types
mc args = mcfixID
  mcfixID = ID of a Monte Carlo fix designed to work with this fix
```

### 2.113.2 Examples

```
fix 1 all mdi/qm
fix 1 all mdi/qm virial yes
fix 1 all mdi/qm add no every 100 elements C C H O
```

### 2.113.3 Description

Added in version 3Aug2022.

This command enables LAMMPS to act as a client with another server code that will compute the total energy, per-atom forces, and total virial for atom conformations and simulation box size/shapes that LAMMPS sends it.

Typically the server code will be a quantum mechanics (QM) code, hence the name of the fix. However this is not required, the server code could be another classical molecular dynamics code or LAMMPS itself. The server code must support use of the MDI Library as explained below.

Typically, to use this fix, the input script should not define any other classical force field components, e.g. a pair style, bond style, etc.

These are example use cases for this fix, discussed further below:

- perform an ab initio MD (AIMD) simulation with quantum forces
- perform an energy minimization with quantum forces
- perform a nudged elastic band (NEB) calculation with quantum forces
- perform a QM calculation for a series of independent systems which LAMMPS reads or generates once
- run a classical MD simulation and calculate QM energy/forces once every N steps on the current configuration

More generally any command which calculates per-atom forces can instead use quantum forces by defining this fix. Examples are the Monte Carlo commands *fix gcmc* and *fix atom/swap*, as well as the *compute born/matrix* command. The only requirement is that internally the command invokes the post_force() method of fixes such as this one, which will trigger the quantum calculation.

The coupling performed by this command is done via the MDI library. LAMMPS runs as an MDI driver (client), and sends MDI commands to an external MDI engine code (server), e.g. a QM code which has support for MDI. See the *Howto mdi* page for more information about how LAMMPS can operate as either an MDI driver or engine.

The `examples/mdi` directory contains input scripts using this fix in the various use cases discussed below. In each case, two instances of LAMMPS are used, once as an MDI driver, once as an MDI engine (surrogate for a QM code). The `examples/mdi/README` file explains how to launch two codes so that they communicate via the MDI library using either MPI or sockets. Any QM code that supports MDI could be used in place of LAMMPS acting as a QM surrogate. See the *Howto mdi* page for a current list (March 2022) of such QM codes. The `examples/QUANTUM` directory has examples for coupling LAMMPS to 3 QM codes either via this fix or the *fix mdi/qmmm* command.

Note that an engine code can support MDI in either or both of two modes. It can be used as a stand-alone code, launched at the same time as LAMMPS. Or it can be used as a plugin library, which LAMMPS loads. See the *mdi plugin* command for how to trigger LAMMPS to load a plugin library. The `examples/mdi/README` file and `examples/QUANTUM/QM-code/README` files explain how to launch the two codes in either mode.

The *virial* keyword setting of yes or no determines whether LAMMPS will request the QM code to also compute and return the QM contribution to a stress tensor for the system which LAMMPS will convert to a 6-element symmetric virial tensor.

The *add* keyword setting of *yes* or *no* determines whether the energy and forces and virial returned by the QM code will be added to the LAMMPS internal energy and forces and virial or not. If the setting is *no* then the default *fix_modify energy* and *fix_modify virial* settings are also set to *no* and your input scripts should not set them to yes. See more details on these fix_modify settings below.

Whatever the setting for the *add* keyword, the QM energy, forces, and virial will be stored by the fix, so they can be accessed by other commands. See details below.

The *every* keyword determines how often the QM code will be invoked during a dynamics run with the current LAMMPS simulation box and configuration of atoms. The QM code will be called once every *Nevery* timesteps. By default *Nevery* = 1.

The *connect* keyword determines whether this fix performs a one-time connection to the QM code. The default is *yes*. The only time a *no* is needed is if this command is used multiple times in an input script and the MDI coupling is between two stand-alone codes (not plugin mode). E.g. if it used inside a loop which also uses the *clear* command to destroy the system (including this fix). See the `examples/mdi/in.series.driver` script as an example of this, where LAMMPS is using the QM code to compute energy and forces for a series of system configurations. In this use case *connect no* is used along with the *mdi connect and exit* command to one-time initiate/terminate the connection outside the loop.

The *elements* keyword allows specification of what element each LAMMPS atom type corresponds to. This is specified by the chemical symbol of the element, e.g. C or Al or Si. A symbol must be specified for each of the ntypes LAMMPS atom types. Multiple LAMMPS types can represent the same element. Ntypes is typically specified via the *create_box* command or in the data file read by the *read_data* command.

If this keyword is specified, then this fix will send the MDI ">ELEMENTS" command to the engine, to ensure the two codes are consistent in their definition of atomic species. If this keyword is not specified, then this fix will send the MDI >TYPES command to the engine. This is fine if both the LAMMPS driver and the MDI engine are initialized so that the atom type values are consistent in both codes.

The *mc* keyword enables this fix to be used with a Monte Carlo (MC) fix to calculate before/after quantum energies as part of the MC accept/reject criterion. The *fix gcmc* and *fix atom/swap* commands can be used in this manner. Specify the ID of the MC fix following the *mc* keyword. This allows the two fixes to coordinate when MC events are being calculated versus MD timesteps between the MC events.

---

The following 3 example use cases are illustrated in the `examples/mdi` directory. See its README file for more details.

(1) To run an ab initio MD (AIMD) dynamics simulation, or an energy minimization with QM forces, or a multi-replica NEB calculation, use *add yes* and *every 1* (the defaults). This is so that every time LAMMPS needs energy and forces, the QM code will be invoked.

Both LAMMPS and the QM code should define the same system (simulation box, atoms and their types) in their respective input scripts. Note that on this scenario, it may not be necessary for LAMMPS to define a pair style or use a neighbor list.

LAMMPS will then perform the timestepping or minimization iterations for the simulation. At the point in each timestep or iteration when LAMMPS needs the force on each atom, it communicates with the engine code. It sends the current simulation box size and shape (if they change dynamically, e.g. during an NPT simulation), and the current atom coordinates. The engine code computes quantum forces on each atom and the total energy of the system and returns them to LAMMPS.

Note that if the AIMD simulation is an NPT or NPH model, or the energy minimization includes *fix box relax* to equilibrate the box size/shape, then LAMMPS computes a pressure. This means the *virial* keyword should be set to *yes* so that the QM contribution to the pressure can be included.

(2) To run dynamics with a LAMMPS interatomic potential, and evaluate the QM energy and forces once every 1000 steps, use *add no* and *every 1000*. This could be useful for using an MD run to generate randomized configurations which are then passed to the QM code to produce training data for a machine learning potential. A *dump custom* command could be invoked every 1000 steps to dump the atom coordinates and QM forces to a file. Likewise the QM energy and virial could be output with the *thermo_style custom* command.

(3) To do a QM evaluation of energy and forces for a series of $N$ independent systems (simulation box and atoms), use *add no* and *every 1*. Write a LAMMPS input script which loops over the $N$ systems. See the *Howto multiple* doc page for details on looping and removing old systems. The series of systems could be initialized by reading them from data files with *read_data* commands. Or, for example, by using the *lattice* , *create_atoms*, *delete_atoms*, and/or *displace_atoms random* commands to generate a series of different systems. At the end of the loop perform *run 0* and *write_dump* commands to invoke the QM code and output the QM energy and forces. As in (2) this be useful to produce QM data for training a machine learning potential.

---

### 2.113.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify energy* option is supported by this fix to add the potential energy computed by the QM code to the global potential energy of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify energy yes*, unless the *add* keyword is set to *no*, in which case the default setting is *no*.

The *fix_modify virial* option is supported by this fix to add the contribution computed by the QM code to the global pressure of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify virial yes*, unless the *add* keyword is set to *no*, in which case the default setting is *no*.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the energy returned by the QM code. The scalar value calculated by this fix is "extensive".

This fix also computes a global vector with of length 6 which contains the symmetric virial tensor values returned by the QM code. It can likewise be accessed by various *output commands*.

The ordering of values in the symmetric virial tensor is as follows: vxx, vyy, vzz, vxy, vxz, vyz. The values will be in pressure *units*.

This fix also computes a peratom array with 3 columns which contains the peratom forces returned by the QM code. It can likewise be accessed by various *output commands*.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

Assuming the *add* keyword is set to *yes* (the default), the forces computed by the QM code are used during an energy minimization, invoked by the *minimize* command.

**Note**

If you want the potential energy associated with the QM forces to be included in the total potential energy of the system (the quantity being minimized), you MUST not disable the *fix_modify energy* option for this fix, which means the *add* keyword should also be set to *yes* (the default).

### 2.113.5 Restrictions

This fix is part of the MDI package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

To use LAMMPS as an MDI driver in conjunction with other MDI-enabled codes (MD or QM codes), the *units* command should be used to specify *real* or *metal* units. This will ensure the correct unit conversions between LAMMPS and MDI units. The other code will also perform similar unit conversions into its preferred units.

LAMMPS can also be used as an MDI driver in other unit choices it supports, e.g. *lj*, but then no unit conversion to MDI units is performed.

If this fix is used in conjunction with a QM code that does not support periodic boundary conditions (more specifically, a QM code that does not support the >CELL MDI command), the LAMMPS system must be fully non-periodic. I.e. no dimension of the system can be periodic.

### 2.113.6 Related commands

*mdi plugin*, *mdi engine*, *fix mdi/qmmm*

### 2.113.7 Default

The default for the optional keywords are virial = no, add = yes, every = 1, connect = yes.

## 2.114 fix mdi/qmmm command

### 2.114.1 Syntax

```
fix ID group-ID mdi/qmmm mode keyword value(s) keyword value(s) ...
```

- ID, group-ID are documented in *fix* command
- mdi/qmmm = style name of this fix command
- mode = *direct* or *potential*
- zero or more keyword/value pairs may be appended
- keyword = *virial* or *add* or *every* or *connect* or *elements*

```
virial args = yes or no
  yes = request virial tensor from server code
  no = do not request virial tensor from server code
connect args = yes or no
  yes = perform a one-time connection to the MDI engine code
  no = do not perform the connection operation
elements args = N_1 N_2 ... N_ntypes
  N_1,N_2,...N_ntypes = chemical symbol for each of ntypes LAMMPS atom types
```

### 2.114.2 Examples

```
fix 1 all mdi/qmmm direct
fix 1 all mdi/qmmm potential virial yes
fix 1 all mdi/qmmm potential virial yes elements 13 29
```

### 2.114.3 Description

Added in version 28Mar2023.

This command enables LAMMPS to act as a client with another server code to perform a coupled QM/MM (quantum-mechanics/molecular-mechanics) simulation. LAMMPS will perform classical MD (molecular mechanics or MM) for the (typically larger) MM portion of the system. A quantum mechanics code will calculate quantum energy and forces for the QM portion of the system. The two codes work together to calculate the energy and forces due to the cross interactions between QM and MM atoms. The QM server code must support use of the MDI library as explained below.

The partitioning of the system between QM and MM atoms is as follows. Atoms in the specified group are QM atoms; the remaining atoms are MM atoms. The input script should thus define this partitioning. See additional information below about other requirements for an input script to use this fix and perform a QM/MM simulation.

The code coupling performed by this command is done via the MDI Library. LAMMPS runs as an MDI driver (client), and sends MDI commands to an external MDI engine code (server), in this case a QM code which has support for MDI. See the *Howto mdi* page for more information about how LAMMPS can operate as either an MDI driver or engine.

The `examples/QUANTUM` directory has sub-directories with example input scripts using this fix in tandem with different QM codes. The README files in the sub-directories explain how to download and build the various QM codes. They also explain how to launch LAMMPS and the QM code so that they communicate via the MDI library using either MPI or sockets. Any QM code that supports MDI could be used in addition to those discussed in the sub-directories. See the *Howto mdi* page for a current list (March 2022) of such QM codes.

Note that an engine code can support MDI in either or both of two modes. It can be used as a stand-alone code, launched at the same time as LAMMPS. Or it can be used as a plugin library, which LAMMPS loads. See the *mdi plugin* command for how to trigger LAMMPS to load a plugin library. The `examples/QUANTUM` sub-directory README files explains how to launch the two codes in either mode.

---

The *mode* setting determines which QM/MM coupling algorithm is used. LAMMPS currently supports *direct* and *potential* algorithms, based on the *mode* setting. Both algorithms should give reasonably accurate results, but some QM codes support only one of the two modes. E.g. in the `examples/QUANTUM` directory, PySCF supports only *direct*, NWChem supports only *potential*, and LATTE currently supports neither, so it cannot be used for QM/MM simulations using this fix.

The *direct* option passes the coordinates and charges of each MM atom to the quantum code, in addition to the coordinates of each QM atom. The quantum code returns forces on each QM atom as well as forces on each MM atom. The latter is effectively the force on MM atoms due to the QM atoms.

The input script for performing a *direct* mode QM/MM simulation should do the following:

- delete all bonds (angles, dihedrals, etc) between QM atoms
- set the charge on each QM atom to zero
- define no bonds (angles, dihedrals, etc) which involve both QM and MM atoms
- define a force field (pair, bonds, angles, optional kspace) for the entire system

The first two bullet can be performed using the *delete_bonds* and *set* commands.

The third bullet is required to have a consistent model, but is not checked by LAMMPS.

The fourth bullet implies that non-bonded non-Coulombic interactions (e.g. van der Waals) between QM/QM and QM/MM pairs of atoms are computed by LAMMPS.

See the `examples/QUANTUM/PySCF/in.*` files for examples of input scripts for QM/MM simulations using the *direct* mode.

The *potential* option passes the coordinates of each QM atom and a Coulomb potential for each QM atom to the quantum code. The latter is calculated by performing a Coulombics-only calculation for the entire system, subtracting all QM/QM pairwise Coulombic terms, and dividing the Coulomb energy on each QM atom by the charge of the QM atom. The potential value represents the Coulombic influence of all the MM atoms on each QM atom.

The quantum code returns forces and charge on each QM atom. The new charges on the QM atom are used to recalculate the MM force field, resulting in altered forces on the MM atoms.

The input script for performing a *potential* mode QM/MM simulation should do the following:

- delete all bonds (angles, dihedrals, etc) between QM atoms
- define a hybrid pair style which includes a Coulomb-only pair sub-style

- define no bonds (angles, dihedrals, etc) which involve both QM and MM atoms
- define a force field (pair, bonds, angles, optional kspace) for the entire system

The first operation can be performed using the *delete_bonds* command. See the `examples/QUANTUM/NWChem/in.*` files for examples of how to do this.

The second operation is necessary so that this fix can calculate the Coulomb potential for the QM atoms.

The third bullet is required to have a consistent model, but is not checked by LAMMPS.

The fourth bullet implies that non-bonded non-Coulombic interactions (e.g. van der Waals) between QM/QM and QM/MM pairs of atoms are computed by LAMMPS. However, some QM codes do not want the MM code (LAMMPS) to compute QM/QM van der Waals interactions. NWChem is an example. In this case, the coefficients for those interactions need to be turned off, which typically requires the atom types for the QM atoms be different than those for the MM atoms.

See the `examples/QUANTUM/NWChem/in.*` files for examples of input scripts for QM/MM simulations using the *potential* mode. Those scripts also illustrate how to turn off QM/QM van der Waals interactions.

---

The *virial* keyword setting of yes or no determines whether LAMMPS will request the QM code to also compute and return the QM contribution to a stress tensor for the system which LAMMPS will convert to a 6-element symmetric virial tensor.

The *connect* keyword determines whether this fix performs a one-time connection to the QM code. The default is *yes*. The only time a *no* is needed is if this command is used multiple times in an input script. E.g. if it used inside a loop which also uses the *clear* command to destroy the system (including this fix). As example would be a script which loop over a series of independent QM/MM simulations, e.g. each with their own data file. In this use case *connect no* could be used along with the *mdi connect and exit* command to one-time initiate/terminate the connection outside the loop.

The *elements* keyword allows specification of what element each LAMMPS atom type corresponds to. This is specified by the chemical symbol of the element, e.g. C or Al or Si. A symbol must be specified for each of the ntypes LAMMPS atom types. Multiple LAMMPS types can represent the same element. Ntypes is typically specified via the *create_box* command or in the data file read by the *read_data* command.

If this keyword is specified, then this fix will send the MDI ">ELEMENTS" command to the engine, to insure the two codes are consistent in their definition of atomic species. If this keyword is not specified, then this fix will send the MDI >TYPES command to the engine. This is fine if both the LAMMPS driver and the MDI engine are initialized so that the atom type values are consistent in both codes.

---

### 2.114.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify energy* option is supported by this fix to add the potential energy computed by the QM code to the global potential energy of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify energy yes*.

The *fix_modify virial* option is supported by this fix to add the contribution computed by the QM code to the global pressure of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify virial yes*.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the energy returned by the QM code. The scalar value calculated by this fix is "extensive".

This fix also computes a global vector with of length 6 which contains the symmetric virial tensor values returned by the QM code. It can likewise be accessed by various *output commands*.

The ordering of values in the symmetric virial tensor is as follows: vxx, vyy, vzz, vxy, vxz, vyz. The values will be in pressure *units*.

This fix also computes a peratom array with 3 columns which contains the peratom forces returned by the QM code. It can likewise be accessed by various *output commands*. Note that for *direct* mode this will be quantum forces on both QM and MM atoms. For *potential* mode it will only be quantum forces on QM atoms; the forces for MM atoms will be zero.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The forces computed by the QM code are used during an energy minimization, invoked by the *minimize* command.

**Note**

If you want the potential energy associated with the QM forces to be included in the total potential energy of the system (the quantity being minimized), you MUST not disable the *fix_modify energy* option for this fix.

### 2.114.5 Restrictions

This command is part of the MDI package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

To use LAMMPS as an MDI driver in conjunction with other MDI-enabled codes (MD or QM codes), the *units* command should be used to specify *real* or *metal* units. This will ensure the correct unit conversions between LAMMPS and MDI units. The other code will also perform similar unit conversions into its preferred units.

If this fix is used in conjunction with a QM code that does not support periodic boundary conditions (more specifically, a QM code that does not support the >CELL MDI command), the LAMMPS system must be fully non-periodic. I.e. no dimension of the system can be periodic.

### 2.114.6 Related commands

*mdi plugin*, *mdi engine*, *fix mdi/qm*

### 2.114.7 Default

The default for the optional keywords are virial = no and connect = yes.

## 2.115 fix meso/move command

### 2.115.1 Syntax

```
fix ID group-ID meso/move style args keyword values ...
```

- ID, group-ID are documented in *fix* command
- meso/move = style name of this fix command
- style = *linear* or *wiggle* or *rotate* or *variable*

```
linear args = Vx Vy Vz
  Vx,Vy,Vz = components of velocity vector (velocity units), any component can be␣
↪specified as NULL
wiggle args = Ax Ay Az period
  Ax,Ay,Az = components of amplitude vector (distance units), any component can be␣
↪specified as NULL
  period = period of oscillation (time units)
```

```
rotate args = Px Py Pz Rx Ry Rz period
  Px,Py,Pz = origin point of axis of rotation (distance units)
  Rx,Ry,Rz = axis of rotation vector
  period = period of rotation (time units)
variable args = v_dx v_dy v_dz v_vx v_vy v_vz
  v_dx,v_dy,v_dz = 3 variable names that calculate x,y,z displacement as function␣
↪of time, any component can be specified as NULL
  v_vx,v_vy,v_vz = 3 variable names that calculate x,y,z velocity as function of␣
↪time, any component can be specified as NULL
```

- zero or more keyword/value pairs may be appended
- keyword = *units*

  ```
  units value = box or lattice
  ```

## 2.115.2 Examples

```
fix 1 boundary meso/move wiggle 3.0 0.0 0.0 1.0 units box
fix 2 boundary meso/move rotate 0.0 0.0 0.0 0.0 0.0 1.0 5.0
fix 2 boundary meso/move variable v_myx v_myy NULL v_VX v_VY NULL
```

## 2.115.3 Description

Perform updates of position, velocity, internal energy and local density for mesoscopic particles in the group each timestep using the specified settings or formulas, without regard to forces on the particles. This can be useful for boundary, solid bodies or other particles, whose movement can influence nearby particles.

The operation of this fix is exactly like that described by the *fix move* command, except that particles' density, internal energy and extrapolated velocity are also updated.

**Note**

The particles affected by this fix should not be time integrated by other fixes (e.g. *fix sph*, *fix sph/stationary*), since that will change their positions and velocities twice.

**Note**

As particles move due to this fix, they will pass through periodic boundaries and be remapped to the other side of the simulation box, just as they would during normal time integration (e.g. via the *fix sph* command). It is up to you to decide whether periodic boundaries are appropriate with the kind of particle motion you are prescribing with this fix.

**Note**

As discussed below, particles are moved relative to their initial position at the time the fix is specified. These initial coordinates are stored by the fix in "unwrapped" form, by using the image flags associated with each particle. See the *dump custom* command for a discussion of "unwrapped" coordinates. See the Atoms section of the *read_data* command for a discussion of image flags and how they are set for each particle. You can reset the image flags (e.g. to 0) before invoking this fix by using the *set image* command.

The *linear* style moves particles at a constant velocity, so that their position $X$ = (x,y,z) as a function of time is given in vector notation as

```
X(t) = X0 + V * delta
```

where *X0* = (x0,y0,z0) is their position at the time the fix is specified, $V$ is the specified velocity vector with components (Vx,Vy,Vz), and *delta* is the time elapsed since the fix was specified. This style also sets the velocity of each particle to V = (Vx,Vy,Vz). If any of the velocity components is specified as NULL, then the position and velocity of that component is time integrated the same as the *fix sph* command would perform, using the corresponding force component on the particle.

Note that the *linear* style is identical to using the *variable* style with an *equal-style variable* that uses the vdisplace() function. E.g.

```
variable V equal 10.0
variable x equal vdisplace(0.0,$V)
fix 1 boundary move variable v_x NULL NULL v_V NULL NULL
```

The *wiggle* style moves particles in an oscillatory fashion, so that their position $X$ = (x,y,z) as a function of time is given in vector notation as

```
X(t) = X0 + A sin(omega*delta)
```

where *X0* = (x0,y0,z0) is their position at the time the fix is specified, $A$ is the specified amplitude vector with components (Ax,Ay,Az), *omega* is 2 PI / *period*, and *delta* is the time elapsed since the fix was specified. This style also sets the velocity of each particle to the time derivative of this expression. If any of the amplitude components is specified as NULL, then the position and velocity of that component is time integrated the same as the *fix sph* command would perform, using the corresponding force component on the particle.

Note that the *wiggle* style is identical to using the *variable* style with *equal-style variables* that use the swiggle() and cwiggle() functions. E.g.

```
variable A equal 10.0
variable T equal 5.0
variable omega equal 2.0*PI/$T
variable x equal swiggle(0.0,$A,$T)
variable v equal v_omega*($A-cwiggle(0.0,$A,$T))
fix 1 boundary move variable v_x NULL NULL v_v NULL NULL
```

The *rotate* style rotates particles around a rotation axis $R$ = (Rx,Ry,Rz) that goes through a point $P$ = (Px,Py,Pz). The *period* of the rotation is also specified. The direction of rotation for the particles around the rotation axis is consistent with the right-hand rule: if your right-hand thumb points along $R$, then your fingers wrap around the axis in the direction of rotation.

This style also sets the velocity of each particle to (omega cross Rperp) where omega is its angular velocity around the rotation axis and Rperp is a perpendicular vector from the rotation axis to the particle.

The *variable* style allows the position and velocity components of each particle to be set by formulas specified via the *variable* command. Each of the 6 variables is specified as an argument to the fix as v_name, where name is the variable name that is defined elsewhere in the input script.

Each variable must be of either the *equal* or *atom* style. *Equal*-style variables compute a single numeric quantity, that can be a function of the timestep as well as of other simulation values. *Atom*-style variables compute a numeric quantity for each particle, that can be a function per-atom quantities, such as the particle's position, as well as of the timestep and other simulation values. Note that this fix stores the original coordinates of each particle (see note below) so that per-atom quantity can be used in an atom-style variable formula. See the *variable* command for details.

The first 3 variables (v_dx,v_dy,v_dz) specified for the *variable* style are used to calculate a displacement from the particle's original position at the time the fix was specified. The second 3 variables (v_vx,v_vy,v_vz) specified are

used to compute a velocity for each particle.

Any of the 6 variables can be specified as NULL. If both the displacement and velocity variables for a particular x,y,z component are specified as NULL, then the position and velocity of that component is time integrated the same as the *fix sph* command would perform, using the corresponding force component on the particle. If only the velocity variable for a component is specified as NULL, then the displacement variable will be used to set the position of the particle, and its velocity component will not be changed. If only the displacement variable for a component is specified as NULL, then the velocity variable will be used to set the velocity of the particle, and the position of the particle will be time integrated using that velocity.

The *units* keyword determines the meaning of the distance units used to define the *linear* velocity and *wiggle* amplitude and *rotate* origin. This setting is ignored for the *variable* style. A *box* value selects standard units as defined by the *units* command, e.g. velocity in Angstroms/fs and amplitude and position in Angstroms for units = real. A *lattice* value means the velocity units are in lattice spacings per time and the amplitude and position are in lattice spacings. The *lattice* command must have been previously used to define the lattice spacing. Each of these 3 quantities may be dependent on the x,y,z dimension, since the lattice spacings can be different in x,y,z.

---

### 2.115.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the original coordinates of moving particles to *binary restart files*, as well as the initial timestep, so that the motion can be continuous in a restarted simulation. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

> **Note**
>
> Because the move positions are a function of the current timestep and the initial timestep, you cannot reset the timestep to a different value after reading a restart file, if you expect a fix move command to work in an uninterrupted fashion.

None of the *fix_modify* options are relevant to this fix.

This fix produces a per-atom array which can be accessed by various *output commands*. The number of columns for each atom is 3, and the columns store the original unwrapped x,y,z coords of each particle. The per-atom values can be accessed on any timestep.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

This fix is not invoked during *energy minimization*.

### 2.115.5 Restrictions

This fix is part of the DPD-SMOOTH package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix requires that atoms store density and internal energy as defined by the *atom_style sph* command.

All particles in the group must be mesoscopic SPH/SDPD particles.

Changed in version 29Aug2024.

This fix is incompatible with deformation controls that remap velocity, for instance the *remap v* option of *fix deform*.

### 2.115.6 Related commands

*fix move*, *fix sph*, *displace_atoms*

### 2.115.7 Default

The option default is units = lattice.

## 2.116 fix mol/swap command

### 2.116.1 Syntax

```
fix ID group-ID mol/swap N X itype jtype seed T keyword value ...
```

- ID, group-ID are documented in *fix* command
- atom/swap = style name of this fix command
- N = invoke this fix every N steps
- X = number of swaps to attempt every N steps
- itype,jtype = two atom types (1-Ntypes or type label) to swap with each other
- seed = random # seed (positive integer)
- T = scaling temperature of the MC swaps (temperature units)
- zero or more keyword/value pairs may be appended to args
- keyword = *ke*

  *ke* value = *no* or *yes*
    *no* = no conservation of kinetic energy after atom swaps
    *yes* = kinetic energy is conserved after atom swaps

### 2.116.2 Examples

```
fix 2 all mol/swap 100 1 2 3 29494 300.0 ke no
fix mySwap fluid mol/swap 500 10 1 2 482798 1.0

labelmap atom 1 A 2 B
fix mySwap fluid mol/swap 500 10 A B 482798 1.0
```

### 2.116.3 Description

This fix performs Monte Carlo swaps of two specified atom types within a randomly selected molecule. Two possible use cases are as follows.

First, consider a mixture of some molecules with atoms of itype and other molecules with atoms of jtype. The fix will select a random molecule and attempt to swap all the itype atoms to jtype for the first kind of molecule, or all the jtype atoms to itype for the second kind. Because the swap will only take place if it is energetically favorable, the fix can be used to determine the miscibility of 2 different kinds of molecules much more quickly than just dynamics would do it.

Second, consider diblock co-polymers with two types of monomers itype and jtype. The fix will select a random molecule and attempt to do a itype <–> jtype swap of all those monomers within the molecule. Thus the fix can be used to find the energetically favorable fractions of two flavors of diblock co-polymers.

Intra-molecular swaps of atom types are attempted every N timesteps. On that timestep, X swaps are attempted. For each attempt a single molecule ID is randomly selected. The range of possible molecule IDs from loID to hiID is pre-computed before each run begins. The loID/hiID is set for the molecule with the smallest/largest ID which has any itype or jtype atoms in it. Note that if you define a system with many molecule IDs between loID and hiID which have no itype or jtype atoms, then the fix will be inefficient at performing swaps. Also note that if atoms with molecule ID = 0 exist, they are not considered molecules by this fix; they are assumed to be solvent atoms or molecules.

Candidate atoms for swapping must also be in the fix group. Atoms within the selected molecule which are not itype or jtype are ignored.

When an atom is swapped from itype to jtype (or vice versa), if charges are defined, the charge values for itype versus jtype atoms are also swapped. This requires that all itype atoms in the system have the same charge value. Likewise all jtype atoms in the system must have the same charge value. If this is not the case, LAMMPS issues a warning that it cannot swap charge values.

If the *ke* keyword is set to yes, which is the default, and the masses of itype and jtype atoms are different, then when a swap occurs, the velocity of the swapped atom is rescaled by the sqrt of the mass ratio, so as to conserve the kinetic energy of the atom.

---

The potential energy of the entire system is computed before and after each swap is performed within a single molecule. The specified temperature T is used in the Metropolis criterion to accept or reject the attempted swap. If the swap is rejected all swapped values are reversed.

The potential energy calculations can include systems and models with the following features:

- manybody pair styles, including EAM
- hybrid pair styles
- long-range electrostatics (kspace)
- triclinic systems
- potential energy contributions from other fixes

For the last bullet point, fixes can have an associated potential energy. Examples of such fixes include: *efield*, *gravity*, *addforce*, *langevin*, *restrain*, *temp/berendsen*, *temp/rescale*, and *wall fixes*. For that energy to be included in the total potential energy of the system (the quantity used for the swap accept/reject decision), you MUST enable the *fix_modify energy* option for that fix. The doc pages for individual *fix* commands specify if this should be done.

**Note**

One comment on computational efficiency. If the cutoff lengths defined for the pair style are different for itype versus jtype atoms (for any of their interactions with any other atom type), then a new neighbor list needs to be generated for every attempted swap. This is potentially expensive if N is small or X is large.

---

### 2.116.4 Dump image info

Added in version 11Feb2026.

Fix *mol/swap* supports the *fix* keyword of *dump image*. The fix will pass geometry information about atoms involved in a swap to *dump image* so that these atoms can be highlighted in the visualization as additional spheres. For how long those additional spheres will be shown depends on the value of the *vizsteps* setting (default is 1000) which can be changed by using the *fix_modify command*. If an atom is involved in multiple swaps, the check on showing the additional graphics depends on the timestep of its last swap.

The color of the additional spheres is by default that of the atom type *before* the swap when using color styles "type" or "element". With color style "const" the default value of "white" can be changed using *dump_modify fcolor*. The transparency is by default fully opaque and can be changed with *dump_modify ftrans*.

The *fflag1* setting of *dump image fix* has no effect.

The *fflag2* setting allows you to set the radius of the added spheres, since the radius is set to zero internally.

---

### 2.116.5 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the state of the fix to *binary restart files*. This includes information about the random number generator seed, the next timestep for MC exchanges, the number of exchange attempts and successes etc. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

> **Note**
>
> For this to work correctly, the timestep must **not** be changed after reading the restart with *reset_timestep*. The fix will try to detect it and stop with an error.

None of the *fix_modify* options are relevant to this fix.

This fix computes a global vector of length 2, which can be accessed by various *output commands*. The vector values are the following global cumulative quantities:

1. swap attempts
2. swap accepts

The vector values calculated by this fix are "intensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.116.6 Restrictions

This fix is part of the MC package. It is only enabled if LAMMPS was built with that package. See the *Build package* doc page for more info.

### 2.116.7 Related commands

*fix atom/swap*, *fix gcmc*

### 2.116.8 Default

The option default is ke = yes.

## 2.117 fix momentum command

Accelerator Variants: *momentum/kk*

# 2.118 fix momentum/chunk command

## 2.118.1 Syntax

```
fix ID group-ID momentum N keyword values ...
```

- ID, group-ID are documented in *fix* command
- momentum = style name of this fix command
- N = adjust the momentum every this many timesteps one or more keyword/value pairs may be appended

```
fix ID group-ID momentum/chunk N chunkID keyword values ...
```

- ID, group-ID are documented in *fix* command
- momentum/chunk = style name of this fix command
- N = adjust the momentum per chunk every this many timesteps
- chunkID = ID of *compute chunk/atom* command

  one or more keyword/value settings may be appended to each of the fix commands:

- keyword = *linear* or *angular* or *rescale*

  ```
  linear values = xflag yflag zflag
    xflag,yflag,zflag = 0/1 to exclude/include each dimension
  angular values = none
  rescale values = none
  ```

## 2.118.2 Examples

```
fix 1 all momentum 1 linear 1 1 0
fix 1 all momentum 1 linear 1 1 1 rescale
fix 1 all momentum 100 linear 1 1 1 angular
fix 1 all momentum/chunk 100 molchunk linear 1 1 1 angular
```

## 2.118.3 Description

Fix momentum zeroes the linear and/or angular momentum of the group of atoms every N timesteps by adjusting the velocities of the atoms. Fix momentum/chunk works equivalently, but operates on a per-chunk basis.

One (or both) of the *linear* or *angular* keywords **must** be specified.

If the *linear* keyword is used, the linear momentum is zeroed by subtracting the center-of-mass velocity of the group or chunk from each atom. This does not change the relative velocity of any pair of atoms. One or more dimensions can be excluded from this operation by setting the corresponding flag to 0.

If the *angular* keyword is used, the angular momentum is zeroed by subtracting a rotational component from each atom.

This command can be used to ensure the entire collection of atoms (or a subset of them) does not drift or rotate during the simulation due to random perturbations (e.g. *fix langevin* thermostatting).

The *rescale* keyword enables conserving the kinetic energy of the group or chunk of atoms by rescaling the velocities after the momentum was removed.

Note that the *velocity* command can be used to create initial velocities with zero aggregate linear and/or angular momentum.

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 2.118.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.118.5 Restrictions

Fix momentum/chunk is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.118.6 Related commands

*fix recenter*, *velocity*

### 2.118.7 Default

none

## 2.119 fix move command

### 2.119.1 Syntax

```
fix ID group-ID move style args keyword values ...
```

- ID, group-ID are documented in *fix* command
- move = style name of this fix command
- style = *linear* or *wiggle* or *rotate* or *transrot* or *variable*

```
linear args = Vx Vy Vz
  Vx,Vy,Vz = components of velocity vector (velocity units), any component can be␣
↪specified as NULL
wiggle args = Ax Ay Az period
  Ax,Ay,Az = components of amplitude vector (distance units), any component can be␣
↪specified as NULL
  period = period of oscillation (time units)
rotate args = Px Py Pz Rx Ry Rz period
  Px,Py,Pz = origin point of axis of rotation (distance units)
  Rx,Ry,Rz = axis of rotation vector
  period = period of rotation (time units)
```

```
transrot args = Vx Vy Vz Px Py Pz Rx Ry Rz period
  Vx,Vy,Vz = components of velocity vector (velocity units)
  Px,Py,Pz = origin point of axis of rotation (distance units)
  Rx,Ry,Rz = axis of rotation vector
  period = period of rotation (time units)
variable args = v_dx v_dy v_dz v_vx v_vy v_vz
  v_dx,v_dy,v_dz = 3 variable names that calculate x,y,z displacement as function␣
↪of time, any component can be specified as NULL
  v_vx,v_vy,v_vz = 3 variable names that calculate x,y,z velocity as function of␣
↪time, any component can be specified as NULL
```

- zero or more keyword/value pairs may be appended
- keyword = *units* or *update*

```
units value = box or lattice
update value = dipole
```

## 2.119.2 Examples

```
fix 1 boundary move wiggle 3.0 0.0 0.0 1.0 units box
fix 2 boundary move rotate 0.0 0.0 0.0 0.0 0.0 1.0 5.0
fix 2 boundary move variable v_myx v_myy NULL v_VX v_VY NULL
fix 3 boundary move transrot 0.1 0.1 0.0 0.0 0.0 0.0 0.0 0.0 1.0 5.0 units box update␣
↪dipole
```

## 2.119.3 Description

Perform updates of position and velocity for atoms in the group each timestep using the specified settings or formulas, without regard to forces on the atoms. This can be useful for boundary or other atoms, whose movement can influence nearby atoms.

**Note**

The atoms affected by this fix should not normally be time integrated by other fixes (e.g. *fix nve*, *fix nvt*), since that will change their positions and velocities twice.

**Note**

As atoms move due to this fix, they will pass through periodic boundaries and be remapped to the other side of the simulation box, just as they would during normal time integration (e.g. via the *fix nve* command). It is up to you to decide whether periodic boundaries are appropriate with the kind of atom motion you are prescribing with this fix.

**Note**

As discussed below, atoms are moved relative to their initial position at the time the fix is specified. These initial coordinates are stored by the fix in "unwrapped" form, by using the image flags associated with each atom. See the *dump custom* command for a discussion of "unwrapped" coordinates. See the Atoms section of the *read_data* command for a discussion of image flags and how they are set for each atom. You can reset the image flags (e.g. to 0) before invoking this fix by using the *set image* command.

The *linear* style moves atoms at a constant velocity, so that their position $X$ = (x,y,z) as a function of time is given in vector notation as

```
X(t) = X0 + V * delta
```

where *X0* = (x0,y0,z0) is their position at the time the fix is specified, *V* is the specified velocity vector with components (Vx,Vy,Vz), and *delta* is the time elapsed since the fix was specified. This style also sets the velocity of each atom to V = (Vx,Vy,Vz). If any of the velocity components is specified as NULL, then the position and velocity of that component is time integrated the same as the *fix nve* command would perform, using the corresponding force component on the atom.

Note that the *linear* style is identical to using the *variable* style with an *equal-style variable* that uses the vdisplace() function. E.g.

```
variable V equal 10.0
variable x equal vdisplace(0.0,$V)
fix 1 boundary move variable v_x NULL NULL v_V NULL NULL
```

The *wiggle* style moves atoms in an oscillatory fashion, so that their position $X$ = (x,y,z) as a function of time is given in vector notation as

```
X(t) = X0 + A sin(omega*delta)
```

where *X0* = (x0,y0,z0) is their position at the time the fix is specified, *A* is the specified amplitude vector with components (Ax,Ay,Az), *omega* is 2 PI / *period*, and *delta* is the time elapsed since the fix was specified. This style also sets the velocity of each atom to the time derivative of this expression. If any of the amplitude components is specified as NULL, then the position and velocity of that component is time integrated the same as the *fix nve* command would perform, using the corresponding force component on the atom.

Note that the *wiggle* style is identical to using the *variable* style with *equal-style variables* that use the swiggle() and cwiggle() functions. E.g.

```
variable A equal 10.0
variable T equal 5.0
variable omega equal 2.0*PI/$T
variable x equal swiggle(0.0,$A,$T)
variable v equal v_omega*($A-cwiggle(0.0,$A,$T))
fix 1 boundary move variable v_x NULL NULL v_v NULL NULL
```

The *rotate* style rotates atoms around a rotation axis $R$ = (Rx,Ry,Rz) that goes through a point $P$ = (Px,Py,Pz). The *period* of the rotation is also specified. The direction of rotation for the atoms around the rotation axis is consistent with the right-hand rule: if your right-hand thumb points along $R$, then your fingers wrap around the axis in the direction of rotation.

This style also sets the velocity of each atom to (omega cross Rperp) where omega is its angular velocity around the rotation axis and Rperp is a perpendicular vector from the rotation axis to the atom. If the defined *atom_style* assigns an angular velocity or angular momentum or orientation to each atom (*atom styles* sphere, ellipsoid, line, tri, body), then those properties are also updated appropriately to correspond to the atom's motion and rotation over time.

The *transrot* style combines the effects of *rotate* and *linear* so that it is possible to prescribe a rotating group of atoms that also moves at a constant velocity. The arguments are for the translation first and then for the rotation. Since the rotation affects all coordinate components, it is not possible to set any of the translation vector components to NULL.

The *variable* style allows the position and velocity components of each atom to be set by formulas specified via the *variable* command. Each of the 6 variables is specified as an argument to the fix as v_name, where name is the variable name that is defined elsewhere in the input script.

Each variable must be of either the *equal* or *atom* style. *Equal*-style variables compute a single numeric quantity, that can be a function of the timestep as well as of other simulation values. *Atom*-style variables compute a numeric quantity

for each atom, that can be a function per-atom quantities, such as the atom's position, as well as of the timestep and other simulation values. Note that this fix stores the original coordinates of each atom (see note below) so that per-atom quantity can be used in an atom-style variable formula. See the *variable* command for details.

The first 3 variables (v_dx,v_dy,v_dz) specified for the *variable* style are used to calculate a displacement from the atom's original position at the time the fix was specified. The second 3 variables (v_vx,v_vy,v_vz) specified are used to compute a velocity for each atom.

Any of the 6 variables can be specified as NULL. If both the displacement and velocity variables for a particular x,y,z component are specified as NULL, then the position and velocity of that component is time integrated the same as the *fix nve* command would perform, using the corresponding force component on the atom. If only the velocity variable for a component is specified as NULL, then the displacement variable will be used to set the position of the atom, and its velocity component will not be changed. If only the displacement variable for a component is specified as NULL, then the velocity variable will be used to set the velocity of the atom, and the position of the atom will be time integrated using that velocity.

The *units* keyword determines the meaning of the distance units used to define the *linear* velocity and *wiggle* amplitude and *rotate* origin. This setting is ignored for the *variable* style. A *box* value selects standard units as defined by the *units* command, e.g. velocity in Angstroms/fs and amplitude and position in Angstroms for units = real. A *lattice* value means the velocity units are in lattice spacings per time and the amplitude and position are in lattice spacings. The *lattice* command must have been previously used to define the lattice spacing. Each of these 3 quantities may be dependent on the x,y,z dimension, since the lattice spacings can be different in x,y,z.

Added in version 2Apr2025.

If the *update dipole* keyword/value pair is used together with the *rotate* or *transrot* style, then the orientation of the dipole moment of each particle is also updated appropriately to correspond with the rotation. This option should be used for models where a dipole moment is assigned to finite-size particles, e.g. spheroids via use of the *atom_style hybrid sphere dipole* command.

---

### 2.119.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the original coordinates of moving atoms to *binary restart files*, as well as the initial timestep, so that the motion can be continuous in a restarted simulation. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

> **Note**
>
> Because the move positions are a function of the current timestep and the initial timestep, you cannot reset the timestep to a different value after reading a restart file, if you expect a fix move command to work in an uninterrupted fashion.

None of the *fix_modify* options are relevant to this fix.

This fix produces a per-atom array which can be accessed by various *output commands*. The number of columns for each atom is 3, and the columns store the original unwrapped x,y,z coords of each atom. The per-atom values can be accessed on any timestep.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

For *rRESPA time integration*, this fix adjusts the position and velocity of atoms on the outermost rRESPA level.

### 2.119.5 Restrictions

none

### 2.119.6 Related commands

*fix nve*, *displace_atoms*

### 2.119.7 Default

none

The option default is units = lattice.

## 2.120 fix msst command

### 2.120.1 Syntax

```
fix ID group-ID msst dir shockvel keyword value ...
```

- ID, group-ID are documented in *fix* command
- msst = style name of this fix
- dir = *x* or *y* or *z*
- shockvel = shock velocity (strictly positive, distance/time units)
- zero or more keyword value pairs may be appended
- keyword = *q* or *mu* or *p0* or *v0* or *e0* or *tscale* or *beta* or *dftb*

```
q value = cell mass-like parameter (mass^2/distance^4 units)
mu value = artificial viscosity (mass/length/time units)
p0 value = initial pressure in the shock equations (pressure units)
v0 value = initial simulation cell volume in the shock equations (distance^3 units)
e0 value = initial total energy (energy units)
tscale value = reduction in initial temperature (unitless fraction between 0.0 and␣
↪1.0)
dftb value = yes or no for whether using MSST in conjunction with DFTB+
beta value = scale factor for improved energy conservation
```

### 2.120.2 Examples

```
fix 1 all msst y 100.0 q 1.0e5 mu 1.0e5
fix 2 all msst z 50.0 q 1.0e4 mu 1.0e4  v0 4.3419e+03 p0 3.7797e+03 e0 -9.72360e+02␣
↪tscale 0.01
fix 1 all msst y 100.0 q 1.0e5 mu 1.0e5 dftb yes beta 0.5
```

### 2.120.3 Description

This command performs the Multi-Scale Shock Technique (MSST) integration to update positions and velocities each timestep to mimic a compressive shock wave passing over the system. See *(Reed)* for a detailed description of this method. The MSST varies the cell volume and temperature in such a way as to restrain the system to the shock Hugoniot and the Rayleigh line. These restraints correspond to the macroscopic conservation laws dictated by a shock front. *shockvel* determines the steady shock velocity that will be simulated.

To perform a simulation, choose a value of *q* that provides volume compression on the timescale of 100 fs to 1 ps. If the volume is not compressing, either the shock speed is chosen to be below the material sound speed or *p0* has been chosen inaccurately. Volume compression at the start can be sped up by using a non-zero value of *tscale*. Use the smallest value of *tscale* that results in compression.

Under some special high-symmetry conditions, the pressure (volume) and/or temperature of the system may oscillate for many cycles even with an appropriate choice of mass-like parameter *q*. Such oscillations have physical significance in some cases. The optional *mu* keyword adds an artificial viscosity that helps break the system symmetry to equilibrate to the shock Hugoniot and Rayleigh line more rapidly in such cases.

The keyword *tscale* is a factor between 0 and 1 that determines what fraction of thermal kinetic energy is converted to compressive strain kinetic energy at the start of the simulation. Setting this parameter to a non-zero value may assist in compression at the start of simulations where it is slow to occur.

If keywords *e0*, *p0*,or *v0* are not supplied, these quantities will be calculated on the first step, after the energy specified by *tscale* is removed. The value of *e0* is not used in the dynamical equations, but is used in calculating the deviation from the Hugoniot.

The keyword *beta* is a scaling term that can be added to the MSST ionic equations of motion to account for drift in the conserved quantity during long timescale simulations, similar to a Berendsen thermostat. See *(Reed)* and *(Goldman)* for more details. The value of *beta* must be between 0.0 and 1.0 inclusive. A value of 0.0 means no contribution, a value of 1.0 means a full contribution.

Values of shockvel less than a critical value determined by the material response will not have compressive solutions. This will be reflected in lack of significant change of the volume in the MSST.

For all pressure styles, the simulation box stays orthogonal in shape. Parrinello-Rahman boundary conditions (tilted box) are supported by LAMMPS, but are not implemented for MSST.

This fix computes a temperature and pressure and potential energy each timestep. To do this, the fix creates its own computes of style "temp" "pressure", and "pe", as if these commands had been issued:

```
compute fix-ID_MSST_temp all temp
compute fix-ID_MSST_press all pressure fix-ID_MSST_temp

compute fix-ID_MSST_pe all pe
```

See the *compute temp* and *compute pressure* commands for details. Note that the IDs of the new computes are the fix-ID + "_MSST_temp" or "MSST_press" or "_MSST_pe". The group for the new computes is "all".

---

The *dftb* keyword is to allow this fix to be used when LAMMPS is being driven by DFTB+, a density-functional tight-binding code. If the keyword *dftb* is used with a value of *yes*, then the MSST equations are altered to account for the electron entropy contribution to the Hugonio relations and total energy. See *(Reed2)* and *(Goldman)* for details on this contribution. In this case, you must define a *fix external* command in your input script, which is used to callback to DFTB+ during the LAMMPS timestepping. DFTB+ will communicate its info to LAMMPS via that fix.

---

### 2.120.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the state of all internal variables to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The cumulative energy change in the system imposed by this fix is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* doc page for details.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the same cumulative energy change due to this fix described in the previous paragraph. The scalar value calculated by this fix is "extensive".

The progress of the MSST can be monitored by printing the global scalar and global vector quantities computed by the fix.

As mentioned above, the scalar is the cumulative energy change due to the fix. By monitoring the thermodynamic *econserve* output, this can be used to test if the MD timestep is sufficiently small for accurate integration of the dynamic equations.

The global vector contains four values in the following order. The vector values output by this fix are "intensive".

[*dhugoniot*, *drayleigh*, *lagrangian_speed*, *lagrangian_position*]

1. *dhugoniot* is the departure from the Hugoniot (temperature units).
2. *drayleigh* is the departure from the Rayleigh line (pressure units).
3. *lagrangian_speed* is the laboratory-frame Lagrangian speed (particle velocity) of the computational cell (velocity units).
4. *lagrangian_position* is the computational cell position in the reference frame moving at the shock speed. This is usually a good estimate of distance of the computational cell behind the shock front.

To print these quantities to the log file with descriptive column headers, the following LAMMPS commands are suggested:

```
fix             msst all msst z
variable dhug    equal f_msst[1]
variable dray    equal f_msst[2]
variable lgr_vel equal f_msst[3]
variable lgr_pos equal f_msst[4]
thermo_style     custom step temp ke pe lz pzz econserve v_dhug v_dray v_lgr_vel v_lgr_
↪pos f_msst
```

### 2.120.5 Restrictions

This fix style is part of the SHOCK package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

All cell dimensions must be periodic. This fix can not be used with a triclinic cell. The MSST fix has been tested only for the group-ID all.

### 2.120.6 Related commands

*fix nphug*, *fix deform*

### 2.120.7 Default

The keyword defaults are q = 10, mu = 0, tscale = 0.01, dftb = no, beta = 0.0. Note that p0, v0, and e0 are calculated on the first timestep.

---

**(Reed)** Reed, Fried, and Joannopoulos, Phys. Rev. Lett., 90, 235503 (2003).

**(Reed2)** Reed, J. Phys. Chem. C, 116, 2205 (2012).

**(Goldman)** Goldman, Srinivasan, Hamel, Fried, Gaus, and Elstner, J. Phys. Chem. C, 117, 7885 (2013).

# 2.121 fix mvv/dpd command

# 2.122 fix mvv/edpd command

# 2.123 fix mvv/tdpd command

## 2.123.1 Syntax

```
fix ID group-ID mvv/dpd lambda

fix ID group-ID mvv/edpd lambda

fix ID group-ID mvv/tdpd lambda
```

- ID, group-ID are documented in *fix* command
- mvv/dpd, mvv/edpd, mvv/tdpd = style name of this fix command
- lambda = (optional) relaxation parameter (unitless)

## 2.123.2 Examples

```
fix 1 all mvv/dpd
fix 1 all mvv/dpd 0.5
fix 1 all mvv/edpd
fix 1 all mvv/edpd 0.5
fix 1 all mvv/tdpd
fix 1 all mvv/tdpd 0.5
```

## 2.123.3 Description

Perform time integration using the modified velocity-Verlet (MVV) algorithm to update position and velocity (fix mvv/dpd), or position, velocity and temperature (fix mvv/edpd), or position, velocity and concentration (fix mvv/tdpd) for particles in the group each timestep.

The modified velocity-Verlet (MVV) algorithm aims to improve the stability of the time integrator by using an extrapolated version of the velocity for the force evaluation:

$$
\begin{aligned}
v(t+\frac{\Delta t}{2}) =& v(t) + \frac{\Delta t}{2} \cdot a(t) \\
r(t+\Delta t) =& r(t) + \Delta t \cdot v(t+\frac{\Delta t}{2}) \\
a(t+\Delta t) =& \frac{1}{m} \cdot F\left[r(t+\Delta t), v(t) + \lambda \cdot \Delta t \cdot a(t)\right] \\
v(t+\Delta t) =& v(t+\frac{\Delta t}{2}) + \frac{\Delta t}{2} \cdot a(t+\Delta t)
\end{aligned}
$$

where the parameter $\lambda$ depends on the specific choice of DPD parameters, and needs to be tuned on a case-by-case basis. Specification of a *lambda* value is optional. If specified, the setting must be from 0.0 to 1.0. If not specified, a default value of 0.5 is used, which effectively reproduces the standard velocity-Verlet (VV) scheme. For more details, see *Groot*.

Fix *mvv/dpd* updates the position and velocity of each atom. It can be used with the *pair_style mdpd* command or other pair styles such as *pair dpd*.

Fix *mvv/edpd* updates the per-atom temperature, in addition to position and velocity, and must be used with the *pair_style edpd* command.

Fix *mvv/tdpd* updates the per-atom chemical concentration, in addition to position and velocity, and must be used with the *pair_style tdpd* command.

---

### 2.123.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.123.5 Restrictions

These fixes are part of the DPD-MESO package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Changed in version 29Aug2024.

This fix is incompatible with deformation controls that remap velocity, for instance the *remap v* option of *fix deform*.

### 2.123.6 Related commands

*pair_style mdpd*, *pair_style edpd*, *pair_style tdpd*

### 2.123.7 Default

The default value for the optional *lambda* parameter is 0.5.

---

**(Groot)** Groot and Warren, J Chem Phys, 107: 4423-4435 (1997). DOI: 10.1063/1.474784

## 2.124 fix neb command

### 2.124.1 Syntax

```
fix ID group-ID neb Kspring keyword value
```

- ID, group-ID are documented in *fix* command
- neb = style name of this fix command
- Kspring = spring constant for parallel nudging force (force/distance units or force units, see parallel keyword)
- zero or more keyword/value pairs may be appended
- keyword = *parallel* or *perp* or *end*

```
parallel value = neigh or ideal or equal
  neigh = parallel nudging force based on distance to neighbor replicas (Kspring = force/
↪distance units)
  ideal = parallel nudging force based on interpolated ideal position (Kspring = force␣
↪units)
  equal = parallel nudging force based on interpolated ideal position before climbing,␣
↪then interpolated ideal energy whilst climbing (Kspring = force units)
perp value = Kspring2
```

```
  Kspring2 = spring constant for perpendicular nudging force (force/distance units)
end values = estyle Kspring3
  estyle = first or last or last/efirst or last/efirst/middle
    first = apply force to first replica
    last = apply force to last replica
    last/efirst = apply force to last replica and set its target energy to that of first␣
↪replica
    last/efirst/middle = same as last/efirst plus prevent middle replicas having lower␣
↪energy than first replica
  Kspring3 = spring constant for target energy term (1/distance units)
```

### 2.124.2 Examples

```
fix 1 active neb 10.0
fix 2 all neb 1.0 perp 1.0 end last
fix 2 all neb 1.0 perp 1.0 end first 1.0 end last 1.0
fix 1 all neb 1.0 parallel ideal end last/efirst 1
```

### 2.124.3 Description

Add nudging forces to atoms in the group for a multi-replica simulation run via the *neb* command to perform a nudged elastic band (NEB) calculation for finding the transition state. Hi-level explanations of NEB are given with the *neb* command and on the *Howto replica* doc page. The fix neb command must be used with the "neb" command and defines how inter-replica nudging forces are computed. A NEB calculation is divided in two stages. In the first stage n replicas are relaxed toward a MEP until convergence. In the second stage, the climbing image scheme (see *(Henkelman2)*) is enabled, so that the replica having the highest energy relaxes toward the saddle point (i.e. the point of highest energy along the MEP), and a second relaxation is performed.

A key purpose of the nudging forces is to keep the replicas equally spaced. During the NEB calculation, the $3N$-length vector of interatomic force $F_i = -\nabla V$ for each replica $i$ is altered. For all intermediate replicas (i.e. for $1 < i < N$, except the climbing replica) the force vector becomes:

$$F_i = -\nabla V + (\nabla V \cdot T')T' + F_{\parallel} + F_{\perp}$$

T' is the unit "tangent" vector for replica $i$ and is a function of $R_i, R_{i-1}, R_{i+1}$, and the potential energy of the 3 replicas; it points roughly in the direction of $R_{i+i} - R_{i-1}$; see the *(Henkelman1)* paper for details. $R_i$ are the atomic coordinates of replica $i$; $R_{i-1}$ and $R_{i+1}$ are the coordinates of its neighbor replicas. The term $\nabla V \cdot T'$ is used to remove the component of the gradient parallel to the path which would tend to distribute the replica unevenly along the path. $F_{\parallel}$ is an artificial nudging force which is applied only in the tangent direction and which maintains the equal spacing between replicas (see below for more information). $F_{\perp}$ is an optional artificial spring which is applied in a direction perpendicular to the tangent direction and which prevent the paths from forming acute kinks (see below for more information).

In the second stage of the NEB calculation, the interatomic force $F_i$ for the climbing replica (the replica of highest energy after the first stage) is changed to:

$$F_i = -\nabla V + 2(\nabla V \cdot T')T' + F_{\perp}$$

and the relaxation procedure is continued to a new converged MEP.

---

The keyword *parallel* specifies how the parallel nudging force is computed. With a value of *neigh*, the parallel nudging force is computed as in *(Henkelman1)* by connecting each intermediate replica with the previous and the next image:

$$F_{\parallel} = Kspring \cdot \left(|R_{i+1} - R_i| - |R_i - R_{i-1}|\right)$$

Note that in this case the specified *Kspring* is in force/distance units.

With a value of *ideal*, the spring force is computed as suggested in *(WeinanE)*

$$F_{\parallel} = -Kspring \cdot (RD - RD_{ideal})/(2 \cdot meanDist)$$

where *RD* is the "reaction coordinate" see *neb* section, and $RD_{ideal}$ is the ideal *RD* for which all the images are equally spaced. I.e. $RD_{ideal} = (i-1) \cdot meanDist$ when the climbing replica is off, where *i* is the replica number). The *meanDist* is the average distance between replicas. Note that in this case the specified *Kspring* is in force units. When the climbing replica is on, $RD_{ideal}$ and *meanDist* are calculated separately each side of the climbing image. Note that the *ideal* form of nudging can often be more effective at keeping the replicas equally spaced before climbing, then equally spaced either side of the climbing image whilst climbing.

With a value of *equal* the spring force is computed as for *ideal* when the climbing replica is off, promoting equidistance. When the climbing replica is on, the spring force is computed to promote equidistant absolute differences in energy, rather than distance, each side of the climbing image:

$$F_{\parallel} = -Kspring \cdot (ED - ED_{ideal})/(2 \cdot meanEDist)$$

where *ED* is the cumulative sum of absolute energy differences:

$$ED = \sum_{i<N} |E(R_{i+1}) - E(R_i)| \,,$$

*meanEdist* is the average absolute energy difference between replicas up to the climbing image or from the climbing image to the final image, for images before or after the climbing image respectively. $ED_{ideal}$ is the corresponding cumulative sum of average absolute energy differences in each case, in close analogy to *ideal*. This form of nudging is to aid schemes which integrate forces along, or near to, NEB pathways such as *fix_pafi*.

---

The keyword *perp* specifies if and how a perpendicular nudging force is computed. It adds a spring force perpendicular to the path in order to prevent the path from becoming too strongly kinked. It can significantly improve the convergence of the NEB calculation when the resolution is poor. I.e. when few replicas are used; see *(Maras)* for details.

The perpendicular spring force is given by

$$F_{\perp} = K_{spring2} \cdot F(R_{i-1}, R_i, R_{i+1})(R_{i+1} + R_{i-1} - 2R_i)$$

where *Kspring2* is the specified value. $F(R_{i-1}, R_i, R_{i+1})$ is a smooth scalar function of the angle $R_{i-1}R_iR_{i+1}$. It is equal to 0.0 when the path is straight and is equal to 1 when the angle $R_{i-1}R_iR_{i+1}$ is acute. $F(R_{i-1}, R_i, R_{i+1})$ is defined in *(Jonsson)*.

If *Kspring2* is set to 0.0 (the default) then no perpendicular spring force is added.

---

By default, no additional forces act on the first and last replicas during the NEB relaxation, so these replicas simply relax toward their respective local minima. By using the keyword *end*, additional forces can be applied to the first and/or last replicas, to enable them to relax toward a MEP while constraining their energy E to the target energy ETarget.

If $E_{Target} > E$, the interatomic force $F_i$ for the specified replica becomes:

$$\begin{aligned} F_i &= -\nabla V + (\nabla V \cdot T' + (E - E_{Target}) \cdot K_{spring3})T', \qquad \text{when} \quad \nabla V \cdot T' < 0 \\ F_i &= -\nabla V + (\nabla V \cdot T' + (E_{Target} - E) \cdot K_{spring3})T', \qquad \text{when} \quad \nabla V \cdot T' > 0 \end{aligned}$$

The "spring" constant on the difference in energies is the specified *Kspring3* value.

When *estyle* is specified as *first*, the force is applied to the first replica. When *estyle* is specified as *last*, the force is applied to the last replica. Note that the *end* keyword can be used twice to add forces to both the first and last replicas.

For both these *estyle* settings, the target energy *ETarget* is set to the initial energy of the replica (at the start of the NEB calculation).

If the *estyle* is specified as *last/efirst* or *last/efirst/middle*, force is applied to the last replica, but the target energy *ETarget* is continuously set to the energy of the first replica, as it evolves during the NEB relaxation.

The difference between these two *estyle* options is as follows. When *estyle* is specified as *last/efirst*, no change is made to the inter-replica force applied to the intermediate replicas (neither first or last). If the initial path is too far from the MEP, an intermediate replica may relax "faster" and reach a lower energy than the last replica. In this case the intermediate replica will be relaxing toward its own local minima. This behavior can be prevented by specifying *estyle* as *last/efirst/middle* which will alter the inter-replica force applied to intermediate replicas by removing the contribution of the gradient to the inter-replica force. This will only be done if a particular intermediate replica has a lower energy than the first replica. This should effectively prevent the intermediate replicas from over-relaxing.

After converging a NEB calculation using an *estyle* of *last/efirst/middle*, you should check that all intermediate replicas have a larger energy than the first replica. If this is not the case, the path is probably not a MEP.

Finally, note that the last replica may never reach the target energy if it is stuck in a local minima which has a larger energy than the target energy.

### 2.124.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The forces due to this fix are imposed during an energy minimization, as invoked by the *minimize* command via the *neb* command.

### 2.124.5 Restrictions

This command can only be used if LAMMPS was built with the REPLICA package. See the *Build package* doc page for more info.

### 2.124.6 Related commands

*neb*

### 2.124.7 Default

The option defaults are parallel = neigh, perp = 0.0, ends is not specified (no inter-replica force on the end replicas).

---

**(Henkelman1)** Henkelman and Jonsson, J Chem Phys, 113, 9978-9985 (2000).

**(Henkelman2)** Henkelman, Uberuaga, Jonsson, J Chem Phys, 113, 9901-9904 (2000).

**(WeinanE)** E, Ren, Vanden-Eijnden, Phys Rev B, 66, 052301 (2002).

**(Jonsson)** Jonsson, Mills and Jacobsen, in Classical and Quantum Dynamics in Condensed Phase Simulations, edited by Berne, Ciccotti, and Coker World Scientific, Singapore, 1998, p 385.

**(Maras)** Maras, Trushin, Stukowski, Ala-Nissila, Jonsson, Comp Phys Comm, 205, 13-21 (2016).

# 2.125 fix neb/spin command

## 2.125.1 Syntax

```
fix ID group-ID neb/spin Kspring
```

- ID, group-ID are documented in *fix* command
- neb/spin = style name of this fix command

```
Kspring = spring constant for parallel nudging force
(force/distance units or force units, see parallel keyword)
```

## 2.125.2 Examples

```
fix 1 active neb/spin 1.0
```

## 2.125.3 Description

Add nudging forces to spins in the group for a multi-replica simulation run via the *neb/spin* command to perform a geodesic nudged elastic band (GNEB) calculation for finding the transition state. Hi-level explanations of GNEB are given with the *neb/spin* command and on the *Howto replica* doc page. The fix neb/spin command must be used with the "neb/spin" command and defines how inter-replica nudging forces are computed. A GNEB calculation is divided in two stages. In the first stage n replicas are relaxed toward a MEP until convergence. In the second stage, the climbing image scheme is enabled, so that the replica having the highest energy relaxes toward the saddle point (i.e. the point of highest energy along the MEP), and a second relaxation is performed.

The nudging forces are calculated as explained in *(Bessarab)*). See this reference for more explanation about their expression.

## 2.125.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The forces due to this fix are imposed during an energy minimization, as invoked by the *minimize* command via the *neb/spin* command.

## 2.125.5 Restrictions

This command can only be used if LAMMPS was built with the SPIN package. See the *Build package* doc page for more info.

## 2.125.6 Related commands

*neb_spin*

## 2.125.7 Default

none

---

**(Bessarab)** Bessarab, Uzdin, Jonsson, Comp Phys Comm, 196, 335-347 (2015).

# 2.126 fix neighbor/swap command

## 2.126.1 Syntax

```
fix ID group-ID neighbor/swap N X seed T R0 voro-ID keyword values ...
```

- ID, group-ID are documented in *fix* command
- neighbor/swap = style name of this fix command
- N = invoke this fix every N steps
- X = number of swaps to attempt every N steps
- seed = random # seed (positive integer)
- T = scaling temperature of the MC swaps (temperature units)
- R0 = scaling swap probability of the MC swaps (distance units)
- voro-ID = valid voronoi compute id (compute voronoi/atom)
- one or more keyword/value pairs may be appended to args
- keywords *types* and *diff* are mutually exclusive, but one must be specified
- keyword = *types* or *diff* or *ke* or *region* or *rates*

```
types values = two or more atom types (Integers in range [1,Ntypes] or type labels)
diff values = one atom type
ke value = yes or no
  yes = kinetic energy is conserved after atom swaps
  no = no conservation of kinetic energy after atom swaps
region value = region-ID
  region-ID = ID of region to use as an exchange/move volume
rates values = V1 V2 . . . Vntypes values to conduct variable diffusion for␣
↪different atom types (unitless)
```

## 2.126.2 Examples

```
compute voroN all voronoi/atom neighbors yes
fix mc all neighbor/swap 10 160 15238 1000.0 3.0 voroN diff 2
fix myFix all neighbor/swap 100 1 12345 298.0 3.0 voroN region my_swap_region types 5 6
fix kmc all neighbor/swap 1 100 345 1.0 3.0 voroN diff 3 rates 3 1 6
```

## 2.126.3 Description

Added in version 22Jul2025.

This fix performs Monte-Carlo (MC) evaluations to enable kinetic Monte Carlo (kMC)-type behavior during MD simulation by allowing neighboring atoms to swap their positions. In contrast to the *fix atom/swap* command which swaps pairs of atoms anywhere in the simulation domain, the restriction of the MC swapping to neighbors enables a hybrid MD/kMC-like simulation.

Neighboring atoms are defined by using a Voronoi tesselation performed by the *compute voronoi/atom* command. Two atoms are neighbors if their Voronoi cells share a common face (3d) or edge (2d).

The selection of a swap neighbor is made using a distance-based criterion for weighting the selection probability of each swap, in the same manner as kMC selects a next event using relative probabilities. The acceptance or rejection of each swap is determined via the Metropolis criterion after evaluating the change in system energy due to the swap.

A detailed explanation of the original implementation of this algorithm can be found in *(Tavenner 2023)* where it was used to simulated accelerated diffusion in an MD context.

Simulating inherently kinetically-limited behaviors which rely on rare events (such as atomic diffusion in a solid) is challenging for traditional MD since its relatively short timescale will not naturally sample many events. This fix addresses this challenge by allowing rare neighbor hopping events to be sampled in a kMC-like fashion at a much faster rate (set by the specified $N$ and $X$ parameters). This enables the processes of atomic diffusion to be approximated during an MD simulation, effectively decoupling the MD atomic vibrational timescale and the atomic hopping (kMC event) timescale.

The algorithm implemented by this fix is as follows:

- The MD simulation is paused every $N$ steps
- A Voronoi tesselation is performed for the current atom configuration.
- Then $X$ atom swaps are attempted, one after the other.
- For each swap, an atom $I$ is selected randomly from the list of atom types specified by either the *types* or *diff* keywords.
- One of $I$'s Voronoi neighbors $J$ is selected using the distance-weighted probability for each neighbor detailed below.
- The $I$,$J$ atom IDs are communicated to all processors so that a global energy evaluation can be performed for the post-swap state of the system.
- The swap is accepted or rejected based on the Metropolis criterion using the energy change of the system and the specified temperature $T$.

**Note**

To run an MC-only simulation (no MD), you should define no time-integration fix, set the *thermo* command to 1, set $N$ to 1, and set $X$ small enough to see the MC evolution of the system. But if $X$ is too small, the overhead at the start and stop of MC moves each timestep will slow down the simulation.

Here are a few comments on the computational cost of the swapping algorithm.

1. The cost of a global energy evaluation is similar to that of an MD timestep.
2. Similar to other MC algorithms in LAMMPS, improved parallel efficiency is achieved with a smaller number of atoms per processor than would typically be used in an standard MD simulation. This is because the per-energy evaluation cost increases relative to the balance of MD/MC steps as indicated by 1., but the communication cost remains relatively constant for a given number of MD steps.
3. The MC portion of the simulation will run dramatically slower if the pair style uses different cutoffs for different atom types (or type pairs). This is because each atom swap then requires a rebuild of the neighbor list to ensure the post-swap global energy can be computed correctly.

Limitations are imposed on selection of $I$,$J$ atom pairs to avoid swapping of atoms which are outside of a reasonable cutoff (e.g. due to a Voronoi tesselation near free surfaces) though the use of a distance-weighted probability scaling.

---

This section gives more details on other arguments and keywords.

The random number generator (RNG) used by all the processors for MC operations is initialized with the specified *seed*.

The distance-based probability is weighted by the specified *R0* which sets the radius $r_0$ in this formula

$$p_{ij} = e^{(\frac{r_{ij}}{r_0})^2}$$

where $p_{ij}$ is the probability of selecting atom $j$ to swap with atom $i$. Typically, a value for *R0* around the average nearest-neighbor spacing is appropriate. Since this is simply a probability weighting, the swapping behavior is not very sensitive to the exact value of *R0*.

The required *voro-ID* value is the compute-ID of a *compute voronoi/atom* command like this:

```
compute compute-ID group-ID voronoi/atom neighbors yes
```

It must return per-atom list of valid neighbor IDs as in the *compute voronoi/atom* command.

The keyword *types* takes two or more atom types as its values. Only atoms *I* of the first atom type will be selected. Only atoms *J* of the remaining atom types will be considered as potential swap partners.

The keyword *diff* take a single atom type as its value. Only atoms *I* of the that atom type will be selected. Atoms *J* of all remaining atom types will be considered as potential swap partners. This includes the atom type specified with the *diff* keyword to account for self-diffusive hops between two atoms of the same type.

Note that the *neighbors yes* option must be enabled for use with this fix. The group-ID should include all the atoms which this fix will potentially select. I.e. the group-ID used in the voronoi compute should include the same atoms as that indicated by the *types* keyword. If the *diff* keyword is used, the group-ID should include atoms of all types in the simulation.

The keyword *ke* takes *yes* (default) or *no* as its value. It two atoms are swapped with different masses, then a value of *yes* will rescale their respective velocities to conserve the kinetic energy of the system. A value of *no* will perform no rescaling, so that kinetic energy is not conserved. See the restriction on this keyword below.

The *region* keyword takes a *region-ID* as its value. If specified, then only atoms *I* and *J* within the geometric region will be considered as swap partners. See the *region* command for details. This means the group-ID for the *compute voronoi/atom* command also need only contain atoms within the region.

The keyword *rates* can modify the swap rate based on the type of atom *J*. Ntype values must be specified, where Ntype = the number of atom types in the system. Each value is used to scale the probability weighting given by the equation above. In the third example command above, a simulation has 3 atoms types. Atom *I*s of type 1 are eligible for swapping. Swaps may occur with atom *J*s of all 3 types. Assuming all *J* atoms are equidistant from an atom *I*, *J* atoms of type 1 will be 3x more likely to be selected as a swap partner than atoms of type 2. And *J* atoms of type 3 will be 6.5x more likely to be selected than atoms of type 2. If the *rates* keyword is not used, all atom types will be treated with the same probability during selection of swap attempts.

---

### 2.126.4 Dump image info

Added in version 11Feb2026.

Fix *neighbor/swap* supports the *fix* keyword of *dump image*. The fix will pass geometry information about atoms involved in a swap to *dump image* so that these atoms can be highlighted in the visualization as additional spheres. For how long those additional spheres will be shown depends on the value of the *vizsteps* setting (default is 1000) which can be changed by using the *fix_modify command*. If an atom is involved in multiple swaps, the check on showing the additional graphics depends on the timestep of its last swap.

The color of the additional spheres is by default that of the atom type *before* the swap when using color styles "type" or "element". With color style "const" the default value of "white" can be changed using *dump_modify fcolor*. The transparency is by default fully opaque and can be changed with *dump_modify ftrans*.

The *fflag1* setting of *dump image fix* has no effect.

The *fflag2* setting allows you to set the radius of the added spheres, since the radius is set to zero internally.

---

### 2.126.5 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the state of the fix to *binary restart files*. This includes information about the random number generator seed, the next timestep for MC exchanges, and the number of exchange attempts and successes. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

None of the *fix_modify* options are relevant to this fix.

This fix computes a global vector of length 2, which can be accessed by various *output commands*. The vector values are the following global cumulative quantities:

1. swap attempts
2. swap accepts

The vector values calculated by this fix are "intensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.126.6 Restrictions

This fix is part of the MC package. It is only enabled if LAMMPS was built with that package. See the *Build package* doc page for more info. Also this fix requires that the *VORONOI package* is installed, otherwise the fix will not be compiled.

The *compute voronoi/atom* command referenced by the required voro-ID must return neighboring atoms as illustrated in the examples above.

If this fix is used with systems that do not have per-type masses (e.g. atom style sphere), the *ke* keyword must be set to *off* since the implemented algorithm will not be able to re-scale velocities properly.

### 2.126.7 Related commands

*fix nvt*, *compute voronoi/atom delete_atoms*, *fix gcmc*, *fix atom/swap*, *fix mol/swap*, *fix sgcmc*

### 2.126.8 Default

The option defaults are *ke* = yes and *rates* = 1 for all atom types.

---

**(Tavenner 2023) J Tavenner, M Mendelev, J Lawson, Computational** Materials Science, 218, 111929 (2023).

## 2.127 fix nvt command

Accelerator Variants: *nvt/gpu*, *nvt/intel*, *nvt/kk*, *nvt/omp*

## 2.128 fix npt command

Accelerator Variants: *npt/gpu*, *npt/intel*, *npt/kk*, *npt/omp*

# 2.129 fix nph command

Accelerator Variants: *nph/kk*, *nph/omp*

## 2.129.1 Syntax

```
fix ID group-ID style_name keyword value ...
```

- ID, group-ID are documented in *fix* command
- style_name = *nvt* or *npt* or *nph*
- one or more keyword/value pairs may be appended

```
keyword = temp or iso or aniso or tri or x or y or z or xy or yz or xz or couple or␣
↪tchain or pchain or mtk or tloop or ploop or nreset or drag or ptemp or dilate␣
↪or scalexy or scaleyz or scalexz or flip or isochoric or fixedpoint or update
  temp values = Tstart Tstop Tdamp
    Tstart,Tstop = external temperature at start/end of run
    Tdamp = temperature damping parameter (time units)
  iso or aniso or tri values = Pstart Pstop Pdamp
    Pstart,Pstop = scalar external pressure at start/end of run (pressure units)
    Pdamp = pressure damping parameter (time units)
  x or y or z or xy or yz or xz values = Pstart Pstop Pdamp
    Pstart,Pstop = external stress tensor component at start/end of run (pressure␣
↪units)
    Pdamp = stress damping parameter (time units)
  couple = none or xyz or xy or yz or xz
  tchain value = N
    N = length of thermostat chain (1 = single thermostat)
  pchain value = N
    N length of thermostat chain on barostat (0 = no thermostat)
  mtk value = yes or no = add in MTK adjustment term or not
  tloop value = M
    M = number of sub-cycles to perform on thermostat
  ploop value = M
    M = number of sub-cycles to perform on barostat thermostat
  nreset value = reset reference cell every this many timesteps
  drag value = Df
    Df = drag factor added to barostat/thermostat (0.0 = no drag)
  ptemp value = Ttarget
    Ttarget = target temperature for barostat
  dilate value = dilate-group-ID
    dilate-group-ID = only dilate atoms in this group due to barostat volume changes
  scalexy value = yes or no = scale xy with ly
  scaleyz value = yes or no = scale yz with lz
  scalexz value = yes or no = scale xz with lz
  flip value = yes or no = allow or disallow box flips when it becomes highly skewed
  isochoric = x or y or z or xy or yz or xz
  fixedpoint values = x y z
    x,y,z = perform barostat dilation/contraction around this point (distance units)
  update value = dipole or dipole/dlm
    dipole = update dipole orientation (only for sphere variants)
    dipole/dlm = use DLM integrator to update dipole orientation (only for sphere␣
↪variants)
```

### 2.129.2 Examples

```
fix 1 all nvt temp 300.0 300.0 100.0
fix 1 water npt temp 300.0 300.0 100.0 iso 0.0 0.0 1000.0
fix 2 jello npt temp 300.0 300.0 100.0 tri 5.0 5.0 1000.0
fix 2 ice nph x 1.0 1.0 0.5 y 2.0 2.0 0.5 z 3.0 3.0 0.5 yz 0.1 0.1 0.5 xz 0.2 0.2 0.5 xy␣
↪0.3 0.3 0.5 nreset 1000
```

### 2.129.3 Description

These commands perform time integration on Nose-Hoover style non-Hamiltonian equations of motion which are designed to generate positions and velocities sampled from the canonical (nvt), isothermal-isobaric (npt), and isenthalpic (nph) ensembles. This updates the position and velocity for atoms in the group each timestep.

The thermostatting and barostatting is achieved by adding some dynamic variables which are coupled to the particle velocities (thermostatting) and simulation domain dimensions (barostatting). In addition to basic thermostatting and barostatting, these fixes can also create a chain of thermostats coupled to the particle thermostat, and another chain of thermostats coupled to the barostat variables. The barostat can be coupled to the overall box volume, or to individual dimensions, including the *xy*, *xz* and *yz* tilt dimensions. The external pressure of the barostat can be specified as either a scalar pressure (isobaric ensemble) or as components of a symmetric stress tensor (constant stress ensemble). When used correctly, the time-averaged temperature and stress tensor of the particles will match the target values specified by Tstart/Tstop and Pstart/Pstop.

The equations of motion used are those of Shinoda et al in *(Shinoda)*, which combine the hydrostatic equations of Martyna, Tobias and Klein in *(Martyna)* with the strain energy proposed by Parrinello and Rahman in *(Parrinello)*. The time integration schemes closely follow the time-reversible measure-preserving Verlet and rRESPA integrators derived by Tuckerman et al in *(Tuckerman)*.

---

The thermostat parameters for fix styles *nvt* and *npt* are specified using the *temp* keyword. Other thermostat-related keywords are *tchain*, *tloop* and *drag*, which are discussed below.

The thermostat is applied to only the translational degrees of freedom for the particles. The translational degrees of freedom can also have a bias velocity removed before thermostatting takes place; see the description below. The desired temperature at each timestep is a ramped value during the run from *Tstart* to *Tstop*. The *Tdamp* parameter is specified in time units and determines how rapidly the temperature is relaxed. For example, a value of 10.0 means to relax the temperature in a timespan of (roughly) 10 time units (e.g. $\tau$ or fs or ps - see the *units* command). The atoms in the fix group are the only ones whose velocities and positions are updated by the velocity/position update portion of the integration.

**Note**

A Nose-Hoover thermostat will not work well for arbitrary values of *Tdamp*. If *Tdamp* is too small, the temperature can fluctuate wildly; if it is too large, the temperature will take a very long time to equilibrate. A good choice for many models is a *Tdamp* of around 100 timesteps. Note that this is NOT the same as 100 time units for most *units* settings. A simple way to ensure this, is via using an *immediate variable* expression accessing the thermo property 'dt', which is the length of the time step. Example:

```
fix 1 all nvt temp 300.0 300.0 $(100.0*dt)
```

---

The barostat parameters for fix styles *npt* and *nph* is specified using one or more of the *iso*, *aniso*, *tri*, *x*, *y*, *z*, *xy*, *xz*, *yz*, and *couple* keywords. These keywords give you the ability to specify all 6 components of an external stress tensor,

and to couple various of these components together so that the dimensions they represent are varied together during a constant-pressure simulation.

Other barostat-related keywords are *pchain*, *mtk*, *ploop*, *nreset*, *drag*, and *dilate*, which are discussed below.

Orthogonal simulation boxes have 3 adjustable dimensions (x,y,z). Triclinic (non-orthogonal) simulation boxes have 6 adjustable dimensions (x,y,z,xy,xz,yz). The *create_box*, *read data*, and *read_restart* commands specify whether the simulation box is orthogonal or non-orthogonal (triclinic) and explain the meaning of the xy,xz,yz tilt factors.

The target pressures for each of the 6 components of the stress tensor can be specified independently via the *x*, *y*, *z*, *xy*, *xz*, *yz* keywords, which correspond to the 6 simulation box dimensions. For each component, the external pressure or tensor component at each timestep is a ramped value during the run from *Pstart* to *Pstop*. If a target pressure is specified for a component, then the corresponding box dimension will change during a simulation. For example, if the *y* keyword is used, the y-box length will change. If the *xy* keyword is used, the xy tilt factor will change. A box dimension will not change if that component is not specified, although you have the option to change that dimension via the *fix deform* command.

Note that in order to use the *xy*, *xz*, or *yz* keywords, the simulation box must be triclinic, even if its initial tilt factors are 0.0.

For all barostat keywords, the *Pdamp* parameter operates like the *Tdamp* parameter, determining the time scale on which pressure is relaxed. For example, a value of 10.0 means to relax the pressure in a timespan of (roughly) 10 time units (e.g. $\tau$ or fs or ps - see the *units* command).

**Note**

A Nose-Hoover barostat will not work well for arbitrary values of *Pdamp*. If *Pdamp* is too small, the pressure and volume can fluctuate wildly; if it is too large, the pressure will take a very long time to equilibrate. A good choice for many models is a *Pdamp* of around 1000 timesteps. However, note that *Pdamp* is specified in time units, and that timesteps are NOT the same as time units for most *units* settings.

The relaxation rate of the barostat is set by its inertia $W$:

$$W = (N+1)k_B T_{\text{target}} P_{\text{damp}}^2$$

where $N$ is the number of atoms, $k_B$ is the Boltzmann constant, and $T_{\text{target}}$ is the target temperature of the barostat *(Martyna)*. If a thermostat is defined, $T_{\text{target}}$ is the target temperature of the thermostat. If a thermostat is not defined, $T_{\text{target}}$ is set to the current temperature of the system when the barostat is initialized. If this temperature is too low the simulation will quit with an error. Note: in previous versions of LAMMPS, $T_{\text{target}}$ would default to a value of 1.0 for *lj* units and 300.0 otherwise if the system had a temperature of exactly zero.

If a thermostat is not specified by this fix, $T_{\text{target}}$ can be manually specified using the *Ptemp* parameter. This may be useful if the barostat is initialized when the current temperature does not reflect the steady state temperature of the system. This keyword may also be useful in athermal simulations where the temperature is not well defined.

Regardless of what atoms are in the fix group (the only atoms which are time integrated), a global pressure or stress tensor is computed for all atoms. Similarly, when the size of the simulation box is changed, all atoms are re-scaled to new positions, unless the keyword *dilate* is specified with a *dilate-group-ID* for a group that represents a subset of the atoms. This can be useful, for example, to leave the coordinates of atoms in a solid substrate unchanged and controlling the pressure of a surrounding fluid. This option should be used with care, since it can be unphysical to dilate some atoms and not others, because it can introduce large, instantaneous displacements between a pair of atoms (one dilated, one not) that are far from the dilation origin. Also note that for atoms not in the fix group, a separate time integration fix like *fix nve* or *fix nvt* can be used on them, independent of whether they are dilated or not.

---

The *couple* keyword allows two or three of the diagonal components of the pressure tensor to be "coupled" together. The value specified with the keyword determines which are coupled. For example, *xz* means the *Pxx* and *Pzz* components

of the stress tensor are coupled. *Xyz* means all 3 diagonal components are coupled. Coupling means two things: the instantaneous stress will be computed as an average of the corresponding diagonal components, and the coupled box dimensions will be changed together in lockstep, meaning coupled dimensions will be dilated or contracted by the same percentage every timestep. The *Pstart*, *Pstop*, *Pdamp* parameters for any coupled dimensions must be identical. *Couple xyz* can be used for a 2d simulation; the *z* dimension is simply ignored.

---

The *iso*, *aniso*, and *tri* keywords are simply shortcuts that are equivalent to specifying several other keywords together.

The keyword *iso* means couple all 3 diagonal components together when pressure is computed (hydrostatic pressure), and dilate/contract the dimensions together. Using "iso Pstart Pstop Pdamp" is the same as specifying these 4 keywords:

```
x Pstart Pstop Pdamp
y Pstart Pstop Pdamp
z Pstart Pstop Pdamp
couple xyz
```

The keyword *aniso* means *x*, *y*, and *z* dimensions are controlled independently using the *Pxx*, *Pyy*, and *Pzz* components of the stress tensor as the driving forces, and the specified scalar external pressure. Using "aniso Pstart Pstop Pdamp" is the same as specifying these 4 keywords:

```
x Pstart Pstop Pdamp
y Pstart Pstop Pdamp
z Pstart Pstop Pdamp
couple none
```

The keyword *tri* means *x*, *y*, *z*, *xy*, *xz*, and *yz* dimensions are controlled independently using their individual stress components as the driving forces, and the specified scalar pressure as the external normal stress. Using "tri Pstart Pstop Pdamp" is the same as specifying these 7 keywords:

```
x Pstart Pstop Pdamp
y Pstart Pstop Pdamp
z Pstart Pstop Pdamp
xy 0.0 0.0 Pdamp
yz 0.0 0.0 Pdamp
xz 0.0 0.0 Pdamp
couple none
```

---

In some cases (e.g. for solids) the pressure (volume) and/or temperature of the system can oscillate undesirably when a Nose/Hoover barostat and thermostat is applied. The optional *drag* keyword will damp these oscillations, although it alters the Nose/Hoover equations. A value of 0.0 (no drag) leaves the Nose/Hoover formalism unchanged. A non-zero value adds a drag term; the larger the value specified, the greater the damping effect. Performing a short run and monitoring the pressure and temperature is the best way to determine if the drag term is working. Typically a value between 0.2 to 2.0 is sufficient to damp oscillations after a few periods. Note that use of the drag keyword will interfere with energy conservation and will also change the distribution of positions and velocities so that they do not correspond to the nominal NVT, NPT, or NPH ensembles.

An alternative way to control initial oscillations is to use chain thermostats. The keyword *tchain* determines the number of thermostats in the particle thermostat. A value of 1 corresponds to the original Nose-Hoover thermostat. The keyword *pchain* specifies the number of thermostats in the chain thermostatting the barostat degrees of freedom. A value of 0 corresponds to no thermostatting of the barostat variables.

The *mtk* keyword controls whether or not the correction terms due to Martyna, Tuckerman, and Klein are included in the equations of motion *(Martyna)*. Specifying *no* reproduces the original Hoover barostat, whose volume probability

distribution function differs from the true NPT and NPH ensembles by a factor of 1/V. Hence using *yes* is more correct, but in many cases the difference is negligible.

The keyword *tloop* can be used to improve the accuracy of integration scheme at little extra cost. The initial and final updates of the thermostat variables are broken up into *tloop* sub-steps, each of length *dt*/*tloop*. This corresponds to using a first-order Suzuki-Yoshida scheme *(Tuckerman)*. The keyword *ploop* does the same thing for the barostat thermostat.

The keyword *nreset* controls how often the reference dimensions used to define the strain energy are reset. If this keyword is not used, or is given a value of zero, then the reference dimensions are set to those of the initial simulation domain and are never changed. If the simulation domain changes significantly during the simulation, then the final average pressure tensor will differ significantly from the specified values of the external stress tensor. A value of *nstep* means that every *nstep* timesteps, the reference dimensions are set to those of the current simulation domain.

The *scaleyz*, *scalexz*, and *scalexy* keywords control whether or not the corresponding tilt factors are scaled with the associated box dimensions when barostatting triclinic periodic cells. The default values *yes* will turn on scaling, which corresponds to adjusting the linear dimensions of the cell while preserving its shape. Choosing *no* ensures that the tilt factors are not scaled with the box dimensions. See below for restrictions and default values in different situations. In older versions of LAMMPS, scaling of tilt factors was not performed. The old behavior can be recovered by setting all three scale keywords to *no*.

The *flip* keyword allows the tilt factors for a triclinic box to exceed half the distance of the parallel box length, as discussed below. If the *flip* value is set to *yes*, the bound is enforced by flipping the box when it is exceeded. If the *flip* value is set to *no*, the tilt will continue to change without flipping. Note that if applied stress induces large deformations (e.g. in a liquid), this means the box shape can tilt dramatically and LAMMPS will run less efficiently, due to the large volume of communication needed to acquire ghost atoms around a processor's irregular-shaped subdomain. For extreme values of tilt, LAMMPS may also lose atoms and generate an error.

Added in version 30Mar2026.

The *isochoric* keyword allows to maintain constant volume when barostatting up to two dimensions with this fix. The values following the isochoric keyword indicates the dimensions to use in that regard: "x" indicates the x dimension, "yz" (no space) indicates the y and z dimensions, etc. The selected dimensions are scaled to compensate the strain induced by the barostat and keep the system at a constant volume (or area in 2d). It is not possible to use this keyword if *all* dimensions are coupled to a barostat. In the case of 2d simulations, only x and y dimensions can be used to maintain a constant plane area. If you want to perform strain with constant volume, the *fix deform* command using *volume* keyword is more likely to suit your needs.

**Note**

If large strains are caused by the barostat because the initial configuration is far from pressure equilibrium or equilibrated too fast, the system will see large strains on the other dimensions as well. It is recommended to perform preliminary NPT equilibration if necessary using standard NPT simulations.

The *fixedpoint* keyword specifies the fixed point for barostat volume changes. By default, it is the center of the box. Whatever point is chosen will not move during the simulation. For example, if the lower periodic boundaries pass through (0,0,0), and this point is provided to *fixedpoint*, then the lower periodic boundaries will remain at (0,0,0), while the upper periodic boundaries will move twice as far. In all cases, the particle trajectories are unaffected by the chosen value, except for a time-dependent constant translation of positions.

If the *update* keyword is used with the *dipole* value, then the orientation of the dipole moment of each particle is also updated during the time integration. This option should be used for models where a dipole moment is assigned to finite-size particles, e.g. spheroids via use of the *atom_style hybrid sphere dipole* command.

The default dipole orientation integrator can be changed to the Dullweber-Leimkuhler-McLachlan integration scheme *(Dullweber)* when using *update* with the value *dipole/dlm*. This integrator is symplectic and time-reversible, giving better energy conservation and allows slightly longer timesteps at only a small additional computational cost.

**Note**

Using a barostat coupled to tilt dimensions *xy*, *xz*, *yz* can sometimes result in arbitrarily large values of the tilt dimensions, i.e. a dramatically deformed simulation box. LAMMPS allows the tilt factors to grow a small amount beyond the normal limit of half the box length (0.6 times the box length), and then performs a box "flip" to an equivalent periodic cell. See the discussion of the *flip* keyword above, to allow this bound to be exceeded, if desired.

The flip operation is described in more detail in the page for *fix deform*. Both the barostat dynamics and the atom trajectories are unaffected by this operation. However, if a tilt factor is incremented by a large amount (1.5 times the box length) on a single timestep, LAMMPS can not accommodate this event and will terminate the simulation with an error. This error typically indicates that there is something badly wrong with how the simulation was constructed, such as specifying values of *Pstart* that are too far from the current stress value, or specifying a timestep that is too large. Triclinic barostatting should be used with care. This also is true for other barostat styles, although they tend to be more forgiving of insults. In particular, it is important to recognize that equilibrium liquids can not support a shear stress and that equilibrium solids can not support shear stresses that exceed the yield stress.

One exception to this rule is if the first dimension in the tilt factor (x for xy) is non-periodic. In that case, the limits on the tilt factor are not enforced, since flipping the box in that dimension does not change the atom positions due to non-periodicity. In this mode, if you tilt the system to extreme angles, the simulation will simply become inefficient due to the highly skewed simulation box.

**Note**

Unlike the *fix temp/berendsen* command which performs thermostatting but NO time integration, these fixes perform thermostatting/barostatting AND time integration. Thus you should not use any other time integration fix, such as *fix nve* on atoms to which this fix is applied. Likewise, fix nvt and fix npt should not normally be used on atoms that also have their temperature controlled by another fix - e.g. by *fix langevin* or *fix temp/rescale* commands.

See the *Howto thermostat* and *Howto barostat* doc pages for a discussion of different ways to compute temperature and perform thermostatting and barostatting.

---

These fixes compute a temperature and pressure each timestep. To do this, the thermostat and barostat fixes create their own computes of style "temp" and "pressure", as if one of these sets of commands had been issued:

For fix nvt:

```
compute fix-ID_temp group-ID temp
```

For fix npt and fix nph:

```
compute fix-ID_temp all temp
compute fix-ID_press all pressure fix-ID_temp
```

For fix nvt, the group for the new temperature compute is the same as the fix group. For fix npt and fix nph, the group for both the new temperature and pressure compute is "all" since pressure is computed for the entire system. In the case of fix nph, the temperature compute is not used for thermostatting, but just for a kinetic-energy contribution to the pressure. See the *compute temp* and *compute pressure* commands for details. Note that the IDs of the new computes are the fix-ID + underscore + "temp" or fix_ID + underscore + "press".

Note that these are NOT the computes used by thermodynamic output (see the *thermo_style* command) with ID = *thermo_temp* and *thermo_press*. This means you can change the attributes of these fix's temperature or pressure via the *compute_modify* command. Or you can print this temperature or pressure during thermodynamic output

via the *thermo_style custom* command using the appropriate compute-ID. It also means that changing attributes of *thermo_temp* or *thermo_press* will have no effect on this fix.

Like other fixes that perform thermostatting, this fix can be used with *compute commands* that remove a "bias" from the atom velocities. E.g. to apply the thermostat only to atoms within a spatial *region*, or to remove the center-of-mass velocity from a group of atoms, or to remove the x-component of velocity from the calculation.

This is not done by default, but only if the *fix_modify* command is used to assign a temperature compute to this fix that includes such a bias term. See the doc pages for individual *compute temp commands* to determine which ones include a bias. In this case, the thermostat works in the following manner: bias is removed from each atom, thermostatting is performed on the remaining thermal degrees of freedom, and the bias is added back in.

---

These fixes can be used with either the *verlet* or *respa integrators*. When using one of the barostat fixes with *respa*, LAMMPS uses an integrator constructed according to the following factorization of the Liouville propagator (for two rRESPA levels):

$$\begin{aligned}\exp\left(iL\Delta t\right) =& \hat{E}\exp\left(iL_{\text{T-baro}}\frac{\Delta t}{2}\right)\exp\left(iL_{\text{T-part}}\frac{\Delta t}{2}\right)\exp\left(iL_{\varepsilon,2}\frac{\Delta t}{2}\right)\exp\left(iL_2^{(2)}\frac{\Delta t}{2}\right)\\ &\times\left[\exp\left(iL_2^{(1)}\frac{\Delta t}{2n}\right)\exp\left(iL_{\varepsilon,1}\frac{\Delta t}{2n}\right)\exp\left(iL_1\frac{\Delta t}{n}\right)\exp\left(iL_{\varepsilon,1}\frac{\Delta t}{2n}\right)\exp\left(iL_2^{(1)}\frac{\Delta t}{2n}\right)\right]^n\\ &\times\exp\left(iL_2^{(2)}\frac{\Delta t}{2}\right)\exp\left(iL_{\varepsilon,2}\frac{\Delta t}{2}\right)\exp\left(iL_{\text{T-part}}\frac{\Delta t}{2}\right)\exp\left(iL_{\text{T-baro}}\frac{\Delta t}{2}\right)\\ &+\mathcal{O}\left(\Delta t^3\right)\end{aligned}$$

This factorization differs somewhat from that of Tuckerman et al, in that the barostat is only updated at the outermost rRESPA level, whereas Tuckerman's factorization requires splitting the pressure into pieces corresponding to the forces computed at each rRESPA level. In theory, the latter method will exhibit better numerical stability. In practice, because Pdamp is normally chosen to be a large multiple of the outermost rRESPA timestep, the barostat dynamics are not the limiting factor for numerical stability. Both factorizations are time-reversible and can be shown to preserve the phase space measure of the underlying non-Hamiltonian equations of motion.

**Note**

This implementation has been shown to conserve linear momentum up to machine precision under NVT dynamics. Under NPT dynamics, for a system with zero initial total linear momentum, the total momentum fluctuates close to zero. It may occasionally undergo brief excursions to non-negligible values, before returning close to zero. Over long simulations, this has the effect of causing the center-of-mass to undergo a slow random walk. This can be mitigated by resetting the momentum at infrequent intervals using the *fix momentum* command.

---

The fix npt and fix nph commands can be used with rigid bodies or mixtures of rigid bodies and non-rigid particles (e.g. solvent). But there are also *fix rigid/npt* and *fix rigid/nph* commands, which are typically a more natural choice. See the page for those commands for more discussion of the various ways to do this.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.129.4 Restart, fix_modify, output, run start/stop, minimize info

These fixes writes the state of all the thermostat and barostat variables to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The *fix_modify* *temp* and *press* options are supported by these fixes. You can use them to assign a *compute* you have defined to this fix which will be used in its thermostatting or barostatting procedure, as described above. If you do this, note that the kinetic energy derived from the compute temperature should be consistent with the virial term computed using all atoms for the pressure. LAMMPS will warn you if you choose to compute temperature on a subset of atoms.

**Note**

If both the *temp* and *press* keywords are used in a single thermo_modify command (or in two separate commands), then the order in which the keywords are specified is important. Note that a *pressure compute* defines its own temperature compute as an argument when it is specified. The *temp* keyword will override this (for the pressure compute being used by fix npt), but only if the *temp* keyword comes after the *press* keyword. If the *temp* keyword comes before the *press* keyword, then the new pressure compute specified by the *press* keyword will be unaffected by the *temp* setting.

The cumulative energy change in the system imposed by these fixes, via either thermostatting and/or barostatting, is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* page for details.

These fixes compute a global scalar which can be accessed by various *output commands*. The scalar is the same cumulative energy change due to this fix described in the previous paragraph. The scalar value calculated by this fix is "extensive".

These fixes compute also compute a global vector of quantities, which can be accessed by various *output commands*. The vector values are "intensive".

The vector stores internal Nose/Hoover thermostat and barostat variables. The number and meaning of the vector values depends on which fix is used and the settings for keywords *tchain* and *pchain*, which specify the number of Nose/Hoover chains for the thermostat and barostat. If no thermostatting is done, then *tchain* is 0. If no barostatting is done, then *pchain* is 0. In the following list, "ndof" is 0, 1, 3, or 6, and is the number of degrees of freedom in the barostat. Its value is 0 if no barostat is used, else its value is 6 if any off-diagonal stress tensor component is barostatted, else its value is 1 if *couple xyz* is used or *couple xy* for a 2d simulation, otherwise its value is 3.

The order of values in the global vector and their meaning is as follows. The notation means there are tchain values for eta, followed by tchain for eta_dot, followed by ndof for omega, etc:

- eta[tchain] = particle thermostat displacements (unitless)
- eta_dot[tchain] = particle thermostat velocities (1/time units)
- omega[ndof] = barostat displacements (unitless)
- omega_dot[ndof] = barostat velocities (1/time units)
- etap[pchain] = barostat thermostat displacements (unitless)
- etap_dot[pchain] = barostat thermostat velocities (1/time units)
- PE_eta[tchain] = potential energy of each particle thermostat displacement (energy units)

- KE_eta_dot[tchain] = kinetic energy of each particle thermostat velocity (energy units)
- PE_omega[ndof] = potential energy of each barostat displacement (energy units)
- KE_omega_dot[ndof] = kinetic energy of each barostat velocity (energy units)
- PE_etap[pchain] = potential energy of each barostat thermostat displacement (energy units)
- KE_etap_dot[pchain] = kinetic energy of each barostat thermostat velocity (energy units)
- PE_strain[1] = scalar strain energy (energy units)

Added in version 10Dec2025.

This fix supports automatically generated thermo column names when using *thermo_modify colname auto*. The thermo column names are "f_", followed by the fix ID, followed by a colon, followed by a keyword listed above, followed by an index for that keyword. Indices range from 1 to the number of values for that keyword. E.g., the first example in the Examples section above would print a thermo column name of "f_1:eta[1]", compared to the default column output name of "f_1[1]". Similarly, "f_1:eta_dot[1]" would be printed instead of the default "f_1[4].

These fixes can ramp their external temperature and pressure over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

These fixes are not invoked during *energy minimization*.

---

## 2.129.5 Restrictions

*X*, *y*, *z* cannot be barostatted if the associated dimension is not periodic. *Xy*, *xz*, and *yz* can only be barostatted if the simulation domain is triclinic and the second dimension in the keyword (*y* dimension in *xy*) is periodic. *Z*, *xz*, and *yz*, cannot be barostatted for 2D simulations. The *create_box*, *read data*, and *read_restart* commands specify whether the simulation box is orthogonal or non-orthogonal (triclinic) and explain the meaning of the xy,xz,yz tilt factors.

For the *temp* keyword, the final Tstop cannot be 0.0 since it would make the external T = 0.0 at some timestep during the simulation which is not allowed in the Nose/Hoover formulation.

The *scaleyz yes* and *scalexz yes* keyword/value pairs can not be used for 2D simulations. The *scaleyz yes*, *scalexz yes*, and *scalexy yes* options can only be used if the second dimension in the keyword is periodic, and if the tilt factor is not coupled to the barostat via keywords *tri*, *yz*, *xz*, and *xy*.

These fixes can be used with dynamic groups as defined by the *group* command. Likewise they can be used with groups to which atoms are added or deleted over time, e.g. a deposition simulation. However, the conservation properties of the thermostat and barostat are defined for systems with a static set of atoms. You may observe odd behavior if the atoms in a group vary dramatically over time or the atom count becomes very small.

## 2.129.6 Related commands

*fix nve*, *fix_modify*, *run_style*

## 2.129.7 Default

The keyword defaults are tchain = 3, pchain = 3, mtk = yes, tloop = 1, ploop = 1, nreset = 0, drag = 0.0, dilate = all, couple = none, flip = yes, scaleyz = scalexz = scalexy = yes if periodic in second dimension and not coupled to barostat, otherwise no.

---

**(Martyna)** Martyna, Tobias and Klein, J Chem Phys, 101, 4177 (1994).

**(Parrinello)** Parrinello and Rahman, J Appl Phys, 52, 7182 (1981).

**(Tuckerman)** Tuckerman, Alejandre, Lopez-Rendon, Jochim, and Martyna, J Phys A: Math Gen, 39, 5629 (2006).

**(Shinoda)** Shinoda, Shiga, and Mikami, Phys Rev B, 69, 134103 (2004).

**(Dullweber)** Dullweber, Leimkuhler and McLachlan, J Chem Phys, 107, 5840 (1997).

# 2.130 fix nvt/eff command

# 2.131 fix npt/eff command

# 2.132 fix nph/eff command

## 2.132.1 Syntax

```
fix ID group-ID style_name keyword value ...
```

- ID, group-ID are documented in *fix* command
- style_name = *nvt/eff* or *npt/eff* or *nph/eff*

  one or more keyword value pairs may be appended
  keyword = *temp* or *iso* or *aniso* or *tri* or *x* or *y* or *z* or *xy* or *yz* or *xz* or *couple* or *tchain* or *pchain* or *mtk* or *tloop* or *ploop* or *nreset* or *drag* or *dilate*
    *temp* values = Tstart Tstop Tdamp
      Tstart,Tstop = external temperature at start/end of run
      Tdamp = temperature damping parameter (time units)
    *iso* or *aniso* or *tri* values = Pstart Pstop Pdamp
      Pstart,Pstop = scalar external pressure at start/end of run (pressure units)
      Pdamp = pressure damping parameter (time units)
    *x* or *y* or *z* or *xy* or *yz* or *xz* values = Pstart Pstop Pdamp
      Pstart,Pstop = external stress tensor component at start/end of run (pressure units)
      Pdamp = stress damping parameter (time units)
    *couple* = *none* or *xyz* or *xy* or *yz* or *xz*
    *tchain* value = length of thermostat chain (1 = single thermostat)
    *pchain* values = length of thermostat chain on barostat (0 = no thermostat)
    *mtk* value = *yes* or *no* = add in MTK adjustment term or not
    *tloop* value = number of sub-cycles to perform on thermostat
    *ploop* value = number of sub-cycles to perform on barostat thermostat
    *nreset* value = reset reference cell every this many timesteps
    *drag* value = drag factor added to barostat/thermostat (0.0 = no drag)
    *dilate* value = *all* or *partial*

## 2.132.2 Examples

```
fix 1 all nvt/eff temp 300.0 300.0 0.1
fix 1 part npt/eff temp 300.0 300.0 0.1 iso 0.0 0.0 1.0
fix 2 part npt/eff temp 300.0 300.0 0.1 tri 5.0 5.0 1.0
fix 2 ice nph/eff x 1.0 1.0 0.5 y 2.0 2.0 0.5 z 3.0 3.0 0.5 yz 0.1 0.1 0.5 xz 0.2 0.2 0.5 xy 0.3 0.3 0.5 nreset 1000
```

### 2.132.3 Description

These commands perform time integration on Nose-Hoover style non-Hamiltonian equations of motion for nuclei and electrons in the group for the *electron force field* model. The fixes are designed to generate positions and velocities sampled from the canonical (nvt), isothermal-isobaric (npt), and isenthalpic (nph) ensembles. This is achieved by adding some dynamic variables which are coupled to the particle velocities (thermostatting) and simulation domain dimensions (barostatting). In addition to basic thermostatting and barostatting, these fixes can also create a chain of thermostats coupled to the particle thermostat, and another chain of thermostats coupled to the barostat variables. The barostat can be coupled to the overall box volume, or to individual dimensions, including the *xy*, *xz* and *yz* tilt dimensions. The external pressure of the barostat can be specified as either a scalar pressure (isobaric ensemble) or as components of a symmetric stress tensor (constant stress ensemble). When used correctly, the time-averaged temperature and stress tensor of the particles will match the target values specified by Tstart/Tstop and Pstart/Pstop.

The operation of these fixes is exactly like that described by the *fix nvt, npt, and nph* commands, except that the radius and radial velocity of electrons are also updated. Likewise the temperature and pressure calculated by the fix, using the computes it creates (as discussed in the *fix nvt, npt, and nph* doc page), are performed with computes that include the eFF contribution to the temperature or kinetic energy from the electron radial velocity.

**Note**

there are two different pressures that can be reported for eFF when defining the pair_style (see *pair eff/cut* to understand these settings), one (default) that considers electrons do not contribute radial virial components (i.e. electrons treated as incompressible 'rigid' spheres) and one that does. The radial electronic contributions to the virials are only tallied if the flexible pressure option is set, and this will affect both global and per-atom quantities. In principle, the true pressure of a system is somewhere in between the rigid and the flexible eFF pressures, but, for most cases, the difference between these two pressures will not be significant over long-term averaged runs (i.e. even though the energy partitioning changes, the total energy remains similar).

**Note**

currently, there is no available option for the user to set or create temperature distributions that include the radial electronic degrees of freedom with the *velocity* command, so the the user must allow for these degrees of freedom to equilibrate (i.e. equi-partitioning of energy) through time integration.

### 2.132.4 Restart, fix_modify, output, run start/stop, minimize info

See the page for the *fix nvt, npt, and nph* commands for details.

### 2.132.5 Restrictions

This fix is part of the EFF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Other restriction discussed on the page for the *fix nvt, npt, and nph* commands also apply.

**Note**

The temperature for systems (regions or groups) with only electrons and no nuclei is 0.0 (i.e. not defined) in the current temperature calculations, a practical example would be a uniform electron gas or a very hot plasma, where electrons remain delocalized from the nuclei. This is because, even though electron virials are included in the temperature calculation, these are averaged over the nuclear degrees of freedom only. In such cases a corrective term must be added to the pressure to get the correct kinetic contribution.

### 2.132.6 Related commands

*fix nvt*, *fix nph*, *fix npt*, *fix_modify*, *run_style*

### 2.132.7 Default

The keyword defaults are tchain = 3, pchain = 3, mtk = yes, tloop = ploop = 1, nreset = 0, drag = 0.0, dilate = all, and couple = none.

---

**(Martyna)** Martyna, Tobias and Klein, J Chem Phys, 101, 4177 (1994).

**(Parrinello)** Parrinello and Rahman, J Appl Phys, 52, 7182 (1981).

**(Tuckerman)** Tuckerman, Alejandre, Lopez-Rendon, Jochim, and Martyna, J Phys A: Math Gen, 39, 5629 (2006).

**(Shinoda)** Shinoda, Shiga, and Mikami, Phys Rev B, 69, 134103 (2004).

## 2.133 fix nvt/uef command

## 2.134 fix npt/uef command

### 2.134.1 Syntax

```
fix ID group-ID style_name erate edot_x edot_y temp Tstart Tstop Tdamp keyword value ...
```

- ID, group-ID are documented in *fix* command
- style_name = *nvt/uef* or *npt/uef*
- *Tstart*, *Tstop*, and *Tdamp* are documented in the *fix npt* command
- *edot_x* and *edot_y* are the strain rates in the x and y directions (1/(time units))
- one or more keyword/value pairs may be appended

```
keyword = erate or ext or strain or temp or iso or x or y or z or tchain or pchain␣
↪or tloop or ploop or mtk
  erate values = e_x e_y = true strain rates (required)
  ext value = x or y or z or xy or yz or xz = external dimensions
    sets the external dimensions used to calculate the scalar pressure
  strain values = e_x e_y = initial strain
    usually not needed, but may be needed to resume a run with a data file.
  temp, iso, x, y, z, tchain, pchain, tloop, ploop, mtk
    keywords documented by the fix npt command
```

### 2.134.2 Examples

```
fix uniax_nvt all nvt/uef temp 400 400 100 erate 0.00001 -0.000005
fix biax_nvt all nvt/uef temp 400 400 100 erate 0.000005 0.000005
fix uniax_npt all npt/uef temp 400 400 300 iso 1 1 3000 erate 0.00001 -0.000005 ext yz
fix biax_npt all npt/uef temp 400 400 100 erate -0.00001 0.000005 x 1 1 3000
```

### 2.134.3 Description

These fixes can be used to simulate non-equilibrium molecular dynamics (NEMD) under diagonal flow fields, including uniaxial and bi-axial flow. Simulations under continuous extensional flow may be carried out for an indefinite amount of time. It is an implementation of the boundary conditions from *(Dobson)*, and also uses numerical lattice reduction as was proposed by *(Hunt)*. The lattice reduction algorithm is from *(Semaev)*. The fix is intended for simulations of homogeneous flows, and integrates the SLLOD equations of motion, originally proposed by Hoover and Ladd (see *(Evans and Morriss)*). Additional detail about this implementation can be found in *(Nicholson and Rutledge)*.

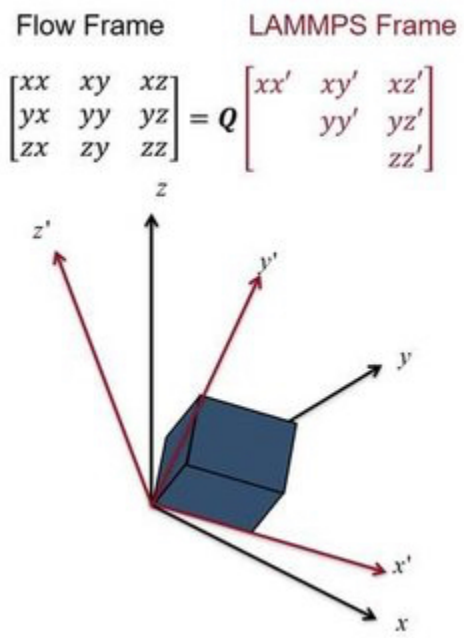
Note that NEMD simulations of a continuously strained system can be performed using the *fix deform*, *fix nvt/sllod*, and *compute temp/deform* commands.

The applied flow field is set by the *erate* keyword. The values *edot_x* and *edot_y* correspond to the strain rates in the xx and yy directions. It is implicitly assumed that the flow field is traceless, and therefore the strain rate in the zz direction is eqal to -(*edot_x* + *edot_y*).

**Note**

Due to an instability in the SLLOD equations under extension, *fix momentum* should be used to regularly reset the linear momentum.

The boundary conditions require a simulation box that does not have a consistent alignment relative to the applied flow field. Since LAMMPS utilizes an upper-triangular simulation box, it is not possible to express the evolving simulation box in the same coordinate system as the flow field. These fixes keep track of two coordinate systems: the flow frame, and the upper triangular LAMMPS frame. The coordinate systems are related to each other through the QR decomposition, as is illustrated in the image below.




During most molecular dynamics operations, the system is represented in the LAMMPS frame. Only when the positions and velocities are updated is the system rotated to the flow frame, and it is rotated back to the LAMMPS frame immediately afterwards. For this reason, all vector-valued quantities (except for the tensors from *compute pressure/uef* and *compute temp/uef*) will be computed in the LAMMPS frame. Rotationally invariant scalar quantities like the temperature and hydrostatic pressure are frame-invariant and will be computed correctly. Additionally, the system is in the LAMMPS frame during all of the output steps, and therefore trajectory files made using the dump command will be in the LAMMPS frame unless the *dump cfg/uef* command is used.

Temperature control is achieved with the default Nose-Hoover style thermostat documented in *fix nvt*. When this fix is active, only the peculiar velocity of each atom is stored, defined as the velocity relative to the streaming velocity. This is in contrast to *fix nvt/sllod*, which uses a lab-frame velocity, and removes the contribution from the streaming velocity in order to compute the temperature.

Pressure control is achieved using the default Nose-Hoover barostat documented in *fix npt*. There are two ways to control the pressure using this fix. The first method involves using the *ext* keyword along with the *iso* pressure style. With this method, the pressure is controlled by scaling the simulation box isotropically to achieve the average pressure only in the directions specified by *ext*. For example, if the *ext* value is set to *xy*, the average pressure (Pxx+Pyy)/2 will be controlled.

This example command will control the total hydrostatic pressure under uniaxial tension:

```
fix f1 all npt/uef temp 0.7 0.7 0.5 iso 1 1 5 erate -0.5 -0.5 ext xyz
```

This example command will control the average stress in compression directions, which would typically correspond to free surfaces under drawing with uniaxial tension:

```
fix f2 all npt/uef temp 0.7 0.7 0.5 iso 1 1 5 erate -0.5 -0.5 ext xy
```

The second method for pressure control involves setting the normal stresses using the *x*, *y*, and/or *z* keywords. When using this method, the same pressure must be specified via *Pstart* and *Pstop* for all dimensions controlled. Any choice of pressure conditions that would cause LAMMPS to compute a deviatoric stress are not permissible and will result in an error. Additionally, all dimensions with controlled stress must have the same applied strain rate. The *ext* keyword must be set to the default value (*xyz*) when using this method.

For example, the following commands will work:

```
fix f3 all npt/uef temp 0.7 0.7 0.5 x 1 1 5 y 1 1 5 erate -0.5 -0.5
fix f4 all npt/uef temp 0.7 0.7 0.5 z 1 1 5 erate 0.5 0.5
```

The following commands will not work:

```
fix f5 all npt/uef temp 0.7 0.7 0.5 x 1 1 5 z 1 1 5 erate -0.5 -0.5
fix f6 all npt/uef temp 0.7 0.7 0.5 x 1 1 5 z 2 2 5 erate 0.5 0.5
```

---

These fixes compute a temperature and pressure each timestep. To do this, they create their own computes of style "temp/uef" and "pressure/uef", as if one of these two sets of commands had been issued:

```
compute fix-ID_temp group-ID temp/uef
compute fix-ID_press group-ID pressure/uef fix-ID_temp

compute fix-ID_temp all temp/uef
compute fix-ID_press all pressure/uef fix-ID_temp
```

See the *compute temp/uef* and *compute pressure/uef* commands for details. Note that the IDs of the new computes are the fix-ID + underscore + "temp" or fix_ID + underscore + "press".

### 2.134.4 Restart, fix_modify, output, run start/stop, minimize info

The fix writes the state of all the thermostat and barostat variables, as well as the cumulative strain applied, to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

> **Note**
>
> It is not necessary to set the *strain* keyword when resuming a run from a restart file. Only for resuming from data files, which do not contain the cumulative applied strain, will this keyword be necessary.

These fixes can be used with the *fix_modify* *temp* and *press* options. The temperature and pressure computes used must be of type *temp/uef* and *pressure/uef*.

These fixes compute the same global scalar and vector quantities as *fix nvt andnpt*.

These fixes are not invoked during *energy minimization*.

### 2.134.5 Restrictions

These fixes are part of the UEF package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Due to requirements of the boundary conditions, when the *strain* keyword is set to zero (or unset), the initial simulation box must be cubic and have style triclinic. If the box is initially of type ortho, use *change_box* before invoking the fix.

### 2.134.6 Related commands

*fix nvt*, *fix npt*, *fix nvt/sllod*, *compute temp/uef*, *compute pressure/uef*, *dump cfg/uef*

### 2.134.7 Default

The default keyword values specific to these fixes are exy = xyz, strain = 0 0. The remaining defaults are the same as for *fix nvt or npt* except tchain = 1. The reason for this change is given in *fix nvt/sllod*.

---

**(Dobson)** Dobson, J Chem Phys, 141, 184103 (2014).

**(Hunt)** Hunt, Mol Simul, 42, 347 (2016).

**(Semaev)** Semaev, Cryptography and Lattices, 181 (2001).

**(Evans and Morriss)** Evans and Morriss, Phys Rev A, 30, 1528 (1984).

**(Nicholson and Rutledge)** Nicholson and Rutledge, J Chem Phys, 145, 244903 (2016).

## 2.135 fix nonaffine/displacement command

### 2.135.1 Syntax

```
fix ID group nonaffine/displacement nevery style args reference/style nstep keyword␣
↪values
```

- ID, group are documented in *fix* command
- nonaffine/displacement = style name of this fix command
- nevery = calculate nonaffine displacement every this many timesteps
- style = *d2min* or *integrated*

```
d2min args = cutoff args
  cutoff = type or radius or custom
    type args = none, cutoffs determined by atom types
```

```
    radius args = none, cutoffs determined based on atom diameters (atom style␣
↪sphere)
    custom args = rmax, cutoff set by a constant numeric value rmax (distance units)
integrated args = none
```

- reference/style = *fixed* or *update* or *offset*

```
fixed = use a fixed reference frame at nstep
update = update the reference frame every nstep timesteps
offset = update the reference frame nstep timesteps before calculating the␣
↪nonaffine displacement
```

- zero or more keyword/value pairs may be appended

```
z/min values = zmin
  zmin = minimum coordination number to calculate D2min
intensive/d2min values = yes or no
  specifies whether output uses an intensive or extensive value of d2min
```

## 2.135.2 Examples

```
fix 1 all nonaffine/displacement 100 integrated update 100
fix 1 all nonaffine/displacement 1000 d2min type fixed 0
fix 1 all nonaffine/displacement 1000 d2min custom 2.0 offset 100 intensive/d2min no
```

## 2.135.3 Description

Added in version 7Feb2024.

This fix computes different metrics of the nonaffine displacement of particles. The first metric, *d2min* calculates the $D^2_{\mathrm{min}}$ nonaffine displacement by Falk and Langer in *(Falk)*. For each atom, the fix computes the two tensors

$$X = \sum_{\mathrm{neighbors}} \vec{r}\,(\vec{r}_0)^T$$

and

$$Y = \sum_{\mathrm{neighbors}} \vec{r}_0\,(\vec{r}_0)^T$$

where the neighbors include all other atoms within the distance criterion set by the cutoff option, discussed below, $\vec{r}$ is the current displacement between particles, and $\vec{r}_0$ is the reference displacement. A deformation gradient tensor is then calculated as $F = XY^{-1}$ from which

$$D^2_{\mathrm{min}} = \frac{1}{N_n} \sum_{\mathrm{neighbors}} |\vec{r} - F\vec{r}_0|^2$$

and a strain tensor is calculated $E = FF^T - I$ where $I$ is the identity tensor and $N_n$ is the number of neighbors. This calculation is only performed on timesteps that are a multiple of *nevery* (including timestep zero). Data accessed before this occurs will simply be zeroed.

This formulation of D2min is intensive in the sense that it is normalized by the number of neighbors that contribute to it. Alternatively, this factor can be removed using the *intensive/d2min* option to calculate a quantity extensive in the number of neighbors.

For particles with low coordination numbers, calculations of $D^2_{\mathrm{min}}$ may not be accurate. An optional minimum coordination number can be defined using the *z/min* keyword. If any particle has fewer than the specified number of particles in the cutoff distance or in contact, the above calculations will be skipped and the corresponding peratom array entries will be zero.

The *integrated* style simply integrates the velocity of particles every timestep to calculate a displacement. This style only works if used in conjunction with another fix that deforms the box and displaces atom positions such as *fix deform* with remap x, *fix press/berendsen*, or *fix nh*.

Both of these methods require defining a reference state. With the *fixed* reference style, the user picks a specific timestep *nstep* at which particle positions are saved. If peratom data is accessed from this compute prior to this timestep, it will simply be zeroed. The *update* reference style implies the reference state will be updated every *nstep* timesteps. The *offset* reference will update the reference state *nstep* timesteps before a multiple of *nevery* timesteps.

---

### 2.135.4 Restart, fix_modify, output, run start/stop, minimize info

The reference state is saved to *binary restart files*.

None of the *fix_modify* options are relevant to this fix.

This fix computes a peratom array with either 3 or 9 columns, which can be accessed by indices 1-9 using any command that uses per-atom values from a fix as input.

For the *integrated* style, the three columns are the nonaffine displacements in the x, y, and z directions. For the *d2min* style, the first three columns are the calculated $\sqrt{D^2_{\text{min}}}$, the volumetric strain, and the deviatoric strain. The following 6 columns are the xx, yy, zz, xy, xz, and yz components of the calculated strain tensor.

### 2.135.5 Restrictions

This compute is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

As this fix depends on a run including specific reference timesteps, it currently does not update peratom values if used in conjunction with the *rerun command* since it cannot ensure the necessary reference timesteps are included.

### 2.135.6 Related commands

none

### 2.135.7 Default

intensive/d2min = yes

---

**(Falk)** Falk and Langer PRE, 57, 7192 (1998).

## 2.136 fix nph/asphere command

Accelerator Variants: *nph/asphere/omp*

### 2.136.1 Syntax

```
fix ID group-ID nph/asphere args keyword value ...
```

- ID, group-ID are documented in *fix* command
- nph/asphere = style name of this fix command
- additional barostat related keyword/value pairs from the *fix nph* command can be appended

### 2.136.2 Examples

```
fix 1 all nph/asphere iso 0.0 0.0 1000.0
fix 2 all nph/asphere x 5.0 5.0 1000.0
fix 2 all nph/asphere x 5.0 5.0 1000.0 drag 0.2
fix 2 water nph/asphere aniso 0.0 0.0 1000.0 dilate partial
```

### 2.136.3 Description

Perform constant NPH integration to update position, velocity, orientation, and angular velocity each timestep for aspherical or ellipsoidal particles in the group using a Nose/Hoover pressure barostat. P is pressure; H is enthalpy. This creates a system trajectory consistent with the isenthalpic ensemble.

This fix differs from the *fix nph* command, which assumes point particles and only updates their position and velocity.

Additional parameters affecting the barostat are specified by keywords and values documented with the *fix nph* command. See, for example, discussion of the *aniso*, and *dilate* keywords.

The particles in the fix group are the only ones whose velocities and positions are updated by the velocity/position update portion of the NPH integration.

Regardless of what particles are in the fix group, a global pressure is computed for all particles. Similarly, when the size of the simulation box is changed, all particles are re-scaled to new positions, unless the keyword *dilate* is specified with a value of *partial*, in which case only the particles in the fix group are re-scaled. The latter can be useful for leaving the coordinates of particles in a solid substrate unchanged and controlling the pressure of a surrounding fluid.

---

This fix computes a temperature and pressure each timestep. To do this, the fix creates its own computes of style "temp/asphere" and "pressure", as if these commands had been issued:

```
compute fix-ID_temp all temp/asphere
compute fix-ID_press all pressure fix-ID_temp
```

See the *compute temp/asphere* and *compute pressure* commands for details. Note that the IDs of the new computes are the fix-ID + underscore + "temp" or fix_ID + underscore + "press", and the group for the new computes is "all" since pressure is computed for the entire system.

Note that these are NOT the computes used by thermodynamic output (see the *thermo_style* command) with ID = *thermo_temp* and *thermo_press*. This means you can change the attributes of this fix's temperature or pressure via the *compute_modify* command or print this temperature or pressure during thermodynamic output via the *thermo_style custom* command using the appropriate compute-ID. It also means that changing attributes of *thermo_temp* or *thermo_press* will have no effect on this fix.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 2.136.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the state of the Nose/Hoover barostat to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The *fix_modify* *temp* and *press* options are supported by this fix. You can use them to assign a *compute* you have defined to this fix which will be used in its thermostatting or barostatting procedure. If you do this, note that the kinetic energy derived from the compute temperature should be consistent with the virial term computed using all atoms for the pressure. LAMMPS will warn you if you choose to compute temperature on a subset of atoms.

The cumulative energy change in the system imposed by this fix is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* doc page for details.

This fix computes the same global scalar and global vector of quantities as does the *fix nph* command.

This fix can ramp its target pressure over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.136.5 Restrictions

This fix is part of the ASPHERE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix requires that atoms store torque and angular momentum and a quaternion as defined by the *atom_style ellipsoid* command.

All particles in the group must be finite-size. They cannot be point particles, but they can be aspherical or spherical as defined by their shape attribute.

### 2.136.6 Related commands

*fix nph*, *fix nve_asphere*, *fix nvt_asphere*, *fix npt_asphere*, *fix_modify*

### 2.136.7 Default

none

## 2.137 fix nph/body command

### 2.137.1 Syntax

```
fix ID group-ID nph/body args keyword value ...
```

- ID, group-ID are documented in *fix* command
- nph/body = style name of this fix command
- additional barostat related keyword/value pairs from the *fix nph* command can be appended

### 2.137.2 Examples

```
fix 1 all nph/body iso 0.0 0.0 1000.0
fix 2 all nph/body x 5.0 5.0 1000.0
fix 2 all nph/body x 5.0 5.0 1000.0 drag 0.2
fix 2 water nph/body aniso 0.0 0.0 1000.0 dilate partial
```

### 2.137.3 Description

Perform constant NPH integration to update position, velocity, orientation, and angular velocity each timestep for body particles in the group using a Nose/Hoover pressure barostat. P is pressure; H is enthalpy. This creates a system trajectory consistent with the isenthalpic ensemble.

This fix differs from the *fix nph* command, which assumes point particles and only updates their position and velocity.

Additional parameters affecting the barostat are specified by keywords and values documented with the *fix nph* command. See, for example, discussion of the *aniso*, and *dilate* keywords.

The particles in the fix group are the only ones whose velocities and positions are updated by the velocity/position update portion of the NPH integration.

Regardless of what particles are in the fix group, a global pressure is computed for all particles. Similarly, when the size of the simulation box is changed, all particles are re-scaled to new positions, unless the keyword *dilate* is specified with a value of *partial*, in which case only the particles in the fix group are re-scaled. The latter can be useful for leaving the coordinates of particles in a solid substrate unchanged and controlling the pressure of a surrounding fluid.

---

This fix computes a temperature and pressure each timestep. To do this, the fix creates its own computes of style "temp/body" and "pressure", as if these commands had been issued:

```
compute fix-ID_temp all temp/body
compute fix-ID_press all pressure fix-ID_temp
```

See the *compute temp/body* and *compute pressure* commands for details. Note that the IDs of the new computes are the fix-ID + underscore + "temp" or fix_ID + underscore + "press", and the group for the new computes is "all" since pressure is computed for the entire system.

Note that these are NOT the computes used by thermodynamic output (see the *thermo_style* command) with ID = *thermo_temp* and *thermo_press*. This means you can change the attributes of this fix's temperature or pressure via the *compute_modify* command or print this temperature or pressure during thermodynamic output via the *thermo_style custom* command using the appropriate compute-ID. It also means that changing attributes of *thermo_temp* or *thermo_press* will have no effect on this fix.

---

### 2.137.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the state of the Nose/Hoover barostat to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The *fix_modify temp* and *press* options are supported by this fix. You can use them to assign a *compute* you have defined to this fix which will be used in its thermostatting or barostatting procedure. If you do this, note that the kinetic energy derived from the compute temperature should be consistent with the virial term computed using all atoms for the pressure. LAMMPS will warn you if you choose to compute temperature on a subset of atoms.

The cumulative energy change in the system imposed by this fix is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* doc page for details.

This fix computes the same global scalar and global vector of quantities as does the *fix nph* command.

This fix can ramp its target pressure over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.137.5 Restrictions

This fix is part of the BODY package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix requires that atoms store torque and angular momentum and a quaternion as defined by the *atom_style body* command.

### 2.137.6 Related commands

*fix nph*, *fix nve_body*, *fix nvt_body*, *fix npt_body*, *fix_modify*

### 2.137.7 Default

none

## 2.138 fix nph/sphere command

Accelerator Variants: *nph/sphere/kk*, *nph/sphere/omp*

### 2.138.1 Syntax

```
fix ID group-ID nph/sphere args keyword value ...
```

- ID, group-ID are documented in *fix* command
- nph/sphere = style name of this fix command
- keyword = *disc*

  *disc* value = none = treat particles as 2d discs, not spheres

- additional barostat related keyword/value pairs from the *fix nph* command can be appended

### 2.138.2 Examples

```
fix 1 all nph/sphere iso 0.0 0.0 1000.0
fix 2 all nph/sphere x 5.0 5.0 1000.0
fix 2 all nph/sphere x 5.0 5.0 1000.0 disc
fix 2 all nph/sphere x 5.0 5.0 1000.0 drag 0.2
fix 2 water nph/sphere aniso 0.0 0.0 1000.0 dilate partial
```

### 2.138.3 Description

Perform constant NPH integration to update position, velocity, and angular velocity each timestep for finite-size spherical particles in the group using a Nose/Hoover pressure barostat. P is pressure; H is enthalpy. This creates a system trajectory consistent with the isenthalpic ensemble.

This fix differs from the *fix nph* command, which assumes point particles and only updates their position and velocity.

If the *disc* keyword is used, then each particle is treated as a 2d disc (circle) instead of as a sphere. This is only possible for 2d simulations, as defined by the *dimension* keyword. The only difference between discs and spheres in this context is their moment of inertia, as used in the time integration.

Additional parameters affecting the barostat are specified by keywords and values documented with the *fix nph* command. See, for example, discussion of the *aniso*, and *dilate* keywords.

The particles in the fix group are the only ones whose velocities and positions are updated by the velocity/position update portion of the NPH integration.

Regardless of what particles are in the fix group, a global pressure is computed for all particles. Similarly, when the size of the simulation box is changed, all particles are re-scaled to new positions, unless the keyword *dilate* is specified with a value of *partial*, in which case only the particles in the fix group are re-scaled. The latter can be useful for leaving the coordinates of particles in a solid substrate unchanged and controlling the pressure of a surrounding fluid.

---

This fix computes a temperature and pressure each timestep. To do this, the fix creates its own computes of style "temp/sphere" and "pressure", as if these commands had been issued:

```
compute fix-ID_temp all temp/sphere
compute fix-ID_press all pressure fix-ID_temp
```

See the *compute temp/sphere* and *compute pressure* commands for details. Note that the IDs of the new computes are the fix-ID + underscore + "temp" or fix_ID + underscore + "press", and the group for the new computes is "all" since pressure is computed for the entire system.

Note that these are NOT the computes used by thermodynamic output (see the *thermo_style* command) with ID = *thermo_temp* and *thermo_press*. This means you can change the attributes of this fix's temperature or pressure via the *compute_modify* command or print this temperature or pressure during thermodynamic output via the *thermo_style custom* command using the appropriate compute-ID. It also means that changing attributes of *thermo_temp* or *thermo_press* will have no effect on this fix.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 2.138.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the state of the Nose/Hoover barostat to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The *fix_modify temp* and *press* options are supported by this fix. You can use them to assign a *compute* you have defined to this fix which will be used in its thermostatting or barostatting procedure. If you do this, note that the kinetic energy derived from the compute temperature should be consistent with the virial term computed using all atoms for the pressure. LAMMPS will warn you if you choose to compute temperature on a subset of atoms.

The cumulative energy change in the system imposed by this fix is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* doc page for details.

This fix computes the same global scalar and global vector of quantities as does the *fix nph* command.

This fix can ramp its target pressure over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.138.5 Restrictions

This fix requires that atoms store torque and angular velocity (omega) and a radius as defined by the *atom_style sphere* command.

All particles in the group must be finite-size spheres. They cannot be point particles.

Use of the *disc* keyword is only allowed for 2d simulations, as defined by the *dimension* keyword.

### 2.138.6 Related commands

*fix nph*, *fix nve_sphere*, *fix nvt_sphere*, *fix npt_sphere*, *fix_modify*

### 2.138.7 Default

none

## 2.139 fix nphug command

Accelerator Variants: *nphug/omp*

### 2.139.1 Syntax

```
fix ID group-ID nphug keyword value ...
```

- ID, group-ID are documented in *fix* command

```
one or more keyword value pairs may be appended
keyword = temp or iso or aniso or tri or x or y or z or couple or tchain or pchain␣
↪or mtk or tloop or ploop or nreset or drag or dilate or scaleyz or scalexz or␣
↪scalexy
  temp values = Value1 Value2 Tdamp
    Value1, Value2 = Nose-Hoover target temperatures, ignored by Hugoniostat
    Tdamp = temperature damping parameter (time units)
  iso or aniso or tri values = Pstart Pstop Pdamp
    Pstart,Pstop = scalar external pressures, must be equal (pressure units)
    Pdamp = pressure damping parameter (time units)
  x or y or z or xy or yz or xz values = Pstart Pstop Pdamp
    Pstart,Pstop = external stress tensor components, must be equal (pressure units)
    Pdamp = stress damping parameter (time units)
  couple = none or xyz or xy or yz or xz
  tchain value = length of thermostat chain (1 = single thermostat)
  pchain values = length of thermostat chain on barostat (0 = no thermostat)
  mtk value = yes or no = add in MTK adjustment term or not
  tloop value = number of sub-cycles to perform on thermostat
  ploop value = number of sub-cycles to perform on barostat thermostat
  nreset value = reset reference cell every this many timesteps
  drag value = drag factor added to barostat/thermostat (0.0 = no drag)
  dilate value = all or partial
  scaleyz value = yes or no = scale yz with lz
  scalexz value = yes or no = scale xz with lz
  scalexy value = yes or no = scale xy with ly
```

### 2.139.2 Examples

```
fix myhug all nphug temp 1.0 1.0 10.0 z 40.0 40.0 70.0
fix myhug all nphug temp 1.0 1.0 10.0 iso 40.0 40.0 70.0 drag 200.0 tchain 1 pchain 0
```

### 2.139.3 Description

This command is a variant of the Nose-Hoover *fix npt* fix style. It performs time integration of the Hugoniostat equations of motion developed by Ravelo et al. *(Ravelo)*. These equations compress the system to a state with average axial stress or pressure equal to the specified target value and that satisfies the Rankine-Hugoniot (RH) jump conditions for steady shocks.

The compression can be performed either hydrostatically (using keyword *iso*, *aniso*, or *tri*) or uniaxially (using keywords *x*, *y*, or *z*). In the hydrostatic case, the cell dimensions change dynamically so that the average axial stress in all three directions converges towards the specified target value. In the uniaxial case, the chosen cell dimension changes dynamically so that the average axial stress in that direction converges towards the target value. The other two cell dimensions are kept fixed (zero lateral strain).

This leads to the following additional restrictions on the keywords:

- One and only one of the following keywords should be used: *iso*, *aniso*, *tri*, *x*, *y*, *z*
- The specified initial and final target pressures must be the same.
- The keywords *xy*, *xz*, *yz* may not be used.
- The only admissible value for the couple keyword is *xyz*, which has the same effect as keyword *iso*
- The *temp* keyword must be used to specify the time constant for kinetic energy relaxation, but initial and final target temperature values are ignored.

Essentially, a Hugoniostat simulation is an NPT simulation in which the user-specified target temperature is replaced with a time-dependent target temperature Tt obtained from the following equation:

$$T_t - T = \frac{\left(\frac{1}{2}(P + P_0)(V_0 - V) + E_0 - E\right)}{N_{dof} k_B} = \Delta$$

where $T$ and $T_t$ are the instantaneous and target temperatures, $P$ and $P_0$ are the instantaneous and reference pressures or axial stresses, depending on whether hydrostatic or uniaxial compression is being performed, $V$ and $V_0$ are the instantaneous and reference volumes, $E$ and $E_0$ are the instantaneous and reference internal energy (potential plus kinetic), $N_{dof}$ is the number of degrees of freedom used in the definition of temperature, and $k_B$ is the Boltzmann constant. $\Delta$ is the negative deviation of the instantaneous temperature from the target temperature. When the system reaches a stable equilibrium, the value of $\Delta$ should fluctuate about zero.

The values of $E_0$, $V_0$, and $P_0$ are the instantaneous values at the start of the simulation. These can be overridden using the fix_modify keywords *e0*, *v0*, and *p0* described below.

---

**Note**

Unlike the *fix temp/berendsen* command which performs thermostatting but NO time integration, this fix performs thermostatting/barostatting AND time integration. Thus you should not use any other time integration fix, such as *fix nve* on atoms to which this fix is applied. Likewise, this fix should not be used on atoms that have their temperature controlled by another fix - e.g. by *fix langevin* or *fix temp/rescale* commands.

---

This fix computes a temperature and pressure at each timestep. To do this, the fix creates its own computes of style "temp" and "pressure", as if one of these two sets of commands had been issued:

```
compute fix-ID_temp group-ID temp
compute fix-ID_press group-ID pressure fix-ID_temp

compute fix-ID_temp all temp
compute fix-ID_press all pressure fix-ID_temp
```

See the *compute temp* and *compute pressure* commands for details. Note that the IDs of the new computes are the fix-ID + underscore + "temp" or fix_ID + underscore + "press". The group for the new computes is "all" since pressure is computed for the entire system.

Note that these are NOT the computes used by thermodynamic output (see the *thermo_style* command) with ID = *thermo_temp* and *thermo_press*. This means you can change the attributes of this fix's temperature or pressure via the *compute_modify* command or print this temperature or pressure during thermodynamic output via the *thermo_style custom* command using the appropriate compute-ID. It also means that changing attributes of *thermo_temp* or *thermo_press* will have no effect on this fix.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.139.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the values of $E_0$, $V_0$, and $P_0$, as well as the state of all the thermostat and barostat variables to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The *fix_modify e0*, *v0* and *p0* keywords can be used to define the values of $E_0$, $V_0$, and $P_0$. Note the the values for *e0* and *v0* are extensive, and so must correspond to the total energy and volume of the entire system, not energy and volume per atom. If any of these quantities are not specified, then the instantaneous value in the system at the start of the simulation is used.

The *fix_modify temp* and *press* options are supported by this fix. You can use them to assign a *compute* you have defined to this fix which will be used in its thermostatting or barostatting procedure, as described above. If you do this, note that the kinetic energy derived from the compute temperature should be consistent with the virial term computed using all atoms for the pressure. LAMMPS will warn you if you choose to compute temperature on a subset of atoms.

The cumulative energy change in the system imposed by this fix is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* doc page for details. Note that this energy is *not* included in the definition of internal energy E when calculating the value of Delta in the above equation.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the same cumulative energy change due to this fix described in the previous paragraph. The scalar value calculated by this fix is "extensive".

This fix also computes a global vector of quantities, which can be accessed by various *output commands*. The scalar The vector values are "intensive".

The vector stores three quantities unique to this fix (Δ, Us, and up), followed by all the internal Nose/Hoover thermostat and barostat variables defined for *fix npt*. Delta is the deviation of the temperature from the target temperature, given by the above equation. Us and up are the shock and particle velocity corresponding to a steady shock calculated from the RH conditions. They have units of distance/time.

### 2.139.5 Restrictions

This fix style is part of the SHOCK package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

All the usual restrictions for *fix npt* apply, plus the additional ones mentioned above.

### 2.139.6 Related commands

*fix msst*, *fix npt*, *fix_modify*

### 2.139.7 Default

The keyword defaults are the same as those for *fix npt*

---

**(Ravelo)** Ravelo, Holian, Germann and Lomdahl, Phys Rev B, 70, 014103 (2004).

## 2.140 fix npt/asphere command

Accelerator Variants: *npt/asphere/omp*

### 2.140.1 Syntax

```
fix ID group-ID npt/asphere keyword value ...
```

- ID, group-ID are documented in *fix* command
- npt/asphere = style name of this fix command
- additional thermostat and barostat related keyword/value pairs from the *fix npt* command can be appended

### 2.140.2 Examples

```
fix 1 all npt/asphere temp 300.0 300.0 100.0 iso 0.0 0.0 1000.0
fix 2 all npt/asphere temp 300.0 300.0 100.0 x 5.0 5.0 1000.0
fix 2 all npt/asphere temp 300.0 300.0 100.0 x 5.0 5.0 1000.0 drag 0.2
fix 2 water npt/asphere temp 300.0 300.0 100.0 aniso 0.0 0.0 1000.0 dilate partial
```

### 2.140.3 Description

Perform constant NPT integration to update position, velocity, orientation, and angular velocity each timestep for aspherical or ellipsoidal particles in the group using a Nose/Hoover temperature thermostat and Nose/Hoover pressure barostat. P is pressure; T is temperature. This creates a system trajectory consistent with the isothermal-isobaric ensemble.

This fix differs from the *fix npt* command, which assumes point particles and only updates their position and velocity.

The thermostat is applied to both the translational and rotational degrees of freedom for the aspherical particles, assuming a compute is used which calculates a temperature that includes the rotational degrees of freedom (see below). The

translational degrees of freedom can also have a bias velocity removed from them before thermostatting takes place; see the description below.

Additional parameters affecting the thermostat and barostat are specified by keywords and values documented with the *fix npt* command. See, for example, discussion of the *temp*, *iso*, *aniso*, and *dilate* keywords.

The particles in the fix group are the only ones whose velocities and positions are updated by the velocity/position update portion of the NPT integration.

Regardless of what particles are in the fix group, a global pressure is computed for all particles. Similarly, when the size of the simulation box is changed, all particles are re-scaled to new positions, unless the keyword *dilate* is specified with a value of *partial*, in which case only the particles in the fix group are re-scaled. The latter can be useful for leaving the coordinates of particles in a solid substrate unchanged and controlling the pressure of a surrounding fluid.

---

This fix computes a temperature and pressure each timestep. To do this, the fix creates its own computes of style "temp/asphere" and "pressure", as if these commands had been issued:

```
compute fix-ID_temp all temp/asphere
compute fix-ID_press all pressure fix-ID_temp
```

See the *compute temp/asphere* and *compute pressure* commands for details. Note that the IDs of the new computes are the fix-ID + underscore + "temp" or fix_ID + underscore + "press", and the group for the new computes is "all" since pressure is computed for the entire system.

Note that these are NOT the computes used by thermodynamic output (see the *thermo_style* command) with ID = *thermo_temp* and *thermo_press*. This means you can change the attributes of this fix's temperature or pressure via the *compute_modify* command or print this temperature or pressure during thermodynamic output via the *thermo_style custom* command using the appropriate compute-ID. It also means that changing attributes of *thermo_temp* or *thermo_press* will have no effect on this fix.

Like other fixes that perform thermostatting, this fix can be used with *compute commands* that remove a "bias" from the atom velocities. E.g. to apply the thermostat only to atoms within a spatial *region*, or to remove the center-of-mass velocity from a group of atoms, or to remove the x-component of velocity from the calculation.

This is not done by default, but only if the *fix_modify* command is used to assign a temperature compute to this fix that includes such a bias term. See the doc pages for individual *compute temp commands* to determine which ones include a bias. In this case, the thermostat works in the following manner: bias is removed from each atom, thermostatting is performed on the remaining thermal degrees of freedom, and the bias is added back in.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 2.140.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the state of the Nose/Hoover thermostat and barostat to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The *fix_modify* *temp* and *press* options are supported by this fix. You can use them to assign a *compute* you have defined to this fix which will be used in its thermostatting or barostatting procedure. If you do this, note that the kinetic energy derived from the compute temperature should be consistent with the virial term computed using all atoms for the pressure. LAMMPS will warn you if you choose to compute temperature on a subset of atoms.

The cumulative energy change in the system imposed by this fix is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* doc page for details.

This fix computes the same global scalar and global vector of quantities as does the *fix npt* command.

This fix can ramp its target temperature and pressure over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.140.5 Restrictions

This fix is part of the ASPHERE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix requires that atoms store torque and angular momentum and a quaternion as defined by the *atom_style ellipsoid* command.

All particles in the group must be finite-size. They cannot be point particles, but they can be aspherical or spherical as defined by their shape attribute.

### 2.140.6 Related commands

*fix npt*, *fix nve_asphere*, *fix nvt_asphere*, *fix_modify*

### 2.140.7 Default

none

## 2.141 fix npt/body command

### 2.141.1 Syntax

```
fix ID group-ID npt/body keyword value ...
```

- ID, group-ID are documented in *fix* command
- npt/body = style name of this fix command
- additional thermostat and barostat related keyword/value pairs from the *fix npt* command can be appended

### 2.141.2 Examples

```
fix 1 all npt/body temp 300.0 300.0 100.0 iso 0.0 0.0 1000.0
fix 2 all npt/body temp 300.0 300.0 100.0 x 5.0 5.0 1000.0
fix 2 all npt/body temp 300.0 300.0 100.0 x 5.0 5.0 1000.0 drag 0.2
fix 2 water npt/body temp 300.0 300.0 100.0 aniso 0.0 0.0 1000.0 dilate partial
```

### 2.141.3 Description

Perform constant NPT integration to update position, velocity, orientation, and angular velocity each timestep for body particles in the group using a Nose/Hoover temperature thermostat and Nose/Hoover pressure barostat. P is pressure; T is temperature. This creates a system trajectory consistent with the isothermal-isobaric ensemble.

This fix differs from the *fix npt* command, which assumes point particles and only updates their position and velocity.

The thermostat is applied to both the translational and rotational degrees of freedom for the body particles, assuming a compute is used which calculates a temperature that includes the rotational degrees of freedom (see below). The translational degrees of freedom can also have a bias velocity removed from them before thermostatting takes place; see the description below.

Additional parameters affecting the thermostat and barostat are specified by keywords and values documented with the *fix npt* command. See, for example, discussion of the *temp*, *iso*, *aniso*, and *dilate* keywords.

The particles in the fix group are the only ones whose velocities and positions are updated by the velocity/position update portion of the NPT integration.

Regardless of what particles are in the fix group, a global pressure is computed for all particles. Similarly, when the size of the simulation box is changed, all particles are re-scaled to new positions, unless the keyword *dilate* is specified with a value of *partial*, in which case only the particles in the fix group are re-scaled. The latter can be useful for leaving the coordinates of particles in a solid substrate unchanged and controlling the pressure of a surrounding fluid.

---

This fix computes a temperature and pressure each timestep. To do this, the fix creates its own computes of style "temp/body" and "pressure", as if these commands had been issued:

```
compute fix-ID_temp all temp/body
compute fix-ID_press all pressure fix-ID_temp
```

See the *compute temp/body* and *compute pressure* commands for details. Note that the IDs of the new computes are the fix-ID + underscore + "temp" or fix_ID + underscore + "press", and the group for the new computes is "all" since pressure is computed for the entire system.

Note that these are NOT the computes used by thermodynamic output (see the *thermo_style* command) with ID = *thermo_temp* and *thermo_press*. This means you can change the attributes of this fix's temperature or pressure via the *compute_modify* command or print this temperature or pressure during thermodynamic output via the *thermo_style custom* command using the appropriate compute-ID. It also means that changing attributes of *thermo_temp* or *thermo_press* will have no effect on this fix.

Like other fixes that perform thermostatting, this fix can be used with *compute commands* that remove a "bias" from the atom velocities. E.g. to apply the thermostat only to atoms within a spatial *region*, or to remove the center-of-mass velocity from a group of atoms, or to remove the x-component of velocity from the calculation.

This is not done by default, but only if the *fix_modify* command is used to assign a temperature compute to this fix that includes such a bias term. See the doc pages for individual *compute temp commands* to determine which ones include a bias. In this case, the thermostat works in the following manner: bias is removed from each atom, thermostatting is performed on the remaining thermal degrees of freedom, and the bias is added back in.

---

### 2.141.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the state of the Nose/Hoover thermostat and barostat to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The *fix_modify temp* and *press* options are supported by this fix. You can use them to assign a *compute* you have defined to this fix which will be used in its thermostatting or barostatting procedure. If you do this, note that the kinetic

energy derived from the compute temperature should be consistent with the virial term computed using all atoms for the pressure. LAMMPS will warn you if you choose to compute temperature on a subset of atoms.

The cumulative energy change in the system imposed by this fix is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* doc page for details.

This fix computes the same global scalar and global vector of quantities as does the *fix npt* command.

This fix can ramp its target temperature and pressure over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.141.5 Restrictions

This fix is part of the BODY package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix requires that atoms store torque and angular momentum and a quaternion as defined by the *atom_style body* command.

### 2.141.6 Related commands

*fix npt*, *fix nve_body*, *fix nvt_body*, *fix_modify*

### 2.141.7 Default

none

## 2.142 fix npt/cauchy command

### 2.142.1 Syntax

```
fix ID group-ID style_name keyword value ...
```

- ID, group-ID are documented in *fix* command
- style_name = *npt/cauchy*
- one or more keyword/value pairs may be appended
- keyword = *temp* or *iso* or *aniso* or *tri* or *x* or *y* or *z* or *xy* or *yz* or *xz* or *couple* or *tchain* or *pchain* or *mtk* or *tloop* or *ploop* or *nreset* or *drag* or *dilate* or *scalexy* or *scaleyz* or *scalexz* or *flip* or *alpha* or *continue* or *fixedpoint*

```
temp values = Tstart Tstop Tdamp
  Tstart,Tstop = external temperature at start/end of run
  Tdamp = temperature damping parameter (time units)
iso or aniso or tri values = Pstart Pstop Pdamp
  Pstart,Pstop = scalar external pressure at start/end of run (pressure units)
  Pdamp = pressure damping parameter (time units)
x or y or z or xy or yz or xz values = Pstart Pstop Pdamp
  Pstart,Pstop = external stress tensor component at start/end of run (pressure␣
↪units)
  Pdamp = stress damping parameter (time units)
couple = none or xyz or xy or yz or xz
tchain value = N
  N = length of thermostat chain (1 = single thermostat)
pchain values = N
```

```
  N length of thermostat chain on barostat (0 = no thermostat)
mtk value = yes or no = add in MTK adjustment term or not
tloop value = M
  M = number of sub-cycles to perform on thermostat
ploop value = M
  M = number of sub-cycles to perform on barostat thermostat
nreset value = reset reference cell every this many timesteps
drag value = Df
  Df = drag factor added to barostat/thermostat (0.0 = no drag)
dilate value = dilate-group-ID
  dilate-group-ID = only dilate atoms in this group due to barostat volume changes
scalexy value = yes or no = scale xy with ly
scaleyz value = yes or no = scale yz with lz
scalexz value = yes or no = scale xz with lz
flip value = yes or no = allow or disallow box flips when it becomes highly skewed
alpha value = strength of Cauchy stress control parameter
continue value = yes or no = whether of not to continue from a previous run
fixedpoint values = x y z
  x,y,z = perform barostat dilation/contraction around this point (distance units)
```

### 2.142.2 Examples

```
fix 1 water npt/cauchy temp 300.0 300.0 100.0 iso 0.0 0.0 1000.0 alpha 0.001
```

### 2.142.3 Description

This command performs time integration on Nose-Hoover style non-Hamiltonian equations of motion which are designed to generate positions and velocities sampled from the isothermal-isobaric (npt) ensembles. This updates the position and velocity for atoms in the group each timestep and the box dimensions.

The thermostatting and barostatting is achieved by adding some dynamic variables which are coupled to the particle velocities (thermostatting) and simulation domain dimensions (barostatting). In addition to basic thermostatting and barostatting, this fix can also create a chain of thermostats coupled to the particle thermostat, and another chain of thermostats coupled to the barostat variables. The barostat can be coupled to the overall box volume, or to individual dimensions, including the *xy*, *xz* and *yz* tilt dimensions. The external pressure of the barostat can be specified as either a scalar pressure (isobaric ensemble) or as components of a symmetric stress tensor (constant stress ensemble). When used correctly, the time-averaged temperature and stress tensor of the particles will match the target values specified by Tstart/Tstop and Pstart/Pstop.

The equations of motion used are those of Shinoda et al in *(Shinoda)*, which combine the hydrostatic equations of Martyna, Tobias and Klein in *(Martyna)* with the strain energy proposed by Parrinello and Rahman in *(Parrinello)*. The time integration schemes closely follow the time-reversible measure-preserving Verlet and rRESPA integrators derived by Tuckerman et al in *(Tuckerman)*.

---

The thermostat parameters are specified using the *temp* keyword. Other thermostat-related keywords are *tchain*, *tloop* and *drag*, which are discussed below.

The thermostat is applied to only the translational degrees of freedom for the particles. The translational degrees of freedom can also have a bias velocity removed before thermostatting takes place; see the description below. The desired temperature at each timestep is a ramped value during the run from *Tstart* to *Tstop*. The *Tdamp* parameter is specified in time units and determines how rapidly the temperature is relaxed. For example, a value of 10.0 means to relax the temperature in a timespan of (roughly) 10 time units (e.g. $\tau$ or fs or ps - see the *units* command). The atoms in the fix group are the only ones whose velocities and positions are updated by the velocity/position update portion of the integration.

**Note**

A Nose-Hoover thermostat will not work well for arbitrary values of *Tdamp*. If *Tdamp* is too small, the temperature can fluctuate wildly; if it is too large, the temperature will take a very long time to equilibrate. A good choice for many models is a *Tdamp* of around 100 timesteps. Note that this is NOT the same as 100 time units for most *units* settings.

---

The barostat parameters are specified using one or more of the *iso*, *aniso*, *tri*, *x*, *y*, *z*, *xy*, *xz*, *yz*, and *couple* keywords. These keywords give you the ability to specify all 6 components of an external stress tensor, and to couple various of these components together so that the dimensions they represent are varied together during a constant-pressure simulation.

Other barostat-related keywords are *pchain*, *mtk*, *ploop*, *nreset*, *drag*, and *dilate*, which are discussed below.

Orthogonal simulation boxes have 3 adjustable dimensions (x,y,z). Triclinic (non-orthogonal) simulation boxes have 6 adjustable dimensions (x,y,z,xy,xz,yz). The *create_box*, *read data*, and *read_restart* commands specify whether the simulation box is orthogonal or non-orthogonal (triclinic) and explain the meaning of the xy,xz,yz tilt factors.

The target pressures for each of the 6 components of the stress tensor can be specified independently via the *x*, *y*, *z*, *xy*, *xz*, *yz* keywords, which correspond to the 6 simulation box dimensions. For each component, the external pressure or tensor component at each timestep is a ramped value during the run from *Pstart* to *Pstop*. If a target pressure is specified for a component, then the corresponding box dimension will change during a simulation. For example, if the *y* keyword is used, the y-box length will change. If the *xy* keyword is used, the xy tilt factor will change. A box dimension will not change if that component is not specified, although you have the option to change that dimension via the *fix deform* command.

Note that in order to use the *xy*, *xz*, or *yz* keywords, the simulation box must be triclinic, even if its initial tilt factors are 0.0.

For all barostat keywords, the *Pdamp* parameter operates like the *Tdamp* parameter, determining the time scale on which pressure is relaxed. For example, a value of 10.0 means to relax the pressure in a timespan of (roughly) 10 time units (e.g. $\tau$ or fs or ps - see the *units* command).

**Note**

A Nose-Hoover barostat will not work well for arbitrary values of *Pdamp*. If *Pdamp* is too small, the pressure and volume can fluctuate wildly; if it is too large, the pressure will take a very long time to equilibrate. A good choice for many models is a *Pdamp* of around 1000 timesteps. However, note that *Pdamp* is specified in time units, and that timesteps are NOT the same as time units for most *units* settings.

Regardless of what atoms are in the fix group (the only atoms which are time integrated), a global pressure or stress tensor is computed for all atoms. Similarly, when the size of the simulation box is changed, all atoms are re-scaled to new positions, unless the keyword *dilate* is specified with a *dilate-group-ID* for a group that represents a subset of the atoms. This can be useful, for example, to leave the coordinates of atoms in a solid substrate unchanged and controlling the pressure of a surrounding fluid. This option should be used with care, since it can be unphysical to dilate some atoms and not others, because it can introduce large, instantaneous displacements between a pair of atoms (one dilated, one not) that are far from the dilation origin. Also note that for atoms not in the fix group, a separate time integration fix like *fix nve* or *fix nvt* can be used on them, independent of whether they are dilated or not.

---

The *couple* keyword allows two or three of the diagonal components of the pressure tensor to be "coupled" together. The value specified with the keyword determines which are coupled. For example, *xz* means the *Pxx* and *Pzz* components of the stress tensor are coupled. *Xyz* means all 3 diagonal components are coupled. Coupling means two things: the

instantaneous stress will be computed as an average of the corresponding diagonal components, and the coupled box dimensions will be changed together in lockstep, meaning coupled dimensions will be dilated or contracted by the same percentage every timestep. The *Pstart*, *Pstop*, *Pdamp* parameters for any coupled dimensions must be identical. *Couple xyz* can be used for a 2d simulation; the *z* dimension is simply ignored.

---

The *iso*, *aniso*, and *tri* keywords are simply shortcuts that are equivalent to specifying several other keywords together.

The keyword *iso* means couple all 3 diagonal components together when pressure is computed (hydrostatic pressure), and dilate/contract the dimensions together. Using "iso Pstart Pstop Pdamp" is the same as specifying these 4 keywords:

```
x Pstart Pstop Pdamp
y Pstart Pstop Pdamp
z Pstart Pstop Pdamp
couple xyz
```

The keyword *aniso* means *x*, *y*, and *z* dimensions are controlled independently using the *Pxx*, *Pyy*, and *Pzz* components of the stress tensor as the driving forces, and the specified scalar external pressure. Using "aniso Pstart Pstop Pdamp" is the same as specifying these 4 keywords:

```
x Pstart Pstop Pdamp
y Pstart Pstop Pdamp
z Pstart Pstop Pdamp
couple none
```

The keyword *tri* means *x*, *y*, *z*, *xy*, *xz*, and *yz* dimensions are controlled independently using their individual stress components as the driving forces, and the specified scalar pressure as the external normal stress. Using "tri Pstart Pstop Pdamp" is the same as specifying these 7 keywords:

```
x Pstart Pstop Pdamp
y Pstart Pstop Pdamp
z Pstart Pstop Pdamp
xy 0.0 0.0 Pdamp
yz 0.0 0.0 Pdamp
xz 0.0 0.0 Pdamp
couple none
```

---

In some cases (e.g. for solids) the pressure (volume) and/or temperature of the system can oscillate undesirably when a Nose/Hoover barostat and thermostat is applied. The optional *drag* keyword will damp these oscillations, although it alters the Nose/Hoover equations. A value of 0.0 (no drag) leaves the Nose/Hoover formalism unchanged. A non-zero value adds a drag term; the larger the value specified, the greater the damping effect. Performing a short run and monitoring the pressure and temperature is the best way to determine if the drag term is working. Typically a value between 0.2 to 2.0 is sufficient to damp oscillations after a few periods. Note that use of the drag keyword will interfere with energy conservation and will also change the distribution of positions and velocities so that they do not correspond to the nominal NVT, NPT, or NPH ensembles.

An alternative way to control initial oscillations is to use chain thermostats. The keyword *tchain* determines the number of thermostats in the particle thermostat. A value of 1 corresponds to the original Nose-Hoover thermostat. The keyword *pchain* specifies the number of thermostats in the chain thermostatting the barostat degrees of freedom. A value of 0 corresponds to no thermostatting of the barostat variables.

The *mtk* keyword controls whether or not the correction terms due to Martyna, Tuckerman, and Klein are included in the equations of motion *(Martyna)*. Specifying *no* reproduces the original Hoover barostat, whose volume probability distribution function differs from the true NPT and NPH ensembles by a factor of 1/V. Hence using *yes* is more correct, but in many cases the difference is negligible.

The keyword *tloop* can be used to improve the accuracy of integration scheme at little extra cost. The initial and final updates of the thermostat variables are broken up into *tloop* sub-steps, each of length *dt*/*tloop*. This corresponds to using a first-order Suzuki-Yoshida scheme *(Tuckerman)*. The keyword *ploop* does the same thing for the barostat thermostat.

The keyword *nreset* controls how often the reference dimensions used to define the strain energy are reset. If this keyword is not used, or is given a value of zero, then the reference dimensions are set to those of the initial simulation domain and are never changed. If the simulation domain changes significantly during the simulation, then the final average pressure tensor will differ significantly from the specified values of the external stress tensor. A value of *nstep* means that every *nstep* timesteps, the reference dimensions are set to those of the current simulation domain.

The *scaleyz*, *scalexz*, and *scalexy* keywords control whether or not the corresponding tilt factors are scaled with the associated box dimensions when barostatting triclinic periodic cells. The default values *yes* will turn on scaling, which corresponds to adjusting the linear dimensions of the cell while preserving its shape. Choosing *no* ensures that the tilt factors are not scaled with the box dimensions. See below for restrictions and default values in different situations. In older versions of LAMMPS, scaling of tilt factors was not performed. The old behavior can be recovered by setting all three scale keywords to *no*.

The *flip* keyword allows the tilt factors for a triclinic box to exceed half the distance of the parallel box length, as discussed below. If the *flip* value is set to *yes*, the bound is enforced by flipping the box when it is exceeded. If the *flip* value is set to *no*, the tilt will continue to change without flipping. Note that if applied stress induces large deformations (e.g. in a liquid), this means the box shape can tilt dramatically and LAMMPS will run less efficiently, due to the large volume of communication needed to acquire ghost atoms around a processor's irregular-shaped subdomain. For extreme values of tilt, LAMMPS may also lose atoms and generate an error.

The *fixedpoint* keyword specifies the fixed point for barostat volume changes. By default, it is the center of the box. Whatever point is chosen will not move during the simulation. For example, if the lower periodic boundaries pass through (0,0,0), and this point is provided to *fixedpoint*, then the lower periodic boundaries will remain at (0,0,0), while the upper periodic boundaries will move twice as far. In all cases, the particle trajectories are unaffected by the chosen value, except for a time-dependent constant translation of positions.

---

**Note**

Using a barostat coupled to tilt dimensions *xy*, *xz*, *yz* can sometimes result in arbitrarily large values of the tilt dimensions, i.e. a dramatically deformed simulation box. LAMMPS allows the tilt factors to grow a small amount beyond the normal limit of half the box length (0.6 times the box length), and then performs a box "flip" to an equivalent periodic cell. See the discussion of the *flip* keyword above, to allow this bound to be exceeded, if desired.

The flip operation is described in more detail in the page for *fix deform*. Both the barostat dynamics and the atom trajectories are unaffected by this operation. However, if a tilt factor is incremented by a large amount (1.5 times the box length) on a single timestep, LAMMPS can not accommodate this event and will terminate the simulation with an error. This error typically indicates that there is something badly wrong with how the simulation was constructed, such as specifying values of *Pstart* that are too far from the current stress value, or specifying a timestep that is too large. Triclinic barostatting should be used with care. This also is true for other barostat styles, although they tend to be more forgiving of insults. In particular, it is important to recognize that equilibrium liquids can not support a shear stress and that equilibrium solids can not support shear stresses that exceed the yield stress.

One exception to this rule is if the first dimension in the tilt factor (x for xy) is non-periodic. In that case, the limits on the tilt factor are not enforced, since flipping the box in that dimension does not change the atom positions due to non-periodicity. In this mode, if you tilt the system to extreme angles, the simulation will simply become inefficient due to the highly skewed simulation box.

**Note**

Unlike the *fix temp/berendsen* command which performs thermostatting but NO time integration, this fix performs thermostatting/barostatting AND time integration. Thus you should not use any other time integration fix, such as *fix nve* on atoms to which this fix is applied. Likewise, fix npt/cauchy should not normally be used on atoms that also have their temperature controlled by another fix - e.g. by *fix langevin* or *fix temp/rescale* commands.

See the *Howto thermostat* and *Howto barostat* doc pages for a discussion of different ways to compute temperature and perform thermostatting and barostatting.

---

This fix compute a temperature and pressure each timestep. To do this, the fix creates its own computes of style "temp" and "pressure", as if one of these sets of commands had been issued:

```
compute fix-ID_temp all temp
compute fix-ID_press all pressure fix-ID_temp
```

The group for both the new temperature and pressure compute is "all" since pressure is computed for the entire system. See the *compute temp* and *compute pressure* commands for details. Note that the IDs of the new computes are the fix-ID + underscore + "temp" or fix_ID + underscore + "press".

Note that these are NOT the computes used by thermodynamic output (see the *thermo_style* command) with ID = *thermo_temp* and *thermo_press*. This means you can change the attributes of these fix's temperature or pressure via the *compute_modify* command. Or you can print this temperature or pressure during thermodynamic output via the *thermo_style custom* command using the appropriate compute-ID. It also means that changing attributes of *thermo_temp* or *thermo_press* will have no effect on this fix.

Like other fixes that perform thermostatting, this fix can be used with *compute commands* that remove a "bias" from the atom velocities. E.g. to apply the thermostat only to atoms within a spatial *region*, or to remove the center-of-mass velocity from a group of atoms, or to remove the x-component of velocity from the calculation.

This is not done by default, but only if the *fix_modify* command is used to assign a temperature compute to this fix that includes such a bias term. See the doc pages for individual *compute temp commands* to determine which ones include a bias. In this case, the thermostat works in the following manner: bias is removed from each atom, thermostatting is performed on the remaining thermal degrees of freedom, and the bias is added back in.

---

This fix can be used with either the *verlet* or *respa integrators*. When using this fix with *respa*, LAMMPS uses an integrator constructed according to the following factorization of the Liouville propagator (for two rRESPA levels):

$$\begin{aligned}\exp\left(iL\Delta t\right) =& \hat{E}\exp\left(iL_{\text{T-baro}}\frac{\Delta t}{2}\right)\exp\left(iL_{\text{T-part}}\frac{\Delta t}{2}\right)\exp\left(iL_{\varepsilon,2}\frac{\Delta t}{2}\right)\exp\left(iL_2^{(2)}\frac{\Delta t}{2}\right)\\ &\times\left[\exp\left(iL_2^{(1)}\frac{\Delta t}{2n}\right)\exp\left(iL_{\varepsilon,1}\frac{\Delta t}{2n}\right)\exp\left(iL_1\frac{\Delta t}{n}\right)\exp\left(iL_{\varepsilon,1}\frac{\Delta t}{2n}\right)\exp\left(iL_2^{(1)}\frac{\Delta t}{2n}\right)\right]^n\\ &\times\exp\left(iL_2^{(2)}\frac{\Delta t}{2}\right)\exp\left(iL_{\varepsilon,2}\frac{\Delta t}{2}\right)\exp\left(iL_{\text{T-part}}\frac{\Delta t}{2}\right)\exp\left(iL_{\text{T-baro}}\frac{\Delta t}{2}\right)\\ &+\mathcal{O}\left(\Delta t^3\right)\end{aligned}$$

This factorization differs somewhat from that of Tuckerman et al, in that the barostat is only updated at the outermost rRESPA level, whereas Tuckerman's factorization requires splitting the pressure into pieces corresponding to the forces computed at each rRESPA level. In theory, the latter method will exhibit better numerical stability. In practice, because Pdamp is normally chosen to be a large multiple of the outermost rRESPA timestep, the barostat dynamics are not the limiting factor for numerical stability. Both factorizations are time-reversible and can be shown to preserve the phase space measure of the underlying non-Hamiltonian equations of motion.

**Note**

Under NPT dynamics, for a system with zero initial total linear momentum, the total momentum fluctuates close to zero. It may occasionally undergo brief excursions to non-negligible values, before returning close to zero. Over long simulations, this has the effect of causing the center-of-mass to undergo a slow random walk. This can be mitigated by resetting the momentum at infrequent intervals using the *fix momentum* command.

---

### 2.142.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the state of all the thermostat and barostat variables to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The *fix_modify* *temp* and *press* options are supported by this fix. You can use them to assign a *compute* you have defined to this fix which will be used in its thermostatting or barostatting procedure, as described above. If you do this, note that the kinetic energy derived from the compute temperature should be consistent with the virial term computed using all atoms for the pressure. LAMMPS will warn you if you choose to compute temperature on a subset of atoms.

**Note**

If both the *temp* and *press* keywords are used in a single thermo_modify command (or in two separate commands), then the order in which the keywords are specified is important. Note that a *pressure compute* defines its own temperature compute as an argument when it is specified. The *temp* keyword will override this (for the pressure compute being used by fix npt), but only if the *temp* keyword comes after the *press* keyword. If the *temp* keyword comes before the *press* keyword, then the new pressure compute specified by the *press* keyword will be unaffected by the *temp* setting.

The cumulative energy change in the system imposed by this fix, due to thermostatting and/or barostatting, is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* page for details.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the same cumulative energy change due to this fix described in the previous paragraph. The scalar value calculated by this fix is "extensive".

This fix also computes a global vector of quantities, which can be accessed by various *output commands*. Rhe vector values are "intensive".

The vector stores internal Nose/Hoover thermostat and barostat variables. The number and meaning of the vector values depends on which fix is used and the settings for keywords *tchain* and *pchain*, which specify the number of Nose/Hoover chains for the thermostat and barostat. If no thermostatting is done, then *tchain* is 0. If no barostatting is done, then *pchain* is 0. In the following list, "ndof" is 0, 1, 3, or 6, and is the number of degrees of freedom in the barostat. Its value is 0 if no barostat is used, else its value is 6 if any off-diagonal stress tensor component is barostatted, else its value is 1 if *couple xyz* is used or *couple xy* for a 2d simulation, otherwise its value is 3.

The order of values in the global vector and their meaning is as follows. The notation means there are tchain values for eta, followed by tchain for eta_dot, followed by ndof for omega, etc:

- eta[tchain] = particle thermostat displacements (unitless)
- eta_dot[tchain] = particle thermostat velocities (1/time units)
- omega[ndof] = barostat displacements (unitless)
- omega_dot[ndof] = barostat velocities (1/time units)
- etap[pchain] = barostat thermostat displacements (unitless)

- etap_dot[pchain] = barostat thermostat velocities (1/time units)
- PE_eta[tchain] = potential energy of each particle thermostat displacement (energy units)
- KE_eta_dot[tchain] = kinetic energy of each particle thermostat velocity (energy units)
- PE_omega[ndof] = potential energy of each barostat displacement (energy units)
- KE_omega_dot[ndof] = kinetic energy of each barostat velocity (energy units)
- PE_etap[pchain] = potential energy of each barostat thermostat displacement (energy units)
- KE_etap_dot[pchain] = kinetic energy of each barostat thermostat velocity (energy units)
- PE_strain[1] = scalar strain energy (energy units)

This fix can ramp its external temperature and pressure over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

---

### 2.142.5 Restrictions

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

*X*, *y*, *z* cannot be barostatted if the associated dimension is not periodic. *Xy*, *xz*, and *yz* can only be barostatted if the simulation domain is triclinic and the second dimension in the keyword (*y* dimension in *xy*) is periodic. *Z*, *xz*, and *yz*, cannot be barostatted for 2D simulations. The *create_box*, *read data*, and *read_restart* commands specify whether the simulation box is orthogonal or non-orthogonal (triclinic) and explain the meaning of the xy,xz,yz tilt factors.

For the *temp* keyword, the final Tstop cannot be 0.0 since it would make the external T = 0.0 at some timestep during the simulation which is not allowed in the Nose/Hoover formulation.

The *scaleyz yes* and *scalexz yes* keyword/value pairs can not be used for 2D simulations. *scaleyz yes*, *scalexz yes*, and *scalexy yes* options can only be used if the second dimension in the keyword is periodic, and if the tilt factor is not coupled to the barostat via keywords *tri*, *yz*, *xz*, and *xy*.

The *alpha* keyword modifies the barostat as per Miller et al. (Miller)_"#nc-Miller" so that the Cauchy stress is controlled. *alpha* is the non-dimensional parameter, typically set to 0.001 or 0.01 that determines how aggressively the algorithm drives the system towards the set Cauchy stresses. Larger values of *alpha* will modify the system more quickly, but can lead to instabilities. Smaller values will lead to longer convergence time. Since *alpha* also influences how much the stress fluctuations deviate from the equilibrium fluctuations, it should be set as small as possible.

A *continue* value of *yes* indicates that the fix is subsequent to a previous run with the npt/cauchy fix, and the intention is to continue from the converged stress state at the end of the previous run. This may be required, for example, when implementing a multi-step loading/unloading sequence over several fixes.

Setting *alpha* to zero is not permitted. To "turn off" the cauchystat control and thus restore the equilibrium stress fluctuations, two subsequent fixes should be used. In the first, fix npt/cauchy is used and the simulation box equilibrates to the correct shape for the desired stresses. In the second, *fix npt* is used instead which uses the original Parrinello-Rahman algorithm, but now with the corrected simulation box shape from using fix npt/cauchy.

This fix can be used with dynamic groups as defined by the *group* command. Likewise it can be used with groups to which atoms are added or deleted over time, e.g. a deposition simulation. However, the conservation properties of the thermostat and barostat are defined for systems with a static set of atoms. You may observe odd behavior if the atoms in a group vary dramatically over time or the atom count becomes very small.

### 2.142.6 Related commands

*fix nve*, *fix_modify*, *run_style*

### 2.142.7 Default

The keyword defaults are tchain = 3, pchain = 3, mtk = yes, tloop = ploop = 1, nreset = 0, drag = 0.0, dilate = all, couple = none, cauchystat = no, scaleyz = scalexz = scalexy = yes if periodic in second dimension and not coupled to barostat, otherwise no.

---

**(Martyna)** Martyna, Tobias and Klein, J Chem Phys, 101, 4177 (1994).

**(Parrinello)** Parrinello and Rahman, J Appl Phys, 52, 7182 (1981).

**(Tuckerman)** Tuckerman, Alejandre, Lopez-Rendon, Jochim, and Martyna, J Phys A: Math Gen, 39, 5629 (2006).

**(Shinoda)** Shinoda, Shiga, and Mikami, Phys Rev B, 69, 134103 (2004).

**(Miller)** Miller, Tadmor, Gibson, Bernstein and Pavia, J Chem Phys, 144, 184107 (2016).

## 2.143 fix npt/sphere command

Accelerator Variants: *npt/sphere/kk*, *npt/sphere/omp*

### 2.143.1 Syntax

```
fix ID group-ID npt/sphere keyword value ...
```

- ID, group-ID are documented in *fix* command npt/sphere = style name of this fix command zero or more keyword/value pairs may be appended
- keyword = *disc*

*disc* value = none = treat particles as 2d discs, not spheres

- NOTE: additional thermostat and barostat and dipole related keyword/value pairs from the *fix npt* command can be appended

### 2.143.2 Examples

```
fix 1 all npt/sphere temp 300.0 300.0 100.0 iso 0.0 0.0 1000.0
fix 2 all npt/sphere temp 300.0 300.0 100.0 x 5.0 5.0 1000.0
fix 2 all npt/sphere temp 300.0 300.0 100.0 x 5.0 5.0 1000.0 disc
fix 2 all npt/sphere temp 300.0 300.0 100.0 x 5.0 5.0 1000.0 drag 0.2
fix 2 all npt/sphere temp 300.0 300.0 100.0 x 5.0 5.0 1000.0 update dipole
fix 2 water npt/sphere temp 300.0 300.0 100.0 aniso 0.0 0.0 1000.0 dilate partial
```

### 2.143.3 Description

Perform constant NPT integration to update position, velocity, and angular velocity each timestep for finite-sizex spherical particles in the group using a Nose/Hoover temperature thermostat and Nose/Hoover pressure barostat. P is pressure; T is temperature. This creates a system trajectory consistent with the isothermal-isobaric ensemble.

This fix differs from the *fix npt* command, which assumes point particles and only updates their position and velocity.

The thermostat is applied to both the translational and rotational degrees of freedom for the spherical particles, assuming a compute is used which calculates a temperature that includes the rotational degrees of freedom (see below). The

translational degrees of freedom can also have a bias velocity removed from them before thermostatting takes place; see the description below.

If the *disc* keyword is used, then each particle is treated as a 2d disc (circle) instead of as a sphere. This is only possible for 2d simulations, as defined by the *dimension* keyword. The only difference between discs and spheres in this context is their moment of inertia, as used in the time integration.

Additional parameters affecting the thermostat and barostat are specified by keywords and values documented with the *fix npt* command. See, for example, discussion of the *temp*, *iso*, *aniso*, and *dilate* keywords.

The particles in the fix group are the only ones whose velocities and positions are updated by the velocity/position update portion of the NPT integration.

Regardless of what particles are in the fix group, a global pressure is computed for all particles. Similarly, when the size of the simulation box is changed, all particles are re-scaled to new positions, unless the keyword *dilate* is specified with a value of *partial*, in which case only the particles in the fix group are re-scaled. The latter can be useful for leaving the coordinates of particles in a solid substrate unchanged and controlling the pressure of a surrounding fluid.

---

This fix computes a temperature and pressure each timestep. To do this, the fix creates its own computes of style "temp/sphere" and "pressure", as if these commands had been issued:

```
compute fix-ID_temp all temp/sphere
compute fix-ID_press all pressure fix-ID_temp
```

See the *compute temp/sphere* and *compute pressure* commands for details. Note that the IDs of the new computes are the fix-ID + underscore + "temp" or fix_ID + underscore + "press", and the group for the new computes is "all" since pressure is computed for the entire system.

Note that these are NOT the computes used by thermodynamic output (see the *thermo_style* command) with ID = *thermo_temp* and *thermo_press*. This means you can change the attributes of this fix's temperature or pressure via the *compute_modify* command or print this temperature or pressure during thermodynamic output via the *thermo_style custom* command using the appropriate compute-ID. It also means that changing attributes of *thermo_temp* or *thermo_press* will have no effect on this fix.

Like other fixes that perform thermostatting, this fix can be used with *compute commands* that remove a "bias" from the atom velocities. E.g. to apply the thermostat only to atoms within a spatial *region*, or to remove the center-of-mass velocity from a group of atoms, or to remove the x-component of velocity from the calculation.

This is not done by default, but only if the *fix_modify* command is used to assign a temperature compute to this fix that includes such a bias term. See the doc pages for individual *compute temp commands* to determine which ones include a bias. In this case, the thermostat works in the following manner: bias is removed from each atom, thermostatting is performed on the remaining thermal degrees of freedom, and the bias is added back in.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 2.143.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the state of the Nose/Hoover thermostat and barostat to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The *fix_modify* *temp* and *press* options are supported by this fix. You can use them to assign a *compute* you have defined to this fix which will be used in its thermostatting or barostatting procedure. If you do this, note that the kinetic energy derived from the compute temperature should be consistent with the virial term computed using all atoms for the pressure. LAMMPS will warn you if you choose to compute temperature on a subset of atoms.

The cumulative energy change in the system imposed by this fix is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* doc page for details.

This fix computes the same global scalar and global vector of quantities as does the *fix npt* command.

This fix can ramp its target temperature and pressure over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.143.5 Restrictions

This fix requires that atoms store torque and angular velocity (omega) and a radius as defined by the *atom_style sphere* command.

All particles in the group must be finite-size spheres. They cannot be point particles.

Use of the *disc* keyword is only allowed for 2d simulations, as defined by the *dimension* keyword.

### 2.143.6 Related commands

***fix npt***, ***fix nve_sphere***,
*fix nvt_sphere*, *fix npt_asphere*, *fix_modify*

### 2.143.7 Default

none

## 2.144 fix numdiff command

### 2.144.1 Syntax

```
fix ID group-ID numdiff Nevery delta
```

- ID, group-ID are documented in *fix* command
- numdiff = style name of this fix command
- Nevery = calculate force by finite difference every this many timesteps
- delta = size of atom displacements (distance units)

### 2.144.2 Examples

```
fix 1 all numdiff 10 1e-6
fix 1 movegroup numdiff 100 0.01
```

### 2.144.3 Description

Calculate forces through finite difference calculations of energy versus position. These forces can be compared to analytic forces computed by pair styles, bond styles, etc. This can be useful for debugging or other purposes.

The group specified with the command means only atoms within the group have their averages computed. Results are set to 0.0 for atoms not in the group.

This fix performs a loop over all atoms in the group. For each atom and each component of force it adds *delta* to the position, and computes the new energy of the entire system. It then subtracts *delta* from the original position and again computes the new energy of the system. It then restores the original position. That component of force is calculated as the difference in energy divided by two times *delta*.

**Note**

It is important to choose a suitable value for delta, the magnitude of atom displacements that are used to generate finite difference approximations to the exact forces. For typical systems, a value in the range of 1 part in 1e4 to 1e5 of the typical separation distance between atoms in the liquid or solid state will be sufficient. However, the best value will depend on a multitude of factors including the stiffness of the interatomic potential, the thermodynamic state of the material being probed, and so on. The only way to be sure that you have made a good choice is to do a sensitivity study on a representative atomic configuration, sweeping over a wide range of values of delta. If delta is too small, the output forces will vary erratically due to truncation effects. If delta is increased beyond a certain point, the output forces will start to vary smoothly with delta, due to growing contributions from higher order derivatives. In between these two limits, the numerical force values should be largely independent of delta.

**Note**

The cost of each energy evaluation is essentially the cost of an MD timestep. Thus invoking this fix once for a 3d system has a cost of 6N timesteps, where N is the total number of atoms in the system. So this fix can be very expensive to use for large systems. One expedient alternative is to define the fix for a group containing only a few atoms.

---

The *Nevery* argument specifies on what timesteps the force will be used calculated by finite difference.

The *delta* argument specifies the size of the displacement each atom will undergo.

---

### 2.144.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix produces a per-atom array which can be accessed by various *output commands*, which stores the components of the force on each atom as calculated by finite difference. The per-atom values can only be accessed on timesteps that are multiples of *Nevery* since that is when the finite difference forces are calculated. See the examples in *examples/numdiff* directory to see how this fix can be used to directly compare with the analytic forces computed by LAMMPS.

The array values calculated by this compute will be in force *units*.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is invoked during *energy minimization*.

### 2.144.5 Restrictions

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.144.6 Related commands

*dynamical_matrix*, *fix numdiff/virial*,

### 2.144.7 Default

none

## 2.145 fix numdiff/virial command

### 2.145.1 Syntax

```
fix ID group-ID numdiff/virial Nevery delta
```

- ID, group-ID are documented in *fix* command
- numdiff/virial = style name of this fix command
- Nevery = calculate virial by finite difference every this many timesteps
- delta = magnitude of strain fields (dimensionless)

### 2.145.2 Examples

```
fix 1 all numdiff/stress 10 1e-6
```

### 2.145.3 Description

Added in version 17Feb2022.

Calculate the virial stress tensor through a finite difference calculation of energy versus strain. These values can be compared to the analytic virial tensor computed by pair styles, bond styles, etc. This can be useful for debugging or other purposes. The specified group must be "all".

This fix applies linear strain fields of magnitude *delta* to all the atoms relative to a point at the center of the box. The strain fields are in six different directions, corresponding to the six Cartesian components of the stress tensor defined by LAMMPS. For each direction it applies the strain field in both the positive and negative senses, and the new energy of the entire system is calculated after each. The difference in these two energies divided by two times *delta*, approximates the corresponding component of the virial stress tensor, after applying a suitable unit conversion.

**Note**

It is important to choose a suitable value for delta, the magnitude of strains that are used to generate finite difference approximations to the exact virial stress. For typical systems, a value in the range of 1 part in 1e5 to 1e6 will be sufficient. However, the best value will depend on a multitude of factors including the stiffness of the interatomic potential, the thermodynamic state of the material being probed, and so on. The only way to be sure that you have made a good choice is to do a sensitivity study on a representative atomic configuration, sweeping over a wide range of values of delta. If delta is too small, the output values will vary erratically due to truncation effects. If delta is increased beyond a certain point, the output values will start to vary smoothly with delta, due to growing

contributions from higher order derivatives. In between these two limits, the numerical virial values should be largely independent of delta.

---

The *Nevery* argument specifies on what timesteps the force will be used calculated by finite difference.

The *delta* argument specifies the size of the displacement each atom will undergo.

---

### 2.145.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix produces a global vector which can be accessed by various *output commands*, which stores the components of the virial stress tensor as calculated by finite difference. The global vector can only be accessed on timesteps that are multiples of *Nevery* since that is when the finite difference virial is calculated. See the examples in *examples/numdiff* directory to see how this fix can be used to directly compare with the analytic virial stress tensor computed by LAMMPS.

The order of the virial stress tensor components is *xx*, *yy*, *zz*, *yz*, *xz*, and *xy*, consistent with Voigt notation. Note that the vector produced by *compute pressure* uses a different ordering, with *yz* and *xy* swapped.

The vector values calculated by this compute are "intensive". The vector values will be in pressure *units*.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is invoked during *energy minimization*.

### 2.145.5 Restrictions

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.145.6 Related commands

*fix numdiff* , *compute pressure*

### 2.145.7 Default

none

## 2.146 fix nve command

Accelerator Variants: *nve/gpu*, *nve/intel*, *nve/kk*, *nve/omp*

### 2.146.1 Syntax

```
fix ID group-ID nve
```

- ID, group-ID are documented in *fix* command
- nve = style name of this fix command

### 2.146.2 Examples

```
fix 1 all nve
```

### 2.146.3 Description

Perform plain time integration to update position and velocity for atoms in the group each timestep. This creates a system trajectory consistent with the microcanonical ensemble (NVE) provided there are (full) periodic boundary conditions and no other "manipulations" of the system (e.g. fixes that modify forces or velocities).

This fix invokes the velocity form of the Stoermer-Verlet time integration algorithm (velocity-Verlet). Other time integration options can be invoked using the *run_style* command.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.146.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.146.5 Restrictions

none

### 2.146.6 Related commands

*fix nvt*, *fix npt*, *run_style*

### 2.146.7 Default

none

## 2.147 fix nve/asphere command

Accelerator Variants: *nve/asphere/gpu*, *nve/asphere/intel*, *nve/asphere/kk*

### 2.147.1 Syntax

```
fix ID group-ID nve/asphere
```

- ID, group-ID are documented in *fix* command
- nve/asphere = style name of this fix command

### 2.147.2 Examples

```
fix 1 all nve/asphere
```

### 2.147.3 Description

Perform constant NVE integration to update position, velocity, orientation, and angular velocity for aspherical particles in the group each timestep. V is volume; E is energy. This creates a system trajectory consistent with the microcanonical ensemble.

This fix differs from the *fix nve* command, which assumes point particles and only updates their position and velocity.

### 2.147.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.147.5 Restrictions

This fix is part of the ASPHERE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix requires that atoms store torque and angular momentum and a quaternion as defined by the *atom_style ellipsoid* command.

All particles in the group must be finite-size. They cannot be point particles, but they can be aspherical or spherical as defined by their shape attribute.

### 2.147.6 Related commands

*fix nve*, *fix nve/sphere*

### 2.147.7 Default

none

## 2.148 fix nve/asphere/noforce command

### 2.148.1 Syntax

```
fix ID group-ID nve/asphere/noforce
```

- ID, group-ID are documented in *fix* command
- nve/asphere/noforce = style name of this fix command

### 2.148.2 Examples

```
fix 1 all nve/asphere/noforce
```

### 2.148.3 Description

Perform updates of position and orientation, but not velocity or angular momentum for atoms in the group each timestep. In other words, the force and torque on the atoms is ignored and their velocity and angular momentum are not updated. The atom velocities and angular momenta are used to update their positions and orientation.

This is useful as an implicit time integrator for Fast Lubrication Dynamics, since the velocity and angular momentum are updated by the *pair_style lubricuteU* command.

---

### 2.148.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.148.5 Restrictions

This fix is part of the ASPHERE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix requires that atoms store torque and angular momentum and a quaternion as defined by the *atom_style ellipsoid* command.

All particles in the group must be finite-size. They cannot be point particles, but they can be aspherical or spherical as defined by their shape attribute.

### 2.148.6 Related commands

*fix nve/noforce*, *fix nve/asphere*

### 2.148.7 Default

none

# 2.149 fix nve/body command

## 2.149.1 Syntax

```
fix ID group-ID nve/body
```

- ID, group-ID are documented in *fix* command
- nve/body = style name of this fix command

## 2.149.2 Examples

```
fix 1 all nve/body
```

## 2.149.3 Description

Perform constant NVE integration to update position, velocity, orientation, and angular velocity for body particles in the group each timestep. V is volume; E is energy. This creates a system trajectory consistent with the microcanonical ensemble. See the *Howto body* page for more details on using body particles.

This fix differs from the *fix nve* command, which assumes point particles and only updates their position and velocity.

## 2.149.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

## 2.149.5 Restrictions

This fix is part of the BODY package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix requires that atoms store torque and angular momentum and a quaternion as defined by the *atom_style body* command.

All particles in the group must be body particles. They cannot be point particles.

## 2.149.6 Related commands

*fix nve*, *fix nve/sphere*, *fix nve/asphere*

## 2.149.7 Default

none

# 2.150 fix nve/bpm/sphere command

## 2.150.1 Syntax

```
fix ID group-ID nve/bpm/sphere
```

- ID, group-ID are documented in *fix* command
- nve/bpm/sphere = style name of this fix command

- zero or more keyword/value pairs may be appended
- keyword = *update* or *disc*

```
update value = dipole or dipole/dlm
  dipole = update orientation of dipole moment during integration
  dipole/dlm = use DLM integrator to update dipole orientation
disc value = none = treat particles as 2d discs, not spheres
```

### 2.150.2 Examples

```
fix 1 all nve/bpm/sphere
fix 1 all nve/bpm/sphere disc
```

### 2.150.3 Description

Added in version 4May2022.

Perform constant NVE integration to update position, velocity, angular velocity, and quaternion orientation for finite-size spherical particles in the group each timestep. V is volume; E is energy. This creates a system trajectory consistent with the microcanonical ensemble.

This fix differs from the *fix nve* command, which assumes point particles and only updates their position and velocity. It also differs from the *fix nve/sphere* command which assumes finite-size spheroid particles which do not store a quaternion. It thus does not update a particle's orientation or quaternion.

If the *disc* keyword is used, then each particle is treated as a 2d disc (circle) instead of as a sphere. This is only possible for 2d simulations, as defined by the *dimension* keyword. The only difference between discs and spheres in this context is their moment of inertia, as used in the time integration.

Added in version 4Jul2026.

If the *update* keyword is used with the *dipole* value, then the orientation of the dipole moment of each particle is also updated during the time integration, similar to the *fix nve/sphere* command. This option should be used for models where a dipole moment is assigned to finite-size particles, e.g. spheroids via use of the *atom_style hybrid bpm/sphere dipole* command.

The default dipole orientation integrator can be changed to the Dullweber-Leimkuhler-McLachlan integration scheme *(Dullweber)* when using *update* with the value *dipole/dlm*. This integrator is symplectic and time-reversible, giving better energy conservation and allows slightly longer timesteps at only a small additional computational cost.

---

### 2.150.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.150.5 Restrictions

This fix is part of the BPM package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix requires that atoms store torque, angular velocity (omega), a radius, and a quaternion as defined by the *atom_style bpm/sphere* command.

All particles in the group must be finite-size spheres with quaternions. They cannot be point particles.

Use of the *disc* keyword is only allowed for 2d simulations, as defined by the *dimension* keyword.

### 2.150.6 Related commands

*fix nve*, *fix nve/sphere*

### 2.150.7 Default

none

## 2.151 fix nve/dot command

### 2.151.1 Syntax

```
fix ID group-ID nve/dot
```

- ID, group-ID are documented in *fix* command
- nve/dot = style name of this fix command

### 2.151.2 Examples

```
fix 1 all nve/dot
```

### 2.151.3 Description

Apply a rigid-body integrator as described in *(Davidchack)* to a group of atoms, but without Langevin dynamics. This command performs Molecular dynamics (MD) via a velocity-Verlet algorithm and an evolution operator that rotates the quaternion degrees of freedom, similar to the scheme outlined in *(Miller)*.

This command is the equivalent of the *fix nve/dotc/langevin* without damping and noise and can be used to determine the stability range in a NVE ensemble prior to using the Langevin-type DOTC-integrator (see also *fix nve/dotc/langevin*). The command is equivalent to the *fix nve*. The particles are always considered to have a finite size.

An example input file can be found in /examples/PACKAGES/cgdna/examples/duplex1/. Further details of the implementation and stability of the integrator are contained in *(Henrich)*. The preprint version of the article can be found here.

---

### 2.151.4 Restrictions

These pair styles can only be used if LAMMPS was built with the *CG-DNA* package and the MOLECULE and ASPHERE package. See the *Build package* page for more info.

### 2.151.5 Related commands

*fix nve/dotc/langevin*, *fix nve*

### 2.151.6 Default

none

---

**(Davidchack)** R.L Davidchack, T.E. Ouldridge, and M.V. Tretyakov. J. Chem. Phys. 142, 144114 (2015).

**(Miller)** T. F. Miller III, M. Eleftheriou, P. Pattnaik, A. Ndirango, G. J. Martyna, J. Chem. Phys., 116, 8649-8659 (2002).

**(Henrich)** O. Henrich, Y. A. Gutierrez-Fosado, T. Curk, T. E. Ouldridge, Eur. Phys. J. E 41, 57 (2018).

# 2.152 fix nve/dotc/langevin command

## 2.152.1 Syntax

```
fix ID group-ID nve/dotc/langevin Tstart Tstop damp seed keyword value
```

- ID, group-ID are documented in *fix* command
- nve/dotc/langevin = style name of this fix command
- Tstart,Tstop = desired temperature at start/end of run (temperature units)
- damp = damping parameter (time units)
- seed = random number seed to use for white noise (positive integer)
- keyword = *angmom*

```
angmom value = factor
  factor = do thermostat rotational degrees of freedom via the angular momentum and␣
↪apply numeric scale factor as discussed below
```

## 2.152.2 Examples

```
fix 1 all nve/dotc/langevin 1.0 1.0 0.03 457145 angmom 10
fix 1 all nve/dotc/langevin 0.1 0.1 78.9375 457145 angmom 10
```

## 2.152.3 Description

Apply a rigid-body Langevin-type integrator of the kind "Langevin C" as described in *(Davidchack)* to a group of atoms, which models an interaction with an implicit background solvent. This command performs Brownian dynamics (BD) via a technique that splits the integration into a deterministic Hamiltonian part and the Ornstein-Uhlenbeck process for noise and damping. The quaternion degrees of freedom are updated though an evolution operator which performs a rotation in quaternion space, preserves the quaternion norm and is akin to *(Miller)*.

In terms of syntax this command has been closely modelled on the *fix langevin* and its *angmom* option. But it combines the *fix nve* and the *fix langevin* in one single command. The main feature is improved stability over the standard integrator, permitting slightly larger timestep sizes.

**Note**

Unlike the *fix langevin* this command performs also time integration of the translational and quaternion degrees of freedom.

The total force on each atom will have the form:

$$
\begin{aligned}
F &= F_c + F_f + F_r \\
F_f &= -\frac{m}{\text{damp}} v \\
F_r &\propto \sqrt{\frac{k_B T m}{dt\ \text{damp}}}
\end{aligned}
$$

$F_c$ is the conservative force computed via the usual inter-particle interactions (*pair_style*, *bond_style*, etc). The $F_f$ and $F_r$ terms are implicitly taken into account by this fix on a per-particle basis.

$F_f$ is a frictional drag or viscous damping term proportional to the particle's velocity. The proportionality constant for each atom is computed as $\frac{m}{\text{damp}}$, where $m$ is the mass of the particle and damp is the damping factor specified by the user.

$F_r$ is a force due to solvent atoms at a temperature $T$ randomly bumping into the particle. As derived from the fluctuation/dissipation theorem, its magnitude as shown above is proportional to $\sqrt{\frac{k_B T m}{dt\ \text{damp}}}$, where $k_B$ is the Boltzmann constant, $T$ is the desired temperature, $m$ is the mass of the particle, $dt$ is the timestep size, and damp is the damping factor. Random numbers are used to randomize the direction and magnitude of this force as described in *(Dunweg)*, where a uniform random number is used (instead of a Gaussian random number) for speed.

---

*Tstart* and *Tstop* have to be constant values, i.e. they cannot be variables. If used together with the oxDNA force field for coarse-grained simulation of DNA please note that T = 0.1 in oxDNA units corresponds to T = 300 K.

The *damp* parameter is specified in time units and determines how rapidly the temperature is relaxed. For example, a value of 0.03 means to relax the temperature in a timespan of (roughly) 0.03 time units $\tau$ (see the *units* command). The damp factor can be thought of as inversely related to the viscosity of the solvent, i.e. a small relaxation time implies a high-viscosity solvent and vice versa. See the discussion about gamma and viscosity in the documentation for the *fix viscous* command for more details. Note that the value 78.9375 in the second example above corresponds to a diffusion constant, which is about an order of magnitude larger than realistic ones. This has been used to sample configurations faster in Brownian dynamics simulations.

The random # *seed* must be a positive integer. A Marsaglia random number generator is used. Each processor uses the input seed to generate its own unique seed and its own stream of random numbers. Thus the dynamics of the system will not be identical on two runs on different numbers of processors.

The keyword/value option has to be used in the following way:

This fix has to be used together with the *angmom* keyword. The particles are always considered to have a finite size. The keyword *angmom* enables thermostatting of the rotational degrees of freedom in addition to the usual translational degrees of freedom.

The scale factor after the *angmom* keyword gives the ratio of the rotational to the translational friction coefficient.

An example input file can be found in examples/PACKAGES/cgdna/examples/duplex2/. Further details of the implementation and stability of the integrators are contained in *(Henrich)*. The preprint version of the article can be found here.

---

### 2.152.4 Restrictions

These pair styles can only be used if LAMMPS was built with the *CG-DNA* package and the MOLECULE and ASPHERE package. See the *Build package* page for more info.

### 2.152.5 Related commands

*fix nve*, *fix langevin*, *fix nve/dot*, *bond_style oxdna/fene*, *bond_style oxdna2/fene*, *pair_style oxdna/excv*, *pair_style oxdna2/excv*

### 2.152.6 Default

none

---

**(Davidchack)** R.L Davidchack, T.E. Ouldridge, M.V. Tretyakov. J. Chem. Phys. 142, 144114 (2015).

**(Miller)** T. F. Miller III, M. Eleftheriou, P. Pattnaik, A. Ndirango, G. J. Martyna, J. Chem. Phys., 116, 8649-8659 (2002).

**(Dunweg)** B. Dunweg, W. Paul, Int. J. Mod. Phys. C, 2, 817-27 (1991).

**(Henrich)** O. Henrich, Y. A. Gutierrez-Fosado, T. Curk, T. E. Ouldridge, Eur. Phys. J. E 41, 57 (2018).

# 2.153 fix nve/eff command

## 2.153.1 Syntax

```
fix ID group-ID nve/eff
```

- ID, group-ID are documented in *fix* command
- nve/eff = style name of this fix command

## 2.153.2 Examples

```
fix 1 all nve/eff
```

## 2.153.3 Description

Perform constant NVE integration to update position and velocity for nuclei and electrons in the group for the *electron force field* model. V is volume; E is energy. This creates a system trajectory consistent with the microcanonical ensemble.

The operation of this fix is exactly like that described by the *fix nve* command, except that the radius and radial velocity of electrons are also updated.

## 2.153.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

## 2.153.5 Restrictions

This fix is part of the EFF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

## 2.153.6 Related commands

*fix nve*, *fix nvt/eff* , *fix npt/eff*

## 2.153.7 Default

none

# 2.154 fix nve/limit command

Accelerator Variants: *nve/limit/kk*

### 2.154.1 Syntax

```
fix ID group-ID nve/limit xmax
```

- ID, group-ID are documented in *fix* command
- nve = style name of this fix command
- xmax = maximum distance an atom can move in one timestep (distance units)

### 2.154.2 Examples

```
fix 1 all nve/limit 0.1
```

### 2.154.3 Description

Perform constant NVE updates of position and velocity for atoms in the group each timestep. A limit is imposed on the maximum distance an atom can move in one timestep. This is useful when starting a simulation with a configuration containing highly overlapped atoms. Normally this would generate huge forces which would blow atoms out of the simulation box, causing LAMMPS to stop with an error.

Using this fix can overcome that problem. Forces on atoms must still be computable (which typically means two atoms must have a separation distance > 0.0). But large velocities generated by large forces are reset to a value that corresponds to a displacement of length *xmax* in a single timestep. *Xmax* is specified in distance units; see the *units* command for details. The value of *xmax* should be consistent with the neighbor skin distance and the frequency of neighbor list re-building, so that pairwise interactions are not missed on successive timesteps as atoms move. See the *neighbor* and *neigh_modify* commands for details.

Note that if a velocity reset occurs the integrator will not conserve energy. On steps where no velocity resets occur, this integrator is exactly like the *fix nve* command. Since forces are unaltered, pressures computed by thermodynamic output will still be very large for overlapped configurations.

**Note**

You should not use *fix shake* in conjunction with this fix. That is because fix shake applies constraint forces based on the predicted positions of atoms after the next timestep. It has no way of knowing the timestep may change due to this fix, which will cause the constraint forces to be invalid. A better strategy is to turn off fix shake when performing initial dynamics that need this fix, then turn fix shake on when doing normal dynamics with a fixed-size timestep.

### 2.154.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the count of how many updates of atom's velocity/position were limited by the maximum distance criterion. This should be roughly the number of atoms so affected, except that updates occur at both the beginning and end of a timestep in a velocity Verlet timestepping algorithm. This is a cumulative quantity for the current run, but is re-initialized to zero each time a run is performed. The scalar value calculated by this fix is "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages*

page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.154.5 Restrictions

none

### 2.154.6 Related commands

*fix nve*, *fix nve/noforce*, *pair_style soft*

### 2.154.7 Default

none

## 2.155 fix nve/line command

### 2.155.1 Syntax

```
fix ID group-ID nve/line
```

- ID, group-ID are documented in *fix* command
- nve/line = style name of this fix command

### 2.155.2 Examples

```
fix 1 all nve/line
```

### 2.155.3 Description

Perform constant NVE integration to update position, velocity, orientation, and angular velocity for line segment particles in the group each timestep. V is volume; E is energy. This creates a system trajectory consistent with the microcanonical ensemble. See *Howto spherical* page for an overview of using line segment particles.

This fix differs from the *fix nve* command, which assumes point particles and only updates their position and velocity.

### 2.155.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.155.5 Restrictions

This fix is part of the ASPHERE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix requires that particles be line segments as defined by the *atom_style line* command.

### 2.155.6 Related commands

*fix nve*, *fix nve/asphere*

### 2.155.7 Default

none

## 2.156 fix nve/manifold/rattle command

### 2.156.1 Syntax

```
fix ID group-ID nve/manifold/rattle tol maxit manifold manifold-args keyword value ...
```

- ID, group-ID are documented in *fix* command
- nve/manifold/rattle = style name of this fix command
- tol = tolerance to which Newton iteration must converge
- maxit = maximum number of iterations to perform
- manifold = name of the manifold
- manifold-args = parameters for the manifold
- one or more keyword/value pairs may be appended

```
keyword = every
  every values = N
    N = print info about iteration every N steps. N = 0 means no output
```

### 2.156.2 Examples

```
fix 1 all nve/manifold/rattle 1e-4 10 sphere 5.0
fix step all nve/manifold/rattle 1e-8 100 ellipsoid 2.5 2.5 5.0 every 25
```

### 2.156.3 Description

Perform constant NVE integration to update position and velocity for atoms constrained to a curved surface (manifold) in the group each timestep. The constraint is handled by RATTLE *(Andersen)* written out for the special case of single-particle constraints as explained in *(Paquay)*. V is volume; E is energy. This way, the dynamics of particles constrained to curved surfaces can be studied. If combined with *fix langevin*, this generates Brownian motion of particles constrained to a curved surface. For a list of currently supported manifolds and their parameters, see the *Howto manifold* doc page.

Note that the particles must initially be close to the manifold in question. If not, RATTLE will not be able to iterate until the constraint is satisfied, and an error is generated. For simple manifolds this can be achieved with *region* and *create_atoms* commands, but for more complex surfaces it might be more useful to write a script.

The manifold args may be equal-style variables, like so:

```
variable R equal "ramp(5.0,3.0)"
fix shrink_sphere all nve/manifold/rattle 1e-4 10 sphere v_R
```

In this case, the manifold parameter will change in time according to the variable. This is not a problem for the time integrator as long as the change of the manifold is slow with respect to the dynamics of the particles. Note that if the manifold has to exert work on the particles because of these changes, the total energy might not be conserved.

---

### 2.156.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

---

### 2.156.5 Restrictions

This fix is part of the MANIFOLD package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

---

### 2.156.6 Related commands

*fix nvt/manifold/rattle*, *fix manifoldforce*

### 2.156.7 Default

every = 0, tchain = 3

---

**(Andersen)** Andersen, J. Comp. Phys. 52, 24, (1983).

**(Paquay)** Paquay and Kusters, Biophys. J., 110, 6, (2016). preprint available at arXiv:1411.3019.

## 2.157 fix nve/noforce command

Accelerator Variant: *nve/noforce/kk*

### 2.157.1 Syntax

```
fix ID group-ID nve
```

- ID, group-ID are documented in *fix* command
- nve/noforce = style name of this fix command

### 2.157.2 Examples

```
fix 3 wall nve/noforce
```

### 2.157.3 Description

Perform updates of position, but not velocity for atoms in the group each timestep. In other words, the force on the atoms is ignored and their velocity is not updated. The atom velocities are used to update their positions.

This can be useful for wall atoms, when you set their velocities, and want the wall to move (or stay stationary) in a prescribed fashion.

This can also be accomplished via the *fix setforce* command, but with fix nve/noforce, the forces on the wall atoms are unchanged, and can thus be printed by the *dump* command or queried with an equal-style *variable* that uses the fcm() group function to compute the total force on the group of atoms.

### 2.157.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.157.5 Restrictions

none

### 2.157.6 Related commands

*fix nve*

### 2.157.7 Default

none

## 2.158 fix nve/sphere command

Accelerator Variants: *nve/sphere/omp*, *nve/sphere/kk*

### 2.158.1 Syntax

```
fix ID group-ID nve/sphere
```

- ID, group-ID are documented in *fix* command
- nve/sphere = style name of this fix command

- zero or more keyword/value pairs may be appended
- keyword = *update* or *disc*

```
update value = dipole or dipole/dlm
  dipole = update orientation of dipole moment during integration
  dipole/dlm = use DLM integrator to update dipole orientation
disc value = none = treat particles as 2d discs, not spheres
```

### 2.158.2 Examples

```
fix 1 all nve/sphere
fix 1 all nve/sphere update dipole
fix 1 all nve/sphere disc
fix 1 all nve/sphere update dipole/dlm
```

### 2.158.3 Description

Perform constant NVE integration to update position, velocity, and angular velocity for finite-size spherical particles in the group each timestep. V is volume; E is energy. This creates a system trajectory consistent with the microcanonical ensemble.

This fix differs from the *fix nve* command, which assumes point particles and only updates their position and velocity.

If the *update* keyword is used with the *dipole* value, then the orientation of the dipole moment of each particle is also updated during the time integration. This option should be used for models where a dipole moment is assigned to finite-size particles, e.g. spheroids via use of the *atom_style hybrid sphere dipole* command.

The default dipole orientation integrator can be changed to the Dullweber-Leimkuhler-McLachlan integration scheme *(Dullweber)* when using *update* with the value *dipole/dlm*. This integrator is symplectic and time-reversible, giving better energy conservation and allows slightly longer timesteps at only a small additional computational cost.

If the *disc* keyword is used, then each particle is treated as a 2d disc (circle) instead of as a sphere. This is only possible for 2d simulations, as defined by the *dimension* keyword. The only difference between discs and spheres in this context is their moment of inertia, as used in the time integration.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.158.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.158.5 Restrictions

This fix requires that atoms store torque and angular velocity (omega) and a radius as defined by the *atom_style sphere* command. If the *dipole* keyword is used, then they must also store a dipole moment as defined by the *atom_style dipole* command.

All particles in the group must be finite-size spheres. They cannot be point particles.

Use of the *disc* keyword is only allowed for 2d simulations, as defined by the *dimension* keyword.

### 2.158.6 Related commands

*fix nve*, *fix nve/asphere*

### 2.158.7 Default

none

---

**(Dullweber)** Dullweber, Leimkuhler and McLachlan, J Chem Phys, 107, 5840 (1997).

## 2.159 fix nve/spin command

### 2.159.1 Syntax

```
fix ID group-ID nve/spin keyword values
```

- ID, group-ID are documented in *fix* command
- nve/spin = style name of this fix command
- keyword = *lattice*

  *lattice* value = *moving* or *frozen*
    moving = integrate both spin and atomic degress of freedom
    frozen = integrate spins on a fixed lattice

### 2.159.2 Examples

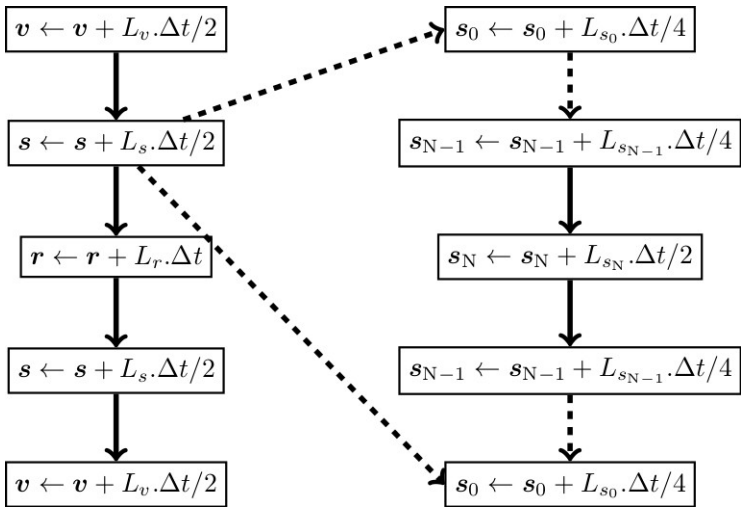
```
fix 3 all nve/spin lattice moving
fix 1 all nve/spin lattice frozen
```

### 2.159.3 Description

Perform a symplectic integration for the spin or spin-lattice system.

The *lattice* keyword defines if the spins are integrated on a lattice of fixed atoms (lattice = frozen), or if atoms are moving (lattice = moving). The first case corresponds to a spin dynamics calculation, and the second to a spin-lattice calculation. By default a spin-lattice integration is performed (lattice = moving).

The *nve/spin* fix applies a Suzuki-Trotter decomposition to the equations of motion of the spin lattice system, following the scheme:




according to the implementation reported in *(Omelyan)*.

A sectoring method enables this scheme for parallel calculations. The implementation of this sectoring algorithm is reported in *(Tranchida)*.

---

### 2.159.4 Restrictions

This fix style can only be used if LAMMPS was built with the SPIN package. See the *Build package* page for more info.

To use the spin algorithm, it is necessary to define a map with the atom_modify command. Typically, by adding the command:

```
atom_modify map array
```

before you create the simulation box. Note that the keyword "hash" instead of "array" is also valid.

### 2.159.5 Related commands

*atom_style spin*, *fix nve*

### 2.159.6 Default

The option default is lattice = moving.

---

**(Omelyan)** Omelyan, Mryglod, and Folk. Phys. Rev. Lett. 86(5), 898. (2001).

**(Tranchida)** Tranchida, Plimpton, Thibaudeau and Thompson, Journal of Computational Physics, 372, 406-425, (2018).

## 2.160 fix nve/tri command

### 2.160.1 Syntax

```
fix ID group-ID nve/tri
```

- ID, group-ID are documented in *fix* command
- nve/tri = style name of this fix command

### 2.160.2 Examples

```
fix 1 all nve/tri
```

### 2.160.3 Description

Perform constant NVE integration to update position, velocity, orientation, and angular momentum for triangular particles in the group each timestep. V is volume; E is energy. This creates a system trajectory consistent with the microcanonical ensemble. See the *Howto spherical* page for an overview of using triangular particles.

This fix differs from the *fix nve* command, which assumes point particles and only updates their position and velocity.

### 2.160.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.160.5 Restrictions

This fix is part of the ASPHERE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix requires that particles be triangles as defined by the *atom_style tri* command.

### 2.160.6 Related commands

*fix nve*, *fix nve/asphere*

### 2.160.7 Default

none

## 2.161 fix nvk command

### 2.161.1 Syntax

```
fix ID group-ID nvk
```

- ID, group-ID are documented in *fix* command
- nvk = style name of this fix command

### 2.161.2 Examples

```
fix 1 all nvk
```

### 2.161.3 Description

Perform constant kinetic energy integration using the Gaussian thermostat to update position and velocity for atoms in the group each timestep. V is volume; K is kinetic energy. This creates a system trajectory consistent with the isokinetic ensemble.

The equations of motion used are those of Minary et al in *(Minary)*, a variant of those initially given by Zhang in *(Zhang)*.

The kinetic energy will be held constant at its value given when fix nvk is initiated. If a different kinetic energy is desired, the *velocity* command should be used to change the kinetic energy prior to this fix.

---

### 2.161.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.161.5 Restrictions

The Gaussian thermostat only works when it is applied to all atoms in the simulation box. Therefore, the group must be set to all.

This fix has not yet been implemented to work with the RESPA integrator.

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.161.6 Related commands

none

### 2.161.7 Default

none

---

**(Minary)** Minary, Martyna, and Tuckerman, J Chem Phys, 18, 2510 (2003).

**(Zhang)** Zhang, J Chem Phys, 106, 6102 (1997).

## 2.162 fix nvt/asphere command

Accelerator Variants: *nvt/asphere/omp*

### 2.162.1 Syntax

```
fix ID group-ID nvt/asphere keyword value ...
```

- ID, group-ID are documented in *fix* command
- nvt/asphere = style name of this fix command
- additional thermostat related keyword/value pairs from the *fix nvt* command can be appended

### 2.162.2 Examples

```
fix 1 all nvt/asphere temp 300.0 300.0 100.0
fix 1 all nvt/asphere temp 300.0 300.0 100.0 drag 0.2
```

### 2.162.3 Description

Perform constant NVT integration to update position, velocity, orientation, and angular velocity each timestep for aspherical or ellipsoidal particles in the group using a Nose/Hoover temperature thermostat. V is volume; T is temperature. This creates a system trajectory consistent with the canonical ensemble.

This fix differs from the *fix nvt* command, which assumes point particles and only updates their position and velocity.

The thermostat is applied to both the translational and rotational degrees of freedom for the aspherical particles, assuming a compute is used which calculates a temperature that includes the rotational degrees of freedom (see below). The translational degrees of freedom can also have a bias velocity removed from them before thermostatting takes place; see the description below.

Additional parameters affecting the thermostat are specified by keywords and values documented with the *fix nvt* command. See, for example, discussion of the *temp* and *drag* keywords.

This fix computes a temperature each timestep. To do this, the fix creates its own compute of style "temp/asphere", as if this command had been issued:

```
compute fix-ID_temp group-ID temp/asphere
```

See the *compute temp/asphere* command for details. Note that the ID of the new compute is the fix-ID + underscore + "temp", and the group for the new compute is the same as the fix group.

Note that this is NOT the compute used by thermodynamic output (see the *thermo_style* command) with ID = *thermo_temp*. This means you can change the attributes of this fix's temperature (e.g. its degrees-of-freedom) via the *compute_modify* command or print this temperature during thermodynamic output via the *thermo_style custom* command using the appropriate compute-ID. It also means that changing attributes of *thermo_temp* will have no effect on this fix.

Like other fixes that perform thermostatting, this fix can be used with *compute commands* that remove a "bias" from the atom velocities. E.g. to apply the thermostat only to atoms within a spatial *region*, or to remove the center-of-mass velocity from a group of atoms, or to remove the x-component of velocity from the calculation.

This is not done by default, but only if the *fix_modify* command is used to assign a temperature compute to this fix that includes such a bias term. See the doc pages for individual *compute temp commands* to determine which ones include a bias. In this case, the thermostat works in the following manner: bias is removed from each atom, thermostatting is performed on the remaining thermal degrees of freedom, and the bias is added back in.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 2.162.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the state of the Nose/Hoover thermostat to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The *fix_modify* *temp* option is supported by this fix. You can use it to assign a *compute* you have defined to this fix which will be used in its thermostatting procedure.

The cumulative energy change in the system imposed by this fix is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* doc page for details.

This fix computes the same global scalar and global vector of quantities as does the *fix nvt* command.

This fix can ramp its target temperature over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.162.5 Restrictions

This fix is part of the ASPHERE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix requires that atoms store torque and angular momentum and a quaternion as defined by the *atom_style ellipsoid* command.

All particles in the group must be finite-size. They cannot be point particles, but they can be aspherical or spherical as defined by their shape attribute.

### 2.162.6 Related commands

*fix nvt*, *fix nve_asphere*, *fix npt_asphere*, *fix_modify*

### 2.162.7 Default

none

## 2.163 fix nvt/body command

### 2.163.1 Syntax

```
fix ID group-ID nvt/body keyword value ...
```

- ID, group-ID are documented in *fix* command
- nvt/body = style name of this fix command
- additional thermostat related keyword/value pairs from the *fix nvt* command can be appended

### 2.163.2 Examples

```
fix 1 all nvt/body temp 300.0 300.0 100.0
fix 1 all nvt/body temp 300.0 300.0 100.0 drag 0.2
```

### 2.163.3 Description

Perform constant NVT integration to update position, velocity, orientation, and angular velocity each timestep for body particles in the group using a Nose/Hoover temperature thermostat. V is volume; T is temperature. This creates a system trajectory consistent with the canonical ensemble.

This fix differs from the *fix nvt* command, which assumes point particles and only updates their position and velocity.

The thermostat is applied to both the translational and rotational degrees of freedom for the body particles, assuming a compute is used which calculates a temperature that includes the rotational degrees of freedom (see below). The translational degrees of freedom can also have a bias velocity removed from them before thermostatting takes place; see the description below.

Additional parameters affecting the thermostat are specified by keywords and values documented with the *fix nvt* command. See, for example, discussion of the *temp* and *drag* keywords.

This fix computes a temperature each timestep. To do this, the fix creates its own compute of style "temp/body", as if this command had been issued:

```
compute fix-ID_temp group-ID temp/body
```

See the *compute temp/body* command for details. Note that the ID of the new compute is the fix-ID + underscore + "temp", and the group for the new compute is the same as the fix group.

Note that this is NOT the compute used by thermodynamic output (see the *thermo_style* command) with ID = *thermo_temp*. This means you can change the attributes of this fix's temperature (e.g. its degrees-of-freedom) via the *compute_modify* command or print this temperature during thermodynamic output via the *thermo_style custom* command using the appropriate compute-ID. It also means that changing attributes of *thermo_temp* will have no effect on this fix.

Like other fixes that perform thermostatting, this fix can be used with *compute commands* that remove a "bias" from the atom velocities. E.g. to apply the thermostat only to atoms within a spatial *region*, or to remove the center-of-mass velocity from a group of atoms, or to remove the x-component of velocity from the calculation.

This is not done by default, but only if the *fix_modify* command is used to assign a temperature compute to this fix that includes such a bias term. See the doc pages for individual *compute temp commands* to determine which ones include a bias. In this case, the thermostat works in the following manner: bias is removed from each atom, thermostatting is performed on the remaining thermal degrees of freedom, and the bias is added back in.

---

### 2.163.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the state of the Nose/Hoover thermostat to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The *fix_modify temp* option is supported by this fix. You can use it to assign a *compute* you have defined to this fix which will be used in its thermostatting procedure.

The cumulative energy change in the system imposed by this fix is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* doc page for details.

This fix computes the same global scalar and global vector of quantities as does the *fix nvt* command.

This fix can ramp its target temperature over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.163.5 Restrictions

This fix is part of the BODY package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix requires that atoms store torque and angular momentum and a quaternion as defined by the *atom_style body* command.

### 2.163.6 Related commands

*fix nvt*, *fix nve_body*, *fix npt_body*, *fix_modify*

### 2.163.7 Default

none

## 2.164 fix nvt/manifold/rattle command

### 2.164.1 Syntax

```
fix ID group-ID nvt/manifold/rattle tol maxit manifold manifold-args keyword value ...
```

- ID, group-ID are documented in *fix* command
- nvt/manifold/rattle = style name of this fix command
- tol = tolerance to which Newton iteration must converge
- maxit = maximum number of iterations to perform
- manifold = name of the manifold
- manifold-args = parameters for the manifold
- one or more keyword/value pairs may be appended

```
keyword = temp or tchain or every
  temp values = Tstart Tstop Tdamp
    Tstart, Tstop = external temperature at start/end of run
    Tdamp = temperature damping parameter (time units)
  tchain value = N
    N = length of thermostat chain (1 = single thermostat)
  every value = N
    N = print info about iteration every N steps. N = 0 means no output
```

### 2.164.2 Examples

```
fix 1 all nvt/manifold/rattle 1e-4 10 cylinder 3.0 temp 1.0 1.0 10.0
```

### 2.164.3 Description

This fix combines the RATTLE-based *(Andersen)* time integrator of *fix nve/manifold/rattle (Paquay)* with a Nose-Hoover-chain thermostat to sample the canonical ensemble of particles constrained to a curved surface (manifold). This sampling does suffer from discretization bias of O(dt). For a list of currently supported manifolds and their parameters, see the *Howto manifold* doc page.

### 2.164.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

---

### 2.164.5 Restrictions

This fix is part of the MANIFOLD package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

---

### 2.164.6 Related commands

*fix nve/manifold/rattle*, *fix manifoldforce* **Default:** every = 0

---

**(Andersen)** Andersen, J. Comp. Phys. 52, 24, (1983).

**(Paquay)** Paquay and Kusters, Biophys. J., 110, 6, (2016). preprint available at arXiv:1411.3019.

## 2.165 fix nvt/sllod command

Accelerator Variants: *nvt/sllod/intel*, *nvt/sllod/omp*, *nvt/sllod/kk*

### 2.165.1 Syntax

```
fix ID group-ID nvt/sllod keyword value ...
```

- ID, group-ID are documented in *fix* command
- nvt/sllod = style name of this fix command
- zero or more keyword/value pairs may be appended

```
keyword = psllod or peculiar or kick or integrator
  psllod value = no or yes = use SLLOD or p-SLLOD variant, respectively
  peculiar value = no or yes = store velocity in the lab frame or peculiar frame,␣
↪respectively
  kick value = no or yes = whether to superimpose streaming velocity
  integrator value = reversible or legacy
    reversible = use reversible integration scheme
    legacy = use old integration scheme (see below for details)
```

- additional thermostat related keyword/value pairs from the *fix nvt* command can be appended

### 2.165.2 Examples

```
fix 1 all nvt/sllod temp 300.0 300.0 100.0
fix 1 all nvt/sllod temp 300.0 300.0 100.0 drag 0.2
```

### 2.165.3 Description

Perform constant NVT integration to update positions and velocities each timestep for atoms in the group using a Nose/Hoover temperature thermostat. V is volume; T is temperature. This creates a system trajectory consistent with the canonical ensemble.

This thermostat is used for a simulation box that is changing size and/or shape, for example in a non-equilibrium MD (NEMD) simulation. The size/shape change is induced by use of the *fix deform* command, so each point in the simulation box can be thought of as having a "streaming" velocity. This position-dependent streaming velocity is subtracted from each atom's actual velocity to yield a thermal velocity which is used for temperature computation and thermostatting. For example, if the box is being sheared in x, relative to y, then points at the bottom of the box (low y) have a small x velocity, while points at the top of the box (hi y) have a large x velocity. These velocities do not contribute to the thermal "temperature" of the atom.

**Note**

*Fix deform* has an option for remapping either atom coordinates or velocities to the changing simulation box. To use fix nvt/sllod, fix deform should NOT remap atom positions, because fix nvt/sllod adjusts the atom positions and velocities to create a velocity profile that matches the changing box size/shape. For *peculiar* = *no*, fix deform SHOULD remap atom velocities when atoms cross periodic boundaries since that is consistent with maintaining the velocity profile created by fix nvt/sllod. For *peculiar* = *yes*, fix deform SHOULD NOT remap velocities. LAMMPS will give an error if this setting is not consistent.

The SLLOD equations of motion, originally proposed by Hoover and Ladd (see *(Evans and Morriss)*), were proven to be equivalent to Newton's equations of motion for shear flow by *(Evans and Morriss)*. They were later shown to generate the desired velocity gradient and the correct production of work by stresses for all forms of homogeneous flow by *(Daivis and Todd)*.

Changed in version 8Feb2023.

For the default (*psllod* = *no*), the LAMMPS implementation adheres to the standard SLLOD equations of motion, as defined by *(Evans and Morriss)*. The option *psllod* = *yes* invokes the slightly different SLLOD variant first introduced by *(Tuckerman et al.)* as g-SLLOD and later by *(Edwards)* as p-SLLOD. In all cases, the equations of motion are coupled to a Nose/Hoover chain thermostat in a velocity Verlet formulation, closely following the implementation used for the *fix nvt* command.

**Note**

A recent (2017) book by *(Todd and Daivis)* discusses use of the SLLOD method and non-equilibrium MD (NEMD) thermostatting generally, for both simple and complex fluids, e.g. molecular systems. The latter can be tricky to do correctly.

Changed in version 11Feb2026.

With *integrator* = *reversible* (the default), the numerical integration scheme closely follows the one described by *(Sanderson and Searles)*, and has been validated to produce work equal to the expected analytical value and thereby preserve a conserved quantity. With *integrator* = *legacy* (the behavior of previous LAMMPS versions), this quantity is not precisely conserved, and stresses may be slightly too high under high flow rates.

For SLLOD simulations with a constant flow tensor, e.g. for calculating viscosity, *fix deform* should be used with the *trate* style for x/y/z deformation and either the *erate* or *erate/rescale* style for xy/xz/yz. For mixed flows, *erate/rescale* is required to maintain a constant flow tensor, and LAMMPS will issue a warning in such cases. These warnings can be safely ignored if using fix nvt/sllod only to adjust the system size before an equilibration step.

The *peculiar* flag specifies whether velocity should be stored in the peculiar frame of reference (i.e. relative to the

flow), or in the laboratory frame. With *peculiar* = *no* (the default), velocity is stored in the lab-frame, and will include the streaming component due to the flow. This is needed for calculating properties like angular velocity, but makes the SLLOD equations of motion more difficult to integrate and can be less performant. With *peculiar* = *yes*, the streaming component is NOT stored, so the velocities reported by LAMMPS are the ones relative to the streaming velocity. If lab-frame velocities are not required, storing velocity in the peculiar frame should generally give the same results with better performance.

A key aspect of the SLLOD algorithm is that when the flow is "turned on", particles receive an initial "kick" to their momentum as viewed from the lab-frame, equivalent to superimposing the streaming velocity. This reduces the time required to reach a steady state, and is required to preserve connections with response theory. This "kick" can be applied manually using the *velocity* command, but care must be taken to ensure velocities are compatible with the box deformation, which requires treating the effective origin for elongational flows as the center of the box (ignoring tilt factors), while using the lower box corner as the origin for shear flows. Instead, to apply the kick automatically when *run* or another similar command is next called, set *kick* = *yes* (the default when *peculiar* = *no*). The kick will only be applied ONCE. If another kick is required (e.g. if changing parameters of *fix deform*), it can be queued up by using the *fix_modify* command. Note, the kick will be applied even when using `run 0`, which can be useful to first remove the current streaming velocity and then apply a new one. For example:

```
fix 1 all deform 0 xy erate 0.5 remap v
fix 2 all nvt/sllod temp 1 1 0.1 peculiar no kick yes
run 100    # Streaming velocity superimposed here
# Remove current streaming velocity
unfix 1
fix 1 all deform 0 xy erate -0.5 remap v
fix_modify 2 kick yes
run 0
# Continue with new streaming velocity
unfix 1
fix 1 all deform 0 xy erate 1.0 remap v
fix_modify 2 kick yes
run 100
```

If velocity is stored in the peculiar frame (*peculiar* = *yes*), then the *kick* flag is ignored and the same behavior as above can be achieved with

```
fix 1 all deform 0 xy erate 0.5 remap v
fix 2 all nvt/sllod temp 1 1 0.1 peculiar yes
run 100
# Switch to new deformation rate
unfix 1
fix 1 all deform 0 xy erate 1.0 remap v
run 100
```

**Note**

Some *fix deform* styles like *final*, *scale*, *vel* and *delta* set the deformation rate to zero when "run 0" is called, in which case no kick will be applied for those components of the flow. If a kick is desired in such cases, it is recommended to initialize fix nvt/sllod with "kick no", and then set "kick yes" using *fix_modify* immediately before the next *run* command with non-zero length. Similarly, the *volume*, *wiggle* and *variable* styles do not set a deformation rate during run initialization and are therefore not compatible with the "kick yes" option.

Additional parameters affecting the thermostat are specified by keywords and values documented with the *fix nvt* command. See, for example, discussion of the *temp* and *drag* keywords.

This fix computes a temperature each timestep. To do this, the fix creates its own compute of style "temp" (if *peculiar* = *yes*) or "temp/deform" (if *peculiar* = *no*), as if this command had been issued:

```
compute fix-ID_temp group-ID temp          # if peculiar = yes
compute fix-ID_temp group-ID temp/deform   # if peculiar = no
```

See the *compute temp/deform* command for details. Note that the ID of the new compute is the fix-ID + underscore + "temp", and the group for the new compute is the same as the fix group.

Note that this is NOT the compute used by thermodynamic output (see the *thermo_style* command) with ID = *thermo_temp*. This means you can change the attributes of this fix's temperature (e.g. its degrees-of-freedom) via the *compute_modify* command or print this temperature during thermodynamic output via the *thermo_style custom* command using the appropriate compute-ID. It also means that changing attributes of *thermo_temp* will have no effect on this fix.

Like other fixes that perform thermostatting, this fix can be used with *compute commands* that remove a "bias" from the atom velocities. E.g. to apply the thermostat only to atoms within a spatial *region*, or to remove the center-of-mass velocity from a group of atoms, or to remove the x-component of velocity from the calculation.

This is not done by default, but only if the *fix_modify* command is used to assign a temperature compute to this fix that includes such a bias term. See the doc pages for individual *compute temp commands* to determine which ones include a bias. In this case, if *integrator* = *legacy* or *integrator* = *reversible* and *peculiar* = *yes*, the thermostat works in the following manner: bias is removed from each atom, thermostatting is performed on the remaining thermal degrees of freedom, and the bias is added back in. If *integrator* = *reversible* and *peculiar* = *no* the temperature compute MUST be of type *compute temp/deform*, but it may use a user-specified internal temperature compute to achieve the same effect.

If *integrator* = *reversible*, no bias is removed when applying SLLOD to atom velocities when *peculiar* = *yes*, and only the bias due to box deformation is removed when *peculiar* = *no*. If *integrator* = *legacy*, all bias components (box deformation plus any bias from the internal temperature compute of compute temp/deform) are removed before applying SLLOD to the velocities.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 2.165.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the state of the Nose/Hoover thermostat to *binary restart files*. Similarly, it stores whether or not the velocity kick has been applied so that it is not applied a second time after a restart. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The *fix_modify* *temp* option is supported by this fix. You can use it to assign a *compute* you have defined to this fix which will be used in its thermostatting procedure. The *kick* option is also supported as described above.

The cumulative energy change in the system imposed by this fix is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* doc page for details. Note, this does NOT include the work done to drive the flow, so those values are expected to change with time. Details of the expected rate of change in *econserve* are discussed in *(Sanderson and Searles)*.

This fix computes the same global scalar and global vector of quantities as does the *fix nvt* command.

This fix can ramp its target temperature over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.165.5 Restrictions

This fix works best without Nose-Hoover chain thermostats, i.e. using *tchain* = 1. Setting *tchain* to larger values can result in poor equilibration.

### 2.165.6 Related commands

*fix nve*, *fix nvt*, *fix temp/rescale*, *fix langevin*, *fix_modify*, *compute temp/deform*

### 2.165.7 Default

Same as *fix nvt*, except *tchain* = 1, *psllod* = *no*, *peculiar* = *no*, *kick* = *yes*, *integrator* = *reversible*.

---

**(Evans and Morriss)** Evans and Morriss, Phys Rev A, 30, 1528 (1984).

**(Daivis and Todd)** Daivis and Todd, J Chem Phys, 124, 194103 (2006).

**(Todd and Daivis)** Todd and Daivis, Nonequilibrium Molecular Dynamics (book), Cambridge University Press, (2017) https://doi.org/10.1017/9781139017848.

**(Tuckerman et al.)** Tuckerman, Mundy, Balasubramanian, and Klein, J Chem Phys 106, 5615 (1997).

**(Edwards)** Edwards, Baig, and Keffer, J Chem Phys 124, 194104 (2006).

**(Sanderson and Searles)** Sanderson and Searles, J Chem Phys, 164, 084121 (2026) https://doi.org/10.1063/5.0315430.

## 2.166 fix nvt/sllod/eff command

### 2.166.1 Syntax

```
fix ID group-ID nvt/sllod/eff keyword value ...
```

- ID, group-ID are documented in *fix* command
- nvt/sllod/eff = style name of this fix command
- zero or more keyword/value pairs may be appended

  keyword = *psllod*
    *psllod* value = *no* or *yes* = use SLLOD or p-SLLOD variant, respectively

- additional thermostat related keyword/value pairs from the *fix nvt/eff* command may be appended, too.

### 2.166.2 Examples

```
fix 1 all nvt/sllod/eff temp 300.0 300.0 0.1
fix 1 all nvt/sllod/eff temp 300.0 300.0 0.1 drag 0.2
```

### 2.166.3 Description

Perform constant NVT integration to update positions and velocities each timestep for nuclei and electrons in the group for the *electron force field* model, using a Nose/Hoover temperature thermostat. V is volume; T is temperature. This creates a system trajectory consistent with the canonical ensemble.

The operation of this fix is exactly like that described by the *fix nvt/sllod* command, except that the radius and radial velocity of electrons are also updated and thermostatted. Likewise the temperature calculated by the fix, using the compute it creates (as discussed in the *fix nvt, npt, and nph* doc page), is performed with a *compute temp/deform/eff* command (if *peculiar* = *no*) or a *compute temp/eff* command (if *peculiar* = *yes*) that includes the eFF contribution to the temperature from the electron radial velocity.

### 2.166.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the state of the Nose/Hoover thermostat to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The *fix_modify* *temp* option is supported by this fix. You can use it to assign a *compute* you have defined to this fix which will be used in its thermostatting procedure. The *kick* option is also supported as described by *fix nvt/sllod*.

The cumulative energy change in the system imposed by this fix is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* doc page for details. Note, as for *fix nvt/sllod* this does NOT include the work done to drive the flow, so those values are expected to change with time.

This fix computes the same global scalar and global vector of quantities as does the *fix nvt/eff* command.

This fix can ramp its target temperature over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.166.5 Restrictions

This fix is part of the EFF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix works best without Nose-Hoover chain thermostats, i.e. using tchain = 1. Setting tchain to larger values can result in poor equilibration.

### 2.166.6 Related commands

*fix nve/eff*, *fix nvt/eff*, *fix langevin/eff*, *fix nvt/sllod*, *fix_modify*, *compute temp/deform/eff*

### 2.166.7 Default

Same as *fix nvt/eff*, except *tchain* = 1, *psllod* = *no*, *peculiar* = *no*, *kick* = *yes*, *integrator* = *reversible*.

---

**(Tuckerman)** Tuckerman, Mundy, Balasubramanian, Klein, J Chem Phys, 106, 5615 (1997).

## 2.167 fix nvt/sphere command

Accelerator Variants: *nvt/sphere/kk*, *nvt/sphere/omp*

### 2.167.1 Syntax

```
fix ID group-ID nvt/sphere keyword value ...
```

- ID, group-ID are documented in *fix* command
- nvt/sphere = style name of this fix command
- zero or more keyword/value pairs may be appended
- keyword = *disc*

  *disc* value = none = treat particles as 2d discs, not spheres

- NOTE: additional thermostat and dipole related keyword/value pairs from the *fix nvt* command can be appended

### 2.167.2 Examples

```
fix 1 all nvt/sphere temp 300.0 300.0 100.0
fix 1 all nvt/sphere temp 300.0 300.0 100.0 disc
fix 1 all nvt/sphere temp 300.0 300.0 100.0 drag 0.2
fix 1 all nvt/sphere temp 300.0 300.0 100.0 update dipole
```

### 2.167.3 Description

Perform constant NVT integration to update position, velocity, and angular velocity each timestep for finite-size spherical particles in the group using a Nose/Hoover temperature thermostat. V is volume; T is temperature. This creates a system trajectory consistent with the canonical ensemble.

This fix differs from the *fix nvt* command, which assumes point particles and only updates their position and velocity.

The thermostat is applied to both the translational and rotational degrees of freedom for the spherical particles, assuming a compute is used which calculates a temperature that includes the rotational degrees of freedom (see below). The translational degrees of freedom can also have a bias velocity removed from them before thermostatting takes place; see the description below.

If the *disc* keyword is used, then each particle is treated as a 2d disc (circle) instead of as a sphere. This is only possible for 2d simulations, as defined by the *dimension* keyword. The only difference between discs and spheres in this context is their moment of inertia, as used in the time integration.

Additional parameters affecting the thermostat are specified by keywords and values documented with the *fix nvt* command. See, for example, discussion of the *temp* and *drag* keywords.

This fix computes a temperature each timestep. To do this, the fix creates its own compute of style "temp/sphere", as if this command had been issued:

```
compute fix-ID_temp group-ID temp/sphere
```

See the *compute temp/sphere* command for details. Note that the ID of the new compute is the fix-ID + underscore + "temp", and the group for the new compute is the same as the fix group.

Note that this is NOT the compute used by thermodynamic output (see the *thermo_style* command) with ID = *thermo_temp*. This means you can change the attributes of this fix's temperature (e.g. its degrees-of-freedom) via the *compute_modify* command or print this temperature during thermodynamic output via the *thermo_style custom* command using the appropriate compute-ID. It also means that changing attributes of *thermo_temp* will have no effect on this fix.

Like other fixes that perform thermostatting, this fix can be used with *compute commands* that remove a "bias" from the atom velocities. E.g. to apply the thermostat only to atoms within a spatial *region*, or to remove the center-of-mass velocity from a group of atoms, or to remove the x-component of velocity from the calculation.

This is not done by default, but only if the *fix_modify* command is used to assign a temperature compute to this fix that includes such a bias term. See the doc pages for individual *compute temp commands* to determine which ones include a bias. In this case, the thermostat works in the following manner: bias is removed from each atom, thermostatting is performed on the remaining thermal degrees of freedom, and the bias is added back in.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 2.167.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the state of the Nose/Hoover thermostat to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The *fix_modify* *temp* option is supported by this fix. You can use it to assign a *compute* you have defined to this fix which will be used in its thermostatting procedure.

The cumulative energy change in the system imposed by this fix is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* doc page for details.

This fix computes the same global scalar and global vector of quantities as does the *fix nvt* command.

This fix can ramp its target temperature over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.167.5 Restrictions

This fix requires that atoms store torque and angular velocity (omega) and a radius as defined by the *atom_style sphere* command.

All particles in the group must be finite-size spheres. They cannot be point particles.

Use of the *disc* keyword is only allowed for 2d simulations, as defined by the *dimension* keyword.

### 2.167.6 Related commands

***fix nvt***, ***fix nve_sphere***,
*fix nvt_asphere*, *fix npt_sphere*, *fix_modify*

### 2.167.7 Default

none

# 2.168 fix oneway command

Accelerator Variant: *oneway/kk*

## 2.168.1 Syntax

```
fix ID group-ID oneway N region-ID direction
```

- ID, group-ID are documented in *fix* command
- oneway = style name of this fix command
- N = apply this fix every this many timesteps
- region-ID = ID of region where fix is active
- direction = *x* or -*x* or *y* or -*y* or *z* or -*z* = coordinate and direction of the oneway constraint

## 2.168.2 Examples

```
fix 1 ions oneway 10 semi -x
fix 2 all oneway 1 left -z
fix 3 all oneway 1 right z
```

## 2.168.3 Description

Enforce that particles in the group and in a given region can only move in one direction. This is done by reversing a particle's velocity component, if it has the wrong sign in the specified dimension. The effect is that the particle moves in one direction only.

This can be used, for example, as a simple model of a semi-permeable membrane, or as an implementation of Maxwell's demon.

---

## 2.168.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.168.5 Restrictions

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.168.6 Related commands

*fix wall/reflect* command

### 2.168.7 Default

none

## 2.169 fix orient/fcc command

Accelerator Variants: *orient/fcc/omp*

## 2.170 fix orient/bcc command

Accelerator Variants: *orient/bcc/omp*

### 2.170.1 Syntax

```
fix ID group-ID orient/fcc nstats dir alat dE cutlo cuthi file0 file1
fix ID group-ID orient/bcc nstats dir alat dE cutlo cuthi file0 file1
```

- ID, group-ID are documented in *fix* command
- nstats = print stats every this many steps, 0 = never
- dir = 0/1 for which crystal is used as reference
- alat = fcc/bcc cubic lattice constant (distance units)
- dE = energy added to each atom (energy units)
- cutlo,cuthi = values between 0.0 and 1.0, cutlo < cuthi
- file0,file1 = files that specify orientation of each grain

### 2.170.2 Examples

```
fix gb all orient/fcc 0 1 4.032008 0.001 0.25 0.75 xi.vec chi.vec
fix gb all orient/bcc 0 1 2.882 0.001 0.25 0.75 ngb.left ngb.right
```

### 2.170.3 Description

The fix applies an orientation-dependent force to atoms near a planar grain boundary which can be used to induce grain boundary migration (in the direction perpendicular to the grain boundary plane). The motivation and explanation of this force and its application are described in *(Janssens)*. The adaptation to bcc crystals is described in *(Wicaksono1)*. The computed force is only applied to atoms in the fix group.

The basic idea is that atoms in one grain (on one side of the boundary) have a potential energy dE added to them. Atoms in the other grain have 0.0 potential energy added. Atoms near the boundary (whose neighbor environment is intermediate between the two grain orientations) have an energy between 0.0 and dE added. This creates an effective driving force to reduce the potential energy of atoms near the boundary by pushing them towards one of the grain

orientations. For dir = 1 and dE > 0, the boundary will thus move so that the grain described by file0 grows and the grain described by file1 shrinks. Thus this fix is designed for simulations of two-grain systems, either with one grain boundary and free surfaces parallel to the boundary, or a system with periodic boundary conditions and two equal and opposite grain boundaries. In either case, the entire system can displace during the simulation, and such motion should be accounted for in measuring the grain boundary velocity.

The potential energy added to atom I is given by these formulas

$$\xi_i = \sum_{j=1}^{12} \left| \mathbf{r}_j - \mathbf{r}_j^{\mathrm{I}} \right| \qquad (1)$$

$$\xi_{\mathrm{IJ}} = \sum_{j=1}^{12} \left| \mathbf{r}_j^{\mathrm{J}} - \mathbf{r}_j^{\mathrm{I}} \right| \qquad (2)$$

$$\xi_{\mathrm{low}} = \mathrm{cutlo}\, \xi_{\mathrm{IJ}} \qquad (3)$$

$$\xi_{\mathrm{high}} = \mathrm{cuthi}\, \xi_{\mathrm{IJ}} \qquad (4)$$

$$\omega_i = \frac{\pi}{2} \frac{\xi_i - \xi_{\mathrm{low}}}{\xi_{\mathrm{high}} - \xi_{\mathrm{low}}} \qquad (5)$$

$$\begin{aligned} u_i &= 0 && \text{for} \quad \xi_i < \xi_{\mathrm{low}} \\ &= \mathrm{dE}\, \frac{1 - \cos(2\omega_i)}{2} && \text{for} \quad \xi_{\mathrm{low}} < \xi_i < \xi_{\mathrm{high}} \\ &= \mathrm{dE} && \text{for} \quad \xi_{\mathrm{high}} < \xi_i \end{aligned} \qquad (6)$$

which are fully explained in *(Janssens)*. For fcc crystals this order parameter Xi for atom I in equation (1) is a sum over the 12 nearest neighbors of atom I. For bcc crystals it is the corresponding sum of the 8 nearest neighbors. Rj is the vector from atom I to its neighbor J, and RIj is a vector in the reference (perfect) crystal. That is, if dir = 0/1, then RIj is a vector to an atom coord from file 0/1. Equation (2) gives the expected value of the order parameter XiIJ in the other grain. Hi and lo cutoffs are defined in equations (3) and (4), using the input parameters *cutlo* and *cuthi* as thresholds to avoid adding grain boundary energy when the deviation in the order parameter from 0 or 1 is small (e.g. due to thermal fluctuations in a perfect crystal). The added potential energy Ui for atom I is given in equation (6) where it is interpolated between 0 and dE using the two threshold Xi values and the Wi value of equation (5).

The derivative of this energy expression gives the force on each atom which thus depends on the orientation of its neighbors relative to the 2 grain orientations. Only atoms near the grain boundary feel a net force which tends to drive them to one of the two grain orientations.

In equation (1), the reference vector used for each neighbor is the reference vector closest to the actual neighbor position. This means it is possible two different neighbors will use the same reference vector. In such cases, the atom in question is far from a perfect orientation and will likely receive the full dE addition, so the effect of duplicate reference vector usage is small.

The *dir* parameter determines which grain wants to grow at the expense of the other. A value of 0 means the first grain will shrink; a value of 1 means it will grow. This assumes that *dE* is positive. The reverse will be true if *dE* is negative.

The *alat* parameter is the cubic lattice constant for the fcc or bcc material and is only used to compute a cutoff distance of 1.57 * alat / sqrt(2) for finding the 12 or 8 nearest neighbors of each atom (which should be valid for an fcc or bcc crystal). A longer/shorter cutoff can be imposed by adjusting *alat*. If a particular atom has less than 12 or 8 neighbors within the cutoff, the order parameter of equation (1) is effectively multiplied by 12 or 8 divided by the actual number of neighbors within the cutoff.

The *dE* parameter is the maximum amount of additional energy added to each atom in the grain which wants to shrink.

The *cutlo* and *cuthi* parameters are used to reduce the force added to bulk atoms in each grain far away from the boundary. An atom in the bulk surrounded by neighbors at the ideal grain orientation would compute an order parameter of 0 or 1 and have no force added. However, thermal vibrations in the solid will cause the order parameters to be greater than 0 or less than 1. The cutoff parameters mask this effect, allowing forces to only be added to atoms with order-parameters between the cutoff values.

*File0* and *file1* are filenames for the two grains which each contain 6 vectors (6 lines with 3 values per line) which specify the grain orientations. Each vector is a displacement from a central atom (0,0,0) to a nearest neighbor atom in an fcc lattice at the proper orientation. The vector lengths should all be identical since an fcc lattice has a coordination number of 12. Only 6 are listed due to symmetry, so the list must include one from each pair of equal-and-opposite neighbors. A pair of orientation files for a Sigma=5 tilt boundary are shown below. A tutorial that can help for writing the orientation files is given in *(Wicaksono2)*

### 2.170.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify* *energy* option is supported by this fix to add the potential energy of atom interactions with the grain boundary driving force to the global potential energy of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify energy no*.

The *fix_modify* *respa* option is supported by these fixes. This allows to set at which level of the *r-RESPA* integrator a fix is adding its forces. Default is the outermost level.

This fix calculates a global scalar which can be accessed by various *output commands*. The scalar is the potential energy change due to this fix. The scalar value calculated by this fix is "extensive".

This fix also calculates a per-atom array which can be accessed by various *output commands*. The array stores the order parameter Xi and normalized order parameter (0 to 1) for each atom. The per-atom values can be accessed on any timestep.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

This fix is not invoked during *energy minimization*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.170.5 Restrictions

These fixes are part of the ORIENT package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

These fixes should only be used with fcc or bcc lattices.

### 2.170.6 Related commands

*fix_modify*

### 2.170.7 Default

none

---

**(Janssens)** Janssens, Olmsted, Holm, Foiles, Plimpton, Derlet, Nature Materials, 5, 124-127 (2006).

**(Wicaksono1)** Wicaksono, Sinclair, Militzer, Computational Materials Science, 117, 397-405 (2016).

**(Wicaksono2)** Wicaksono, figshare, https://doi.org/10.6084/m9.figshare.1488628.v1 (2015).

---

For illustration purposes, here are example files that specify a Sigma=5 <100> tilt boundary. This is for a lattice constant of 3.5706 Angs.

file0:

```
 0.798410432046075    1.785300000000000    1.596820864092150
-0.798410432046075    1.785300000000000   -1.596820864092150
 2.395231296138225    0.000000000000000    0.798410432046075
 0.798410432046075    0.000000000000000   -2.395231296138225
 1.596820864092150    1.785300000000000   -0.798410432046075
 1.596820864092150   -1.785300000000000   -0.798410432046075
```

file1:

```
-0.798410432046075    1.785300000000000    1.596820864092150
 0.798410432046075    1.785300000000000   -1.596820864092150
 0.798410432046075    0.000000000000000    2.395231296138225
 2.395231296138225    0.000000000000000   -0.798410432046075
 1.596820864092150    1.785300000000000    0.798410432046075
 1.596820864092150   -1.785300000000000    0.798410432046075
```

## 2.171 fix orient/eco command

Accelerator Variants: *orient/eco/omp*

```
fix ID group-ID orient/eco u0 eta cutoff orientationsFile
```

- ID, group-ID are documented in fix command
- u0 = energy added to each atom (energy units)
- eta = cutoff value (usually 0.25)
- cutoff = cutoff radius for orientation parameter calculation
- orientationsFile = file that specifies orientation of each grain

### 2.171.1 Examples

```
fix gb all orient/eco 0.08 0.25 3.524 sigma5.ori
```

## 2.171.2 Description

The fix applies a synthetic driving force to a grain boundary which can be used for the investigation of grain boundary motion. The affiliation of atoms to either of the two grains forming the grain boundary is determined from an orientation-dependent order parameter as described in *(Ulomek)*. The potential energy of atoms is either increased by an amount of 0.5*u0* or -0.5*u0* according to the orientation of the surrounding crystal. This creates a potential energy gradient which pushes atoms near the grain boundary to orient according to the energetically favorable grain orientation. This fix is designed for applications in bicrystal system with one grain boundary and open ends, or two opposite grain boundaries in a periodic system. In either case, the entire system can experience a displacement during the simulation which needs to be accounted for in the evaluation of the grain boundary velocity. While the basic method is described in *(Ulomek)*, the implementation follows the efficient implementation from *(Schratt & Mohles)*. The synthetic potential energy added to an atom j is given by the following formulas

$$w(|\vec{r}_{jk}|) = w_{jk} = \begin{cases} \frac{|\vec{r}_{jk}|^4}{r_{\mathrm{cut}}^4} - 2\frac{|\vec{r}_{jk}|^2}{r_{\mathrm{cut}}^2} + 1, & |\vec{r}_{jk}| < r_{\mathrm{cut}} \\ 0, & |\vec{r}_{jk}| \geq r_{\mathrm{cut}} \end{cases}$$

$$\chi_j = \frac{1}{N}\sum_{l=1}^{3}\left[\left|\psi_l^{\mathrm{I}}(\vec{r}_j)\right|^2 - \left|\psi_l^{\mathrm{II}}(\vec{r}_j)\right|^2\right]$$

$$\psi_l^{\mathrm{X}}(\vec{r}_j) = \sum_{k\in\blacksquare_j} w_{jk}\exp\left(\mathrm{i}\vec{r}_{jk}\cdot\vec{q}_l^{\mathrm{X}}\right)$$

$$u(\chi_j) = \frac{u_0}{2}\begin{cases} 1, & \chi_j \geq \eta \\ \sin\left(\frac{\pi\chi_j}{2\eta}\right), & -\eta < \chi_j < \eta \\ -1, & \chi_j \leq -\eta \end{cases}$$

which are fully explained in *(Ulomek)* and *(Schratt & Mohles)*.

The force on each atom is the negative gradient of the synthetic potential energy. It depends on the surrounding of this atom. An atom far from the grain boundary does not experience a synthetic force as its surrounding is that of an oriented single crystal and thermal fluctuations are masked by the parameter *eta*. Near the grain boundary however, the gradient is nonzero and synthetic force terms are computed. The orientationsFile specifies the perfect oriented crystal basis vectors for the two adjoining crystals. The first three lines (line=row vector) for the energetically penalized and the last three lines for the energetically favored grain assuming *u0* is positive. For negative *u0*, this is reversed. With the *cutoff* parameter, the size of the region around each atom which is used in the order parameter computation is defined. The cutoff must be smaller than the interaction range of the MD potential. It should at least include the nearest neighbor shell. For high temperatures or low angle grain boundaries, it might be beneficial to increase the cutoff in order to get a more precise identification of the atoms surrounding. However, computation time will increase as more atoms are considered in the order parameter and force computation. It is also worth noting that the cutoff radius must not exceed the communication distance for ghost atoms in LAMMPS. With orientationsFile, the 6 oriented crystal basis vectors is specified. Each line of the input file contains the three components of a primitive lattice vector oriented according to the grain orientation in the simulation box. The first (last) three lines correspond to the primitive lattice vectors of the first (second) grain. An example for a $\Sigma\langle 001\rangle$ mis-orientation is given at the end.

If no synthetic energy difference between the grains is created, $u0 = 0$, the force computation is omitted. In this case, still, the order parameter of the driving force is computed and can be used to track the grain boundary motion throughout the simulation.

## 2.171.3 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify energy* option is supported by this fix to add the potential energy of atom interactions with the grain boundary driving force to the global potential energy of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify energy no*.

This fix calculates a per-atom array with 2 columns, which can be accessed by indices 1-1 by any command that uses per-atom values from a fix as input. See the *Howto output* doc page for an overview of LAMMPS output options.

The first column is the order parameter for each atom; the second is the thermal masking value for each atom. Both are described above.

No parameter of this fix can be used with the start/stop keywords of the run command. This fix is not invoked during energy minimization.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.171.4 Restrictions

This fix is part of the ORIENT package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.171.5 Related commands

*fix_modify*

*fix_orient*

### 2.171.6 Default

none

---

**(Ulomek)** Ulomek, Brien, Foiles, Mohles, Modelling Simul. Mater. Sci. Eng. 23 (2015) 025007

**(Schratt & Mohles)** Schratt, Mohles. Comp. Mat. Sci. 182 (2020) 109774

---

For illustration purposes, here is an example file that specifies a $\Sigma = 5\langle 001 \rangle$ tilt grain boundary. This is for a lattice constant of 3.52 Angstrom:

```
sigma5.ori:

1.671685  0.557228  1.76212
0.557228 -1.671685  1.76212
2.228913 -1.114456  0.00000
0.557228  1.671685  1.76212
1.671685 -0.557228  1.76212
2.228913  1.114456  0.00000
```

# 2.172 fix pafi command

## 2.172.1 Syntax

```
fix ID group-ID pafi compute-ID Temp Tdamp seed keyword values...
```

- ID, group-ID are documented in *fix* command
- pafi = style name of this fix command
- compute-ID = ID of a *compute property/atom* that holds data used by this fix
- Temp = desired temperature (temperature units)
- Tdamp = damping parameter (time units)
- seed = random number seed to use for white noise (positive integer)
- keyword = *overdamped* or *com*

```
overdamped value = yes or no or 1 or 0
  yes or 1 = Brownian (overdamped) integration in hyperplane
  no or 0 = Langevin integration in hyperplane
com value = yes or no or 1 or 0
  yes or 1 = zero linear momentum, fixing center or mass (recommended)
  no or 0 = do not zero linear momentum, allowing center of mass drift
```

## 2.172.2 Examples

```
compute pa all property/atom d_nx d_ny d_nz d_dnx d_dny d_dnz d_ddnx d_ddny d_ddnz
run 0 post no
fix hp all pafi pa 500.0 0.01 434 overdamped yes
```

## 2.172.3 Description

Perform Brownian or Langevin integration whilst constraining the system to lie in some hyperplane, which is expected to be the tangent plane to some reference pathway in a solid state system. The instantaneous value of a modified force projection is also calculated, whose time integral can be shown to be equal to the true free energy gradient along the minimum free energy path local to the reference pathway. A detailed discussion of the projection technique can be found in *(Swinburne)*.

This fix can be used with LAMMPS as demonstrated in examples/PACKAGES/pafi, though it is primarily intended to be coupled with the PAFI C++ code, developed at https://github.com/tomswinburne/pafi, which distributes multiple LAMMPS workers in parallel to compute and collate hyperplane-constrained averages, allowing the calculation of free energy barriers and pathways.

A *compute property/atom* must be provided with 9 fields per atom coordinate, which in order are the x,y,z coordinates of a configuration on the reference path, the x,y,z coordinates of the path tangent (derivative of path position with path coordinate) and the x,y,z coordinates of the change in tangent (derivative of path tangent with path coordinate).

A 4-element vector is also calculated by this fix. The 4 components are the modified projected force, its square, the expected projection of the minimum free energy path tangent on the reference path tangent and the minimum image distance between the current configuration and the reference configuration, projected along the path tangent. This latter value should be essentially zero.

**Note**

When com=yes/1, which is recommended, the provided tangent vector must also have zero center of mass. This can be achieved by subtracting from each coordinate of the path tangent the average x,y,z value. The PAFI C++ code (see above) can generate these paths for use in LAMMPS.

**Note**

When overdamped=yes/1, the Tdamp parameter should be around 5-10 times smaller than that used in typical Langevin integration. See *fix langevin* for typical values.

### 2.172.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix produces a global vector each timestep which can be accessed by various *output commands*.

### 2.172.5 Restrictions

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.172.6 Default

The option defaults are com = *yes*, overdamped = *no*

---

**(Swinburne)** Swinburne and Marinica, Physical Review Letters, 120, 1 (2018)

## 2.173 fix pair command

### 2.173.1 Syntax

```
fix ID group-ID pair N pstyle name flag ...
```

- ID, group-ID are documented in *fix* command
- pair = style name of this fix command
- N = invoke this fix once every N timesteps
- pstyle = name of pair style to extract info from (e.g. eam)
- one or more name/flag pairs can be listed
- name = name of quantity the pair style allows extraction of
- flag = 1 if pair style needs to be triggered to produce data for name, 0 if not

### 2.173.2 Examples

```
fix request all pair 100 eam rho 0
fix request all pair 100 amoeba uind 0 uinp 0
```

### 2.173.3 Description

Added in version 15Sep2022.

Extract per-atom quantities from a pair style and store them in this fix so they can be accessed by other LAMMPS commands, e.g. by a *dump* command or by another *fix*, *compute*, or *variable* command.

These are example use cases:

- extract per-atom density from *pair_style eam* to a dump file
- extract induced dipoles from *pair_style amoeba* to a dump file
- extract accuracy metrics from a machine-learned potential to trigger output when a condition is met (see the *dump_modify skip* command)

The *N* argument determines how often the fix is invoked.

The *pstyle* argument is the name of the pair style. It can be a sub-style used in a *pair_style hybrid* command. If there are multiple sub-styles using the same pair style, then *pstyle* should be specified as "style:N", where *N* is the number of the instance of the pair style you wish monitor (e.g., the first or second). For example, *pstyle* could be specified as "pace/extrapolation" or "amoeba" or "eam:1" or "eam:2".

One or more *name/flag* pairs of arguments follow. Each *name* is a per-atom quantity which the pair style must recognize as an extraction request. See the doc pages for individual *pair_styles* to see what fix pair requests (if any) they support.

The *flag* setting determines whether this fix will also trigger the pair style to compute the named quantity so it can be extracted. If the quantity is always computed by the pair style, no trigger is needed; specify *flag* = 0. If the quantity is not always computed (e.g. because it is expensive to calculate), then specify *flag* = 1. This will trigger the quantity to be calculated only on timesteps it is needed. Again, see the doc pages for individual *pair_styles* to determine which fix pair requests (if any) need to be triggered with a *flag* = 1 setting.

The per-atom data extracted from the pair style is stored by this fix as either a per-atom vector or array. If there is only one *name* argument specified and the pair style computes a single value for each atom, then this fix stores it as a per-atom vector. Otherwise a per-atom array is created, with its data in the order of the *name* arguments.

For example, *pair_style amoeba* allows extraction of two named quantities: "uind" and "uinp", both of which are 3-vectors for each atom, i.e. dipole moments. In the example below a 6-column per-atom array will be created. Columns 1-3 will store the "uind" values; columns 4-6 will store the "uinp" values.

```
pair_style amoeba
fix ex all pair 10 amoeba uind 0 uinp 0
```

### 2.173.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

As explained above, this fix produces a per-atom vector or array which can be accessed by various *output commands*. If an array is produced, the number of columns is the sum of the number of per-atom quantities produced by each *name* argument requested from the pair style.

### 2.173.5 Restrictions

none

### 2.173.6 Related commands

*compute pair*

### 2.173.7 Default

none

## 2.174 fix phonon command

### 2.174.1 Syntax

```
fix ID group-ID phonon N Noutput Nwait map_file prefix keyword values ...
```

- ID, group-ID are documented in *fix* command
- phonon = style name of this fix command
- N = measure the Green's function every this many timesteps
- Noutput = output the dynamical matrix every this many measurements
- Nwait = wait this many timesteps before measuring
- map_file = *file* or *GAMMA*

  ```
  file is the file that contains the mapping info between atom ID and the lattice␣
  ↪indices.

  GAMMA flags to treate the whole simulation box as a unit cell, so that the mapping
  info can be generated internally. In this case, dynamical matrix at only the␣
  ↪gamma-point
  will/can be evaluated.
  ```

- prefix = prefix for output files
- one or none keyword/value pairs may be appended
- keyword = *sysdim* or *nasr*

  ```
  sysdim value = d
    d = dimension of the system, usually the same as the MD model dimension
  nasr value = n
    n = number of iterations to enforce the acoustic sum rule
  ```

### 2.174.2 Examples

```
fix 1 all phonon 20 5000 200000 map.in LJ1D sysdim 1
fix 1 all phonon 20 5000 200000 map.in EAM3D
fix 1 all phonon 10 5000 500000 GAMMA EAM0D nasr 100
```

### 2.174.3 Description

Calculate the dynamical matrix from molecular dynamics simulations based on fluctuation-dissipation theory for a group of atoms.

Consider a crystal with $N$ unit cells in three dimensions labeled $l = (l_1, l_2, l_3)$ where $l_i$ are integers. Each unit cell is defined by three linearly independent vectors $\mathbf{a}_1$, $\mathbf{a}_2$, $\mathbf{a}_3$ forming a parallelepiped, containing $K$ basis atoms labeled $k$.

Based on fluctuation-dissipation theory, the force constant coefficients of the system in reciprocal space are given by (*Campana* , *Kong*)

$$\blacksquare_{k\alpha,k'\beta}(\mathbf{q}) = k_B T \mathbf{G}^{-1}_{k\alpha,k'\beta}(\mathbf{q})$$

where **G** is the Green's functions coefficients given by

$$\mathbf{G}_{k\alpha,k'\beta}(\mathbf{q}) = \left\langle \mathbf{u}_{k\alpha}(\mathbf{q}) \bullet \mathbf{u}^*_{k'\beta}(\mathbf{q}) \right\rangle$$

where $\langle \ldots \rangle$ denotes the ensemble average, and

$$\mathbf{u}_{k\alpha}(\mathbf{q}) = \sum_l \mathbf{u}_{lk\alpha} \exp\left(i\mathbf{q}\mathbf{r}_l\right)$$

is the $\alpha$ component of the atomic displacement for the $k$ th atom in the unit cell in reciprocal space at **q**. In practice, the Green's functions coefficients can also be measured according to the following formula,

$$\mathbf{G}_{k\alpha,k'\beta}(\mathbf{q}) = \left\langle \mathbf{R}_{k\alpha}(\mathbf{q}) \bullet \mathbf{R}^*_{k'\beta}(\mathbf{q}) \right\rangle - \langle \mathbf{R} \rangle_{k\alpha}(\mathbf{q}) \bullet \langle \mathbf{R} \rangle^*_{k'\beta}(\mathbf{q})$$

where **R** is the instantaneous positions of atoms, and $\langle \mathbf{R} \rangle$ is the averaged atomic positions. It gives essentially the same results as the displacement method and is easier to implement in an MD code.

Once the force constant matrix is known, the dynamical matrix **D** can then be obtained by

$$\mathbf{D}_{k\alpha,k'\beta}(\mathbf{q}) = (m_k m_{k'})^{-\frac{1}{2}} \blacksquare_{k\alpha,k'\beta}(\mathbf{q})$$

whose eigenvalues are exactly the phonon frequencies at **q**.

This fix uses positions of atoms in the specified group and calculates two-point correlations. To achieve this. the positions of the atoms are examined every *Nevery* steps and are Fourier-transformed into reciprocal space, where the averaging process and correlation computation is then done. After every *Noutput* measurements, the matrix **G**(**q**) is calculated and inverted to obtain the elastic stiffness coefficients. The dynamical matrices are then constructed and written to *prefix*.bin.timestep files in binary format and to the file *prefix*.log for each wave-vector **q**.

A detailed description of this method can be found in (*Kong2011*).

The *sysdim* keyword is optional. If specified with a value smaller than the dimensionality of the LAMMPS simulation, its value is used for the dynamical matrix calculation. For example, using LAMMPS to model a 2D or 3D system, the phonon dispersion of a 1D atomic chain can be computed using *sysdim* = 1.

The *nasr* keyword is optional. An iterative procedure is employed to enforce the acoustic sum rule on $\Phi$ at $\Gamma$, and the number provided by keyword *nasr* gives the total number of iterations. For a system whose unit cell has only one atom, *nasr* = 1 is sufficient; for other systems, *nasr* = 10 is typically sufficient.

The *map_file* contains the mapping information between the lattice indices and the atom IDs, which tells the code which atom sits at which lattice point; the lattice indices start from 0. An auxiliary code, latgen, can be employed to generate the compatible map file for various crystals.

In case one simulates a non-periodic system, where the whole simulation box is treated as a unit cell, one can set *map_file* as *GAMMA*, so that the mapping info will be generated internally and a file is not needed. In this case, the dynamical matrix at only the gamma-point will/can be evaluated. Please keep in mind that fix-phonon is designed for cyrstals, it will be inefficient and even degrade the performance of LAMMPS in case the unit cell is too large.

The calculated dynamical matrix elements are written out in *energy/distance^2/mass* units. The coordinates for *q* points in the log file is in the units of the basis vectors of the corresponding reciprocal lattice.

### 2.174.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify temp* option is supported by this fix. You can use it to change the temperature compute from thermo_temp to the one that reflects the true temperature of atoms in the group.

No global scalar or vector or per-atom quantities are stored by this fix for access by various *output commands*.

Instead, this fix outputs its initialization information (including mapping information) and the calculated dynamical matrices to the file *prefix*.log, with the specified *prefix*. The dynamical matrices are also written to files *prefix*.bin.timestep in binary format. These can be read by the post-processing tool in tools/phonon to compute the phonon density of states and/or phonon dispersion curves.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

This fix is not invoked during *energy minimization*.

### 2.174.5 Restrictions

This fix assumes a crystalline system with periodical lattice. The temperature of the system should not exceed the melting temperature to keep the system in its solid state.

This fix is part of the PHONON package. It is only enabled if LAMMPS was built with that package. This fix also requires LAMMPS to be built with 3d-FFT support which is included in the KSPACE package. See the *Build package* page for more info.

### 2.174.6 Related commands

*compute msd*, *dynamical_matrix*

### 2.174.7 Default

The option defaults are sysdim = the same dimension as specified by the *dimension* command, and nasr = 20.

---

**(Campana)** C. Campana and M. H. Muser, *Practical Green's function approach to the simulation of elastic semi-infinite solids*, Phys. Rev. B [74], 075420 (2006)

**(Kong)** L.T. Kong, G. Bartels, C. Campana, C. Denniston, and Martin H. Muser, *Implementation of Green's function molecular dynamics: An extension to LAMMPS*, Computer Physics Communications [180](6):1004-1010 (2009).

L.T. Kong, C. Denniston, and Martin H. Muser, *An improved version of the Green's function molecular dynamics method*, Computer Physics Communications [182](2):540-541 (2011).

**(Kong2011)** L.T. Kong, *Phonon dispersion measured directly from molecular dynamics simulations*, Computer Physics Communications [182](10):2201-2207, (2011).

## 2.175 fix pimd/langevin command

## 2.176 fix pimd/nvt command

## 2.177 fix pimd/langevin/bosonic command

## 2.178 fix pimd/nvt/bosonic command

### 2.178.1 Syntax

```
fix ID group-ID style keyword value ...
```

- ID, group-ID are documented in *fix* command
- style = *pimd/langevin* or *pimd/nvt* or *pimd/langevin/bosonic* or *pimd/nvt/bosonic* = style name of this fix command
- zero or more keyword/value pairs may be appended

- keywords for style *pimd/nvt*

```
keywords = method or fmass or sp or temp or nhc
method value = pimd or nmpimd or cmd
fmass value = scaling factor on mass
sp value = scaling factor on Planck constant
temp value = temperature (temperature units)
nhc value = Nc = number of chains in Nose-Hoover thermostat
```

- keywords for style *pimd/langevin*

```
keywords = method or integrator or ensemble or fmmode or fmass or scale or sp or␣
↪temp or thermostat or tau or iso or aniso or barostat or taup or fixcom or esynch
method value = nmpimd (default) or pimd
integrator value = obabo or baoab
ensemble value = nvt or nve or nph or npt
fmmode value = physical or normal
fmass value = scaling factor on mass
sp value = scaling factor on Planck constant
temp value = temperature (temperature unit)
     temperature = target temperature of the thermostat
thermostat values = style seed
     style value = PILE_L
     seed = random number generator seed
tau value = thermostat damping parameter (time unit)
scale value = scaling factor of the damping times of non-centroid modes of PILE_L␣
↪thermostat
iso or aniso values = pressure (pressure unit)
    pressure = scalar external pressure of the barostat
barostat value = BZP or MTTK
taup value = barostat damping parameter (time unit)
fixcom value = yes or no
esynch value = yes or no (only in pimd/langevin/bosonic)
```

## 2.178.2 Examples

```
fix 1 all pimd/nvt method nmpimd fmass 1.0 sp 2.0 temp 300.0 nhc 4
fix 1 all pimd/langevin ensemble npt integrator obabo temp 113.15 thermostat PILE_L 1234␣
↪tau 1.0 iso 1.0 barostat BZP taup 1.0
fix 1 all pimd/nvt/bosonic method pimd fmass 1.0 sp 1.0 temp 2.0 nhc 4
fix 1 all pimd/langevin/bosonic integrator obabo temp 113.15 thermostat PILE_L 1234 tau␣
↪1.0
```

Example input files are provided in the examples/PACKAGES/pimd and examples/PACKAGES/pimd_bosonic directories.

## 2.178.3 Description

Changed in version 28Mar2023.

Fix pimd was renamed to fix *pimd/nvt* and fix *pimd/langevin* was added.

These fix commands perform quantum molecular dynamics simulations based on the Feynman path-integral to include effects of tunneling and zero-point motion. In this formalism, the isomorphism of a quantum partition function for the original system to a classical partition function for a ring-polymer system is exploited, to efficiently sample configurations from the canonical ensemble *(Feynman)*.

Added in version 2Apr2025: Fix *pimd/langevin/bosonic* and *pimd/nvt/bosonic* were added.

Fix *pimd/nvt* and fix *pimd/langevin* simulate *distinguishable* quantum particles. Simulations of bosons, including exchange effects, are supported with the fix *pimd/langevin/bosonic* and the *pimd/nvt/bosonic* commands.

For distinguishable particles, the isomorphic classical partition function and its components are given by the following equations:

$$
\begin{aligned}
Z &= \int d\mathbf{q} d\mathbf{p} \cdot \exp[-\beta H_{eff}] \\
H_{eff} &= \left( \sum_{i=1}^{P} \frac{p_i^2}{2M_i} \right) + V_{eff} \\
V_{eff} &= \sum_{i=1}^{P} \left[ \frac{mP}{2\beta^2\hbar^2}(q_i - q_{i+1})^2 + \frac{1}{P} V(q_i) \right]
\end{aligned}
$$

$M_i$ is the fictitious mass of the $i$-th mode, and m is the actual mass of the atoms.

The interested user is referred to any of the numerous references on this methodology, but briefly, each quantum particle in a path integral simulation is represented by a ring-polymer of P quasi-beads, labeled from 1 to P. During the simulation, each quasi-bead interacts with beads on the other ring-polymers with the same imaginary time index (the second term in the effective potential above). The quasi-beads also interact with the two neighboring quasi-beads through the spring potential in imaginary-time space (first term in effective potential).

For bosons, the method of Hirshberg et. al. *(Hirshberg1)* is employed, which replaces the spring part of $V_{eff}$ by the spring potential $V^{[1,N]}$ defined through recurrence relation:

$$
\begin{aligned}
e^{-\beta V^{[1,N]}} &= \frac{1}{N} \sum_{k=1}^{N} e^{-\beta (V^{[1,N-k]} + E^{[N-K+1,N]})} \\
e^{-\beta V^{[1,0]}} &= 1
\end{aligned}
$$

Here, $E^{[N-K+1,N]}$ is the spring energy of the ring polymer obtained by connecting the beads of particles $N-k+1, N-k+2, ..., N$ in a cycle. The implementation of the potential and forces evaluation uses the algorithm developed by Feldman and Hirshberg, which scales like $N^2 + PN$ *(Feldman)*. The minimum-image convention is employed on the springs to account for periodic boundary conditions; an elaborate discussion of the validity of the approximation is available in *(Higer)*.

To sample the canonical ensemble, any thermostat can be applied.

Fix *pimd/nvt* applies a Nose-Hoover massive chain thermostat *(Tuckerman)*. With the massive chain algorithm, a chain of NH thermostats is coupled to each degree of freedom for each quasi-bead. The keyword *temp* sets the target temperature for the system and the keyword *nhc* sets the number *Nc* of thermostats in each chain. For example, for a simulation of N particles with P beads in each ring-polymer, the total number of NH thermostats would be 3 x N x P x Nc.

Fix *pimd/langevin* implements a Langevin thermostat in the normal mode representation, and also provides a barostat to sample the NPH/NPT ensembles.

**Note**

Both these *fix* styles implement a complete velocity-verlet integrator combined with a thermostat, so no other time integration fix should be used.

The *method* keyword determines what style of PIMD is performed. A value of *pimd* is standard PIMD. A value of *nmpimd* is for normal-mode PIMD. A value of *cmd* is for centroid molecular dynamics (CMD). The difference between the styles is as follows.

In standard PIMD, the value used for a bead's fictitious mass is arbitrary. A common choice is to use $M_i = m/P$, which results in the mass of the entire ring-polymer being equal to the real quantum particle. But it can be difficult to efficiently integrate the equations of motion for the stiff harmonic interactions in the ring polymers.

A useful way to resolve this issue is to integrate the equations of motion in a normal mode representation, using Normal Mode Path-Integral Molecular Dynamics (NMPIMD) *(Cao1)*. In NMPIMD, the NH chains are attached to each normal mode of the ring-polymer and the fictitious mass of each mode is chosen as Mk = the eigenvalue of the Kth normal mode for k > 0. The k = 0 mode, referred to as the zero-frequency mode or centroid, corresponds to overall translation of the ring-polymer and is assigned the mass of the real particle.

**Note**

Motion of the centroid can be effectively uncoupled from the other normal modes by scaling the fictitious masses to achieve a partial adiabatic separation. This is called a Centroid Molecular Dynamics (CMD) approximation *(Cao2)*. The time-evolution (and resulting dynamics) of the quantum particles can be used to obtain centroid time correlation functions, which can be further used to obtain the true quantum correlation function for the original system. The CMD method also uses normal modes to evolve the system, except only the k > 0 modes are thermostatted, not the centroid degrees of freedom.

Added in version 21Nov2023: Mode *pimd* added to fix pimd/langevin.

Fix pimd/langevin supports the *method* values *nmpimd* and *pimd*. The default value is *nmpimd*. If *method* is *nmpimd*, the normal mode representation is used to integrate the equations of motion. The exact solution of harmonic oscillator is used to propagate the free ring polymer part of the Hamiltonian. If *method* is *pimd*, the Cartesian representation is used to integrate the equations of motion. The harmonic force is added to the total force of the system, and the numerical integrator is used to propagate the Hamiltonian.

Fix *pimd/nvt/bosonic* only supports the *pimd* and *nmpimd* methods. Fix *pimd/langevin/bosonic* only supports the *pimd* method, which is the default in this fix. These restrictions are related to the use of normal modes, which change in bosons.

The keyword *integrator* specifies the Trotter splitting method used by *fix pimd/langevin*. See *(Liu)* for a discussion on the OBABO and BAOAB splitting schemes. Typically either of the two should work fine.

The keyword *fmass* sets a further scaling factor for the fictitious masses of beads, which can be used for the Partial Adiabatic CMD *(Hone)*, or to be set as P, which results in the fictitious masses to be equal to the real particle masses.

The keyword *fmmode* of *fix pimd/langevin* determines the mode of fictitious mass preconditioning. There are two options: *physical* and *normal*. If *fmmode* is *physical*, then the physical mass of the particles are used (and then multiplied by *fmass*). If *fmmode* is *normal*, then the physical mass is first multiplied by the eigenvalue of each normal mode, and then multiplied by *fmass*. More precisely, the fictitious mass of *fix pimd/langevin* is determined by two factors: *fmmode* and *fmass*. If *fmmode* is *physical*, then the fictitious mass is

$$M_i = \text{fmass} \times m$$

If *fmmode* is *normal*, then the fictitious mass is

$$M_i = \text{fmass} \times \lambda_i \times m$$

where $\lambda_i$ is the eigenvalue of the $i$-th normal mode.

In *pimd/langevin/bosonic*, *fmmode* should not be used, and would raise an error if set to a value other than *physical*, due to the lack of support for bosonic normal modes.

**Note**

Fictitious mass is only used in the momentum of the equation of motion ($\mathbf{p}_i = M_i\mathbf{v}_i$), and not used in the spring elastic energy ($\sum_{i=1}^{P} \frac{1}{2}m\omega_P^2(q_i - q_{i+1})^2$, $m$ is always the actual mass of the particles).

Changed in version 10Dec2025: *sp* keyword added to *fix pimd/langevin*

The keyword *sp* is a scaling factor on Planck's constant. Scaling the Planck's constant means modifying the "quantumness" of the PIMD simulation. Using the physical value of Planck's constant corresponds to a fully quantum simulation, and 0 corresponds to the classical limit. For unit styles other than *lj*, the default value of 1.0 is appropriate for most situations. For *lj* units, a fully quantum simulation translates into setting *sp* to the de Boer quantumness parameter $\Lambda^*$ (see *de Boer*):

$$\Lambda^* = h/\sigma\sqrt{m\varepsilon}$$

where $h$ is Planck's constant, $\sigma$ is the length scale, $\varepsilon$ is the energy scale, and $m$ is the mass of the particles. For example, for Neon, $m$ = 20.1797 Dalton, $\varepsilon = 3.0747 \times 10^{-3}$ eV and $\sigma$ = 2.7616Å. Then we have

$$\Lambda^* = \frac{4.135667403 \times 10^{-3}\ \text{eV} \cdot\ \text{ps}}{2.7616\ \text{Å} \times \sqrt{20.1797\ \text{Dalton} \times\ 3.0747 \times 10^{-3}\ \text{eV} \times 1.0364269 \times 10^{-4}\ \text{eV} \cdot \text{Dalton}^{-1} \cdot \text{Å}^{-2} \cdot \text{ps}^2}} = 0.600.$$

Thus for a fully quantum simulation of Neon using *lj* units, *sp* should be set to 0.600. The modification of the quantumness should be done by scaling $\Lambda^*$.

The keyword *ensemble* for fix style *pimd/langevin* determines which ensemble is it going to sample. The value can be *nve* (microcanonical), *nvt* (canonical), *nph* (isoenthalpic), and *npt* (isothermal-isobaric). Fix *pimd/langevin/bosonic* currently does not support *ensemble* other than *nve*, *nvt*.

The keyword *temp* specifies temperature parameter for fix styles *pimd/nvt* and *pimd/langevin*. It should read a positive floating-point number.

**Note**

For pimd simulations, a temperature values should be specified even for nve ensemble. Temperature will make a difference for nve pimd, since the spring elastic frequency between the beads will be affected by the temperature.

The keyword *thermostat* reads *style* and *seed* of thermostat for fix style *pimd/langevin*. *style* can only be *PILE_L* (path integral Langevin equation local thermostat, as described in *Ceriotti*), and *seed* should a positive integer number, which serves as the seed of the pseudo random number generator.

**Note**

The fix style *pimd/langevin* uses the stochastic PILE_L thermostat to control temperature. This thermostat works on the normal modes of the ring polymer. The *tau* parameter controls the centroid mode, and the *scale* parameter controls the non-centroid modes.

The keyword *tau* specifies the thermostat damping time parameter for fix style *pimd/langevin*. It is in time unit. It only works on the centroid mode.

The keyword *scale* specifies a scaling parameter for the damping times of the non-centroid modes for fix style *pimd/langevin*. The default damping time of the non-centroid mode $i$ is $\frac{P}{\beta\hbar}\sqrt{\lambda_i \times \text{fmass}}$ (*fmmode* is *physical*) or $\frac{P}{\beta\hbar}\sqrt{\text{fmass}}$ (*fmmode* is *normal*). The damping times of all non-centroid modes are the default values divided by *scale*. This keyword should be used only with *method*=*nmpimd*.

The barostat parameters for fix style *pimd/langevin* with *npt* or *nph* ensemble is specified using one of *iso* and *aniso* keywords. A *pressure* value should be given with pressure unit. The keyword *iso* means couple all 3 diagonal components together when pressure is computed (hydrostatic pressure), and dilate/contract the dimensions together. The keyword *aniso* means x, y, and z dimensions are controlled independently using the Pxx, Pyy, and Pzz components of the stress tensor as the driving forces, and the specified scalar external pressure. These parameters are not supported in *pimd/langevin/bosonic*.

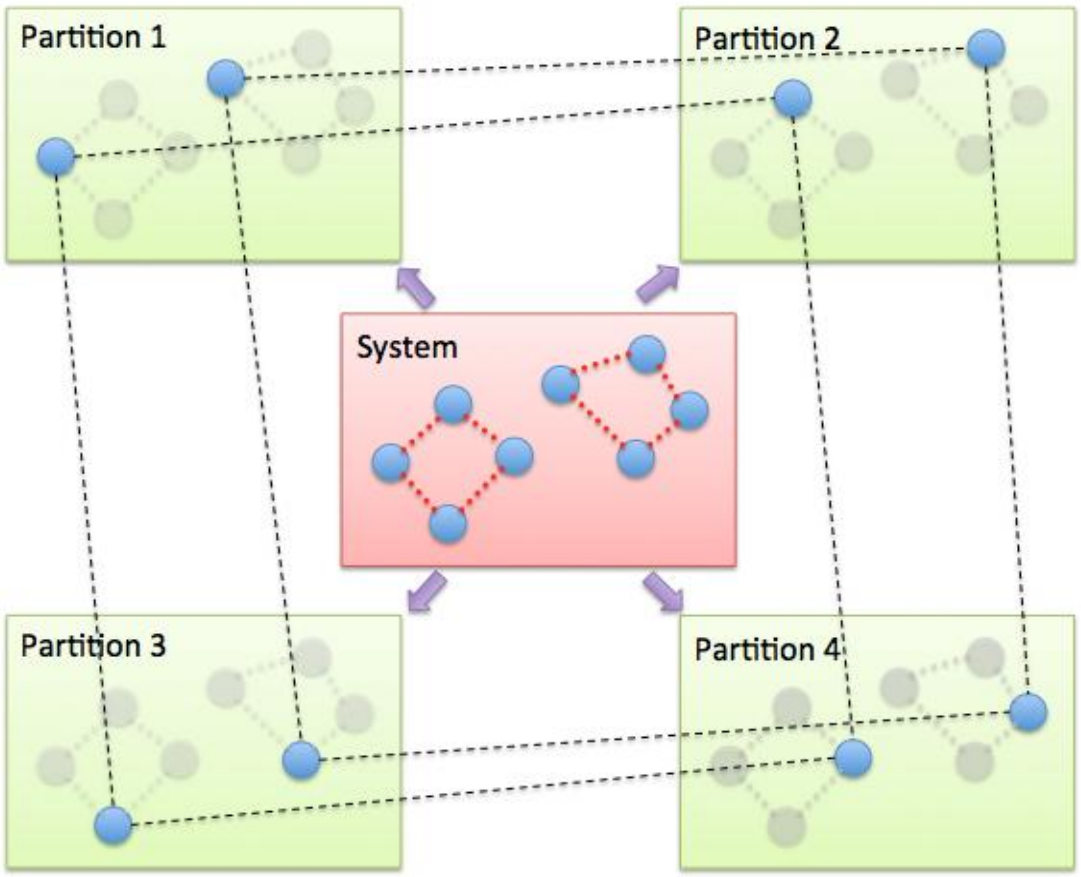
The keyword *barostat* reads *style* of barostat for fix style *pimd/langevin*. *style* can be *BZP* (Bussi-Zykova-Parrinello, as described in *Bussi*) or *MTTK* (Martyna-Tuckerman-Tobias-Klein, as described in *Martyna1* and *Martyna2*).

The keyword *taup* specifies the barostat damping time parameter for fix style *pimd/langevin*. It is in time unit. It is not supported in *pimd/langevin/bosonic*.

The keyword *fixcom* specifies whether the center-of-mass of the extended ring-polymer system is fixed during the pimd simulation. Once *fixcom* is set to be *yes*, the center-of-mass velocity will be distracted from the centroid-mode velocities in each step.

Fix *pimd/langevin/bosonic* also has a keyword not available in fix *pimd/langevin*: *esynch*, with default *yes*. If set to *no*, some time consuming synchronization of spring energies and the primitive kinetic energy estimator between processors is avoided.

The PIMD algorithm in LAMMPS is implemented as a hyper-parallel scheme as described in *Calhoun*. In LAMMPS this is done by using *multi-replica feature* in LAMMPS, where each quasi-particle system is stored and simulated on a separate partition of processors. The following diagram illustrates this approach. The original system with 2 ring polymers is shown in red. Since each ring has 4 quasi-beads (imaginary time slices), there are 4 replicas of the system, each running on one of the 4 partitions of processors. Each replica (shown in green) owns one quasi-bead in each ring.



To run a PIMD simulation with M quasi-beads in each ring polymer using N MPI tasks for each partition's domain-decomposition, you would use P = MxN processors (cores) and run the simulation as follows:

```
mpirun -np P lmp_mpi -partition MxN -in script
```

Note that in the LAMMPS input script for a multi-partition simulation, it is often very useful to define a *uloop-style variable* such as

```
variable ibead uloop M pad
```

where M is the number of quasi-beads (partitions) used in the calculation. The uloop variable can then be used to manage I/O related tasks for each of the partitions, e.g.

```
dump dcd all dcd 10 system_${ibead}.dcd
dump 1 all custom 100 ${ibead}.xyz id type x y z vx vy vz ix iy iz fx fy fz
restart 1000 system_${ibead}.restart1 system_${ibead}.restart2
read_restart system_${ibead}.restart2
```

**Note**

Fix *pimd/langevin* dumps the Cartesian coordinates, but dumps the velocities and forces in the normal mode representation. If the Cartesian velocities and forces are needed, it is easy to perform the transformation when doing post-processing.

It is recommended to dump the image flags (*ix iy iz*) for fix *pimd/langevin*. It will be useful if you want to calculate some estimators during post-processing.

Major differences of *fix pimd/nvt* and *fix pimd/langevin* are:

1. *Fix pimd/nvt* includes Cartesian pimd, normal mode pimd, and centroid md. *Fix pimd/langevin* only intends to support normal mode pimd, as it is commonly enough for thermodynamic sampling.
2. *Fix pimd/nvt* uses Nose-Hoover chain thermostat. *Fix pimd/langevin* uses Langevin thermostat.
3. *Fix pimd/langevin* provides barostat, so the npt ensemble can be sampled. *Fix pimd/nvt* only support nvt ensemble.
4. *Fix pimd/langevin* provides several quantum estimators in output.
5. *Fix pimd/langevin* allows multiple processes for each bead. For *fix pimd/nvt*, there is a large chance that multi-process tasks for each bead may fail.
6. The dump of *fix pimd/nvt* are all Cartesian. *Fix pimd/langevin* dumps normal-mode velocities and forces, and Cartesian coordinates.

Initially, the inter-replica communication and normal mode transformation parts of *fix pimd/langevin* are written based on those of *fix pimd/nvt*, but are significantly revised.

### 2.178.4 Restart, fix_modify, output, run start/stop, minimize info

Fix *pimd/nvt* writes the state of the Nose/Hoover thermostat over all quasi-beads to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

Fix *pimd/langevin* writes the state of the barostat overall beads to *binary restart files*. Since it uses a stochastic thermostat, the state of the thermostat is not written. However, the state of the system can be restored by reading the restart file, except that it will re-initialize the random number generator.

None of the *fix_modify* options are relevant to fix pimd/nvt.

Fix *pimd/nvt* computes a global 3-vector, which can be accessed by various *output commands*. The three quantities in the global vector are:

1. the total spring energy of the quasi-beads,
2. the current temperature of the classical system of ring polymers,
3. the current value of the scalar virial estimator for the kinetic energy of the quantum system *(Herman)*.

The vector values calculated by fix *pimd/nvt* are "extensive", except for the temperature, which is "intensive".

Fix *pimd/nvt/bosonic* computes a global 4-vector. The first three are the same as in *pimd/nvt* (the justification for the correctness of the virial estimator for bosons appears in the supporting information of *(Hirshberg2)*). The fourth is the current value of the scalar primitive estimator for the kinetic energy of the quantum system *(Hirshberg1)*.

Fix *pimd/langevin* computes a global vector of quantities, which can be accessed by various *output commands*. Note that it outputs multiple log files, and different log files contain information about different beads or modes (see detailed explanations below). If *ensemble* is *nve* or *nvt*, the vector has 10 values:

1. kinetic energy of the bead (if *method**=**pimd*) or normal mode (if *method**=**nmpimd*)
2. spring elastic energy of the bead (if *method**=**pimd*) or normal mode (if *method**=**nmpimd*)
3. potential energy of the bead
4. total energy of all beads (conserved if *ensemble* is *nve*)
5. primitive kinetic energy estimator
6. virial energy estimator
7. centroid-virial energy estimator
8. primitive pressure estimator
9. thermodynamic pressure estimator
10. centroid-virial pressure estimator

The first 3 are different for different log files, and the others are the same for different log files.

If *ensemble* is *nph* or *npt*, the vector stores internal variables of the barostat. If *iso* is used, the vector has 15 values:

1. kinetic energy of the normal mode
2. spring elastic energy of the normal mode
3. potential energy of the bead
4. total energy of all beads (conserved if *ensemble* is *nve*)
5. primitive kinetic energy estimator
6. virial energy estimator
7. centroid-virial energy estimator
8. primitive pressure estimator
9. thermodynamic pressure estimator
10. centroid-virial pressure estimator
11. barostat velocity
12. barostat kinetic energy
13. barostat potential energy

14. barostat cell Jacobian
15. enthalpy of the extended system (sum of 4, 12, 13, and 14; conserved if *ensemble* is *nph*)

If *aniso* or *x* or *y* or *z* is used for the barostat, the vector has 17 values:

1. kinetic energy of the normal mode
2. spring elastic energy of the normal mode
3. potential energy of the bead
4. total energy of all beads (conserved if *ensemble* is *nve*)
5. primitive kinetic energy estimator
6. virial energy estimator
7. centroid-virial energy estimator
8. primitive pressure estimator
9. thermodynamic pressure estimator
10. centroid-virial pressure estimator
11. x component of barostat velocity
12. y component of barostat velocity
13. z component of barostat velocity
14. barostat kinetic energy
15. barostat potential energy
16. barostat cell Jacobian
17. enthalpy of the extended system (sum of 4, 14, 15, and 16; conserved if *ensemble* is *nph*)

Fix *pimd/langevin/bosonic* computes a global 6-vector. The quantities in the global vector are:

1. kinetic energy of the beads,
2. spring elastic energy of the beads,
3. potential energy of the bead,
4. total energy of all beads (conserved if *ensemble* is *nve*) if *esynch* is *yes*
5. primitive kinetic energy estimator *(Hirshberg1)*
6. virial energy estimator *(Herman)* (see the justification in the supporting information of *(Hirshberg2)*).

The first three are different for different log files, and the others are the same for different log files, except for the primitive kinetic energy estimator when setting *esynch* to *no*. Then, the primitive kinetic energy estimator is obtained by summing over all log files. Also note that when *esynch* is set to *no*, the fourth output gives the total energy of all beads excluding the spring elastic energy; the total classical energy can then be obtained by adding the sum of second output over all log files. All vector values calculated by fix *pimd/langevin/bosonic* are "extensive".

For both *pimd/nvt/bosonic* and *pimd/langevin/bosonic*, the contribution of the exterior spring to the primitive estimator is printed to the first log file. The contribution of the $P-1$ interior springs is printed to the other $P-1$ log files. The contribution of the constant $\frac{PdN}{2\beta}$ (with $d$ being the dimensionality) is equally divided over log files.

No parameter of fix *pimd/nvt* or *pimd/langevin* can be used with the *start/stop* keywords of the *run* command. Fix *pimd/nvt* or *pimd/langevin* is not invoked during *energy minimization*.

### 2.178.5 Restrictions

These fixes are part of the REPLICA package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Fix *pimd/nvt* cannot be used with *lj units*. Fix *pimd/langevin* can be used with *lj units*. See the documentation above for how to use it.

Only some combinations of fix styles and their options support partitions with multiple processors. LAMMPS will stop with an error if multi-processor partitions are not supported.

A PIMD simulation can be initialized with a single data file read via the *read_data* command. However, this means all quasi-beads in a ring polymer will have identical positions and velocities, resulting in identical trajectories for all quasi-beads. To avoid this, users can simply initialize velocities with different random number seeds assigned to each partition, as defined by the uloop variable, e.g.

```
velocity all create 300.0 1234${ibead} rot yes dist gaussian
```

### 2.178.6 Related commands

*fix ipi*

### 2.178.7 Default

The keyword defaults for fix *pimd/nvt* are method = pimd, fmass = 1.0, sp = 1.0, temp = 300.0, and nhc = 2.

The keyword defaults for fix *pimd/langevin* are integrator = obabo, method = nmpimd, ensemble = nvt, fmmode = physical, fmass = 1.0, scale = 1, temp = 298.15, thermostat = PILE_L, tau = 1.0, iso = 1.0, taup = 1.0, barostat = BZP, fixcom = yes, and sp = 1.0 for all its arguments.

---

**(Feynman)** R. Feynman and A. Hibbs, Chapter 7, Quantum Mechanics and Path Integrals, McGraw-Hill, New York (1965).

**(Tuckerman)** M. Tuckerman and B. Berne, J Chem Phys, 99, 2796 (1993).

**(Cao1)** J. Cao and B. Berne, J Chem Phys, 99, 2902 (1993).

**(Cao2)** J. Cao and G. Voth, J Chem Phys, 100, 5093 (1994).

**(de Boer)** J. de Boer, "Quantum Effects and Exchange Effects on the Thermodynamic Properties of Liquid Helium," Progress in Low Temperature Physics, Volume 2, Pages 1-58 (1957).

**(Hone)** T. Hone, P. Rossky, G. Voth, J Chem Phys, 124, 154103 (2006).

**(Calhoun)** A. Calhoun, M. Pavese, G. Voth, Chem Phys Letters, 262, 415 (1996).

**(Herman)** M. F. Herman, E. J. Bruskin, B. J. Berne, J Chem Phys, 76, 5150 (1982).

**(Bussi)** G. Bussi, T. Zykova-Timan, M. Parrinello, J Chem Phys, 130, 074101 (2009).

**(Ceriotti)** M. Ceriotti, M. Parrinello, T. Markland, D. Manolopoulos, J. Chem. Phys. 133, 124104 (2010).

**(Martyna1)** G. Martyna, D. Tobias, M. Klein, J. Chem. Phys. 101, 4177 (1994).

**(Martyna2)** G. Martyna, A. Hughes, M. Tuckerman, J. Chem. Phys. 110, 3275 (1999).

**(Liu)** J. Liu, D. Li, X. Liu, J. Chem. Phys. 145, 024103 (2016).

**(Hirshberg1)** B. Hirshberg, V. Rizzi, and M. Parrinello, "Path integral molecular dynamics for bosons," Proc. Natl. Acad. Sci. U. S. A. 116, 21445 (2019)

**(Hirshberg2)** B. Hirshberg, M. Invernizzi, and M. Parrinello, "Path integral molecular dynamics for fermions: Alleviating the sign problem with the Bogoliubov inequality," J Chem Phys, 152, 171102 (2020)

**(Feldman)** Y. M. Y. Feldman and B. Hirshberg, "Quadratic scaling bosonic path integral molecular dynamics," J. Chem. Phys. 159, 154107 (2023)

**(Higer)** J. Higer, Y. M. Y. Feldman, and B. Hirshberg, "Periodic Boundary Conditions for Bosonic Path Integral Molecular Dynamics," J. Chem. Phys. 163, 024101 (2025)

# 2.179 fix planeforce command

Accelerator Variant: *planeforce/kk*

## 2.179.1 Syntax

```
fix ID group-ID planeforce x y z
```

- ID, group-ID are documented in *fix* command
- planeforce = style name of this fix command
- x y z = 3-vector that is normal to the plane

## 2.179.2 Examples

```
fix hold boundary planeforce 1.0 0.0 0.0
```

## 2.179.3 Description

Adjust the forces on each atom in the group so that only the components of force in the plane specified by the normal vector (x,y,z) remain. This is done by subtracting out the component of force perpendicular to the plane.

If the initial velocity of the atom is 0.0 (or in the plane), then it should continue to move in the plane thereafter.

## 2.179.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The forces due to this fix are imposed during an energy minimization, invoked by the *minimize* command.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.179.5 Restrictions

none

### 2.179.6 Related commands

*fix lineforce*

### 2.179.7 Default

none

## 2.180 fix plumed command

### 2.180.1 Syntax

```
fix ID group-ID plumed keyword value ...
```

- ID, group-ID are documented in *fix* command
- plumed = style name of this fix command
- keyword = *plumedfile* or *outfile*

  *plumedfile* arg = name of PLUMED input file to use (default: NULL)
  *outfile* arg = name of file on which to write the PLUMED log (default: NULL)

### 2.180.2 Examples

```
fix pl all plumed plumedfile plumed.dat outfile p.log
```

### 2.180.3 Description

This fix instructs LAMMPS to call the PLUMED library, which allows one to perform various forms of trajectory analysis on the fly and to also use methods such as umbrella sampling and metadynamics to enhance the sampling of phase space.

The documentation included here only describes the fix plumed command itself. This command is LAMMPS specific, whereas most of the functionality implemented in PLUMED will work with a range of MD codes, and when PLUMED is used as a stand alone code for analysis. The full documentation for PLUMED is available online and included in the PLUMED source code. The PLUMED library development is hosted at https://github.com/plumed/plumed2 A detailed discussion of the code can be found in *(Tribello)*.

There is an example input for using this package with LAMMPS in the examples/PACKAGES/plumed directory.

---

The command to make LAMMPS call PLUMED during a run requires two keyword value pairs pointing to the PLUMED input file and an output file for the PLUMED log. The user must specify these arguments every time PLUMED is to be used. Furthermore, the fix plumed command should appear in the LAMMPS input file **after** relevant input parameters (e.g. the timestep) have been set.

The *group-ID* entry is ignored. LAMMPS will always pass all the atoms to PLUMED and there can only be one instance of the plumed fix at a time. The way the plumed fix is implemented ensures that the minimum amount of information required is communicated. Furthermore, PLUMED supports multiple, completely independent collective variables, multiple independent biases and multiple independent forms of analysis. There is thus really no restriction in functionality by only allowing only one plumed fix in the LAMMPS input.

The *plumedfile* keyword allows the user to specify the name of the PLUMED input file. Instructions as to what should be included in a plumed input file can be found in the documentation for PLUMED

The *outfile* keyword allows the user to specify the name of a file in which to output the PLUMED log. This log file normally just repeats the information that is contained in the input file to confirm it was correctly read and parsed. The names of the files in which the results are stored from the various analysis options performed by PLUMED will be specified by the user in the PLUMED input file.

### 2.180.4 Restart, fix_modify, output, run start/stop, minimize info

When performing a restart of a calculation that involves PLUMED you must include a RESTART command in the PLUMED input file as detailed in the PLUMED documentation. When the restart command is found in the PLUMED input PLUMED will append to the files that were generated in the run that was performed previously. No part of the PLUMED restart data is included in the LAMMPS restart files. Furthermore, any history dependent bias potentials that were accumulated in previous calculations will be read in when the RESTART command is included in the PLUMED input.

The *fix_modify energy* option is supported by this fix to add the energy change from the biasing force added by PLUMED to the global potential energy of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify energy yes*.

The *fix_modify virial* option is supported by this fix to add the contribution from the biasing force to the global pressure of the system via the *compute pressure* command. This can be accessed by *thermodynamic output*. The default setting for this fix is *fix_modify virial yes*.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the PLUMED energy mentioned above. The scalar value calculated by this fix is "extensive".

Note that other quantities of interest can be output by commands that are native to PLUMED.

### 2.180.5 Restrictions

This fix is part of the PLUMED package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

There can only be one fix plumed command active at a time.

### 2.180.6 Related commands

*fix smd* *fix colvars*

### 2.180.7 Default

The default options are plumedfile = NULL and outfile = NULL

---

**(Tribello)** G.A. Tribello, M. Bonomi, D. Branduardi, C. Camilloni and G. Bussi, Comp. Phys. Comm 185, 604 (2014)

## 2.181 fix poems command

Deprecated since version 10Sep2025.

The POEMS package and the *fix poems* command have been removed from LAMMPS.

## 2.182 fix polarize/bem/gmres command

## 2.183 fix polarize/bem/icc command

## 2.184 fix polarize/functional command

### 2.184.1 Syntax

```
fix ID group-ID style nevery tolerance
```

- ID, group-ID are documented in *fix* command
- style = *polarize/bem/gmres* or *polarize/bem/icc* or *polarize/functional*
- nevery = this fixed is invoked every this many timesteps
- tolerance = the relative tolerance for the iterative solver to stop

### 2.184.2 Examples

```
fix 2 interface polarize/bem/gmres 5 0.0001
fix 1 interface polarize/bem/icc 1 0.0001
fix 3 interface polarize/functional 1 0.0001
```

Used in input scripts:

```
examples/PACKAGES/dielectric/in.confined
examples/PACKAGES/dielectric/in.nopbc
```

### 2.184.3 Description

These fixes compute induced charges at the interface between two impermeable media with different dielectric constants. The interfaces need to be discretized into vertices, each representing a boundary element. The vertices are treated as if they were regular atoms or particles. *atom_style dielectric* should be used since it defines the additional properties of each interface particle such as interface normal vectors, element areas, and local dielectric mismatch. These fixes also require the use of *pair_style* and *kspace_style* with the *dielectric* suffix. At every time step, given a configuration of the physical charges in the system (such as atoms and charged particles) these fixes compute and update the charge of the interface particles. The interfaces are allowed to move during the simulation if the appropriate time integrators are also set (for example, with *fix_rigid*).

Consider an interface between two media: one with dielectric constant of 78 (water), the other of 4 (silica). The interface is discretized into 2000 boundary elements, each represented by an interface particle. Suppose that each interface particle has a normal unit vector pointing from the silica medium to water. The dielectric difference along the normal vector is then 78 - 4 = 74, the mean dielectric value is (78 + 4) / 2 = 41. Each boundary element also has its area and the local mean curvature, which is used by these fixes for computing a correction term in the local electric field. To model charged interfaces, an interface particle will have a non-zero charge value, coming from its area and surface charge density, and its local dielectric constant set to the mean dielectric value.

For non-interface particles such as atoms and charged particles, the interface normal vectors, element area, and dielectric mismatch are irrelevant and unused. Their local dielectric value is used internally to rescale their given charge when computing the Coulombic interactions. For instance, to simulate a cation carrying a charge of +2 (in simulation charge units) in an implicit solvent with a dielectric constant of 40, the cation's charge should be set to +2 and its local dielectric constant property (defined in the *atom_style dielectric*) should be set to 40; there is no need to manually rescale charge. This will produce the proper force for any *pair_style* with the dielectric suffix. It is assumed that the particles

cannot pass through the interface during the simulation because the value of the local dielectric constant property does not change.

There are some example scripts for using these fixes with LAMMPS in the `examples/PACKAGES/dielectric` directory. The README file therein contains specific details on the system setup. Note that the example data files show the additional fields (columns) needed for *atom_style dielectric* beyond the conventional fields *id*, *mol*, *type*, *q*, *x*, *y*, and *z*.

---

For fix *polarize/bem/gmres* and fix *polarize/bem/icc* the induced charges of the atoms in the specified group, which are the vertices on the interface, are computed using the equation:

$$\sigma_b(\mathbf{s}) = \frac{1-\bar{\varepsilon}}{\bar{\varepsilon}}\sigma_f(\mathbf{s}) - \varepsilon_0\frac{\Delta\varepsilon}{\bar{\varepsilon}}\mathbf{E}(\mathbf{s})\cdot\mathbf{n}(\mathbf{s})$$

- $\sigma_b$ is the induced charge density at the interface vertex $\mathbf{s}$.
- $\bar{\varepsilon}$ is the mean dielectric constant at the interface vertex: $\bar{\varepsilon} = (\varepsilon_1 + \varepsilon_2)/2$.
- $\Delta\varepsilon$ is the dielectric constant difference at the interface vertex: $\Delta\varepsilon = \varepsilon_1 - \varepsilon_2$
- $\sigma_f$ is the free charge density at the interface vertex
- $\mathbf{E}(\mathbf{s})$ is the electrical field at the vertex
- $\mathbf{n}(\mathbf{s})$ is the unit normal vector at the vertex pointing from medium with $\varepsilon_2$ to that with $\varepsilon_1$

Fix *polarize/bem/gmres* employs the Generalized Minimum Residual (GMRES) as described in *(Barros)* to solve $\sigma_b$.

Fix *polarize/bem/icc* employs the successive over-relaxation algorithm as described in *(Tyagi)* to solve $\sigma_b$.

The iterative solvers would terminate either when the maximum relative change in the induced charges in consecutive iterations is below the set tolerance, or when the number of iterations reaches *iter_max* (see below).

Fix *polarize/functional* employs the energy functional variation approach as described in *(Jadhao)* to solve $\sigma_b$.

The induced charges computed by these fixes are stored in the *q_scaled* field, and can be accessed as in the following example:

```
compute qs all property/atom q_scaled
dump 1 all custom 1000 all.txt id type q x y z c_qs
```

Note that the *q* field is the regular atom charges, which do not change during the simulation. For interface particles, *q_scaled* is the sum of the real charge, divided by the local dielectric constant *epsilon*, and their induced charges. For non-interface particles, *q_scaled* is the real charge, divided by the local dielectric constant *epsilon*.

More details on the implementation of these fixes and their recommended use are described in *(NguyenTD)*.

### 2.184.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify* command provides the ability to modify certain settings:

```
itr_max arg
   arg = maximum number of iterations for convergence
dielectrics ediff emean epsilon area charge
   ediff = dielectric difference or NULL
   emean = dielectric mean or NULL
   epsilon = local dielectric value or NULL
   area = element area or NULL
   charge = real interface charge or NULL
kspace arg = yes or no
```

```
rand max seed
   max = range of random induced charges to be generated
   seed = random number seed to use when generating random charge
mr arg
   arg = maximum number of q-vectors to use when solving (GMRES only)
omega arg
   arg = relaxation parameter to use when iterating (ICC only)
```

The *itr_max* keyword sets the max number of iterations to be used for solving each step.

The *dielectrics* keyword allows properties of the atoms in group *group-ID* to be modified. Values passed to any of the arguments (*ediff*, *emean*, *epsilon*, *area*, *charge*) will override existing values for all atoms in the group *group-ID*. Passing NULL to any of these arguments will preserve the existing value. Note that setting the properties of the interface this way will change the properties of all atoms associated with the fix (all atoms in *group-ID*), so multiple fix and fix_modify commands would be needed to change the properties of two different interfaces to different values (one fix and fix_modify for each interface group).

The *kspace* keyword turns on long range interactions.

If the arguments of the *rand* keyword are set, then the atoms subject to this fix will be assigned a random initial charge in a uniform distribution from -*max*/2 to *max*/2, using random number seed *seed*.

The *mr* keyword only applies to *style* = *polarize/bem/gmres*. It is the maximum number of q-vectors to use when solving for the surface charge.

The *omega* keyword only applies when using *style* = *polarize/bem/icc*. It is a relaxation parameter defined in *(Tyagi)* that should generally be set between 0 and 2.

Note that the local dielectric constant (epsilon) can also be set independently using the *set* command.

---

*polarize/bem/gmres* or *polarize/bem/icc* compute a global 2-element vector which can be accessed by various *output commands*. The first element is the number of iterations when the solver terminates (of which the upper bound is set by *iter_max*). The second element is the RMS error.

### 2.184.5 Restrictions

These fixes are part of the DIELECTRIC package. They are only enabled if LAMMPS was built with that package, which requires that also the KSPACE package is installed. See the *Build package* page for more info.

Note that the *polarize/bem/gmres* and *polarize/bem/icc* fixes only support *units* *lj*, *real*, *metal*, *si* and *nano* at the moment.

Note that *polarize/functional* does not yet support charged interfaces.

### 2.184.6 Related commands

*pair_coeff*, *fix polarize*, *read_data*, *pair_style lj/cut/coul/long/dielectric*, *kspace_style pppm/dielectric*, *compute efield/atom*

### 2.184.7 Default

*iter_max* = 50

*kspace* = yes

*omega* = 0.7 (ICC only)

*mr* = # atoms in group *group-ID* minus 1 (GMRES only)

No random charge initialization happens by default.

**(Barros)** Barros, Sinkovits, Luijten, J. Chem. Phys, 140, 064903 (2014)

**(Tyagi)** Tyagi, Suzen, Sega, Barbosa, Kantorovich, Holm, J Chem Phys, 132, 154112 (2010)

**(Jadhao)** Jadhao, Solis, Olvera de la Cruz, J Chem Phys, 138, 054119 (2013)

**(NguyenTD)** Nguyen, Li, Bagchi, Solis, Olvera de la Cruz, Comput Phys Commun 241, 80-19 (2019)

## 2.185 fix pour command

### 2.185.1 Syntax

```
fix ID group-ID pour N type seed keyword values ...
```

- ID, group-ID are documented in *fix* command
- pour = style name of this fix command
- N = # of particles to insert
- type = atom type to assign to inserted particles (offset for molecule insertion)
- seed = random # seed (positive integer)
- one or more keyword/value pairs may be appended to args
- keyword = *region* or *diam* or *id* or *vol* or *rate* or *dens* or *vel* or *mol* or *molfrac* or *rigid* or *shake* or *ignore*

```
region value = region-ID
  region-ID = ID of region to use as insertion volume
diam values = dstyle args
  dstyle = one or range or poly
    one args = D
      D = single diameter for inserted particles (distance units)
    range args = Dlo Dhi
      Dlo,Dhi = range of diameters for inserted particles (distance units)
    poly args = Npoly D1 P1 D2 P2 ...
      Npoly = # of (D,P) pairs
      D1,D2,... = diameter for subset of inserted particles (distance units)
      P1,P2,... = percentage of inserted particles with this diameter (0-1)
id values = idflag
  idflag = max or next = how to choose IDs for inserted particles and molecules
vol values = fraction Nattempt
  fraction = desired volume fraction for filling insertion volume
  Nattempt = max # of insertion attempts per particle
rate value = V
  V = z velocity (3d) or y velocity (2d) at which
      insertion volume moves (velocity units)
dens values = Rholo Rhohi
  Rholo,Rhohi = range of densities for inserted particles (mass/volume units)
vel values (3d) = vxlo vxhi vylo vyhi vz
vel values (2d) = vxlo vxhi vy
  vxlo,vxhi = range of x velocities for inserted particles (velocity units)
  vylo,vyhi = range of y velocities for inserted particles (velocity units)
  vz = z velocity (3d) assigned to inserted particles (velocity units)
  vy = y velocity (2d) assigned to inserted particles (velocity units)
```

```
mol value = template-ID
  template-ID = ID of molecule template specified in a separate molecule command
molfrac values = f1 f2 ... fN
  f1 to fN = relative probability of creating each of N molecules in template-ID
rigid value = fix-ID
  fix-ID = ID of fix rigid/small command
shake value = fix-ID
  fix-ID = ID of fix shake command
ignore value = none
  skip any line or triangle particles when detecting possible
    overlaps with inserted particles
```

### 2.185.2 Examples

```
fix 3 all pour 1000 2 29494 region myblock
fix 2 all pour 10000 1 19985583 region disk vol 0.33 100 rate 1.0 diam range 0.9 1.1
fix 2 all pour 10000 1 19985583 region disk diam poly 2 0.7 0.4 1.5 0.6
fix ins all pour 500 1 4767548 vol 0.8 10 region slab mol object rigid myRigid
```

### 2.185.3 Description

Insert finite-size particles or molecules into the simulation box every few timesteps within a specified region until N particles or molecules have been inserted. This is typically used to model the pouring of granular particles into a container under the influence of gravity. For the remainder of this doc page, a single inserted atom or molecule is referred to as a "particle".

If inserted particles are individual atoms, they are assigned the specified atom type. If they are molecules, the type of each atom in the inserted molecule is specified in the file read by the *molecule* command, and those values are added to the specified atom type. E.g. if the file specifies atom types 1,2,3, and those are the atom types you want for inserted molecules, then specify *type* = 0. If you specify *type* = 2, the in the inserted molecule will have atom types 3,4,5.

All atoms in the inserted particle are assigned to two groups: the default group "all" and the group specified in the fix pour command (which can also be "all").

This command must use the *region* keyword to define an insertion volume. The specified region must have been previously defined with a *region* command. It must be of type *block* or a z-axis *cylinder* and must be defined with side = *in*. The cylinder style of region can only be used with 3d simulations.

Individual atoms are inserted, unless the *mol* keyword is used. It specifies a *template-ID* previously defined using the *molecule* command, which reads a file that defines the molecule. The coordinates, atom types, center-of-mass, moments of inertia, etc, as well as any bond/angle/etc and special neighbor information for the molecule can be specified in the molecule file. See the *molecule* command for details. The only settings required to be in this file are the coordinates and types of atoms in the molecule.

If the molecule template contains more than one molecule, the relative probability of depositing each molecule can be specified by the *molfrac* keyword. N relative probabilities, each from 0.0 to 1.0, are specified, where N is the number of molecules in the template. Each time a molecule is inserted, a random number is used to sample from the list of relative probabilities. The N values must sum to 1.0.

If you wish to insert molecules via the *mol* keyword, that will be treated as rigid bodies, use the *rigid* keyword, specifying as its value the ID of a separate *fix rigid/small* command which also appears in your input script.

**Note**

If you wish the new rigid molecules (and other rigid molecules) to be thermostatted correctly via *fix rigid/small/nvt* or *fix rigid/small/npt*, then you need to use the *fix_modify dynamic/dof yes* command for the rigid fix. This is to

inform that fix that the molecule count will vary dynamically.

If you wish to insert molecules via the *mol* keyword, that will have their bonds or angles constrained via SHAKE, use the *shake* keyword, specifying as its value the ID of a separate *fix shake* command which also appears in your input script.

Each timestep particles are inserted, they are placed randomly inside the insertion volume so as to mimic a stream of poured particles. If they are molecules they are also oriented randomly. Each atom in the particle is tested for overlaps with existing particles, including effects due to periodic boundary conditions if applicable. If an overlap is detected, another random insertion attempt is made; see the *vol* keyword discussion below. The larger the volume of the insertion region, the more particles that can be inserted at any one timestep. Particles are inserted again after enough time has elapsed that the previously inserted particles fall out of the insertion volume under the influence of gravity. Insertions continue every so many timesteps until the desired # of particles has been inserted.

**Note**

If you are monitoring the temperature of a system where the particle count is changing due to adding particles, you typically should use the *compute_modify dynamic/dof yes* command for the temperature compute you are using.

**Implementation Notes**

The exact insertion procedure depends on many factors (e.g. the range of diameters inserted or whether molecules are being inserted). However, in the simplest scenario of monodisperse atoms, the procedure works as follows. First, the number of timesteps between two insertion events is calculated as the time for a particle to fall through the insertion region, accounting for gravity and any region motion. Next, the target number of particles inserted per event (assuming no failed insertions due to overlaps) is calculated as the product of the volume fraction and the volume of the insertion region divided by the volume of a particle (or area in 2D). Events are repeated until all N particles have been inserted, where the final event is likely interrupted upon reaching N. Estimates of this process are printed to the log/screen at the start of a run.

---

All other keywords are optional with defaults as shown below.

The *diam* option is only used when inserting atoms and specifies the diameters of inserted particles. There are 3 styles: *one*, *range*, or *poly*. For *one*, all particles will have diameter *D*. For *range*, the diameter of each particle will be chosen randomly and uniformly between the specified *Dlo* and *Dhi* bounds. For *poly*, a series of *Npoly* diameters is specified. For each diameter a percentage value from 0.0 to 1.0 is also specified. The *Npoly* percentages must sum to 1.0. For the example shown above with "diam 2 0.7 0.4 1.5 0.6", all inserted particles will have a diameter of 0.7 or 1.5. 40% of the particles will be small; 60% will be large.

Note that for molecule insertion, the diameters of individual atoms in the molecule can be specified in the file read by the *molecule* command. If not specified, the diameter of each atom in the molecule has a default diameter of 1.0.

The *id* option has two settings which are used to determine the atom or molecule IDs to assign to inserted particles/molecules. In both cases a check is done of the current system to find the maximum current atom and molecule ID of any existing particle. Newly inserted particles and molecules are assigned IDs that increment those max values. For the *max* setting, which is the default, this check is done at every insertion step, which allows for particles to leave the system, and their IDs to potentially be re-used. For the *next* setting this check is done only once when the fix is specified, which can be more efficient if you are sure particles will not be added in some other way.

The *vol* option specifies what volume fraction of the insertion volume will be filled with particles. For particles with a size specified by the *diam range* keyword, they are assumed to all be of maximum diameter *Dhi* for purposes of computing their contribution to the volume fraction.

The higher the volume fraction value, the more particles are inserted each timestep. Since inserted particles cannot overlap, the maximum volume fraction should be no higher than about 0.6. Each timestep particles are inserted, LAMMPS will make up to a total of M tries to insert the new particles without overlaps, where M = # of inserted particles * Nattempt. If LAMMPS is unsuccessful at completing all insertions, it prints a warning.

The *dens* and *vel* options enable inserted particles to have a range of densities or xy velocities. The specific values for a particular inserted particle will be chosen randomly and uniformly between the specified bounds. Internally, the density value for a particle is converted to a mass, based on the radius (volume) of the particle. The *vz* or *vy* value for option *vel* assigns a z-velocity (3d) or y-velocity (2d) to each inserted particle.

The *rate* option moves the insertion volume in the z direction (3d) or y direction (2d). This enables pouring particles from a successively higher height over time.

The *ignore* option is useful when running a simulation that used line segment (2d) or triangle (3d) particles, typically to define boundaries for spherical granular particles to interact with. See the *atom_style line or tri* command for details. Lines and triangles store their size, and if the size is large it may overlap (in a spherical sense) with the insertion region, even if the line/triangle is oriented such that there is no actual overlap. This can prevent particles from being inserted. The *ignore* keyword causes the overlap check to skip any line or triangle particles. Obviously you should only use it if there is in fact no overlap of the line or triangle particles with the insertion region.

---

### 2.185.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. This means you must be careful when restarting a pouring simulation, when the restart file was written in the middle of the pouring operation. Specifically, you should use a new fix pour command in the input script for the restarted simulation that continues the operation. You will need to adjust the arguments of the original fix pour command to do this.

Also note that because the state of the random number generator is not saved in restart files, you cannot do "exact" restarts with this fix, where the simulation continues on the same as if no restart had taken place. However, in a statistical sense, a restarted simulation should produce the same behavior if you adjust the fix pour parameters appropriately.

None of the *fix_modify* options are relevant to this fix. This fix computes a global scalar, which can be accessed by various output commands. The scalar is the cumulative number of insertions. The scalar value calculated by this fix is "intensive". No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.185.5 Restrictions

This fix is part of the GRANULAR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

For 3d simulations, a gravity fix in the -z direction must be defined for use in conjunction with this fix. For 2d simulations, gravity must be defined in the -y direction.

The specified insertion region cannot be a "dynamic" region, as defined by the *region* command.

### 2.185.6 Related commands

*fix deposit*, *fix gravity*, *region*

### 2.185.7 Default

Insertions are performed for individual particles, i.e. no *mol* setting is defined. If the *mol* keyword is used, the default for *molfrac* is an equal probabilities for all molecules in the template. Additional option defaults are diam = one 1.0, dens = 1.0 1.0, vol = 0.25 50, rate = 0.0, vel = 0.0 0.0 0.0 0.0 0.0 (for 3d), vel = 0.0 0.0 0.0 (for 2d), and id = max.

# 2.186 fix precession/spin command

## 2.186.1 Syntax

```
fix ID group precession/spin style args
```

- ID, group are documented in *fix* command
- precession/spin = style name of this fix command
- style = *zeeman* or *anisotropy* or *cubic* or *stt*

```
zeeman args = H x y z
  H = intensity of the magnetic field (in Tesla)
  x y z = vector direction of the field
anisotropy args = K x y z
  K = intensity of the magnetic anisotropy (in eV)
  x y z = vector direction of the anisotropy
cubic args = K1 K2c n1x n1y n1x n2x n2y n2z n3x n3y n3z
  K1 and K2c = intensity of the magnetic anisotropy (in eV)
  n1x to n3z = three direction vectors of the cubic anisotropy
stt args = J x y z
  J = intensity of the spin-transfer torque field
  x y z = vector direction of the field
```

## 2.186.2 Examples

```
fix 1 all precession/spin zeeman 0.1 0.0 0.0 1.0
fix 1 3 precession/spin anisotropy 0.001 0.0 0.0 1.0
fix 1 iron precession/spin cubic 0.001 0.0005 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0
fix 1 all precession/spin zeeman 0.1 0.0 0.0 1.0 anisotropy 0.001 0.0 0.0 1.0
```

## 2.186.3 Description

This fix applies a precession torque to each magnetic spin in the group.

Style *zeeman* is used for the simulation of the interaction between the magnetic spins in the defined group and an external magnetic field:

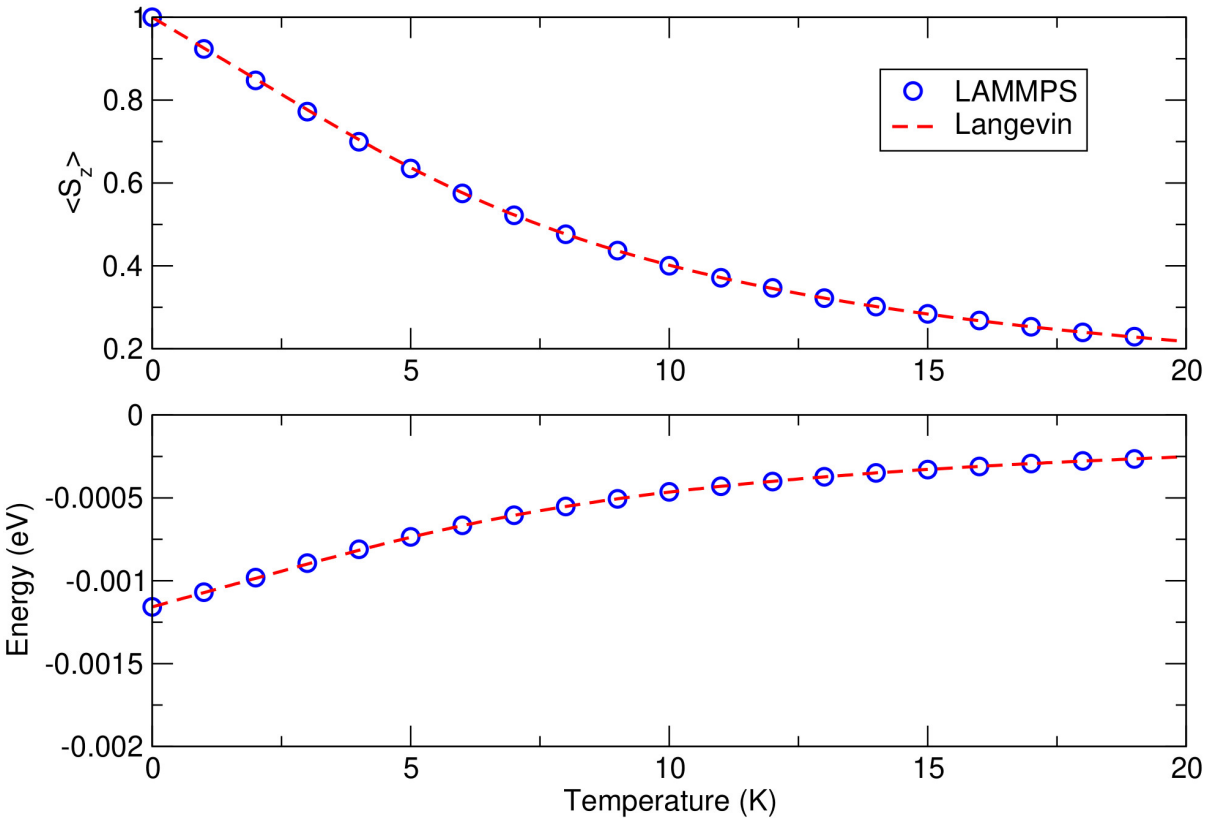
$$H_{Zeeman} = -g \sum_{i=0}^{N} \mu_i \vec{s}_i \cdot \vec{B}_{ext}$$

with:

- $\vec{B}_{ext}$ the external magnetic field (in T)
- $g$ the Lande factor (hard-coded as $g = 2.0$)
- $\vec{s}_i$ the unitary vector describing the orientation of spin $i$
- $\mu_i$ the atomic moment of spin $i$ given as a multiple of the Bohr magneton $\mu_B$ (for example, $\mu_i \approx 2.2$ in bulk iron).

The field value in Tesla is multiplied by the gyromagnetic ratio, $g \cdot \mu_B/\hbar$, converting it into a precession frequency in rad.THz (in metal units and with $\mu_B = 5.788 \cdot 10^{-5}$ eV/T).

As a comparison, the figure below displays the simulation of a single spin (of norm $\mu_i = 1.0$) submitted to an external magnetic field of $|B_{ext}| = 10.0$ Tesla (and oriented along the z axis). The upper plot shows the average magnetization along the external magnetic field axis and the lower plot the Zeeman energy, both as a function of temperature. The reference result is provided by the plot of the Langevin function for the same parameters.




The temperature effects are accounted for by connecting the spin $i$ to a thermal bath using a Langevin thermostat (see *fix langevin/spin* for the definition of this thermostat).

Style *anisotropy* is used to simulate an easy axis or an easy plane for the magnetic spins in the defined group:

$$H_{aniso} = -\sum_{i=1}^{N} K_{an}(\mathbf{r}_i)\ (\vec{s}_i \cdot \vec{n}_i)^2$$

with $n$ defining the direction of the anisotropy, and $K$ (in eV) its intensity. If $K > 0$, an easy axis is defined, and if $K < 0$, an easy plane is defined.

Style *cubic* is used to simulate a cubic anisotropy, with three possible easy axis for the magnetic spins in the defined group:

$$H_{cubic} = -\sum_{i=1}^{N} K_1 \left[ (\vec{s}_i \cdot \vec{n}_1)^2 (\vec{s}_i \cdot \vec{n}_2)^2 + (\vec{s}_i \cdot \vec{n}_2)^2 (\vec{s}_i \cdot \vec{n}_3)^2 + (\vec{s}_i \cdot \vec{n}_1)^2 (\vec{s}_i \cdot \vec{n}_3)^2 \right] + K_2^{(c)} (\vec{s}_i \cdot \vec{n}_1)^2 (\vec{s}_i \cdot \vec{n}_2)^2 (\vec{s}_i \cdot \vec{n}_3)^2$$

with $K_1$ and $K_{2c}$ (in eV) the intensity coefficients and $\vec{n}_1$, $\vec{n}_2$ and $\vec{n}_3$ defining the three anisotropic directions defined by the command (from *n1x* to *n3z*). For $\vec{n}_1 = (100)$, $\vec{n}_2 = (010)$, and $\vec{n}_3 = (001)$, $K_1 < 0$ defines an iron type anisotropy (easy axis along the (001)-type cube edges), and $K_1 > 0$ defines a nickel type anisotropy (easy axis along the (111)-type cube diagonals). $K_2^c > 0$ also defines easy axis along the (111)-type cube diagonals. See chapter 2 of *(Skomski)* for more details on cubic anisotropies.

Style *stt* is used to simulate the interaction between the spins and a spin-transfer torque. See equation (7) of *(Chirac)* for more details about the implemented spin-transfer torque term.

In all cases, the choice of (*xyz*) only imposes the vector directions for the forces. Only the direction of the vector is important; its length is ignored (the entered vectors are normalized).

Those styles can be combined within one single command.

**Note**

The norm of all vectors defined with the precession/spin command have to be non-zero. For example, defining "fix 1 all precession/spin zeeman 0.1 0.0 0.0 0.0" would result in an error message. Since those vector components are used to compute the inverse of the field (or anisotropy) vector norm, setting a zero-vector would result in a division by zero.

---

### 2.186.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify energy* option is supported by this fix to add the energy associated with the spin precession torque to the global potential energy of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify energy no*.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the potential energy (in energy units) discussed in the previous paragraph. The scalar value is an "extensive" quantity.

No information about this fix is written to *binary restart files*.

### 2.186.5 Restrictions

The *precession/spin* style is part of the SPIN package. This style is only enabled if LAMMPS was built with this package, and if the atom_style "spin" was declared. See the *Build package* page for more info.

### 2.186.6 Related commands

*atom_style spin*

### 2.186.7 Default

none

---

**(Skomski)** Skomski, R. (2008). Simple models of magnetism. Oxford University Press.

**(Chirac)** Chirac, Theophile, et al. Ultrafast antiferromagnetic switching in NiO induced by spin transfer torques. Physical Review B 102.13 (2020): 134415.

## 2.187 fix press/berendsen command

### 2.187.1 Syntax

```
fix ID group-ID press/berendsen keyword value ...
```

- ID, group-ID are documented in *fix* command
- press/berendsen = style name of this fix command

```
one or more keyword value pairs may be appended
keyword = iso or aniso or x or y or z or couple or dilate or modulus
  iso or aniso values = Pstart Pstop Pdamp
    Pstart,Pstop = scalar external pressure at start/end of run (pressure units)
    Pdamp = pressure damping parameter (time units)
```

```
  x or y or z values = Pstart Pstop Pdamp
    Pstart,Pstop = external stress tensor component at start/end of run (pressure␣
↪units)
    Pdamp = stress damping parameter (time units)
  couple = none or xyz or xy or yz or xz
  modulus value = bulk modulus of system (pressure units)
  dilate value = all or partial
```

## 2.187.2 Examples

```
fix 1 all press/berendsen iso 0.0 0.0 1000.0
fix 2 all press/berendsen aniso 0.0 0.0 1000.0 dilate partial
```

## 2.187.3 Description

Reset the pressure of the system by using a Berendsen barostat *(Berendsen)*, which rescales the system volume and (optionally) the atoms coordinates within the simulation box every timestep.

Regardless of what atoms are in the fix group, a global pressure is computed for all atoms. Similarly, when the size of the simulation box is changed, all atoms are re-scaled to new positions, unless the keyword *dilate* is specified with a value of *partial*, in which case only the atoms in the fix group are re-scaled. The latter can be useful for leaving the coordinates of atoms in a solid substrate unchanged and controlling the pressure of a surrounding fluid.

**Note**

Unlike the *fix npt* or *fix nph* commands which perform Nose/Hoover barostatting AND time integration, this fix does NOT perform time integration. It only modifies the box size and atom coordinates to effect barostatting. Thus you must use a separate time integration fix, like *fix nve* or *fix nvt* to actually update the positions and velocities of atoms. This fix can be used in conjunction with thermostatting fixes to control the temperature, such as *fix nvt* or *fix langevin* or *fix temp/berendsen*.

See the *Howto barostat* page for a discussion of different ways to perform barostatting.

---

The barostat is specified using one or more of the *iso*, *aniso*, *x*, *y*, *z*, and *couple* keywords. These keywords give you the ability to specify the 3 diagonal components of an external stress tensor, and to couple various of these components together so that the dimensions they represent are varied together during a constant-pressure simulation. Unlike the *fix npt* and *fix nph* commands, this fix cannot be used with triclinic (non-orthogonal) simulation boxes to control all 6 components of the general pressure tensor.

The target pressures for each of the 3 diagonal components of the stress tensor can be specified independently via the *x*, *y*, *z*, keywords, which correspond to the 3 simulation box dimensions. For each component, the external pressure or tensor component at each timestep is a ramped value during the run from *Pstart* to *Pstop*. If a target pressure is specified for a component, then the corresponding box dimension will change during a simulation. For example, if the *y* keyword is used, the y-box length will change. A box dimension will not change if that component is not specified, although you have the option to change that dimension via the *fix deform* command.

For all barostat keywords, the *Pdamp* parameter determines the time scale on which pressure is relaxed. For example, a value of 10.0 means to relax the pressure in a timespan of (roughly) 10 time units (tau or fs or ps - see the *units* command).

**Note**

A Berendsen barostat will not work well for arbitrary values of *Pdamp*. If *Pdamp* is too small, the pressure and volume can fluctuate wildly; if it is too large, the pressure will take a very long time to equilibrate. A good choice for many models is a *Pdamp* of around 1000 timesteps. However, note that *Pdamp* is specified in time units, and that timesteps are NOT the same as time units for most *units* settings.

**Note**

The relaxation time is actually also a function of the bulk modulus of the system (inverse of isothermal compressibility). The bulk modulus has units of pressure and is the amount of pressure that would need to be applied (isotropically) to reduce the volume of the system by a factor of 2 (assuming the bulk modulus was a constant, independent of density, which it's not). The bulk modulus can be set via the keyword *modulus*. The *Pdamp* parameter is effectively multiplied by the bulk modulus, so if the pressure is relaxing faster than expected or desired, increasing the bulk modulus has the same effect as increasing *Pdamp*. The converse is also true. LAMMPS does not attempt to guess a correct value of the bulk modulus; it just uses 10.0 as a default value which gives reasonable relaxation for a Lennard-Jones liquid, but will be way off for other materials and way too small for solids. Thus you should experiment to find appropriate values of *Pdamp* and/or the *modulus* when using this fix.

---

The *couple* keyword allows two or three of the diagonal components of the pressure tensor to be "coupled" together. The value specified with the keyword determines which are coupled. For example, *xz* means the *Pxx* and *Pzz* components of the stress tensor are coupled. *Xyz* means all 3 diagonal components are coupled. Coupling means two things: the instantaneous stress will be computed as an average of the corresponding diagonal components, and the coupled box dimensions will be changed together in lockstep, meaning coupled dimensions will be dilated or contracted by the same percentage every timestep. The *Pstart*, *Pstop*, *Pdamp* parameters for any coupled dimensions must be identical. *Couple xyz* can be used for a 2d simulation; the *z* dimension is simply ignored.

---

The *iso* and *aniso* keywords are simply shortcuts that are equivalent to specifying several other keywords together.

The keyword *iso* means couple all 3 diagonal components together when pressure is computed (hydrostatic pressure), and dilate/contract the dimensions together. Using "iso Pstart Pstop Pdamp" is the same as specifying these 4 keywords:

```
x Pstart Pstop Pdamp
y Pstart Pstop Pdamp
z Pstart Pstop Pdamp
couple xyz
```

The keyword *aniso* means *x*, *y*, and *z* dimensions are controlled independently using the *Pxx*, *Pyy*, and *Pzz* components of the stress tensor as the driving forces, and the specified scalar external pressure. Using "aniso Pstart Pstop Pdamp" is the same as specifying these 4 keywords:

```
x Pstart Pstop Pdamp
y Pstart Pstop Pdamp
z Pstart Pstop Pdamp
couple none
```

---

This fix computes a temperature and pressure each timestep. To do this, the fix creates its own computes of style "temp" and "pressure", as if these commands had been issued:

```
compute fix-ID_temp group-ID temp
compute fix-ID_press group-ID pressure fix-ID_temp
```

See the *compute temp* and *compute pressure* commands for details. Note that the IDs of the new computes are the fix-ID + underscore + "temp" or fix_ID + underscore + "press", and the group for the new computes is the same as the fix group.

Note that these are NOT the computes used by thermodynamic output (see the *thermo_style* command) with ID = *thermo_temp* and *thermo_press*. This means you can change the attributes of this fix's temperature or pressure via the *compute_modify* command or print this temperature or pressure during thermodynamic output via the *thermo_style custom* command using the appropriate compute-ID. It also means that changing attributes of *thermo_temp* or *thermo_press* will have no effect on this fix.

### 2.187.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify temp* and *press* options are supported by this fix. You can use them to assign a *compute* you have defined to this fix which will be used in its temperature and pressure calculations. If you do this, note that the kinetic energy derived from the compute temperature should be consistent with the virial term computed using all atoms for the pressure. LAMMPS will warn you if you choose to compute temperature on a subset of atoms.

No global or per-atom quantities are stored by this fix for access by various *output commands*.

This fix can ramp its target pressure over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.187.5 Restrictions

Any dimension being adjusted by this fix must be periodic.

### 2.187.6 Related commands

*fix nve*, *fix nph*, *fix npt*, *fix temp/berendsen*, *fix_modify*

### 2.187.7 Default

The keyword defaults are dilate = all, modulus = 10.0 in units of pressure for whatever *units* are defined.

---

**(Berendsen)** Berendsen, Postma, van Gunsteren, DiNola, Haak, J Chem Phys, 81, 3684 (1984).

## 2.188 fix press/langevin command

### 2.188.1 Syntax

```
fix ID group-ID press/langevin keyword value ...
```

- ID, group-ID are documented in *fix* command
- press/langevin = style name of this fix command

```
one or more keyword value pairs may be appended
keyword = iso or aniso or tri or x or y or z or xy or xz or yz or couple or dilate␣
↪or modulus or temp or flip
  iso or aniso or tri values = Pstart Pstop Pdamp
    Pstart,Pstop = scalar external pressure at start/end of run (pressure units)
    Pdamp = pressure damping parameter (time units)
  x or y or z or xy or xz or yz values = Pstart Pstop Pdamp
    Pstart,Pstop = external stress tensor component at start/end of run (pressure␣
↪units)
    Pdamp = pressure damping parameter
  flip value = yes or no = allow or disallow box flips when it becomes highly skewed
  couple = none or xyz or xy or yz or xz
  friction value = Friction coefficient for the barostat (time units)
  temp values = Tstart, Tstop, seed
  Tstart, Tstop = target temperature used for the barostat at start/end of run
  seed = seed of the random number generator
  dilate value = all or partial
```

## 2.188.2 Examples

```
fix 1 all press/langevin iso 0.0 0.0 1000.0 temp 300 300 487374
fix 2 all press/langevin aniso 0.0 0.0 1000.0 temp 100 300 238 dilate partial
```

## 2.188.3 Description

Adjust the pressure of the system by using a Langevin stochastic barostat *(Gronbech)*, which rescales the system volume and (optionally) the atoms coordinates within the simulation box every timestep.

The Langevin barostat couple each direction $L$ with a pseudo-particle that obeys the Langevin equation such as:

$$
\begin{aligned}
f_P =& \frac{Nk_BT_{target}}{V} + \frac{1}{Vd}\sum_{i=1}^{N}\vec{r}_i \cdot \vec{f}_i - P_{target} \\
Q\ddot{L} + \alpha\dot{L} =& f_P + \beta(t) \\
L^{n+1} =& L^n + bdt\dot{L}^n + \frac{bdt^2}{2Q}f_P^n + \frac{bdt}{2Q}\beta^{n+1} \\
\dot{L}^{n+1} =& \alpha\dot{L}^n + \frac{dt}{2Q}\left(af_P^n + f_P^{n+1}\right) + \frac{b}{Q}\beta^{n+1} \\
a =& \frac{1-\frac{\alpha dt}{2Q}}{1+\frac{\alpha dt}{2Q}} \\
b =& \frac{1}{1+\frac{\alpha dt}{2Q}} \\
\langle\beta(t)\beta(t')\rangle =& 2\alpha k_BT\,dt
\end{aligned}
$$

Where $dt$ is the timestep $\dot{L}$ and $\ddot{L}$ the first and second derivatives of the coupled direction with regard to time, $\alpha$ is a friction coefficient, $\beta$ is a random gaussian variable and $Q$ the effective mass of the coupled pseudoparticle. The two first terms on the right-hand side of the first equation are the virial expression of the canonical pressure. It is to be noted that the temperature used to compute the pressure is not based on the atom velocities but rather on the canonical target temperature directly. This temperature is specified using the *temp* keyword parameter and should be close to the expected target temperature of the system.

Regardless of what atoms are in the fix group, a global pressure is computed for all atoms. Similarly, when the size of the simulation box is changed, all atoms are re-scaled to new positions, unless the keyword *dilate* is specified with

a value of *partial*, in which case only the atoms in the fix group are re-scaled. The latter can be useful for leaving the coordinates of atoms in a solid substrate unchanged and controlling the pressure of a surrounding fluid.

**Note**

Unlike the *fix npt* or *fix nph* commands which perform Nose-Hoover barostatting AND time integration, this fix does NOT perform time integration of the atoms but only of the barostat coupled coordinate. It then only modifies the box size and atom coordinates to effect barostatting. Thus you must use a separate time integration fix, like *fix nve* or *fix nvt* to actually update the positions and velocities of atoms. This fix can be used in conjunction with thermostatting fixes to control the temperature, such as *fix nvt* or *fix langevin* or *fix temp/berendsen*.

See the *Howto barostat* page for a discussion of different ways to perform barostatting.

---

The barostat is specified using one or more of the *iso*, *aniso*, *tri x*, *y*, *z*, *xy*, *xz*, *yz*, and *couple* keywords. These keywords give you the ability to specify the 3 diagonal components of an external stress tensor, and to couple various of these components together so that the dimensions they represent are varied together during a constant-pressure simulation.

The target pressures for each of the 6 diagonal components of the stress tensor can be specified independently via the *x*, *y*, *z*, keywords, which correspond to the 3 simulation box dimensions, and the *xy*, *xz* and *yz* keywords which corresponds to the 3 simulation box tilt factors. For each component, the external pressure or tensor component at each timestep is a ramped value during the run from *Pstart* to *Pstop*. If a target pressure is specified for a component, then the corresponding box dimension will change during a simulation. For example, if the *y* keyword is used, the y-box length will change. A box dimension will not change if that component is not specified, although you have the option to change that dimension via the *fix deform* command.

The *Pdamp* parameter can be seen in the same way as a Nose-Hoover parameter as it is used to compute the mass of the fictitious particle. Without friction, the barostat can be compared to a single particle Nose-Hoover barostat and should follow a similar decay in time. The mass of the barostat is linked to *Pdamp* by the relation $Q = (N_{at}+1) \cdot k_B T_{target} \cdot P_{damp}^2$. Note that *Pdamp* should be expressed in time units.

**Note**

As for Berendsen barostat, a Langevin barostat will not work well for arbitrary values of *Pdamp*. If *Pdamp* is too small, the pressure and volume can fluctuate wildly; if it is too large, the pressure will take a very long time to equilibrate. A good choice for many models is a *Pdamp* of around 1000 timesteps. However, note that *Pdamp* is specified in time units, and that timesteps are NOT the same as time units for most *units* settings.

---

The *temp* keyword sets the temperature to use in the equation of motion of the barostat. This value is used to compute the value of the force $f_P$ in the equation of motion. It is important to note that this value is not the instantaneous temperature but a target temperature that ramps from *Tstart* to *Tstop*. Also the required argument *seed* sets the seed for the random number generator used in the generation of the random forces.

---

The *couple* keyword allows two or three of the diagonal components of the pressure tensor to be "coupled" together. The value specified with the keyword determines which are coupled. For example, *xz* means the *Pxx* and *Pzz* components of the stress tensor are coupled. *Xyz* means all 3 diagonal components are coupled. Coupling means two things: the instantaneous stress will be computed as an average of the corresponding diagonal components, and the coupled box dimensions will be changed together in lockstep, meaning coupled dimensions will be dilated or contracted by the same percentage every timestep. The *Pstart*, *Pstop*, *Pdamp* parameters for any coupled dimensions must be identical. *Couple xyz* can be used for a 2d simulation; the *z* dimension is simply ignored.

---

The *iso*, *aniso* and *tri* keywords are simply shortcuts that are equivalent to specifying several other keywords together.

The keyword *iso* means couple all 3 diagonal components together when pressure is computed (hydrostatic pressure), and dilate/contract the dimensions together. Using "iso Pstart Pstop Pdamp" is the same as specifying these 4 keywords:

```
x Pstart Pstop Pdamp
y Pstart Pstop Pdamp
z Pstart Pstop Pdamp
couple xyz
```

The keyword *aniso* means $x$, $y$, and $z$ dimensions are controlled independently using the $Pxx$, $Pyy$, and $Pzz$ components of the stress tensor as the driving forces, and the specified scalar external pressure. Using "aniso Pstart Pstop Pdamp" is the same as specifying these 4 keywords:

```
x Pstart Pstop Pdamp
y Pstart Pstop Pdamp
z Pstart Pstop Pdamp
couple none
```

The keyword *tri* is the same as *aniso* but also adds the control on the shear pressure coupled with the tilt factors.

```
x Pstart Pstop Pdamp
y Pstart Pstop Pdamp
z Pstart Pstop Pdamp
xy Pstart Pstop Pdamp
xz Pstart Pstop Pdamp
yz Pstart Pstop Pdamp
couple none
```

---

The *flip* keyword allows the tilt factors for a triclinic box to exceed half the distance of the parallel box length, as discussed below. If the *flip* value is set to *yes*, the bound is enforced by flipping the box when it is exceeded. If the *flip* value is set to *no*, the tilt will continue to change without flipping. Note that if applied stress induces large deformations (e.g. in a liquid), this means the box shape can tilt dramatically and LAMMPS will run less efficiently, due to the large volume of communication needed to acquire ghost atoms around a processor's irregular-shaped subdomain. For extreme values of tilt, LAMMPS may also lose atoms and generate an error.

---

The *friction* keyword sets the friction parameter $\alpha$ in the equations of motion of the barostat. For each barostat direction, the value of $\alpha$ depends on both *Pdamp* and *friction*. The value given as a parameter is the Langevin characteristic time $\tau_L = \frac{Q}{\alpha}$ in time units. The langevin time can be understood as a decorrelation time for the pressure. A long Langevin time value will make the barostat act as an underdamped oscillator while a short value will make it act as an overdamped oscillator. The ideal configuration would be to find the critical parameter of the barostat. Empirically this is observed to occur for $\tau_L \approx P_{damp}$. For this reason, if the *friction* keyword is not used, the default value *Pdamp* is used for each barostat direction.

---

This fix computes pressure each timestep. To do this, the fix creates its own computes of style "pressure", as if this command had been issued:

```
compute fix-ID_press group-ID pressure NULL virial
```

The kinetic contribution to the pressure is taken as the ensemble value $\frac{Nk_bT}{V}$ and computed by the fix itself.

See the *compute pressure* command for details. Note that the IDs of the new compute is the fix-ID + underscore + "press" and the group for the new computes is the same as the fix group.

Note that this is NOT the compute used by thermodynamic output (see the *thermo_style* command) with ID = *thermo_press*. This means you can change the attributes of this fix's pressure via the *compute_modify* command or print this temperature or pressure during thermodynamic output via the *thermo_style custom* command using the appropriate compute-ID. It also means that changing attributes of *thermo_temp* or *thermo_press* will have no effect on this fix.

### 2.188.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify press* option is supported by this fix. You can use it to assign a *compute* you have defined to this fix which will be used in its pressure calculations.

No global or per-atom quantities are stored by this fix for access by various *output commands*.

This fix can ramp its target pressure and temperature over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this. It is recommended that the ramped temperature is the same as the effective temperature of the thermostatted system. That is, if the system's temperature is ramped by other commands, it is recommended to do the same with this pressure control.

This fix is not invoked during *energy minimization*.

### 2.188.5 Restrictions

Any dimension being adjusted by this fix must be periodic.

### 2.188.6 Related commands

*fix press/berendsen*, *fix nve*, *fix nph*, *fix npt*, *fix langevin*, *fix_modify*

### 2.188.7 Default

The keyword defaults are *dilate* = all, *flip* = yes, and *friction* = *Pdamp*.

---

**(Gronbech)** Gronbech-Jensen, Farago, J Chem Phys, 141, 194108 (2014).

## 2.189 fix print command

### 2.189.1 Syntax

```
fix ID group-ID print N string keyword value ...
```

- ID, group-ID are documented in *fix* command
- print = style name of this fix command
- N = print every N steps; N can be a variable (see below)
- string = text string to print with optional variable names
- zero or more keyword/value pairs may be appended
- keyword = *file* or *append* or *screen* or *title*

```
file value = filename
append value = filename
screen value = yes or no
title value = string
  string =  text to print as 1st line of output file
```

### 2.189.2 Examples

```
fix extra all print 100 "Coords of marker atom = $x $y $z"
fix extra all print 100 "Coords of marker atom = $x $y $z" file coord.txt
```

### 2.189.3 Description

Print a text string every N steps during a simulation run. This can be used for diagnostic purposes or as a debugging tool to monitor some quantity during a run. The text string must be a single argument, so it should be enclosed in single or double quotes if it is more than one word. If it contains variables it must be enclosed in double quotes to ensure they are not evaluated when the input script line is read, but will instead be evaluated each time the string is printed.

Added in version 15Jun2023: support for vector style variables

See the *variable* command for a description of *equal* and *vector* style variables which are typically the most useful ones to use with the print command. Equal- and vector-style variables can calculate formulas involving mathematical operations, atom properties, group properties, thermodynamic properties, global values calculated by a *compute* or *fix*, or references to other *variables*. Vector-style variables are printed in a bracketed, comma-separated format, e.g. [1,2,3,4] or [12.5,2,4.6,10.1].

**Note**

As discussed on the *Commands parse* doc page, the text string can use "immediate" variables, specified as $(formula) with parenthesis, where the numeric formula has the same syntax as equal-style variables described on the *variable* doc page. This is a convenient way to evaluate a formula immediately without using the variable command to define a named variable and then use that variable in the text string. The formula can include a trailing colon and format string which determines the precision with which the numeric value is output. This is also explained on the *Commands parse* doc page.

Instead of a numeric value, N can be specified as an *equal-style variable*, which should be specified as v_name, where name is the variable name. In this case, the variable is evaluated at the beginning of a run to determine the **next** timestep at which the string will be written out. On that timestep, the variable will be evaluated again to determine the next timestep, etc. Thus the variable should return timestep values. See the stagger() and logfreq() and stride() math functions for *equal-style variables*, as examples of useful functions to use in this context. For example, the following commands will print output at timesteps 10,20,30,100,200,300,1000,2000,etc:

```
variable        s equal logfreq(10,3,10)
fix extra all print v_s "Coords of marker atom = $x $y $z"
```

The specified group-ID is ignored by this fix.

If the *file* or *append* keyword is used, a filename is specified to which the output generated by this fix will be written. If *file* is used, then the filename is overwritten if it already exists. If *append* is used, then the filename is appended to if it already exists, or created if it does not exist.

If the *screen* keyword is used, output by this fix to the screen and logfile can be turned on or off as desired.

The *title* keyword allow specification of the string that will be printed as the first line of the output file, assuming the *file* keyword was used. By default, the title line is as follows:

```
# Fix print output for fix ID
```

where ID is replaced with the fix-ID.

### 2.189.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.189.5 Restrictions

none

### 2.189.6 Related commands

*variable*, *print*

### 2.189.7 Default

The option defaults are no file output, screen = yes, and title string as described above.

## 2.190 fix propel/self command

### 2.190.1 Syntax

```
fix ID group-ID propel/self mode magnitude keyword values
```

- ID, group-ID are documented in *fix* command
- propel/self = style name of this fix command
- mode = *dipole* or *velocity* or *quat*
- magnitude = magnitude of self-propulsion force
- zero or one keyword/value pairs may be appended
- keyword = *qvector*

```
qvector value = direction of force in ellipsoid frame
  sx, sy, sz = components of qvector
```

### 2.190.2 Examples

```
fix active all propel/self dipole 40.0
fix active all propel/self velocity 10.0
fix active all propel/self quat 15.7 qvector 1.0 0.0 0.0
```

### 2.190.3 Description

Add a force to each atom in the group due to a self-propulsion force. The force is given by

$$F_i = f_P e_i$$

where $i$ is the particle the force is being applied to, $f_P$ is the magnitude of the force, and $e_i$ is the vector direction of the force. The specification of $e_i$ is based on which of the three keywords (*dipole* or *velocity* or *quat*) one selects.

For mode *dipole*, $e_i$ is just equal to the dipole vectors of the atoms in the group. Therefore, if the dipoles are not unit vectors, the $e_i$ will not be unit vectors.

**Note**

If another command changes the magnitude of the dipole, this force will change accordingly (since $|e_i|$ will change, which is physically equivalent to re-scaling $f_P$ while keeping $|e_i|$ constant), and no warning will be provided by LAMMPS. This is almost never what you want, so ensure you are not changing dipole magnitudes with another LAMMPS fix or pair style. Furthermore, self-propulsion forces (almost) always set $e_i$ to be a unit vector for all times, so it's best to set all the dipole magnitudes to 1.0 unless you have a good reason not to (see the *set* command on how to do this).

For mode *velocity*, $e_i$ points in the direction of the current velocity (a unit-vector). This can be interpreted as a velocity-dependent friction, as proposed by e.g. *(Erdmann)*.

For mode *quat*, $e_i$ points in the direction of a unit vector, oriented in the coordinate frame of the ellipsoidal particles, which defaults to point along the x-direction. This default behavior can be changed by via the *quatvec* keyword.

The optional *quatvec* keyword specifies the direction of self-propulsion via a unit vector (sx,sy,sz). The arguments *sx*, *sy*, and *sz*, are defined within the coordinate frame of the atom's ellipsoid. For instance, for an ellipsoid with long axis along its x-direction, if one wanted the self-propulsion force to also be along this axis, set *sx* equal to 1 and *sy*, *sz* both equal to zero. This keyword may only be specified for mode *quat*.

**Note**

In using keyword *quatvec*, the three arguments *sx*, *sy*, and *sz* will be automatically normalized to components of a unit vector internally to avoid users having to explicitly do so themselves. Therefore, in mode *quat*, the vectors $e_i$ will always be of unit length.

Along with adding a force contribution, this fix can also contribute to the virial (pressure) of the system, defined as $f_P \sum_i < e_i.r_i > /(dV)$, where $r_i$ is the *unwrapped* coordinate of particle i in the case of periodic boundary conditions. See *(Winkler)* for a discussion of this active pressure contribution.

For modes *dipole* and *quat*, this fix is by default included in pressure computations.

For mode *velocity*, this fix is by default not included in pressure computations.

**Note**

In contrast to equilibrium systems, pressure of active systems in general depends on the geometry of the container. The active pressure contribution as calculated in this fix is only valid for certain boundary conditions (spherical walls, rectangular walls, or periodic boundary conditions). For other geometries, the pressure must be measured via explicit calculation of the force per unit area on a wall, and so one must not calculate it using this fix. (Use *fix_modify* as described below to turn off the virial contribution of this fix). Again, see *(Winkler)* for discussion of why this is the case.

Furthermore, when dealing with active systems, the temperature is no longer well defined. Therefore, one should ensure that the *virial* flag is used in the *compute pressure* command (turning off temperature contributions).

### 2.190.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify virial* option is supported by this fix to add the contribution due to the added forces on atoms to the system's virial as part of *thermodynamic output*. The default is *virial yes* for keywords *dipole* and *quat*. The default is *virial no* for keyword *velocity*.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

### 2.190.5 Restrictions

This fix is part of the BROWNIAN package. It is only enabled if LAMMPS was built with that package. See the *Build package* doc page for more info.

With keyword *dipole*, this fix only works when the DIPOLE package is also enabled. See the *Build package* page for more info.

### 2.190.6 Related commands

*fix align/self*, *fix efield*, *fix setforce*, *fix addforce*

### 2.190.7 Default

none

---

**(Erdmann)** U. Erdmann , W. Ebeling, L. Schimansky-Geier, and F. Schweitzer, Eur. Phys. J. B 15, 105-113, 2000.

**(Winkler)** Winkler, Wysocki, and Gompper, Soft Matter, 11, 6680 (2015).

## 2.191 fix property/atom command

Accelerator Variants: *property/atom/kk*

### 2.191.1 Syntax

```
fix ID group-ID property/atom name1 name2 ... keyword value ...
```

- ID, group-ID are documented in *fix* command
- property/atom = style name of this fix command
- name1,name2,... = *mol* or *q* or *rmass* or i_name or d_name or i2_name or d2_name

  *mol* = molecule IDs
  *q* = charge
  *rmass* = per-atom mass
  *temperature* = internal temperature of atom
  *heatflow* = internal heat flow of atom
  i_name = new integer vector referenced by name
  d_name = new floating-point vector referenced by name
  i2_name = new integer array referenced by name
     i2_name arg = N = number of columns in the array
  d2_name = new floating-point array referenced by name
     d2_name arg = N = number of columns in the array

- zero of more keyword/value pairs may be appended

- keyword = *ghost*

  *ghost* value = *no* or *yes* for whether ghost atom info in communicated

## 2.191.2 Examples

```
fix 1 all property/atom mol
fix 1 all property/atom i_myflag1 i_myflag2
fix 1 all property/atom d2_sxyz 3 ghost yes
```

## 2.191.3 Description

Create one or more additional per-atom vectors or arrays to store information about atoms and to use during a simulation. The specified *group-ID* is ignored by this fix.

The atom style used for a simulation defines a set of per-atom properties, as explained on the *atom_style* and *read_data* doc pages. The latter command defines these properties for each atom in the system when a data file is read. This fix augments the set of per-atom properties with new custom ones. This can be useful in several scenarios.

If the atom style does not define molecule IDs, per-atom charge, per-atom mass, internal temperature, or internal heat flow, they can be added using the *mol*, *q*, *rmass*, *temperature*, or *heatflow* keywords. This could be useful to define "molecules" to use as rigid bodies with the *fix rigid* command, or to carry around an extra flag with atoms (stored as a molecule ID) that can be used by various commands like *compute chunk/atom* to group atoms without having to use the group command (which is limited to a total of 32 groups including *all*). For finite-size particles, an internal temperature and heat flow can be used to model heat conduction as in the *GRANULAR package*.

Another application is to use the *rmass* flag in order to have per-atom masses instead of per-type masses. This could be used to study isotope effects with partial isotope substitution. *See below* for an example of simulating a mixture of light and heavy water with the TIP4P water potential.

An alternative to using fix *property/atom* for these examples is to use an atom style that does define molecule IDs or charge or per-atom mass (indirectly via diameter and density) or to use a hybrid atom style that combines two or more atom styles to provide the union of all their atom properties. However, this has two practical drawbacks: first, it typically necessitates changing the format of the Atoms section in the data file and second, it may define additional properties that are not needed such as bond lists, which incurs some overhead when there are no bonds.

In the future, we may add additional existing per-atom properties to fix property/atom, similar to *mol*, *q*, *rmass*, *temperature*, or *heatflow* which "turn-on" specific properties defined by some atom styles, so they can be easily used by atom styles that do not define them.

More generally, the *i_name* and *d_name* options allow one or more new custom per-atom vectors to be defined. Likewise the *i2_name* and *d2_name* options allow one or more custom per-atom arrays to be defined. The *i2_name* and *d2_name* options take an argument *N* which specifies the number of columns in the per-atom array, i.e. the number of attributes associated with each atom. *N* >= 1 is required.

Each name must be unique and can use alphanumeric or underscore characters. These vectors and arrays can store whatever values you decide are useful in your simulation. As explained below there are several ways to initialize, access, and output these values, via input script commands, data files, and in new code you add to LAMMPS.

This is effectively a simple way to add per-atom properties to a model without needing to write code for a new *atom style* that defines the properties. Note however that implementing a new atom style allows new atom properties to be more tightly and seamlessly integrated with the rest of the code.

The new atom properties encode values that migrate with atoms to new processors and are written to restart files. If you want the new properties to also be defined for ghost atoms, then use the *ghost* keyword with a value of *yes*. This will invoke extra communication when ghost atoms are created (at every re-neighboring) to ensure the new properties are also defined for the ghost atoms.

**Properties on ghost atoms**

If you use the *mol*, *q* or *rmass* names, you most likely want to set *ghost* yes, since these properties are stored with ghost atoms if you use an *atom_style* that defines them. Many LAMMPS operations that use molecule IDs or charge, such as neighbor lists and pair styles, will expect ghost atoms to have these values. LAMMPS will issue a warning it you define those vectors but do not set *ghost* yes.

**Limitations on ghost atom properties**

The specified properties for ghost atoms are not updated every timestep, but only once every few steps when neighbor lists are re-built. Thus the *ghost* keyword is suitable for static properties, like molecule IDs, but not for dynamic properties that change every step. For the latter, the code you add to LAMMPS to change the properties will also need to communicate their new values to/from ghost atoms, an operation that can be invoked from within a *pair style* or *fix* or *compute* that you write.

---

This fix is one of a small number that can be defined in an input script before the simulation box is created or atoms are defined. This is so it can be used with the *read_data* command as described next.

Per-atom properties that are defined by the *atom style* are initialized when atoms are created, e.g. by the *read_data* or *create_atoms* commands. The per-atom properties defined by this fix are not. So you need to initialize them explicitly. One way to do this is *read_data* command, using its *fix* keyword and passing it the fix-ID of this fix.

Thus these commands:

```
fix prop all property/atom mol d_flag
read_data data.txt fix prop NULL Molecules
```

would allow a data file to have a section like this:

```
Molecules

1 4 1.5
2 4 3.0
3 10 1.0
4 10 1.0
5 10 1.0
...
N 763 4.5
```

where N is the number of atoms, the first field on each line is the atom-ID, the next two are a molecule-ID and a floating point value that will be stored in a new property called "flag". If a per-atom array was specified in the fix property/atom command then the *N* values for that array must be specified consecutively for that property on each line. Note that the order of values on each line corresponds to the order of custom names in the fix property/atom command.

Note that the the lines of per-atom properties can be listed in any order. Also note that all the per-atom properties specified by the fix ID (prop in this case) must be included on each line in the specified data file section (Molecules in this case).

Another way of initializing the new properties is via the *set* command. For example, if you wanted molecules defined for every set of 10 atoms, based on their atom-IDs, these commands could be used:

```
fix prop all property/atom mol
variable cluster atom ((id-1)/10)+1
set atom * mol v_cluster
```

The *atom-style variable* will create values for atoms with IDs 31,32,33,...40 that are 4.0,4.1,4.2,...,4.9. When the *set* commands assigns them to the molecule ID for each atom, they will be truncated to an integer value, so atoms 31-40 will all be assigned a molecule ID of 4.

Note that *atomfile-style variables* can also be used in place of atom-style variables, which means in this case that the molecule IDs could be read-in from a separate file and assigned by the *set* command. This allows you to initialize new per-atom properties in a completely general fashion.

---

For new atom properties specified as *i_name*, *d_name*, *i2_name*, or *d2_name*, the *dump custom* and *compute property/atom* commands can access their values. This means that the values can be used accessed by fixes like *fix ave/atom*, accessed by other computes like *compute reduce*, or used in *atom-style variables*.

For example, these commands will output both the instantaneous and time-averaged values of two new properties to a custom dump file:

```
fix myprops all property/atom i_flag1 d_flag2
compute 1 all property/atom i_flag1 d_flag2
fix 1 all ave/atom 10 10 100 c_1[1] c_1[2]
dump 1 all custom 100 tmp.dump id x y z i_flag1 d_flag2 f_1[1] f_1[2]
```

---

If you wish to add new *pair styles*, *fixes*, or *computes* that use the per-atom properties defined by this fix, see the *Modify atom* doc page which has details on how the custom properties of this fix can be accessed from added classes.

---

Here is an example of using per-atom masses with TIP4P water to study isotope effects. When setting up simulations with the *TIP4P pair styles* for water, you have to provide exactly one atom type each to identify the water oxygen and hydrogen atoms. Since the atom mass is normally tied to the atom type, this makes it impossible to study multiple isotopes in the same simulation. With *fix property/atom rmass* however, the per-type masses are replaced by per-atom masses. Asumming you have a working input deck for regular TIP4P water, where water oxygen is atom type 1 and water hydrogen is atom type 2, the following lines of input script convert this to using per-atom masses:

```
fix Isotopes all property/atom rmass ghost yes
set type 1 mass 15.9994
set type 2 mass 1.008
```

When writing out the system data with the *write_data* command, there will be a new section named with the fix-ID (i.e. *Isotopes* in this case). Alternatively, you can take an existing data file and just add this *Isotopes* section with one line per atom containing atom-ID and mass. Either way, the extended data file can be read back with:

```
fix Isotopes all property/atom rmass ghost yes
read_data tip4p-isotopes.data fix Isotopes NULL Isotopes
```

Please note that the first *Isotopes* refers to the fix-ID and the second to the name of the section. The following input script code will now change the first 100 water molecules in this example to heavy water:

```
group hwat id 2:300:3
group hwat id 3:300:3
set group hwat mass 2.0141018
```

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 2.191.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the per-atom values it stores to *binary restart files*, so that the values can be restored when a simulation is restarted. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

**Warning**

When reading data from a restart file, this fix command has to be specified **after** the *read_restart* command and **exactly** the same was in the input script that created the restart file. LAMMPS will only check whether a fix is of the same style and has the same fix ID and in case of a match will then try to initialize the fix with the data stored in the binary restart file. If the names and associated date types in the new fix property/atom command do not match the old one exactly, data can be corrupted or LAMMPS may crash. If the fix is specified **before** the *read_restart* command its data will not be restored.

None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.191.5 Restrictions

none

### 2.191.6 Related commands

*read_data*, *set*, *compute property/atom*

### 2.191.7 Default

The default keyword value is ghost = no.

## 2.192 fix python/invoke command

### 2.192.1 Syntax

```
fix ID group-ID python/invoke N callback function_name
```

- ID, group-ID are ignored by this fix

- python/invoke = style name of this fix command
- N = execute every N steps
- callback = *post_force* or *end_of_step*

  *post_force* = callback after force computations on atoms every N time steps
  *end_of_step* = callback after every N time steps

### 2.192.2 Examples

```
python post_force_callback here """
from lammps import lammps

def post_force_callback(lammps_ptr, vflag):
    lmp = lammps(ptr=lammps_ptr)
    # access LAMMPS state using Python interface
"""

python end_of_step_callback here """
def end_of_step_callback(lammps_ptr):
    lmp = lammps(ptr=lammps_ptr)
    # access LAMMPS state using Python interface
"""

fix pf  all python/invoke 50 post_force post_force_callback
fix eos all python/invoke 50 end_of_step end_of_step_callback
```

### 2.192.3 Description

This fix allows you to call a Python function during a simulation run. The callback is either executed after forces have been applied to atoms or at the end of every N time steps.

Callback functions must be declared in the global scope of the active Python interpreter. This can either be done by defining it inline using the python command or by importing functions from other Python modules. If LAMMPS is driven using the library interface from Python, functions defined in the driving Python interpreter can also be executed.

Each callback is given a pointer object as first argument. This can be used to initialize an instance of the lammps Python interface, which gives access to the LAMMPS state from Python.

**Warning**

While you can access the state of LAMMPS via library functions from these callbacks, trying to execute input script commands will in the best case not work or in the worst case result in undefined behavior.

### 2.192.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.192.5 Restrictions

This fix is part of the PYTHON package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Building LAMMPS with the PYTHON package will link LAMMPS with the Python library on your system. Settings to enable this are in the lib/python/Makefile.lammps file. See the lib/python/README file for information on those settings.

### 2.192.6 Related commands

*python command*

## 2.193 fix python/move command

### 2.193.1 Syntax

```
fix python/move pymodule.CLASS
```

pymodule.CLASS = use class **CLASS** in module/file **pymodule** to compute how to move atoms

### 2.193.2 Examples

```
fix  1 all python/move py_nve.NVE
fix  1 all python/move py_nve.NVE_OPT
```

### 2.193.3 Description

The *python/move* fix style provides a way to define ways how particles are moved during an MD run from python script code, that is loaded from a file into LAMMPS and executed at the various steps where other fixes can be executed. This python script must contain specific python class definitions.

This allows to implement complex position updates and also modified time integration methods. Due to python being an interpreted language, however, the performance of this fix can be moderately to significantly slower than the corresponding C++ code. For specific cases, this performance penalty can be limited through effective use of NumPy.

---

The python module file has to start with the following code:

```
from __future__ import print_function
import lammps
import ctypes
import traceback
import numpy as np
#
class LAMMPSFix(object):
    def __init__(self, ptr, group_name="all"):
        self.lmp = lammps.lammps(ptr=ptr)
        self.group_name = group_name
#
class LAMMPSFixMove(LAMMPSFix):
    def __init__(self, ptr, group_name="all"):
        super(LAMMPSFixMove, self).__init__(ptr, group_name)
```

(continues on next page)

(continued from previous page)

```
#
    def init(self):
        pass
#
    def initial_integrate(self, vflag):
        pass
#
    def final_integrate(self):
        pass
#
    def initial_integrate_respa(self, vflag, ilevel, iloop):
        pass
#
    def final_integrate_respa(self, ilevel, iloop):
        pass
#
    def reset_dt(self):
        pass
```

Any classes implementing new atom motion functionality have to be derived from the **LAMMPSFixMove** class, overriding the available methods as needed.

Examples for how to do this are in the *examples/python* folder.

---

### 2.193.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.193.5 Restrictions

This pair style is part of the PYTHON package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.193.6 Related commands

*fix nve*, *fix python/invoke*

### 2.193.7 Default

none

## 2.194 fix qbmsst command

### 2.194.1 Syntax

```
fix ID group-ID qbmsst dir shockvel keyword value ...
```

- ID, group-ID are documented in *fix* command
- qbmsst = style name of this fix

- dir = *x* or *y* or *z*
- shockvel = shock velocity (strictly positive, velocity units)
- zero or more keyword/value pairs may be appended
- keyword = *q* or *mu* or *p0* or *v0* or *e0* or *tscale* or *damp* or *seed* or *f_max* or *N_f* or *eta* or *beta* or *T_init*

```
q value = cell mass-like parameter (mass^2/distance^4 units)
mu value = artificial viscosity (mass/distance/time units)
p0 value = initial pressure in the shock equations (pressure units)
v0 value = initial simulation cell volume in the shock equations (distance^3 units)
e0 value = initial total energy (energy units)
tscale value = reduction in initial temperature (unitless fraction between 0.0 and␣
↪1.0)
damp value = damping parameter (time units) inverse of friction gamma
seed value = random number seed (positive integer)
f_max value = upper cutoff frequency of the vibration spectrum (1/time units)
N_f value = number of frequency bins (positive integer)
eta value = coupling constant between the shock system and the quantum thermal bath␣
↪(positive unitless)
beta value = the quantum temperature is updated every beta time steps (positive␣
↪integer)
T_init value = quantum temperature for the initial state (temperature units)
```

## 2.194.2 Examples

```
# (liquid methane modeled with the REAX force field, real units)
fix 1 all qbmsst z 0.122 q 25 mu 0.9 tscale 0.01 damp 200 seed 35082 f_max 0.3 N_f 100␣
↪eta 1 beta 400 T_init 110
# (quartz modeled with the BKS force field, metal units)
fix 2 all qbmsst z 72 q 40 tscale 0.05 damp 1 seed 47508 f_max 120.0 N_f 100 eta 1.0␣
↪beta 500 T_init 300
```

Two example input scripts are given, including shocked $\alpha$-quartz and shocked liquid methane. The input script first equilibrates an initial state with the quantum thermal bath at the target temperature and then applies *fix qbmsst* to simulate shock compression with quantum nuclear correction. The following two figures plot relevant quantities for shocked $\alpha$-quartz.

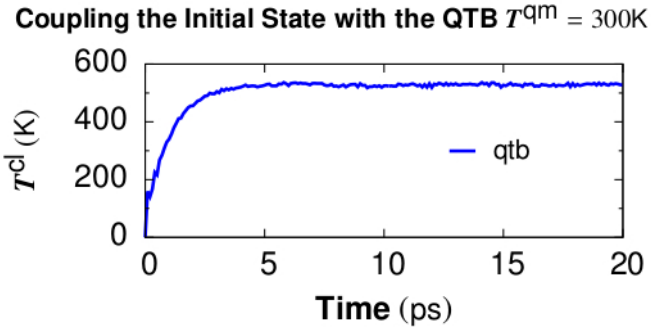



Figure 1. Classical temperature $T_{cl} = \sum \frac{m_i v_i^2}{3Nk_B}$ vs. time for coupling the $\alpha$-quartz initial state with the quantum thermal bath at target quantum temperature $T^{qm} = 300K$. The NpH ensemble is used for time integration while QTB provides the colored random force. $T^{cl}$ converges at the timescale of *damp* which is set to be 1 ps.

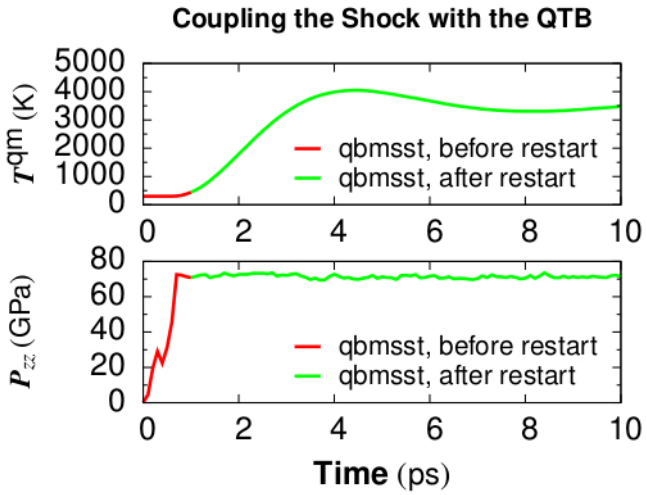



Figure 2. Quantum temperature and pressure vs. time for simulating shocked $\alpha$-quartz with *fix qbmsst*. The shock propagates along the z direction. Restart of the *fix qbmsst* command is demonstrated in the example input script. Thermodynamic quantities stay continuous before and after the restart.

### 2.194.3 Description

This command performs the Quantum-Bath coupled Multi-Scale Shock Technique (QBMSST) integration. See *(Qi)* for a detailed description of this method. QBMSST provides description of the thermodynamics and kinetics of shock processes while incorporating quantum nuclear effects. The *shockvel* setting determines the steady shock velocity that will be simulated along direction *dir*.

Quantum nuclear effects *(fix qtb)* can be crucial especially when the temperature of the initial state is below the classical limit or there is a great change in the zero point energies between the initial and final states. Theoretical post processing quantum corrections of shock compressed water and methane have been reported as much as 30% of the temperatures *(Goldman)*. A self-consistent method that couples the shock to a quantum thermal bath described by a colored noise Langevin thermostat has been developed by Qi et al *(Qi)* and applied to shocked methane. The onset of chemistry is reported to be at a pressure on the shock Hugoniot that is 40% lower than observed with classical molecular dynamics.

It is highly recommended that the system be already in an equilibrium state with a quantum thermal bath at temperature of *T_init*. The fix command *fix qtb* at constant temperature *T_init* could be used before applying this command to introduce self-consistent quantum nuclear effects into the initial state.

The parameters *q*, *mu*, *e0*, *p0*, *v0* and *tscale* are described in the command *fix msst*. The values of *e0*, *p0*, or *v0* will be calculated on the first step if not specified. The parameter of *damp*, *f_max*, and *N_f* are described in the command *fix qtb*.

The *fix qbmsst* command couples the shock system to a quantum thermal bath with a rate that is proportional to the change of the total energy of the shock system, $E^{tot} - E_0^{tot}$. Here $E^{etot}$ consists of both the system energy and a thermal term, see *(Qi)*, and $E_0^{tot} = e0$ is the initial total energy.

The *eta* ($\eta$) parameter is a unitless coupling constant between the shock system and the quantum thermal bath. A small $\eta$ value cannot adjust the quantum temperature fast enough during the temperature ramping period of shock compression while large $\eta$ leads to big temperature oscillation. A value of $\eta$ between 0.3 and 1 is usually appropriate for simulating most systems under shock compression. We observe that different values of $\eta$ lead to almost the same final thermodynamic state behind the shock, as expected.

The quantum temperature is updated every *beta* ($\beta$) steps with an integration time interval $\beta$ times longer than the simulation time step. In that case, $E^{tot}$ is taken as its average over the past $\beta$ steps. The temperature of the quantum thermal bath $T^{qm}$ changes dynamically according to the following equation where $\Delta_t$ is the MD time step and $\gamma$ is the

friction constant which is equal to the inverse of the *damp* parameter.

$$\frac{dT^{qm}}{dt} = \gamma\eta \sum_{l=1}^{\beta} \frac{E^{tot}(t - l\Delta t) - E_0^{tot}}{3\beta N k_B}$$

The parameter *T_init* is the initial temperature of the quantum thermal bath and the system before shock loading.

For all pressure styles, the simulation box stays orthorhombic in shape. Parrinello-Rahman boundary conditions (tilted box) are supported by LAMMPS, but are not implemented for QBMSST.

---

### 2.194.4 Restart, fix_modify, output, run start/stop, minimize info

Because the state of the random number generator is not written to *binary restart files*, this fix cannot be restarted "exactly" in an uninterrupted fashion. However, in a statistical sense, a restarted simulation should produce similar behaviors of the system as if it is not interrupted. To achieve such a restart, one should write explicitly the same value for *q*, *mu*, *damp*, *f_max*, *N_f*, *eta*, and *beta* and set *tscale* = 0 if the system is compressed during the first run.

The cumulative energy change in the system imposed by this fix is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* doc page for details.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the same cumulative energy change due to this fix described in the previous paragraph. The scalar value calculated by this fix is "extensive".

The progress of the QBMSST can be monitored by printing the global scalar and global vector quantities computed by the fix.

As mentioned above, the scalar is the cumulative energy change due to the fix. By monitoring the thermodynamic *econserve* output, this can be used to test if the MD timestep is sufficiently small for accurate integration of the dynamic equations.

The global vector contains five values in the following order. The vector values output by this fix are "intensive".

[*dhugoniot*, *drayleigh*, *lagrangian_speed*, *lagrangian_position*, *quantum_temperature*]

1. *dhugoniot* is the departure from the Hugoniot (temperature units).
2. *drayleigh* is the departure from the Rayleigh line (pressure units).
3. *lagrangian_speed* is the laboratory-frame Lagrangian speed (particle velocity) of the computational cell (velocity units).
4. *lagrangian_position* is the computational cell position in the reference frame moving at the shock speed. This is the distance of the computational cell behind the shock front.
5. *quantum_temperature* is the temperature of the quantum thermal bath $T^{qm}$.

To print these quantities to the log file with descriptive column headers, the following LAMMPS commands are suggested.

```
fix             fix_id all msst z
variable        dhug    equal f_fix_id[1]
variable        dray    equal f_fix_id[2]
variable        lgr_vel equal f_fix_id[3]
variable        lgr_pos equal f_fix_id[4]
variable        T_qm    equal f_fix_id[5]
thermo_style    custom  step temp ke pe lz pzz econserve v_dhug v_dray v_lgr_vel v_lgr_
↪pos v_T_qm f_fix_id
```

It is worth noting that the temp keyword for the *thermo_style* command prints the instantaneous classical temperature $T^{cl}$ as described by the *fix qtb* command.

---

### 2.194.5 Restrictions

This fix style is part of the QTB package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

All cell dimensions must be periodic. This fix can not be used with a triclinic cell. The QBMSST fix has been tested only for the group-ID all.

---

### 2.194.6 Related commands

*fix qtb*, *fix msst*

---

### 2.194.7 Default

The keyword defaults are q = 10, mu = 0, tscale = 0.01, damp = 1, seed = 880302, f_max = 200.0, N_f = 100, eta = 1.0, beta = 100, and T_init=300.0. e0, p0, and v0 are calculated on the first step.

---

**(Goldman)** Goldman, Reed and Fried, J. Chem. Phys. 131, 204103 (2009)

**(Qi)** Qi and Reed, J. Phys. Chem. A 116, 10451 (2012).

## 2.195 fix qeq/point command

Accelerator Variants: *qeq/point/omp*

## 2.196 fix qeq/shielded command

Accelerator Variants: *qeq/shielded/omp*

## 2.197 fix qeq/slater command

Accelerator Variants: *qeq/slater/omp*

## 2.198 fix qeq/ctip command

Accelerator Variants: *qeq/ctip/omp*

## 2.199 fix qeq/dynamic command

Accelerator Variants: *qeq/dynamic/omp*

# 2.200 fix qeq/fire command

Accelerator Variants: *qeq/fire/omp*

## 2.200.1 Syntax

```
fix ID group-ID style Nevery cutoff tolerance maxiter qfile keyword ...
```

- ID, group-ID are documented in *fix* command
- style = *qeq/point* or *qeq/shielded* or *qeq/slater* or *qeq/ctip* or *qeq/dynamic* or *qeq/fire*
- Nevery = perform charge equilibration every this many steps
- cutoff = global cutoff for charge-charge interactions (distance unit)
- tolerance = precision to which charges will be equilibrated
- maxiter = maximum iterations to perform charge equilibration
- qfile = a filename with QEq parameters or *coul/streitz* or *coul/ctip* or *reaxff*
- zero or more keyword/value pairs may be appended
- keyword = *alpha* or *cdamp* or *maxrepeat* or *qdamp* or *qstep* or *warn*

```
alpha value = Slater type orbital exponent (qeq/slater only). Can be followed by␣
↪optional arguments:
  wolf value = width of taper to terminate Coulomb integrals for the Wolf summation␣
↪(default value is zero)
  dsf value = width of taper to terminate Coulomb integrals for the␣
↪Fennell-Gezelter summation (default value is zero)
cdamp value = damping parameter for Coulomb interactions (qeq/ctip only)
maxrepeat value = number of equilibration cycles allowed to ensure no atoms cross␣
↪charge bounds (qeq/ctip only)
qdamp value = damping factor for damped dynamics charge solver (qeq/dynamic and qeq/
↪fire only)
qstep value = time step size for damped dynamics charge solver (qeq/dynamic and qeq/
↪fire only)
warn value = do (=yes) or do not (=no) print a warning when the maximum number of␣
↪iterations is reached
```

## 2.200.2 Examples

```
fix 1 all qeq/point 1 10 1.0e-6 200 param.qeq1
fix 1 qeq qeq/shielded 1 8 1.0e-6 100 param.qeq2
fix 1 all qeq/slater 5 10 1.0e-6 100 params alpha 0.2
fix 1 all qeq/slater 5 10 1.0e-6 100 params alpha 0.2 wolf
fix 1 all qeq/slater 5 10 1.0e-6 100 params alpha 0.2 wolf 2.0
fix 1 all qeq/slater 5 10 1.0e-6 100 params alpha 0.2 dsf
fix 1 all qeq/slater 5 10 1.0e-6 100 params alpha 0.2 dsf 2.0
fix 1 all qeq/ctip 1 12 1.0e-8 100 coul/ctip cdamp 0.30 maxrepeat 10
fix 1 qeq qeq/dynamic 1 12 1.0e-3 100 my_qeq
fix 1 all qeq/fire 1 10 1.0e-3 100 my_qeq qdamp 0.2 qstep 0.1
```

### 2.200.3 Description

Perform the charge equilibration (QEq) method as described in *(Rappe and Goddard)* and formulated in *(Nakano)* (also known as the matrix inversion method) and in *(Rick and Stuart)* (also known as the extended Lagrangian method) based on the electronegativity equilization principle.

These fixes can be used with any *pair style* in LAMMPS, so long as per-atom charges are defined. The most typical use-case is in conjunction with a *pair style* that performs charge equilibration periodically (e.g. every timestep), such as the ReaxFF or Streitz-Mintmire potential. But these fixes can also be used with potentials that normally assume per-atom charges are fixed, e.g. a *Buckingham* or *LJ/Coulombic* potential.

Because the charge equilibration calculation is effectively independent of the pair style, these fixes can also be used to perform a one-time assignment of charges to atoms. For example, you could define the QEq fix, perform a zero-timestep run via the *run* command without any pair style defined which would set per-atom charges (based on the current atom configuration), then remove the fix via the *unfix* command before performing further dynamics.

**Note**

Computing and using charge values different from published values defined for a fixed-charge potential like Buckingham or CHARMM or AMBER, can have a strong effect on energies and forces, and produces a different model than the published versions.

**Note**

The *fix qeq/comb* command must still be used to perform charge equilibration with the *COMB potential*. The *fix qeq/reaxff* command can be used to perform charge equilibration with the *ReaxFF force field*, although fix qeq/shielded yields the same results as fix qeq/reaxff if *Nevery*, *cutoff*, and *tolerance* are the same. Eventually the fix qeq/reaxff command will be deprecated.

The QEq method minimizes the electrostatic energy of the system (or equalizes the derivative of energy with respect to charge of all the atoms) by adjusting the partial charge on individual atoms based on interactions with their neighbors within *cutoff*. It requires a few parameters in the appropriate units for each atom type which are read from a file specified by *qfile*. The file has the following format:

```
1 chi eta gamma zeta qcore
2 chi eta gamma zeta qcore
...
Ntype chi eta gamma zeta qcore
```

except for fix style *qeq/ctip* where the format is:

```
1 chi eta gamma zeta qcore qmin qmax omega
2 chi eta gamma zeta qcore qmin qmax omega
...
Ntype chi eta gamma zeta qcore qmin qmax omega
```

There have to be parameters given for every atom type. Wildcard entries are possible using the same type range syntax as for "coeff" commands (i.e., n*m, n*, *m, *). Later entries will overwrite previous ones. Empty lines or any text following the pound sign (#) are ignored. Each line starts with the atom type followed by eight parameters. Only a subset of the parameters is used by each QEq style as described below, thus the others can be set to 0.0 if desired, but all eight entries per line are required.

- *chi* = electronegativity in energy units

- *eta* = self-Coulomb potential in energy units
- *gamma* = shielded Coulomb constant defined by *ReaxFF force field* in distance units
- *zeta* = Slater type orbital exponent defined by the *Streitz-Mintmire* potential in reverse distance units
- *qcore* = charge of the nucleus defined by the *Streitz-Mintmire potential* potential in charge units
- *qmin* = lower bound on the allowed charge defined by the *CTIP* potential in charge units
- *qmax* = upper bound on the allowed charge defined by the *CTIP* potential in charge units
- *omega* = penalty parameter used to enforce charge bounds defined by the *CTIP* potential in energy units

The fix qeq styles will print a warning if the charges are not equilibrated within *tolerance* by *maxiter* steps, unless the *warn* keyword is used with "no" as argument. This latter option may be useful for testing and benchmarking purposes, as it allows to use a fixed number of QEq iterations when *tolerance* is set to a small enough value to always reach the *maxiter* limit. Turning off warnings will avoid the excessive output in that case.

The *qeq/point* style describes partial charges on atoms as point charges. Interaction between a pair of charged particles is 1/r, which is the simplest description of the interaction between charges. Only the *chi* and *eta* parameters from the *qfile* file are used. Note that Coulomb catastrophe can occur if repulsion between the pair of charged particles is too weak. This style solves partial charges on atoms via the matrix inversion method. A tolerance of 1.0e-6 is usually a good number.

The *qeq/shielded* style describes partial charges on atoms also as point charges, but uses a shielded Coulomb potential to describe the interaction between a pair of charged particles. Interaction through the shielded Coulomb is given by equation (13) of the *ReaxFF force field* paper. The shielding accounts for charge overlap between charged particles at small separation. This style is the same as *fix qeq/reaxff*, and can be used with *pair_style reaxff*. Only the *chi*, *eta*, and *gamma* parameters from the *qfile* file are used. When using the string *reaxff* as filename, these parameters are extracted directly from an active *reaxff* pair style. This style solves partial charges on atoms via the matrix inversion method. A tolerance of 1.0e-6 is usually a good number.

The *qeq/slater* style describes partial charges on atoms as spherical charge densities centered around atoms via the Slater 1*s* orbital, so that the interaction between a pair of charged particles is the product of two Slater 1*s* orbitals. The expression for the Slater 1*s* orbital is given under equation (6) of the *Streitz-Mintmire* paper. Only the *chi*, *eta*, *zeta*, and *qcore* parameters from the *qfile* file are used. When using the string *coul/streitz* as filename, these parameters are extracted directly from an active *coul/streitz* pair style. This style solves partial charges on atoms via the matrix inversion method. A tolerance of 1.0e-6 is usually a good number. Keyword *alpha* can be used to change the Slater type orbital exponent.

Added in version 19Nov2024.

The *qeq/ctip* style describes partial charges on atoms in the same way as style *qeq/shielded* but also enables the definition of charge bounds. Only the *chi*, *eta*, *gamma*, *qmin*, *qmax*, and *omega* parameters from the *qfile* file are used. When using the string *coul/ctip* as filename, these parameters are extracted directly from an active *coul/ctip* pair style. This style solves partial charges on atoms via the matrix inversion method. Keyword *cdamp* can be used to change the damping parameter used to calculate Coulomb interactions. Keyword *maxrepeat* can be used to adjust the number of equilibration cycles allowed to ensure no atoms have crossed the charge bounds. A value of 10 is usually a good choice. A tolerance between 1.0e-6 and 1.0e-8 is usually a good choice but should be checked in conjunction with the timestep for adequate energy conservation during dynamic runs.

The *qeq/dynamic* style describes partial charges on atoms as point charges that interact through 1/r, but the extended Lagrangian method is used to solve partial charges on atoms. Only the *chi* and *eta* parameters from the *qfile* file are used. Note that Coulomb catastrophe can occur if repulsion between the pair of charged particles is too weak. A tolerance of 1.0e-3 is usually a good number. Keyword *qdamp* can be used to change the damping factor, while keyword *qstep* can be used to change the time step size.

The **qeq/fire** style describes the same charge model and charge solver as the *qeq/dynamic* style, but employs a FIRE minimization algorithm to solve for equilibrium charges. Keyword *qdamp* can be used to change the damping factor, while keyword *qstep* can be used to change the time step size.

Note that *qeq/point*, *qeq/shielded*, *qeq/slater*, and *qeq/ctip* describe different charge models, whereas the matrix inversion method and the extended Lagrangian method (*qeq/dynamic* and *qeq/fire*) are different solvers.

Note that *qeq/point*, *qeq/dynamic* and *qeq/fire* styles all describe charges as point charges that interact through 1/r relationship, but solve partial charges on atoms using different solvers. These three styles should yield comparable results if the QEq parameters and *Nevery*, *cutoff*, and *tolerance* are the same. Style *qeq/point* is typically faster, *qeq/dynamic* scales better on larger sizes, and *qeq/fire* is faster than *qeq/dynamic*.

**Note**

In order to solve the self-consistent equations for electronegativity equalization, LAMMPS imposes the additional constraint that all the charges in the fix group must add up to zero. The initial charge assignments should also satisfy this constraint. LAMMPS will print a warning if that is not the case.

**Note**

Developing QEq parameters (chi, eta, gamma, zeta, and qcore) is non-trivial. Charges on atoms are not guaranteed to equilibrate with arbitrary choices of these parameters. We do not develop these QEq parameters. See the examples/qeq directory for some examples.

Added in version 11Feb2026.

In previous versions of LAMMPS, the real-space summations of Coulomb interactions were done by replacing *1/r* using a damped potential *erfc(alpha*r)/r* with the parameter *alpha* controlling the rate of decay. However, any finite value of *alpha* leads to a jump at the cutoff, which interferes with equilibration if atoms move across the cutoff. The charge-neutralized potential of *(Wolf et al.)* (*wolf*) and its extension by *(Fennell and Gezelter)* (*dsf*) solve this problem. An extension was implemented to specify the width of taper (see *(Mei et al.)*) to smoothly terminate the Coulomb integrals at the cutoff. This is done by specifying the optional arguments *wolf* and *dsf* with the value representing the width of taper that smoothly terminates the Coulomb integrals. For example, if the cutoff is 8 A and the taper width is 2 A, the Coulomb integrals are smoothly rescaled from their actual value at r=6 A to zero at r=8 A. For backward compatibility, the default taper width is zero.

### 2.200.4 Restart, fix_modify, output, run start/stop, minimize info

No information about these fixes is written to *binary restart files*. No global scalar or vector or per-atom quantities are stored by these fixes for access by various *output commands*. No parameter of these fixes can be used with the *start/stop* keywords of the *run* command.

Thexe fixes are invoked during *energy minimization*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.200.5 Restrictions

These fixes are part of the QEQ package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

These qeq fixes will ignore electric field contributions from *fix efield*.

### 2.200.6 Related commands

*fix qeq/reaxff*, *fix qeq/comb*

### 2.200.7 Default

warn yes

---

**(Rappe and Goddard)** A. K. Rappe and W. A. Goddard III, J Physical Chemistry, 95, 3358-3363 (1991).

**(Nakano)** A. Nakano, Computer Physics Communications, 104, 59-69 (1997).

**(Rick and Stuart)** S. W. Rick, S. J. Stuart, B. J. Berne, J Chemical Physics 101, 16141 (1994).

**(Streitz-Mintmire)** F. H. Streitz, J. W. Mintmire, Physical Review B, 50, 16, 11996 (1994)

**(CTIP)** G. Plummer, J. P. Tavenner, M. I. Mendelev, Z. Wu, J. W. Lawson, J Chemical Physics, 162, 054709 (2025)

**(ReaxFF)** A. C. T. van Duin, S. Dasgupta, F. Lorant, W. A. Goddard III, J Physical Chemistry, 105, 9396-9049 (2001)

**(QEq/Fire)** T.-R. Shan, A. P. Thompson, S. J. Plimpton, in preparation

**(Wolf)** D. Wolf, P. Keblinski, S. R. Phillpot, J. Eggebrecht, J. Chem. Phys. 110, 8254 (1999).

**(Fennell)** J. Fennell, J. D. Gezelter, J. Chem. Phys. 124, 234104 (2006).

**(Mei)** J. Mei, J. W. Davenport, G. W. Fernando, Phys. Rev. B 43, 4653 (1991).

## 2.201 fix qeq/comb command

Accelerator Variants: *qeq/comb/omp*

### 2.201.1 Syntax

```
fix ID group-ID qeq/comb Nevery precision keyword value ...
```

- ID, group-ID are documented in *fix* command
- qeq/comb = style name of this fix command
- Nevery = perform charge equilibration every this many steps
- precision = convergence criterion for charge equilibration
- zero or more keyword/value pairs may be appended
- keyword = *file*

```
file value = filename
  filename = name of file to write QEQ equilibration info to
```

### 2.201.2 Examples

```
fix 1 surface qeq/comb 10 0.0001
```

### 2.201.3 Description

Perform charge equilibration (QeQ) in conjunction with the COMB (Charge-Optimized Many-Body) potential as described in *(COMB_1)* and *(COMB_2)*. It performs the charge equilibration portion of the calculation using the so-called QEq method, whereby the charge on each atom is adjusted to minimize the energy of the system. This fix can only be used with the COMB potential; see the *fix qeq/reaxff* command for a QeQ calculation that can be used with any potential.

Only charges on the atoms in the specified group are equilibrated. The fix relies on the pair style (COMB in this case) to calculate the per-atom electronegativity (effective force on the charges). An electronegativity equalization calculation (or QEq) is performed in an iterative fashion, which in parallel requires communication at each iteration for processors to exchange charge information about nearby atoms with each other. See *Rappe_and_Goddard* and *Rick_and_Stuart* for details.

During a run, charge equilibration is performed every *Nevery* time steps. Charge equilibration is also always enforced on the first step of each run. The *precision* argument controls the tolerance for the difference in electronegativity for all atoms during charge equilibration. *Precision* is a trade-off between the cost of performing charge equilibration (more iterations) and accuracy.

If the *file* keyword is used, then information about each equilibration calculation is written to the specified file.

**Note**

In order to solve the self-consistent equations for electronegativity equalization, LAMMPS imposes the additional constraint that all the charges in the fix group must add up to zero. The initial charge assignments should also satisfy this constraint. LAMMPS will print a warning if that is not the case.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.201.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify* *respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is performing charge equilibration. Default is the outermost level.

This fix produces a per-atom vector which can be accessed by various *output commands*. The vector stores the gradient of the charge on each atom. The per-atom values be accessed on any timestep.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

This fix can be invoked during *energy minimization*.

### 2.201.5 Restrictions

This fix command currently only supports *pair style *comb**.

### 2.201.6 Related commands

*pair_style comb*

### 2.201.7 Default

No file output is performed.

---

**(COMB_1)** J. Yu, S. B. Sinnott, S. R. Phillpot, Phys Rev B, 75, 085311 (2007),

**(COMB_2)** T.-R. Shan, B. D. Devine, T. W. Kemper, S. B. Sinnott, S. R. Phillpot, Phys Rev B, 81, 125328 (2010).

**(Rappe_and_Goddard)** A. K. Rappe, W. A. Goddard, J Phys Chem 95, 3358 (1991).

**(Rick_and_Stuart)** S. W. Rick, S. J. Stuart, B. J. Berne, J Chem Phys 101, 16141 (1994).

## 2.202 fix qeq/reaxff command

Accelerator Variants: *qeq/reaxff/kk*, *qeq/reaxff/omp*

### 2.202.1 Syntax

```
fix ID group-ID qeq/reaxff Nevery cutlo cuthi tolerance params args
```

- ID, group-ID are documented in *fix* command
- qeq/reaxff = style name of this fix command
- Nevery = perform QEq every this many steps
- cutlo,cuthi = lo and hi cutoff for Taper radius
- tolerance = precision to which charges will be equilibrated
- params = reaxff or a filename
- one or more keywords or keyword/value pairs may be appended

```
keyword = dual or maxiter or nowarn or matfree
  dual = process S and T matrix in parallel (only for qeq/reaxff/omp)
  maxiter N = limit the number of iterations to N
  nowarn = do not print a warning message if the maximum number of iterations was␣
↪reached
  matfree = use a matrix-free approach for applying the H matrix (only for qeq/
↪reaxff/kk)
```

### 2.202.2 Examples

```
fix 1 all qeq/reaxff 1 0.0 10.0 1.0e-6 reaxff
fix 1 all qeq/reaxff 1 0.0 10.0 1.0e-6 param.qeq maxiter 500
```

### 2.202.3 Description

Perform the charge equilibration (QEq) method as described in *(Rappe and Goddard)* and formulated in *(Nakano)*. It is typically used in conjunction with the ReaxFF force field model as implemented in the *pair_style reaxff* command, but it can be used with any potential in LAMMPS, so long as it defines and uses charges on each atom. The *fix qeq/comb* command should be used to perform charge equilibration with the *COMB potential*. For more technical details about the charge equilibration performed by fix qeq/reaxff, see the *(Aktulga)* paper.

The QEq method minimizes the electrostatic energy of the system by adjusting the partial charge on individual atoms based on interactions with their neighbors. It requires some parameters for each atom type. If the *params* setting above is the word "reaxff", then these are extracted from the *pair_style reaxff* command and the ReaxFF force field file it reads in. If a file name is specified for *params*, then the parameters are taken from the specified file and the file must contain one line for each atom type. The latter form must be used when performing QEq with a non-ReaxFF potential. Each line should be formatted as follows:

```
itype chi eta gamma
```

where *itype* is the atom type from 1 to Ntypes, *chi* denotes the electronegativity in eV, *eta* denotes the self-Coulomb potential in eV, and *gamma* denotes the valence orbital exponent. Note that these 3 quantities are also in the ReaxFF potential file, except that eta is defined here as twice the eta value in the ReaxFF file. Note that unlike the rest of LAMMPS, the units of this fix are hard-coded to be A, eV, and electronic charge.

The optional *dual* keyword allows to perform the optimization of the S and T matrices in parallel. This is only supported for the *qeq/reaxff/omp* style. Otherwise they are processed separately. The *qeq/reaxff/kk* style always solves the S and T matrices in parallel.

The optional *maxiter* keyword allows changing the max number of iterations in the linear solver. The default value is 200.

The optional *nowarn* keyword silences the warning message printed when the maximum number of iterations was reached. This can be useful for comparing serial and parallel results where having the same fixed number of QEq iterations is desired, which can be achieved by using a very small tolerance and setting *maxiter* to the desired number of iterations.

The optional *matfree* keyword replaces the sequence of explicitly constructing the H matrix, then (repeatedly) applying it with a matrix-free approach where the H matrix is effectively regenerated each time it is applied. This trades performance for reduced memory requirements because it avoids the overheads of storing the matrix. This is only supported for the *qeq/reaxff/kk* style, with both full and half qeq neighbor lists supported.

**Note**

In order to solve the self-consistent equations for electronegativity equalization, LAMMPS imposes the additional constraint that all the charges in the fix group must add up to zero. The initial charge assignments should also satisfy this constraint. LAMMPS will print a warning if that is not the case.

### 2.202.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. This fix computes a global scalar (the number of iterations) for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

This fix is invoked during *energy minimization*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.202.5 Restrictions

This fix is part of the REAXFF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix does not correctly handle interactions involving multiple periodic images of the same atom. Hence, it should not be used for periodic cell dimensions smaller than the non-bonded cutoff radius, which is typically 10 Å for ReaxFF simulations.

This fix may be used in combination with *fix efield* and will apply the external electric field during charge equilibration, but there may be only one fix efield instance used and the electric field vector may only have components in non-periodic directions. Equal-style variables can be used for electric field vector components without any further settings. Atom-style variables can be used for spatially-varying electric field vector components, but the resulting electric potential must be specified as an atom-style variable using the *potential* keyword for *fix efield*.

### 2.202.6 Related commands

*pair_style reaxff*, *fix qeq/shielded*, *fix acks2/reaxff*, *fix qtpie/reaxff*, *fix qeq/rel/reaxff*

### 2.202.7 Default

maxiter 200

---

**(Rappe)** Rappe and Goddard III, Journal of Physical Chemistry, 95, 3358-3363 (1991).

**(Nakano)** Nakano, Computer Physics Communications, 104, 59-69 (1997).

**(Aktulga)** Aktulga, Fogarty, Pandit, Grama, Parallel Computing, 38, 245-259 (2012).

## 2.203 fix qeq/rel/reaxff command

### 2.203.1 Syntax

```
fix ID group-ID qeq/rel/reaxff Nevery cutlo cuthi tolerance params gfile args
```

- ID, group-ID are documented in *fix* command
- qeq/rel/reaxff = style name of this fix command

- Nevery = perform QEqR every this many steps
- cutlo,cuthi = lo and hi cutoff for Taper radius
- tolerance = precision to which charges will be equilibrated
- params = reaxff or a filename
- gfile = the name of a file containing Gaussian orbital exponents
- one or more keywords or keyword/value pairs may be appended

```
keyword = scale or maxiter or nowarn
  scale beta = set value of scaling factor beta (determines strength of electric␣
↪polarization)
  maxiter N = limit the number of iterations to N
  nowarn = do not print a warning message if the maximum number of iterations is␣
↪reached
```

### 2.203.2 Examples

```
fix 1 all qeq/rel/reaxff 1 0.0 10.0 1.0e-6 reaxff exp.qeqr
fix 1 all qeq/rel/reaxff 1 0.0 10.0 1.0e-6 params.qeqr exp.qeqr scale 1.5 maxiter 500␣
↪nowarn
```

### 2.203.3 Description

Added in version 2Apr2025.

This fix implements the QEqR method *(Lalli)* for charge equilibration, which differs from the QEq charge equilibration method *(Rappe and Goddard)* only in how external electric fields are accounted for. This fix therefore raises a warning when used without *fix efield* since *fix qeq/reaxff* should be used when no external electric field is present. Charges are computed with the QEqR method by minimizing the electrostatic energy of the system in the same way as the QEq method but where the absolute electronegativity, $\chi_i$, of each atom in the QEq method is replaced with an effective electronegativity given by

$$\chi_{ri} = \chi_i + \frac{\sum_{j=1}^{N} \beta(\phi_i - \phi_j) S_{ij}}{\sum_{m=1}^{N} S_{im}},$$

where $N$ is the number of atoms in the system, $\beta$ is a scaling factor, $\phi_i$ and $\phi_j$ are the electric potentials at the positions of atoms $i$ and $j$ due to the external electric field and $S_{ij}$ is the overlap integral between atoms $i$ and $j$. This formulation is advantageous over the method used by *fix qeq/reaxff* to account for an external electric field in that it permits periodic boundaries in the direction of an external electric field and in that it does not worsen long-range charge transfer seen with QEq. See *Lalli* for further details.

This fix is typically used in conjunction with the ReaxFF force field model as implemented in the *pair_style reaxff* command, but it can be used with any potential in LAMMPS, so long as it defines and uses charges on each atom. For more technical details about the charge equilibration performed by *fix qeq/rel/reaxff*, which is the same as in *fix qeq/reaxff* except for the use of $\chi_{ri}$, please refer to *(Aktulga)*. To be explicit, *fix qeq/rel/reaxff* replaces $\chi_k$ of eq. 3 in *(Aktulga)* with $\chi_{rk}$ when an external electric field is applied.

This fix requires the absolute electronegativity, $\chi$, in eV, the self-Coulomb potential, $\eta$, in eV, and the shielded Coulomb constant, $\gamma$, in $\text{Å}^{-1}$. If the *params* setting above is the word "reaxff", then these are extracted from the *pair_style reaxff* command and the ReaxFF force field file it reads in. If a file name is specified for *params*, then the parameters are taken from the specified file and the file must contain one line for each atom type. The latter form must be used when using this fix with a non-ReaxFF potential. Each line should be formatted as follows, ensuring that the parameters are given in units of eV, eV, and $\text{Å}^{-1}$, respectively:

```
itype chi eta gamma
```

where *itype* is the atom type from 1 to Ntypes. Note that eta is defined here as twice the eta value in the ReaxFF file.

The overlap integrals $S_{ij}$ are computed by using normalized 1s Gaussian type orbitals. The Gaussian orbital exponents, $\alpha$, that are needed to compute the overlap integrals are taken from the file given by *gfile*. This file must contain one line for each atom type and provide the Gaussian orbital exponent for each atom type in units of inverse square Bohr radius. Each line should be formatted as follows:

```
itype alpha
```

Empty lines or any text following the pound sign (#) are ignored. An example *gfile* for a system with two atom types is

```
# An example gfile. Exponents are taken from Table 2.2 of Chen, J. (2009).
# Theory and applications of fluctuating-charge models.
# The units of the exponents are 1 / (Bohr radius)^2 .
1  0.2240  # O
2  0.5434  # H
```

The optional *scale* keyword sets the value of $\beta$ in the equation for $\chi_{ri}$. The default value is 1.0.

The optional *maxiter* keyword allows changing the max number of iterations in the linear solver. The default value is 200.

The optional *nowarn* keyword silences the warning message printed when the maximum number of iterations is reached. This can be useful for comparing serial and parallel results where having the same fixed number of iterations is desired, which can be achieved by using a very small tolerance and setting *maxiter* to the desired number of iterations.

**Note**

In order to solve the self-consistent equations for electronegativity equalization, LAMMPS imposes the additional constraint that all the charges in the fix group must add up to zero. The initial charge assignments should also satisfy this constraint. LAMMPS will print a warning if that is not the case.

### 2.203.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. This fix computes a global scalar (the number of iterations) and a per-atom vector (the effective electronegativity), which can be accessed by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

This fix is invoked during *energy minimization*.

### 2.203.5 Restrictions

This fix is part of the REAXFF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix does not correctly handle interactions involving multiple periodic images of the same atom. Hence, it should not be used for periodic cell dimensions smaller than the non-bonded cutoff radius, which is typically 10 Å for ReaxFF simulations.

This fix may be used in combination with *fix efield* and will apply the external electric field during charge equilibration, but there may be only one fix efield instance used and the electric field must be applied to all atoms in the system. Consequently, *fix efield* must be used with *group-ID* all and must not be used with the keyword *region*. Equal-style variables can be used for electric field vector components without any further settings. Atom-style variables can be

used for spatially-varying electric field vector components, but the resulting electric potential must be specified as an atom-style variable using the *potential* keyword for *fix efield*.

### 2.203.6 Related commands

*pair_style reaxff*, *fix qeq/reaxff*, *fix acks2/reaxff*, *fix qtpie/reaxff*

### 2.203.7 Default

scale = 1.0 and maxiter = 200

---

**(Lalli)** Lalli and Giusti, Journal of Chemical Physics, 162, 174311 (2025).

**(Rappe)** Rappe and Goddard III, Journal of Physical Chemistry, 95, 3358-3363 (1991).

**(Aktulga)** Aktulga, Fogarty, Pandit, Grama, Parallel Computing, 38, 245-259 (2012).

## 2.204 fix qmmm command

### 2.204.1 Syntax

```
fix ID group-ID qmmm
```

- ID, group-ID are documented in *fix* command
- qmmm = style name of this fix command

### 2.204.2 Examples

```
fix 1 qmol qmmm
```

### 2.204.3 Description

This fix provides functionality to enable a quantum mechanics/molecular mechanics (QM/MM) coupling of LAMMPS to a quantum mechanical code. The current implementation only supports an ONIOM style mechanical coupling to the Quantum ESPRESSO plane wave DFT package. Electrostatic coupling is in preparation and the interface has been written in a manner that coupling to other QM codes should be possible without changes to LAMMPS itself.

The interface code for this is in the lib/qmmm directory of the LAMMPS distribution and is being made available at this early stage of development in order to encourage contributions for interfaces to other QM codes. This will allow the LAMMPS side of the implementation to be adapted if necessary before being finalized.

Details about how to use this fix are currently documented in the description of the QM/MM interface code itself in lib/qmmm/README.

### 2.204.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global scalar or vector or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.204.5 Restrictions

This fix is part of the QMMM package. It is only enabled if LAMMPS was built with that package. It also requires building a library provided with LAMMPS. See the *Build package* page for more info.

The fix is only functional when LAMMPS is built as a library and linked with a compatible QM program and a QM/MM front end into a QM/MM executable. See the lib/qmmm/README file for details.

### 2.204.6 Related commands

none

### 2.204.7 Default

none

## 2.205 fix qtb command

### 2.205.1 Syntax

```
fix ID group-ID qtb keyword value ...
```

- ID, group-ID are documented in *fix* command
- qtb = style name of this fix
- zero or more keyword/value pairs may be appended
- keyword = *temp* or *damp* or *seed* or *f_max* or *N_f*

  *temp* value = target quantum temperature (temperature units)
  *damp* value = damping parameter (time units) inverse of friction *gamma*
  *seed* value = random number seed (positive integer)
  *f_max* value = upper cutoff frequency of the vibration spectrum (1/time units)
  *N_f* value = number of frequency bins (positive integer)

### 2.205.2 Examples

```
# (liquid methane modeled with the REAX force field, real units)
fix 1 all nve
fix 1 all qtb temp 110 damp 200 seed 35082 f_max 0.3 N_f 100
# (quartz modeled with the BKS force field, metal units)
fix 2 all nph iso 1.01325 1.01325 1
fix 2 all qtb temp 300 damp 1 seed 47508 f_max 120.0 N_f 100
```

### 2.205.3 Description

This command performs the quantum thermal bath scheme proposed by *(Dammak)* to include self-consistent quantum nuclear effects, when used in conjunction with the *fix nve* or *fix nph* commands.

Classical molecular dynamics simulation does not include any quantum nuclear effect. Quantum treatment of the vibrational modes will introduce zero point energy into the system, alter the energy power spectrum and bias the heat capacity from the classical limit. Missing all the quantum nuclear effects, classical MD cannot model systems at temperatures lower than their classical limits. This effect is especially important for materials with a large population of hydrogen atoms and thus higher classical limits.

The equation of motion implemented by this command follows a Langevin form:

$$m_i a_i = f_i + R_i - m_i \gamma v_i$$

Here $m_i, a_i, f_i, R_i, \gamma,$ and$v_i$ represent in this order mass, acceleration, force exerted by all other atoms, random force, frictional coefficient (the inverse of damping parameter damp), and velocity. The random force $R_i$ is "colored" so that any vibrational mode with frequency $\omega$ will have a temperature-sensitive energy $\theta(\omega, T)$ which resembles the energy expectation for a quantum harmonic oscillator with the same natural frequency:

$$\theta(\omega T) = \frac{\hbar}{2} + \hbar\omega \left[\exp(\frac{\hbar\omega}{k_B T}) - 1\right]^{-1}$$

To efficiently generate the random forces, we employ the method of *(Barrat)*, that circumvents the need to generate all random forces for all times before the simulation. The memory requirement of this approach is less demanding and independent of the simulation duration. Since the total random force $R_{tot}$ does not necessarily vanish for a finite number of atoms, $R_i$ is replaced by $R_i - \frac{R_{tot}}{N_{tot}}$ to avoid collective motion of the system.

The *temp* parameter sets the target quantum temperature. LAMMPS will still have an output temperature in its thermo style. That is the instantaneous classical temperature $T^{cl}$ derived from the atom velocities at thermal equilibrium. A non-zero $T^{cl}$ will be present even when the quantum temperature approaches zero. This is associated with zero-point energy at low temperatures.

$$T^{cl} = \sum \frac{m_i v_i^2}{3Nk_B}$$

The *damp* parameter is specified in time units, and it equals the inverse of the frictional coefficient $\gamma$. $\gamma$ should be as small as possible but slightly larger than the timescale of anharmonic coupling in the system which is about 10 ps to 100 ps. When $\gamma$ is too large, it gives an energy spectrum that differs from the desired Bose-Einstein spectrum. When $\gamma$ is too small, the quantum thermal bath coupling to the system will be less significant than anharmonic effects, reducing to a classical limit. We find that setting $\gamma$ between 5 THz and 1 THz could be appropriate depending on the system.

The random number *seed* is a positive integer used to initiate a Marsaglia random number generator. Each processor uses the input seed to generate its own unique seed and its own stream of random numbers. Thus the dynamics of the system will not be identical on two runs on different numbers of processors.

The *f_max* parameter truncate the noise frequency domain so that vibrational modes with frequencies higher than *f_max* will not be modulated. If we denote $\Delta t$ as the time interval for the MD integration, *f_max* is always reset by the code to make $\alpha = (int)(2\, f_max\, \Delta t)^{-1}$ a positive integer and print out relative information. An appropriate value for the cutoff frequency *f_max* would be around 2~3 $f_D$, where $f_D$ is the Debye frequency.

The *N_f* parameter is the frequency grid size, the number of points from 0 to *f_max* in the frequency domain that will be sampled. 3*2*N_f* per-atom random numbers are required in the random force generation and there could be as many atoms as in the whole simulation that can migrate into every individual processor. A larger *N_f* provides a more accurate sampling of the spectrum while consumes more memory. With fixed *f_max* and $\gamma$, *N_f* should be big enough to converge the classical temperature $T^{cl}$ as a function of target quantum bath temperature. Memory usage per processor could be from 10 to 100 MBytes.

**Note**

Unlike the *fix nvt* command which performs Nose/Hoover thermostatting AND time integration, this fix does NOT perform time integration. It only modifies forces to a colored thermostat. Thus you must use a separate time integration fix, like *fix nve* or *fix nph* to actually update the velocities and positions of atoms (as shown in the examples). Likewise, this fix should not normally be used with other fixes or commands that also specify system temperatures , e.g. *fix nvt* and *fix temp/rescale*.

### 2.205.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. Because the state of the random number generator is not saved in restart files, this means you cannot do "exact" restarts with this fix. However, in a statistical sense, a restarted simulation should produce similar behaviors of the system.

This fix is not invoked during *energy minimization*.

---

### 2.205.5 Restrictions

This fix style is part of the QTB package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

---

### 2.205.6 Related commands

*fix nve*, *fix nph*, *fix langevin*, *fix qbmsst*

---

### 2.205.7 Default

The keyword defaults are temp = 300, damp = 1, seed = 880302, f_max=200.0 and N_f = 100.

---

**(Dammak)** Dammak, Chalopin, Laroche, Hayoun, and Greffet, Phys Rev Lett, 103, 190601 (2009).

**(Barrat)** Barrat and Rodney, J. Stat. Phys, 144, 679 (2011).

## 2.206 fix qtpie/reaxff command

### 2.206.1 Syntax

```
fix ID group-ID qtpie/reaxff Nevery cutlo cuthi tolerance params gfile args
```

- ID, group-ID are documented in *fix* command
- qtpie/reaxff = style name of this fix command
- Nevery = perform QTPIE every this many steps
- cutlo,cuthi = lo and hi cutoff for Taper radius
- tolerance = precision to which charges will be equilibrated
- params = reaxff or a filename
- gfile = the name of a file containing Gaussian orbital exponents
- one or more keywords or keyword/value pairs may be appended

```
keyword = scale or maxiter or nowarn
  scale beta = set value of scaling factor beta (determines strength of electric␣
↪polarization)
  maxiter N = limit the number of iterations to N
  nowarn = do not print a warning message if the maximum number of iterations is␣
↪reached
```

### 2.206.2 Examples

```
fix 1 all qtpie/reaxff 1 0.0 10.0 1.0e-6 reaxff exp.qtpie
fix 1 all qtpie/reaxff 1 0.0 10.0 1.0e-6 params.qtpie exp.qtpie scale 1.5 maxiter 500␣
↪nowarn
```

### 2.206.3 Description

Added in version 19Nov2024.

The QTPIE charge equilibration method is an extension of the QEq charge equilibration method. With QTPIE, the partial charges on individual atoms are computed by minimizing the electrostatic energy of the system in the same way as the QEq method but where the absolute electronegativity, $\chi_i$, of each atom in the QEq charge equilibration scheme *(Rappe and Goddard)* is replaced with an effective electronegativity given by *(Chen)*

$$\tilde{\chi}_i = \frac{\sum_{j=1}^{N}(\chi_i - \chi_j)S_{ij}}{\sum_{m=1}^{N} S_{im}},$$

which acts to penalize long-range charge transfer seen with the QEq charge equilibration scheme. In this equation, $N$ is the number of atoms in the system and $S_{ij}$ is the overlap integral between atom $i$ and atom $j$.

The effect of an external electric field can be incorporated into the QTPIE method by modifying the absolute or effective electronegativities of each atom *(Chen)*. This fix models the effect of an external electric field by using the effective electronegativity *(Lalli)*

$$\tilde{\chi}_{ri} = \frac{\sum_{j=1}^{N}(\chi_i - \chi_j + \beta(\phi_i - \phi_j))S_{ij}}{\sum_{m=1}^{N} S_{im}},$$

where $\beta$ is a scaling factor and $\phi_i$ and $\phi_j$ are the electric potentials at the positions of atoms $i$ and $j$ due to the external electric field. Additional details regarding the implementation and performance of this fix are provided in *Lalli*.

This fix is typically used in conjunction with the ReaxFF force field model as implemented in the *pair_style reaxff* command, but it can be used with any potential in LAMMPS, so long as it defines and uses charges on each atom. For more technical details about the charge equilibration performed by *fix qtpie/reaxff*, which is the same as in *fix qeq/reaxff* except for the use of $\tilde{\chi}_i$ or $\tilde{\chi}_{ri}$, please refer to *(Aktulga)*. To be explicit, this fix replaces $\chi_k$ of eq. 3 in *(Aktulga)* with $\tilde{\chi}_k$ when no external electric field is applied and with $\tilde{\chi}_{rk}$ when an external electric field is applied.

This fix requires the absolute electronegativity, $\chi$, in eV, the self-Coulomb potential, $\eta$, in eV, and the shielded Coulomb constant, $\gamma$, in $\text{Å}^{-1}$. If the *params* setting above is the word "reaxff", then these are extracted from the *pair_style reaxff* command and the ReaxFF force field file it reads in. If a file name is specified for *params*, then the parameters are taken from the specified file and the file must contain one line for each atom type. The latter form must be used when performing QTPIE with a non-ReaxFF potential. Each line should be formatted as follows, ensuring that the parameters are given in units of eV, eV, and $\text{Å}^{-1}$, respectively:

```
itype chi eta gamma
```

where *itype* is the atom type from 1 to Ntypes. Note that eta is defined here as twice the eta value in the ReaxFF file.

The overlap integrals $S_{ij}$ are computed by using normalized 1s Gaussian type orbitals. The Gaussian orbital exponents, $\alpha$, that are needed to compute the overlap integrals are taken from the file given by *gfile*. This file must contain one line for each atom type and provide the Gaussian orbital exponent for each atom type in units of inverse square Bohr radius. Each line should be formatted as follows:

```
itype alpha
```

Empty lines or any text following the pound sign (#) are ignored. An example *gfile* for a system with two atom types is

```
# An example gfile. Exponents are taken from Table 2.2 of Chen, J. (2009).
# Theory and applications of fluctuating-charge models.
# The units of the exponents are 1 / (Bohr radius)^2 .
1  0.2240  # O
2  0.5434  # H
```

The optional *scale* keyword sets the value of $\beta$ in the equation for $\tilde{\chi}_{ri}$. This keyword only affects the computed charges when *fix efield* is used. The default value is 1.0.

The optional *maxiter* keyword allows changing the max number of iterations in the linear solver. The default value is 200.

The optional *nowarn* keyword silences the warning message printed when the maximum number of iterations is reached. This can be useful for comparing serial and parallel results where having the same fixed number of iterations is desired, which can be achieved by using a very small tolerance and setting *maxiter* to the desired number of iterations.

**Note**

In order to solve the self-consistent equations for electronegativity equalization, LAMMPS imposes the additional constraint that all the charges in the fix group must add up to zero. The initial charge assignments should also satisfy this constraint. LAMMPS will print a warning if that is not the case.

### 2.206.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. This fix computes a global scalar (the number of iterations) and a per-atom vector (the effective electronegativity), which can be accessed by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

This fix is invoked during *energy minimization*.

### 2.206.5 Restrictions

This fix is part of the REAXFF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix does not correctly handle interactions involving multiple periodic images of the same atom. Hence, it should not be used for periodic cell dimensions smaller than the non-bonded cutoff radius, which is typically 10 Å for ReaxFF simulations.

This fix may be used in combination with *fix efield* and will apply the external electric field during charge equilibration, but there may be only one fix efield instance used and the electric field must be applied to all atoms in the system. Consequently, *fix efield* must be used with *group-ID* all and must not be used with the keyword *region*. Equal-style variables can be used for electric field vector components without any further settings. Atom-style variables can be used for spatially-varying electric field vector components, but the resulting electric potential must be specified as an atom-style variable using the *potential* keyword for *fix efield*.

### 2.206.6 Related commands

*pair_style reaxff*, *fix qeq/reaxff*, *fix acks2/reaxff*, *fix qeq/rel/reaxff*

### 2.206.7 Default

scale = 1.0 and maxiter = 200

**(Rappe)** Rappe and Goddard III, Journal of Physical Chemistry, 95, 3358-3363 (1991).

**(Chen)** Chen, Jiahao. Theory and applications of fluctuating-charge models. University of Illinois at Urbana-Champaign, 2009.

**(Lalli)** Lalli and Giusti, Journal of Chemical Physics, 162, 174311 (2025).

**(Aktulga)** Aktulga, Fogarty, Pandit, Grama, Parallel Computing, 38, 245-259 (2012).

## 2.207 fix reaxff/bonds command

Accelerator Variants: *reaxff/bonds/kk*

### 2.207.1 Syntax

```
fix ID group-ID reaxff/bonds Nevery filename
```

- ID, group-ID are documented in *fix* command
- reaxff/bonds = style name of this fix command
- Nevery = output interval in timesteps
- filename = name of output file

### 2.207.2 Examples

```
fix 1 all reaxff/bonds 100 bonds.reaxff
```

### 2.207.3 Description

Write out the bond information computed by the ReaxFF potential specified by *pair_style reaxff* in the exact same format as the original stand-alone ReaxFF code of Adri van Duin. The bond information is written to *filename* on timesteps that are multiples of *Nevery*, including timestep 0. For time-averaged chemical species analysis, please see the *fix reaxff/species* command.

The specified group-ID is ignored by this fix except for the *dump image* related functionality (see below).

The format of the output file should be reasonably self-explanatory. The meaning of the column header abbreviations is as follows:

- id = atom id
- type = atom type
- nb = number of bonds
- id_1 = atom id of first bond
- id_nb = atom id of Nth bond
- mol = molecule id
- bo_1 = bond order of first bond
- bo_nb = bond order of Nth bond
- abo = atom bond order (sum of all bonds)
- nlp = number of lone pairs
- q = atomic charge

If the filename ends with ".gz" or some *other supported compression format suffix*, the output file is written in compressed format. A compressed output file can be significantly smaller than the text version, but will also take longer to write.

Added in version 2Apr2025.

If the filename contains the wildcard character "*", a new file is created on every timestep where bond information is written. The "*" character is replaced with the timestep value. Note that the *fix_modify pad* command can be used so that all timestep numbers have the same length by adding leading zeroes (e.g. 00010 for a pad value of 5). The default pad value is 0, i.e. no leading zeroes.

Added in version 11Feb2026.

If the filename is "NULL", then no output is created. This can be useful when using fix *reaxff/bonds* in combination with *dump image fix* keyword to visualize the bonds computed by the ReaxFF force field.

### 2.207.4 Dump image info

Added in version 11Feb2026.

Fix *reaxff/bonds* supports the *fix* keyword of *dump image*. The fix will pass geometry information about the bonds computed by the *ReaxFF pair style* to *dump image* so that they can be included in the rendered image. Only bonds where *both* atoms are within the fix group generate graphics objects that are displayed in the dumped images. That group may be a dynamic group.

The color of the bonds is by default that of the atoms when using color styles "type" or "element". With color style "const" the default value of "white" can be changed using *dump_modify fcolor*. The transparency is by default fully opaque and can be changed with *dump_modify ftrans*.

The *fflag1* setting of *dump image fix* determines whether the bonds will be capped with spheres (1) or not (0).

The *fflag2* setting allows to adjust diameter of the cylinders for the bonds. By default, a diameter of 0.5 length units will be used.

---

### 2.207.5 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. This fix supports the *fix_modify pad* option. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.207.6 Restrictions

The fix reaxff/bonds command requires that the *pair_style reaxff* is invoked. This fix is part of the REAXFF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

To write compressed bond files, you must compile LAMMPS with the `-DLAMMPS_GZIP` option. See the *Build settings* doc page for details.

### 2.207.7 Related commands

*pair_style reaxff*, *fix reaxff/species*

### 2.207.8 Default

pad = 0

## 2.208 fix reaxff/species command

Accelerator Variants: *reaxff/species/kk*

### 2.208.1 Syntax

```
fix ID group-ID reaxff/species Nevery Nrepeat Nfreq filename keyword value ...
```

- ID, group-ID are documented in *fix* command
- reaxff/species = style name of this command
- Nevery = sample bond-order every this many timesteps
- Nrepeat = # of bond-order samples used for calculating averages
- Nfreq = calculate average bond-order every this many timesteps
- filename = name of output file
- zero or more keyword/value pairs may be appended
- keyword = *cutoff* or *element* or *position* or *delete* or *delete_rate_limit*

```
cutoff value = I J Cutoff
  I, J = atom types (see asterisk form below)
  Cutoff = Bond-order cutoff value for this pair of atom types
element value = Element1, Element2, ...
position value = posfreq filepos
  posfreq = write position files every this many timestep
  filepos = name of position output file
delete value = filedel keyword value
  filedel = name of delete species output file
  keyword = specieslist or masslimit
    specieslist value = Nspecies Species1 Species2 ...
      Nspecies = number of species in list
    masslimit value = massmin massmax
      massmin = minimum molecular weight of species to delete
      massmax = maximum molecular weight of species to delete
delete_rate_limit value = Nlimit Nsteps
      Nlimit = maximum number of deletions allowed to occur within interval
      Nsteps = the interval (number of timesteps) over which to count deletions
```

*delete_subgroup* value = group-ID
group-ID = name of a *group*

### 2.208.2 Examples

```
fix 1 all reaxff/species 10 10 100 species.out
fix 1 all reaxff/species 1 2 20 species.out cutoff 1 1 0.40 cutoff 1 2*3 0.55
fix 1 all reaxff/species 1 100 100 species.out element Au O H position 1000 AuOH.pos
fix 1 all reaxff/species 1 100 100 species.out delete species.del masslimit 0 50
```

### 2.208.3 Description

Write out the chemical species information computed by the ReaxFF potential specified by *pair_style reaxff*. Bond-order values (either averaged or instantaneous, depending on value of *Nrepeat*) are used to determine chemical bonds. Every *Nfreq* timesteps, chemical species information is written to *filename* as a two line output. The first line is a header containing labels. The second line consists of the following: timestep, total number of molecules, total number of distinct species, number of molecules of each species. In this context, "species" means a unique molecule. The chemical formula of each species is given in the first line.

**Warning**

In order to compute averaged data, it is required that there are no neighbor list rebuilds for at least Nrepeat*Nevery steps preceding each *Nfreq* step. For that reason, fix *reaxff/species* may change your neighbor list settings. Reneighboring will occur no more frequently than every Nrepeat*Nevery timesteps, and will occur less frequently if *Nfreq* is not a multiple of Nrepeat*Nevery. There will be a warning message showing the new settings. Having a *Nfreq* setting that is larger than what is required for correct computation of the ReaxFF force field interactions, in combination with certain *Nrepeat* and *Nevery* settings, can thus lead to incorrect results. For typical ReaxFF calculations, reneighboring only every 100 steps is already quite a low frequency.

If the filename ends with ".gz" or some *other supported compression format suffix*, the output file is written in compressed format. A compressed output file can be significantly smaller than the text version, but will also take longer to write.

Added in version 15Jun2023: Support for wildcards added

Optional keyword *cutoff* can be assigned to change the minimum bond-order values used in identifying chemical bonds between pairs of atoms. Bond-order cutoffs should be carefully chosen, as bond-order cutoffs that are too small may include too many bonds (which will result in an error), while cutoffs that are too large will result in fragmented molecules. The default cutoff of 0.3 usually gives good results. A wildcard asterisk can be used in place of or in conjunction with the I,J arguments to set the bond-order cutoff for multiple pairs of atom types. This takes the form "*" or "*n" or "n*" or "m*n". If *N* is the number of atom types, then an asterisk with no numeric values means all types from 1 to *N*. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to *N* (inclusive). A middle asterisk means all types from m to n (inclusive).

The optional keyword *element* can be used to specify the chemical symbol printed for each LAMMPS atom type. The number of symbols must match the number of LAMMPS atom types and each symbol must consist of 1 or 2 alphanumeric characters. By default, these symbols are the same as the chemical identity of each LAMMPS atom type, as specified by the *ReaxFF pair_coeff* command and the ReaxFF force field file.

The optional keyword *position* writes center-of-mass positions of each identified molecules to file *filepos* every *posfreq* timesteps. The first line contains information on timestep, total number of molecules, total number of distinct species, and box dimensions. The second line is a header containing labels. From the third line downward, each molecule writes a line of output containing the following information: molecule ID, number of atoms in this molecule, chemical

formula, total charge, and center-of-mass xyz positions of this molecule. The xyz positions are in fractional coordinates relative to the box dimensions.

For the keyword *position*, the *filepos* is the name of the output file. It can contain the wildcard character "*". If the "*" character appears in *filepos*, then one file per snapshot is written at *posfreq* and the "*" character is replaced with the timestep value. For example, AuO.pos.* becomes AuO.pos.0, AuO.pos.1000, etc.

Added in version 3Aug2022.

The optional keyword *delete* enables the periodic removal of molecules from the system *(Gissinger)*. Criteria for deletion can be either a list of specific chemical formulae or a range of molecular weights. Molecules are deleted every *Nfreq* timesteps, and bond connectivity is determined using the *Nevery* and *Nrepeat* keywords. The *filedel* argument is the name of the output file that records the species that are removed from the system (see below for output file format options). The *specieslist* keyword permits specific chemical species to be deleted. The *Nspecies* argument specifies how many species are eligible for deletion and is followed by a list of chemical formulae, whose strings are compared to species identified by this fix. For example, "specieslist 2 CO CO2" deletes molecules that are identified as "CO" and "CO2" in the species output file. The *masslimit* keyword permits deletion of molecules with molecular weights between *massmin* and *massmax*. The *specieslist* and *masslimit* keywords cannot both be used in the same *reaxff/species* fix.

The *delete_rate_limit* keyword can enforce an upper limit on the overall rate of molecule deletion. The number of deletion occurrences is limited to Nlimit within an interval of Nsteps timesteps. Nlimit can be specified with an equal-style *variable*. When using the *delete_rate_limit* keyword, no deletions are permitted to occur within the first Nsteps timesteps of the first run (after reading either a data or restart file).

Added in version 10Dec2025.

The *delete_subgroup* keyword enforces a requirement that deleted molecules must have at least one atom in the group specified by the keyword's *group-ID* argument. For example, this keyword can be used to delete an entire molecule when it is partially inside a *region*, as long as the molecule is fully contained with the overall fix group.

The *delete* keyword can output information about the deleted molecules in either the legacy format or in JSON format. The latter is activated when the *filedel* argument has a '.json' extension. The legacy format lists how many of each species are deleted, while the JSON format provides the atom ID, atom type, and coordinates of deleted atoms within each molecule. The format for legacy output changes depending on the keyword used. When using the *specieslist* keyword and legacy format, the *filedel* file has the following format: the first line lists the chemical formulae eligible for deletion, and each additional line contains the timestep on which a molecule deletion occurs and the number of each species deleted on that timestep. When using the *masslimit* keyword and the legacy format, each line of the *filedel* file contains the timestep on which deletions occurs, followed by how many of each species are deleted (with quantities preceding chemical formulae). The JSON format is the same regardless of the keyword, and lists deleted molecules in the style of the *JSON molecule file*, where more discussion of JSON schema can be found. Here is an example of a JSON output file from a simulation during which one water molecule was deleted on the first timestep:

```
{
    "application": "LAMMPS",
    "units": "real",
    "format": "dump",
    "style": "molecules",
    "revision": 1,
    "title": "fix reaxff/species: delete keyword",
    "timesteps": [
        {
            "timestep": 1,
            "molecules": [
                {
                    "types": {
                        "format": ["atom-id", "type"],
```

(continues on next page)

(continued from previous page)

```
                        "data": [
                            [1368, "H"],
                            [1366, "O"],
                            [1367, "H"]
                        ]
                    },
                    "coords": {
                        "format": ["atom-id", "x", "y", "z"],
                        "data": [
                            [1368, 26.787767440427466, 29.785528640296768, 25.
↪85197353660144],
                            [1366, 26.641801222582824, 29.868106247702887, 24.
↪91285138212243],
                            [1367, 25.69611192416744, 30.093425787807448, 24.
↪914380215672846]
                        ]
                    }
                }
            ]
        }
    ]
}
```

The required first-level keys of the JSON format output are "application", "format", "style", "revision", and "timesteps", and optional keys are "units" and "title". The value of the "timesteps" key is an array of objects that contain data for each timestep on which a molecule was deleted, and the other first-level keys identify this JSON schema. The objects in "timesteps" contains two keys, "timestep" and "molecules". The "molecules" key is an array of *LAMMPS molecule JSON* objects, one for each deleted molecule. The "format" keys within molecule JSON objects are only listed once per output file, for brevity. The "atom-id" values are atom IDs from the simulation, and the "type" values are atom types. In the above example, the types were reported as strings corresponding to elements using *type labels*.

---

The *Nevery*, *Nrepeat*, and *Nfreq* arguments specify on what timesteps the bond-order values are sampled to get the average bond order. The species analysis is performed using the average bond-order on timesteps that are a multiple of *Nfreq*. The average is over *Nrepeat* bond-order samples, computed in the preceding portion of the simulation every *Nevery* timesteps. *Nfreq* must be a multiple of *Nevery* and *Nevery* must be non-zero even if *Nrepeat* is 1. Also, the timesteps contributing to the average bond-order cannot overlap, i.e. Nrepeat*Nevery can not exceed Nfreq.

For example, if Nevery=2, Nrepeat=6, and Nfreq=100, then bond-order values on timesteps 90,92,94,96,98,100 will be used to compute the average bond-order for the species analysis output on timestep 100.

---

### 2.208.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix computes both a global vector of length 2 and a per-atom vector, either of which can be accessed by various *output commands*. The values in the global vector are "intensive".

The 2 values in the global vector are as follows:

1. total number of molecules
2. total number of distinct species

The per-atom vector stores the molecule ID for each atom as identified by the fix. If an atom is not in a molecule, its ID will be 0. For atoms in the same molecule, the molecule ID for all of them will be the same, and molecule IDs will range from 1 to the number of molecules.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

This fix supports dynamic groups only if the *Nrepeat* setting is 1, i.e. there is no averaging.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.208.5 Restrictions

The "fix reaxff/species" requires that *pair_style reaxff* is used. This fix is part of the REAXFF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

To write compressed species files, you must compile LAMMPS with the `-DLAMMPS_GZIP` option. See the *Build settings* doc page for details.

### 2.208.6 Related commands

*pair_style reaxff*, *fix reaxff/bonds*

### 2.208.7 Default

The default values for bond-order cutoffs are 0.3 for all I-J pairs. The default element symbols are taken from the ReaxFF pair_coeff command. Position files are not written by default.

**(Gissinger)** Jacob R. Gissinger, Scott R. Zavada, Joseph G. Smith, Josh Kemppainen, Ivan Gallegos, Gregory M. Odegard, Emilie J. Siochi, and Kristopher E. Wise, Carbon, 202, 336-347 (2023).

## 2.209 fix recenter command

Accelerator Variants: *recenter/kk*

### 2.209.1 Syntax

```
fix ID group-ID recenter x y z keyword value ...
```

- ID, group-ID are documented in *fix* command
- recenter = style name of this fix command
- x,y,z = constrain center-of-mass to these coords (distance units), any coord can also be NULL or INIT (see below)

- zero or more keyword/value pairs may be appended
- keyword = *shift* or *units*

```
shift value = group-ID
  group-ID = group of atoms whose coords are shifted
units value = box or lattice or fraction
```

## 2.209.2 Examples

```
fix 1 all recenter 0.0 0.5 0.0
fix 1 all recenter INIT INIT NULL
fix 1 all recenter INIT 0.0 0.0 units box
```

## 2.209.3 Description

Constrain the center-of-mass position of a group of atoms by adjusting the coordinates of the atoms every timestep. This is simply a small shift that does not alter the dynamics of the system or change the relative coordinates of any pair of atoms in the group. This can be used to ensure the entire collection of atoms (or a portion of them) do not drift during the simulation due to random perturbations (e.g. *fix langevin* thermostatting).

Distance units for the x,y,z values are determined by the setting of the *units* keyword, as discussed below. One or more x,y,z values can also be specified as NULL, which means exclude that dimension from this operation. Or it can be specified as INIT which means to constrain the center-of-mass to its initial value at the beginning of the run.

The center-of-mass (COM) is computed for the group specified by the fix. If the current COM is different than the specified x,y,z, then a group of atoms has their coordinates shifted by the difference. By default the shifted group is also the group specified by the fix. A different group can be shifted by using the *shift* keyword. For example, the COM could be computed on a protein to keep it in the center of the simulation box. But the entire system (protein + water) could be shifted.

If the *units* keyword is set to *box*, then the distance units of x,y,z are defined by the *units* command - e.g. Angstroms for *real* units. A *lattice* value means the distance units are in lattice spacings. The *lattice* command must have been previously used to define the lattice spacing. A *fraction* value means a fractional distance between the lo/hi box boundaries, e.g. 0.5 = middle of the box. The default is to use lattice units.

Note that the *velocity* command can be used to create velocities with zero aggregate linear and/or angular momentum.

**Note**

This fix performs its operations at the same point in the timestep as other time integration fixes, such as *fix nve*, *fix nvt*, or *fix npt*. Thus fix recenter should normally be the last such fix specified in the input script, since the adjustments it makes to atom coordinates should come after the changes made by time integration. LAMMPS will warn you if your fixes are not ordered this way.

**Note**

If you use this fix on a small group of atoms (e.g. a molecule in solvent) without using the *shift* keyword to adjust the positions of all atoms in the system, then the results can be unpredictable. For example, if the molecule is pushed consistently in one direction by a flowing solvent, its velocity will increase. But its coordinates will be re-centered, meaning it is moved back towards the force. Thus over time, the velocity and effective temperature of the molecule could become very large, though it won't actually be moving due to the re-centering. If you are thermostatting the entire system, then the solvent would be cooled to compensate. A better solution for this simulation scenario is to use the *fix spring* command to tether the molecule in place.

### 2.209.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the distance the group is moved by fix recenter.

This fix also computes global 3-vector which can be accessed by various *output commands*. The 3 quantities in the vector are xyz components of displacement applied to the group of atoms by the fix.

The scalar and vector values calculated by this fix are "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.209.5 Restrictions

This fix should not be used with an x,y,z setting that causes a large shift in the system on the first timestep, due to the requested COM being very different from the initial COM. This could cause atoms to be lost, especially in parallel. Instead, use the *displace_atoms* command, which can be used to move atoms a large distance.

### 2.209.6 Related commands

*fix momentum*, *velocity*

### 2.209.7 Default

The option defaults are shift = fix group-ID, and units = lattice.

## 2.210 fix restrain command

### 2.210.1 Syntax

```
fix ID group-ID restrain keyword args ...
```

- ID, group-ID are documented in *fix* command
- restrain = style name of this fix command
- one or more keyword/arg pairs may be appended
- keyword = *bond* or *lbound* or *angle* or *dihedral*

```
bond args = atom1 atom2 Kstart Kstop r0start (r0stop)
  atom1,atom2 = IDs of two atoms in bond
  Kstart,Kstop = restraint coefficients at start/end of run (energy units)
  r0start = equilibrium bond distance at start of run (distance units)
  r0stop = equilibrium bond distance at end of run (optional) (distance units). If␣
↪not
    specified it is assumed to be equal to r0start
lbound args = atom1 atom2 Kstart Kstop r0start (r0stop)
  atom1,atom2 = IDs of two atoms in bond
  Kstart,Kstop = restraint coefficients at start/end of run (energy units)
  r0start = equilibrium bond distance at start of run (distance units)
  r0stop = equilibrium bond distance at end of run (optional) (distance units). If␣
↪not
    specified it is assumed to be equal to r0start
angle args = atom1 atom2 atom3 Kstart Kstop theta0
  atom1,atom2,atom3 = IDs of three atoms in angle, atom2 = middle atom
  Kstart,Kstop = restraint coefficients at start/end of run (energy units)
  theta0 = equilibrium angle theta (degrees)
dihedral args = atom1 atom2 atom3 atom4 Kstart Kstop phi0 keyword/value
  atom1,atom2,atom3,atom4 = IDs of 4 atoms in dihedral in linear order
  Kstart,Kstop = restraint coefficients at start/end of run (energy units)
  phi0 = equilibrium dihedral angle phi (degrees)
  keyword/value = optional keyword value pairs. supported keyword/value pairs:
    mult n = dihedral multiplicity n (integer >= 0, default = 1)
```

### 2.210.2 Examples

```
fix holdem all restrain bond 45 48 2000.0 2000.0 2.75
fix holdem all restrain lbound 45 48 2000.0 2000.0 2.75
fix holdem all restrain dihedral 1 2 3 4 2000.0 2000.0 120.0
fix holdem all restrain bond 45 48 2000.0 2000.0 2.75 dihedral 1 2 3 4 2000.0 2000.0 120.
↪0
fix texas_holdem all restrain dihedral 1 2 3 4 0.0 2000.0 120.0 dihedral 1 2 3 5 0.0␣
↪2000.0 -120.0 dihedral 1 2 3 6 0.0 2000.0 0.0
```

### 2.210.3 Description

Restrain the motion of the specified sets of atoms by making them part of a bond or angle or dihedral interaction whose strength can vary over time during a simulation. This is functionally similar to creating a bond or angle or dihedral for the same atoms in a data file, as specified by the *read_data* command, albeit with a time-varying prefactor coefficient, and except for exclusion rules, as explained below.

For the purpose of force field parameter-fitting or mapping a molecular potential energy surface, this fix reduces the hassle and risk associated with modifying data files. In other words, use this fix to temporarily force a molecule to adopt a particular conformation. To create a permanent bond or angle or dihedral, you should modify the data file.

**Note**

Adding a bond/angle/dihedral with this command does not apply the exclusion rules and weighting factors specified by the *special_bonds* command to atoms in the restraint that are now bonded (1-2,1-3,1-4 neighbors) as a result. If they are close enough to interact in a *pair_style* sense (non-bonded interaction), then the bond/angle/dihedral restraint interaction will simply be superposed on top of that interaction.

The group-ID specified by this fix is ignored.

The second example above applies a restraint to hold the dihedral angle formed by atoms 1, 2, 3, and 4 near 120 degrees using a constant restraint coefficient. The fourth example applies similar restraints to multiple dihedral angles using a restraint coefficient that increases from 0.0 to 2000.0 over the course of the run.

**Note**

Adding a force to atoms implies a change in their potential energy as they move due to the applied force field. For dynamics via the *run* command, this energy can be added to the system's potential energy for thermodynamic output (see below). For energy minimization via the *minimize* command, this energy must be added to the system's potential energy to formulate a self-consistent minimization problem (see below).

In order for a restraint to be effective, the restraint force must typically be significantly larger than the forces associated with conventional force field terms. If the restraint is applied during a dynamics run (as opposed to during an energy minimization), a large restraint coefficient can significantly reduce the stable timestep size, especially if the atoms are initially far from the preferred conformation. You may need to experiment to determine what value of $K$ works best for a given application.

For the case of finding a minimum energy structure for a single molecule with particular restraints (e.g. for fitting force field parameters or constructing a potential energy surface), commands such as the following may be useful:

```
# minimize molecule energy with restraints
velocity all create 600.0 8675309 mom yes rot yes dist gaussian
fix NVE all nve
fix TFIX all langevin 600.0 0.0 100 24601
fix REST all restrain dihedral 2 1 3 8 0.0 5000.0 ${angle1} dihedral 3 1 2 9 0.0 5000.0 $
↪{angle2}
fix_modify REST energy yes
run 10000
fix TFIX all langevin 0.0 0.0 100 24601
fix REST all restrain dihedral 2 1 3 8 5000.0 5000.0 ${angle1} dihedral 3 1 2 9 5000.0␣
↪5000.0 ${angle2}
fix_modify REST energy yes
run 10000
# sanity check for convergence
minimize 1e-6 1e-9 1000 100000
# report unrestrained energies
unfix REST
run 0
```

---

The *bond* keyword applies a bond restraint to the specified atoms using the same functional form used by the *bond_style harmonic* command. The potential associated with the restraint is

$$E = K(r - r_0)^2$$

with the following coefficients:

- $K$ (energy/distance^2)
- $r_0$ (distance)

$K$ and $r_0$ are specified with the fix. Note that the usual 1/2 factor is included in $K$.

---

The *lbound* keyword applies a lower bound bond restraint to the specified atoms using the same functional form used by the *bond_style harmonic* command if the distance between the atoms is smaller than the equilibrium bond distance and 0 otherwise. The potential associated with the restraint is

$$E = K(r - r_0)^2, if\ r < r_0$$

$$E = 0 \qquad\qquad , if\ r \geq r_0$$

with the following coefficients:

- $K$ (energy/distance^2)
- $r_0$ (distance)

$K$ and $r_0$ are specified with the fix. Note that the usual 1/2 factor is included in $K$.

---

The *angle* keyword applies an angle restraint to the specified atoms using the same functional form used by the *angle_style harmonic* command. The potential associated with the restraint is

$$E = K(\theta - \theta_0)^2$$

with the following coefficients:

- $K$ (energy)
- $\theta_0$ (degrees)

$K$ and $\theta_0$ are specified with the fix. $\theta_0$ is specified in degrees, but LAMMPS converts it to radians internally; hence $K$ is effectively energy per radian^2. Note that the usual 1/2 factor is included in $K$.

---

The *dihedral* keyword applies a dihedral restraint to the specified atoms using a simplified form of the function used by the *dihedral_style charmm* command. The potential associated with the restraint is

$$E = K[1 + \cos(n\phi - d)]$$

with the following coefficients:

- $K$ (energy)
- $n$ (multiplicity, >= 0)
- $d$ (degrees) = $\phi_0$ + 180

$K$ and $\phi_0$ are specified with the fix. Note that the value of the dihedral multiplicity $n$ is set by default to 1. You can use the optional *mult* keyword to set it to a different positive integer. Also note that the energy will be a minimum when the current dihedral angle $\phi$ is equal to $\phi_0$.

---

### 2.210.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify energy* option is supported by this fix to add the potential energy associated with this fix to the global potential energy of the system as part of *thermodynamic output* The default setting for this fix is *fix_modify energy no*.

The *fix_modify respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is adding its forces. Default is the outermost level.

**Note**

If you want the fictitious potential energy associated with the added forces to be included in the total potential energy of the system (the quantity being minimized), you MUST enable the *fix_modify energy* option for this fix.

This fix computes a global scalar and a global vector of length 3, which can be accessed by various *output commands*. The scalar is the total potential energy for *all* the restraints as discussed above. The vector values are the sum of contributions to the following individual categories:

1. bond energy
2. angle energy
3. dihedral energy

The scalar and vector values calculated by this fix are "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

### 2.210.5 Restrictions

none

### 2.210.6 Related commands

none

### 2.210.7 Default

none

## 2.211 fix rheo command

### 2.211.1 Syntax

```
fix ID group-ID rheo cut kstyle zmin keyword values...
```

- ID, group-ID are documented in *fix* command
- rheo = style name of this fix command
- cut = cutoff for the kernel (distance)
- kstyle = *quintic* or *RK0* or *RK1* or *RK2*
- zmin = minimal number of neighbors for reproducing kernels
- zero or more keyword/value pairs may be appended to args
- keyword = *thermal* or *interface/reconstruct* or *surface/detection* or *shift* or *rho/sum* or *density* or *speed/sound*

```
thermal turns on thermal evolution
  values = none
interface/reconstruct reconstructs interfaces with solid particles
  values = none
surface/detection detects free-surfaces with an absence of particles
  values = sdstyle limit limit/splash
    sdstyle = coordination or divergence
    limit = threshold for surface particles
```

```
    limit/splash = threshold for splash particles (unitless)
  shift turns on velocity shifting
    values = none
    optional args = exclude/type or scale/cross/type
      exclude/type values = types
        types = list of types
      scale/cross/type values = shiftscale cmin rmin
        shiftscale = fraction of shifting in normal direction to preserve (unitless)
        cmin = minimum color function value required for scaling (unitless)
        rmin = minimum local same-type weighted distance required for any shifting␣
↪(unitless)
  rho/sum density evolution performed by a kernel summation
    values = none
    optional args = self/mass
      self/mass values = none, a particle uses its own mass in summation
  density specify equilibrium densities for each atom type
    values = rho01, ... rho0N (density)
  speed/sound specify speeds of sound for each atom type
    values = cs0, ... csN (velocity)
```

### 2.211.2 Examples

```
fix 1 all rheo 3.0 quintic 0 thermal density 0.1 0.1 speed/sound 10.0 1.0
fix 1 all rheo 3.0 RK1 10 shift surface/detection coordination 40
fix 1 all rheo 3.0 RK1 10 shift exclude/type 2*4 scale/cross/type 0.05 0.02 0.5
fix 1 all rheo 3.0 RK1 10 rhosum self/mass
```

### 2.211.3 Description

Added in version 29Aug2024.

Perform time integration for RHEO particles, updating positions, velocities, and densities. For a detailed breakdown of the integration timestep and numerical details, see *(Palermo)*. For an overview and list of other features available in the RHEO package, see *the RHEO howto*.

The type of kernel is specified using *kstyle* and the cutoff is *cut*. Four kernels are currently available. The *quintic* kernel is a standard quintic spline function commonly used in SPH. The other options, *RK0*, *RK1*, and *RK2*, are zeroth, first, and second order reproducing. To generate a reproducing kernel, a particle must have sufficient neighbors inside the kernel cutoff distance (a coordination number) to accurately calculate moments. This threshold is set by *zmin*. If reproducing kernels are requested but a particle has fewer neighbors, then it will revert to a non-reproducing quintic kernel until it gains more neighbors.

To model temperature evolution, one must specify the *thermal* keyword, define a separate instance of *fix rheo/thermal*, and use atom style rheo/thermal.

By default, the density of solid RHEO particles does not evolve and forces with fluid particles are calculated using the current velocity of the solid particle. If the *interface/reconstruct* keyword is used, then the density and velocity of solid particles are alternatively reconstructed for every fluid-solid interaction to ensure no-slip and pressure-balanced boundaries. This is done by estimating the location of the fluid-solid interface and extrapolating fluid particle properties across the interface to calculate a temporary apparent density and velocity for a solid particle. The numerical details are the same as those described in *(Palermo)* except there is an additional restriction that the reconstructed solid density cannot be less than the equilibrium density. This prevents fluid particles from sticking to solid surfaces.

A modified form of Fickian particle shifting can be enabled with the *shift* keyword. This effectively shifts particle positions to generate a more uniform spatial distribution. By default, shifting does not consider the type of a particle and therefore may be inappropriate in systems consisting of multiple atom types representing multiple fluid phases.

However, two optional sub-arguments can follow the *shift* keyword, *exclude/type* and *scale/cross/type* to adjust shifting at fluid interfaces.

The *exclude/type* option lets the user specify a list of atom types which are not shifted, *types*. A wild-card asterisk can be used in place of or in conjunction with the *types* argument to toggle shifting for multiple atom types. This takes the form "*" or "*n" or "m*" or "m*n". If *N* is the number of atom types, then an asterisk with no numeric values means all types from 1 to *N*. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from m to *N* (inclusive). A middle asterisk means all types from m to n (inclusive).

The *scale/cross/type* option is designed to handle interfaces between fluids made up of different atom types. Similar to the method by *(Yang)*, a color function is calculated and used to estimate a local interfacial normal vector. Shifting along this normal direction is rescaled by a factor of *scaleshift*, such that a value of *scaleshift* of zero implies there is no shifting in the normal direction and a value of *scaleshift* of one implies no change in behavior. This scaling is only applied to atoms with a color function value greater than *cmin*. To handle scenarios of a small inclusion of one fluid type (e.g. a single atom) inside another, a weighted distance of same-type support is calculated

$$R_{i,\text{same}} = \sum_j r_{ij} W_{ij} \delta_{ij}$$

where $\delta_{ij}$ is zero if atoms $i$ and $j$ have different types but unity otherwise. If $R_{i,\text{same}}$ is ever less than the specified value of *rmin*, shifting is turned off for particle $i$

In systems with free surfaces (atom-vacuum), the *surface/detection* keyword can classify the location of particles as being within the bulk fluid, on a free surface, or isolated from other particles in a splash or droplet. Shifting is then disabled in the normal direction away from the free surface to prevent particles from diffusing away. Surface detection can also be used to control surface-nucleated effects like oxidation when used in combination with *fix rheo/oxidation*. Surface detection is not performed on solid bodies.

The *surface/detection* keyword takes three arguments: *sdstyle*, *limit*, and *limit/splash*. The first, *sdstyle*, specifies whether surface particles are identified using a coordination number (*coordination*) or the divergence of the local particle positions (*divergence*). The threshold value for a surface particle for either of these criteria is set by the numerical value of *limit*. Additionally, if a particle's coordination number is too low, i.e. if it has separated off from the bulk in a droplet, it is not possible to define surfaces and the particle is classified as a splash. The coordination threshold for this classification is set by the numerical value of *limit/splash*.

By default, RHEO integrates particles' densities using a mass diffusion equation. Alternatively, one can update densities every timestep by performing a kernel summation of the masses of neighboring particles by specifying the *rho/sum* keyword. Following this keyword, one may include the optional *self/mass* sub-argument which modifies the behavior of the density summation. Typically, the density $\rho$ of a particle is calculated as the sum over neighbors

$$\rho_i = \sum_j W_{ij} M_j$$

where $W_{ij}$ is the kernel, and $M_j$ is the mass of particle $j$. The *self/mass* keyword augments this expression by replacing $M_j$ with $M_i$. This may be useful in simulations of multiple fluid phases with large differences in density, *(Hu)*.

The *density* keyword is used to specify the equilibrium density of each of the N particle types. It must be followed by N numerical values specifying each type's equilibrium density *rho0*.

The *speed/sound* keyword is used to specify the speed of sound of each of the N particle types. It must be followed by N numerical values specifying each type's speed of sound *cs*. These values may be ignored if the pressure equation of state has a non-constant speed of sound, as discussed further in *fix rheo/pressure*.

### 2.211.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.211.5 Restrictions

This fix must be used with atom style rheo or rheo/thermal. This fix must be used in conjunction with *fix rheo/pressure*. and *fix rheo/viscosity*. If the *thermal* setting is used, there must also be an instance of *fix rheo/thermal*. The fix group must be set to all. Only one instance of fix rheo may be defined and it must be defined prior to all other RHEO fixes in the input script.

This fix is part of the RHEO package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.211.6 Related commands

*fix rheo/viscosity*, *fix rheo/pressure*, *fix rheo/thermal*, *pair rheo*, *compute rheo/property/atom*

### 2.211.7 Default

*rho0* and *cs* are set to 1.0 for all atom types.

---

**(Palermo)** Palermo, Wolf, Clemmer, O'Connor, Phys. Fluids, 36, 113337 (2024).

**(Yang)** Yang, Rakhsha, Hu, Negrut, J. Comp. Physics, 458, 111079 (2022).

**(Hu)** Hu, and Adams, J. Comp. Physics, 213, 844-861 (2006).

## 2.212 fix rheo/oxidation command

### 2.212.1 Syntax

```
fix ID group-ID rheo/oxidation cut btype rsurf
```

- ID, group-ID are documented in *fix* command
- rheo/oxidation = style name of this fix command
- cut = maximum bond length (distance units)
- btype = type of bonds created
- rsurf = distance from surface to create bonds (distance units)

### 2.212.2 Examples

```
fix 1 all rheo/oxidation 1.5 2 0.0
fix 1 all rheo/oxidation 1.0 1 2.0
```

### 2.212.3 Description

Added in version 29Aug2024.

This fix dynamically creates bonds on the surface of fluids to represent physical processes such as oxidation. It is intended for use with bond style *bond rheo/shell*.

Every timestep, particles check neighbors within a distance of *cut*. This distance must be smaller than the kernel length defined in *fix rheo*. Bonds of type *btype* are created between a fluid particle and either a fluid or solid neighbor. The fluid particles must also be on the fluid surface, or within a distance of *rsurf* from the surface. This process is further described in *(Clemmer)*.

If used in conjunction with solid bodies, such as those generated by the *react* option of *fix rheo/thermal*, it is recommended to use a *hybrid bond style* with different bond types for solid and oxide bonds.

### 2.212.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.212.5 Restrictions

This fix must be used with the bond style *rheo/shell* and *fix rheo* with surface detection enabled.

This fix is part of the RHEO package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.212.6 Related commands

*fix rheo*, *bond rheo/shell*, *compute rheo/property/atom*

### 2.212.7 Default

none

---

**(Clemmer)** Clemmer, Pierce, O'Connor, Nevins, Jones, Lechman, Tencer, Appl. Math. Model., 130, 310-326 (2024).

## 2.213 fix rheo/pressure command

### 2.213.1 Syntax

```
fix ID group-ID rheo/pressure type1 pstyle1 args1 ... typeN pstyleN argsN
```

- ID, group-ID are documented in *fix* command
- rheo/pressure = style name of this fix command
- one or more types and pressure styles must be appended
- types = lists of types (see below)
- pstyle = *linear* or *tait/water* or *tait/general* or *cubic* or *ideal/gas* or *background*

  *linear* args = none
  *tait/water* args = none
  *tait/general* args = exponent *gamma* (unitless)
  *cubic* args = cubic prefactor $A_3$ (pressure/density^2)
  *ideal/gas* args = heat capacity ratio *gamma* (unitless)
  *background* args = background pressure $P[b]$ (pressure)

### 2.213.2 Examples

```
fix 1 all rheo/pressure * linear
fix 1 all rheo/pressure 1 linear 2 cubic 10.0
fix 1 all rheo/pressure * linear * background 0.1
```

### 2.213.3 Description

Added in version 29Aug2024.

This fix defines a pressure equation of state for RHEO particles. One can define different equations of state for different atom types. An equation must be specified for every atom type.

One first defines the atom *types*. A wild-card asterisk can be used in place of or in conjunction with the *types* argument to set values for multiple atom types. This takes the form "*" or "*n" or "m*" or "m*n". If $N$ is the number of atom types, then an asterisk with no numeric values means all types from 1 to $N$. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from m to $N$ (inclusive). A middle asterisk means all types from m to n (inclusive).

The *types* definition is followed by the pressure style, *pstyle*. Current options *linear*, *taitwater*, and *cubic*. Style *linear* is a linear equation of state with a particle pressure $P$ calculated as

$$P = c^2(\rho - \rho_0)$$

where $c$ is the speed of sound, $\rho_0$ is the equilibrium density, and $\rho$ is the current density of a particle. The numerical values of $c$ and $\rho_0$ are set in *fix rheo*. Style *cubic* is a cubic equation of state which has an extra argument $A_3$,

$$P = c^2((\rho - \rho_0) + A_3(\rho - \rho_0)^3).$$

Style *tait/water* is Tait's equation of state:

$$P = \frac{c^2\rho_0}{7}\left[\left(\frac{\rho}{\rho_0}\right)^7 - 1\right].$$

Style *tait/general* generalizes this equation of state

$$P = \frac{c^2\rho_0}{\gamma}\left[\left(\frac{\rho}{\rho_0}\right)^\gamma - 1\right]$$

where $\gamma$ is an exponent.

Style *ideal/gas* is the ideal gas equation of state

$$P = (\gamma - 1)\rho e$$

where $\gamma$ is the heat capacity ratio and $e$ is the internal energy of a particle per unit mass. This style is only compatible with atom style rheo/thermal. Note that when using this style, the speed of sound is no longer constant such that the value of $c$ specified in *fix rheo* is not used.

The *background* style acts differently than the rest as it only adds a constant background pressure shift $P[b]$ to all atoms of the designated types. Therefore, this style must be used in conjunction with another style that specifies an equation of state.

### 2.213.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.213.5 Restrictions

This fix must be used with an atom style that includes density such as atom_style rheo or rheo/thermal. This fix must be used in conjunction with *fix rheo*. The fix group must be set to all. Only one instance of fix rheo/pressure can be defined.

This fix is part of the RHEO package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.213.6 Related commands

*fix rheo*, *pair rheo*, *compute rheo/property/atom*

### 2.213.7 Default

none

## 2.214 fix rheo/thermal command

### 2.214.1 Syntax

```
fix ID group-ID rheo/thermal attribute values ...
```

- ID, group-ID are documented in *fix* command
- rheo/thermal = style name of this fix command
- one or more attributes may be appended
- attribute = *conductivity* or *specific/heat* or *latent/heat* or *Tfreeze* or *react*

```
conductivity args = types style args
  types = lists of types (see below)
  style = constant
    constant arg = conductivity (power/(length*temperature))
specific/heat args = types style args
  types = lists of types (see below)
  style = constant
    constant arg = specific heat (energy/(mass*temperature))
latent/heat args = types style args
  types = lists of types (see below)
  style = constant
    constant arg = latent heat (energy/mass)
Tfreeze args = types style args
  types = lists of types (see below)
  style = constant
    constant arg = freezing temperature (temperature)
react args = cut type
  cut = maximum bond distance
  type = bond type
```

### 2.214.2 Examples

```
fix 1 all rheo/thermal conductivity * constant 1.0 specific/heat * constant 1.0 Tfreeze␣
↪* constant 1.0
fix 1 all rheo/pressure conductivity 1*2 constant 1.0 conductivity 3*4 constant 2.0␣
↪specific/heat * constant 1.0
```

### 2.214.3 Description

Added in version 29Aug2024.

This fix performs time integration of temperature for atom style rheo/thermal. In addition, it defines multiple thermal properties of particles and handles melting/solidification, if applicable. For more details on phase transitions in RHEO, see *the RHEO howto*.

Note that the temperature of a particle is always derived from the energy. This implies the *temperature* attribute of *the set command* does not affect particles. Instead, one should use the *sph/e* attribute.

For each atom type, one can define expressions for the *conductivity*, *specific/heat*, *latent/heat*, and critical temperature (*Tfreeze*). The conductivity and specific heat must be defined for all atom types. The latent heat and critical temperature are optional. However, a critical temperature must be defined to specify a latent heat.

Note, if shifting is turned on in *fix rheo*, the gradient of the energy is used to shift energies. This may be inappropriate in systems with multiple atom types with different specific heats.

For each property, one must first define a list of atom types. A wild-card asterisk can be used in place of or in conjunction with the *types* argument to set values for multiple atom types. This takes the form "*" or "*n" or "m*" or "m*n". If *N* is the number of atom types, then an asterisk with no numeric values means all types from 1 to *N*. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from m to *N* (inclusive). A middle asterisk means all types from m to n (inclusive).

The *types* definition for each property is followed by the style. Currently, the only option is *constant*. Style *constant* simply applies a constant value of respective property to each particle of the assigned type.

The *react* keyword controls whether bonds are created/deleted when particles transition between a fluid and solid state. This option only applies to atom types that have a defined value of *Tfreeze*. When a fluid particle's temperature drops below *Tfreeze*, bonds of type *btype* are created between nearby solid particles within a distance of *cut*. The particle's status also swaps to a solid state. When a solid particle's temperature rises above *Tfreeze*, all bonds of type *btype* are broken and the particle's status swaps to a fluid state.

### 2.214.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.214.5 Restrictions

This fix must be used with an atom style that includes temperature, heatflow, and conductivity such as atom_style rheo/thermal This fix must be used in conjunction with *fix rheo* with the *thermal* setting. The fix group must be set to all. Only one instance of fix rheo/pressure can be defined.

This fix is part of the RHEO package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.214.6 Related commands

*fix rheo*, *pair rheo*, *compute rheo/property/atom*, *fix add/heat*

### 2.214.7 Default

none

## 2.215 fix rheo/viscosity command

### 2.215.1 Syntax

```
fix ID group-ID rheo/viscosity type1 pstyle1 args1 ... typeN pstyleN argsN
```

- ID, group-ID are documented in *fix* command
- rheo/viscosity = style name of this fix command

- one or more types and viscosity styles must be appended
- types = lists of types (see below)
- vstyle = *constant* or *power*

```
constant args = eta
  eta = viscosity

power args = eta, gd0, K, n
  eta = viscosity
  gd0 = critical strain rate
  K = consistency index
  n = power-law exponent
```

## 2.215.2 Examples

```
fix 1 all rheo/viscosity * constant 1.0
fix 1 all rheo/viscosity 1 constant 1.0 2 power 0.1 5e-4 0.001 0.5
```

## 2.215.3 Description

Added in version 29Aug2024.

This fix defines a viscosity for RHEO particles. One can define different viscosities for different atom types, but a viscosity must be specified for every atom type.

One first defines the atom *types*. A wild-card asterisk can be used in place of or in conjunction with the *types* argument to set values for multiple atom types. This takes the form "*" or "*n" or "m*" or "m*n". If $N$ is the number of atom types, then an asterisk with no numeric values means all types from 1 to $N$. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from m to $N$ (inclusive). A middle asterisk means all types from m to n (inclusive).

The *types* definition is followed by the viscosity style, *vstyle*. Two options are available, *constant* and *power*. Style *constant* simply applies a constant value of the viscosity *eta* to each particle of the assigned type. Style *power* is a Hershchel-Bulkley constitutive equation for the stress $\tau$

$$\tau = \left( \frac{\tau_0}{\dot{\gamma}} + K \dot{\gamma}^{n-1} \right) \dot{\gamma}, \tau \geq \tau_0$$

where $\dot{\gamma}$ is the strain rate and $\tau_0$ is the critical yield stress, below which $\dot{\gamma} = 0.0$. To avoid divergences, this expression is regularized by defining a critical strain rate *gd0*. If the local strain rate on a particle falls below this limit, a constant viscosity of *eta* is assigned. This implies a value of

$$\tau_0 = \eta \dot{\gamma}_0 - K \dot{\gamma}_0^N$$

## 2.215.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

## 2.215.5 Restrictions

This fix must be used with an atom style that includes viscosity such as atom_style rheo or rheo/thermal. This fix must be used in conjunction with *fix rheo*. The fix group must be set to all. Only one instance of fix rheo/viscosity can be defined.

This fix is part of the RHEO package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.215.6 Related commands

*fix rheo*, *pair rheo*, *compute rheo/property/atom*

### 2.215.7 Default

none

## 2.216 fix rhok command

### 2.216.1 Syntax

```
fix ID group-ID rhok nx ny nz K a
```

- ID, group-ID are documented in *fix* command
- nx, ny, nz = k-vector of collective density field
- K = spring constant of bias potential
- a = anchor point of bias potential

### 2.216.2 Examples

```
fix bias all rhok 16 0 0 4.0 16.0
fix 1 all npt temp 0.8 0.8 4.0 z 2.2 2.2 8.0
# output of 4 values from fix rhok: U_bias rho_k_RE  rho_k_IM  \|rho_k\|
thermo_style custom step temp pzz lz f_bias f_bias[1] f_bias[2] f_bias[3]
```

### 2.216.3 Description

The fix applies a force to atoms given by the potential

$$U = \frac{1}{2}K(|\rho_{\vec{k}}| - a)^2$$
$$\rho_{\vec{k}} = \sum_j^N \exp(-i\vec{k}\cdot\vec{r}_j)/\sqrt{N}$$
$$\vec{k} = (2\pi n_x/L_x, 2\pi n_y/L_y, 2\pi n_z/L_z)$$

as described in *(Pedersen)*.

This field, which biases configurations with long-range order, can be used to study crystal-liquid interfaces and determine melting temperatures *(Pedersen)*.

An example of using the interface pinning method is located in the *examples/PACKAGES/rhok* directory.

### 2.216.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify energy* option is supported by this fix to add the potential energy calculated by the fix to the global potential energy of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify energy no*.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the potential energy discussed in the preceding paragraph. The scalar stored by this fix is "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

This fix is not invoked during *energy minimization*.

### 2.216.5 Restrictions

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.216.6 Related commands

*thermo_style*

### 2.216.7 Default

none

---

**(Pedersen)** Pedersen, J. Chem. Phys., 139, 104102 (2013).

## 2.217 fix rigid command

Accelerator Variants: *rigid/omp*

## 2.218 fix rigid/nve command

Accelerator Variants: *rigid/nve/omp*

## 2.219 fix rigid/nvt command

Accelerator Variants: *rigid/nvt/omp*

## 2.220 fix rigid/npt command

Accelerator Variants: *rigid/npt/omp*

## 2.221 fix rigid/nph command

Accelerator Variants: *rigid/nph/omp*

## 2.222 fix rigid/small command

Accelerator Variants: *rigid/small/omp*

## 2.223 fix rigid/nve/small command

## 2.224 fix rigid/nvt/small command

## 2.225 fix rigid/npt/small command

## 2.226 fix rigid/nph/small command

### 2.226.1 Syntax

```
fix ID group-ID style bodystyle args keyword values ...
```

- ID, group-ID are documented in *fix* command
- style = *rigid* or *rigid/nve* or *rigid/nvt* or *rigid/npt* or *rigid/nph* or *rigid/small* or *rigid/nve/small* or *rigid/nvt/small* or *rigid/npt/small* or *rigid/nph/small*
- bodystyle = *single* or *molecule* or *group*

```
single args = none
molecule args = none
custom args = i_propname or v_varname
  i_propname = a custom integer vector defined via fix property/atom
  v_varname  = an atom-style or atomfile-style variable
group args = N groupID1 groupID2 ...
  N = # of groups
  groupID1, groupID2, ... = list of N group IDs
```

- zero or more keyword/value pairs may be appended
- keyword = *langevin* or *reinit* or *temp* or *mol* or *iso* or *aniso* or *x* or *y* or *z* or *couple* or *tparam* or *pchain* or *dilate* or *force* or *torque* or *infile* or *gravity*

```
langevin values = Tstart Tstop Tperiod seed
  Tstart,Tstop = desired temperature at start/stop of run (temperature units)
  Tdamp = temperature damping parameter (time units)
  seed = random number seed to use for white noise (positive integer)
reinit value = yes or no
temp values = Tstart Tstop Tdamp
  Tstart,Tstop = desired temperature at start/stop of run (temperature units)
  Tdamp = temperature damping parameter (time units)
mol value = template-ID
  template-ID = ID of molecule template specified in a separate molecule command
iso or aniso values = Pstart Pstop Pdamp
  Pstart,Pstop = scalar external pressure at start/end of run (pressure units)
  Pdamp = pressure damping parameter (time units)
x or y or z values = Pstart Pstop Pdamp
  Pstart,Pstop = external stress tensor component at start/end of run (pressure␣
↪units)
  Pdamp = stress damping parameter (time units)
couple value = none or xyz or xy or yz or xz
tparam values = Tchain Titer Torder
  Tchain = length of Nose/Hoover thermostat chain
  Titer = number of thermostat iterations performed
  Torder = 3 or 5 = Yoshida-Suzuki integration parameters
pchain values = Pchain
```

```
    Pchain = length of the Nose/Hoover thermostat chain coupled with the barostat
  dilate value = dilate-group-ID
    dilate-group-ID = only dilate atoms in this group due to barostat volume changes
  force values = M xflag yflag zflag
    M = which rigid body from 1-Nbody (see asterisk form below)
    xflag,yflag,zflag = off/on if component of center-of-mass force is active
  torque values = M xflag yflag zflag
    M = which rigid body from 1-Nbody (see asterisk form below)
    xflag,yflag,zflag = off/on if component of center-of-mass torque is active
  infile filename
    filename = file with per-body values of mass, center-of-mass, moments of inertia
  gravity values = gravity-ID
    gravity-ID = ID of fix gravity command to add gravitational forces
```

### 2.226.2 Examples

```
fix 1 clump rigid single reinit yes
fix 1 clump rigid/small molecule
fix 1 clump rigid single force 1 off off on langevin 1.0 1.0 1.0 428984
fix 1 polychains rigid/nvt molecule temp 1.0 1.0 5.0 reinit no
fix 1 polychains rigid molecule force 1*5 off off off force 6*10 off off on
fix 1 polychains rigid/small molecule langevin 1.0 1.0 1.0 428984
fix 2 fluid rigid group 3 clump1 clump2 clump3 torque * off off off
fix 1 rods rigid/npt molecule temp 300.0 300.0 100.0 iso 0.5 0.5 10.0
fix 1 particles rigid/npt molecule temp 1.0 1.0 5.0 x 0.5 0.5 1.0 z 0.5 0.5 1.0 couple xz
fix 1 water rigid/nph molecule iso 0.5 0.5 1.0
fix 1 particles rigid/npt/small molecule temp 1.0 1.0 1.0 iso 0.5 0.5 1.0

variable bodyid atom 1.0*gmask(clump1)+2.0*gmask(clump2)+3.0*gmask(clump3)
fix 1 clump rigid custom v_bodyid

variable bodyid atomfile bodies.txt
fix 1 clump rigid custom v_bodyid

fix 0 all property/atom i_bodyid
read_restart data.rigid fix 0 NULL Bodies
fix 1 clump rigid/small custom i_bodyid
```

### 2.226.3 Description

Treat one or more sets of atoms as independent rigid bodies. This means that each timestep the total force and torque on each rigid body is computed as the sum of the forces and torques on its constituent particles. The coordinates, velocities, and orientations of the atoms in each body are then updated so that the body moves and rotates as a single entity. This is implemented by creating internal data structures for each rigid body and performing time integration on these data structures. Positions, velocities, and orientations of the constituent particles are regenerated from the rigid body data structures in every time step. This restricts which operations and fixes can be applied to rigid bodies. See below for a detailed discussion.

Examples of large rigid bodies are a colloidal particle, or portions of a biomolecule such as a protein.

Example of small rigid bodies are patchy nanoparticles, such as those modeled in *this paper* by Sharon Glotzer's group, clumps of granular particles, lipid molecules consisting of one or more point dipoles connected to other spheroids or ellipsoids, irregular particles built from line segments (2d) or triangles (3d), and coarse-grain models of nano or colloidal particles consisting of a small number of constituent particles. Note that the *fix shake* command can also be

used to rigidify small molecules of 2, 3, or 4 atoms, e.g. water molecules. That fix treats the constituent atoms as point masses.

These fixes also update the positions and velocities of the atoms in each rigid body via time integration, in the NVE, NVT, NPT, or NPH ensemble, as described below.

There are two main variants of this fix, fix rigid and fix rigid/small. The NVE/NVT/NPT/NHT versions belong to one of the two variants, as their style names indicate.

**Note**

Not all of the *bodystyle* options and keyword/value options are available for both the *rigid* and *rigid/small* variants. See details below.

The *rigid* styles are typically the best choice for a system with a small number of large rigid bodies, each of which can extend across the domain of many processors. It operates by creating a single global list of rigid bodies, which all processors contribute to. MPI_Allreduce operations are performed each timestep to sum the contributions from each processor to the force and torque on all the bodies. This operation will not scale well in parallel if large numbers of rigid bodies are simulated.

The *rigid/small* styles are typically best for a system with a large number of small rigid bodies. Each body is assigned to the atom closest to the geometrical center of the body. The fix operates using local lists of rigid bodies owned by each processor and information is exchanged and summed via local communication between neighboring processors when ghost atom info is accumulated.

**Note**

To use the *rigid/small* styles the ghost atom cutoff must be large enough to span the distance between the atom that owns the body and every other atom in the body. This distance value is printed out when the rigid bodies are defined. If the *pair_style* cutoff plus neighbor skin does not span this distance, then you should use the *comm_modify cutoff* command with a setting epsilon larger than the distance.

Which of the two variants is faster for a particular problem is hard to predict. The best way to decide is to perform a short test run. Both variants should give identical numerical answers for short runs. Long runs should give statistically similar results, but round-off differences may accumulate to produce divergent trajectories.

**Note**

You should not update the atoms in rigid bodies via other time-integration fixes (e.g. *fix nve*, *fix nvt*, *fix npt*, *fix move*), or you will have conflicting updates to positions and velocities resulting in unphysical behavior in most cases. When performing a hybrid simulation with some atoms in rigid bodies, and some not, a separate time integration fix like *fix nve* or *fix nvt* should be used for the non-rigid particles.

**Note**

These fixes are overkill if you simply want to hold a collection of atoms stationary or have them move with a constant velocity. A simpler way to hold atoms stationary is to not include those atoms in your time integration fix. E.g. use "fix 1 mobile nve" instead of "fix 1 all nve", where "mobile" is the group of atoms that you want to move. You can move atoms with a constant velocity by assigning them an initial velocity (via the *velocity* command), setting the force on them to 0.0 (via the *fix setforce* command), and integrating them as usual (e.g. via the *fix nve* command).

**Warning**

The aggregate properties of each rigid body are calculated at the start of a simulation run and are maintained in internal data structures. The properties include the position and velocity of the center-of-mass of the body, its moments of inertia, and its angular momentum. This is done using the properties of the constituent atoms of the body at that point in time (or see the *infile* keyword option). Thereafter, changing these properties of individual atoms in the body will have no effect on a rigid body's dynamics, unless they effect any computation of per-atom forces or torques. If the keyword *reinit* is set to *yes* (the default), the rigid body data structures will be recreated at the beginning of each *run* command; if the keyword *reinit* is set to *no*, the rigid body data structures will be built only at the very first *run* command and maintained for as long as the rigid fix is defined. For example, you might think you could displace the atoms in a body or add a large velocity to each atom in a body to make it move in a desired direction before a second run is performed, using the *set* or *displace_atoms* or *velocity* commands. But these commands will not affect the internal attributes of the body unless *reinit* is set to *yes*. With *reinit* set to *no* (or using the *infile* option, which implies *reinit no*) the position and velocity of individual atoms in the body will be reset when time integration starts again.

---

Each rigid body must have two or more atoms. An atom can belong to at most one rigid body. Which atoms are in which bodies can be defined via several options.

**Note**

With the *rigid/small* styles, which require that *bodystyle* be specified as *molecule* or *custom*, you can define a system that has no rigid bodies initially. This is useful when you are using the *mol* keyword in conjunction with another fix that is adding rigid bodies on-the-fly as molecules, such as *fix deposit* or *fix pour*.

For bodystyle *single* the entire fix group of atoms is treated as one rigid body. This option is only allowed for the *rigid* styles.

For bodystyle *molecule*, atoms are grouped into rigid bodies by their respective molecule IDs: each set of atoms in the fix group with the same molecule ID is treated as a different rigid body. This option is allowed for both the *rigid* and *rigid/small* styles. Note that atoms with a molecule ID = 0 will be treated as a single rigid body. For a system with atomic solvent (typically this is atoms with molecule ID = 0) surrounding rigid bodies, this may not be what you want. Thus you should be careful to use a fix group that only includes atoms you want to be part of rigid bodies.

Bodystyle *custom* is similar to bodystyle *molecule* except that it is more flexible in using other per-atom properties to define the sets of atoms that form rigid bodies. A custom per-atom integer vector defined by the *fix property/atom* command can be used. Or an *atom-style or atomfile-style variable* can be used; the floating-point value produced by the variable is rounded to an integer. As with bodystyle *molecule*, each set of atoms in the fix groups with the same integer value is treated as a different rigid body. Since fix property/atom custom vectors and atom-style variables produce values for all atoms, you should be careful to use a fix group that only includes atoms you want to be part of rigid bodies.

**Note**

To compute the initial center-of-mass position and other properties of each rigid body, the image flags for each atom in the body are used to "unwrap" the atom coordinates. Thus you must ensure that these image flags are consistent so that the unwrapping creates a valid rigid body (one where the atoms are close together), particularly if the atoms in a single rigid body straddle a periodic boundary. This means the input data file or restart file must define the image flags for each atom consistently or that you have used the *set* command to specify them correctly. If a dimension is non-periodic then the image flag of each atom must be 0 in that dimension, else an error is generated.

The *force* and *torque* keywords discussed next are only allowed for the *rigid* styles.

By default, each rigid body is acted on by other atoms which induce an external force and torque on its center of mass, causing it to translate and rotate. Components of the external center-of-mass force and torque can be turned off by the *force* and *torque* keywords. This may be useful if you wish a body to rotate but not translate, or vice versa, or if you wish it to rotate or translate continuously unaffected by interactions with other particles. Note that if you expect a rigid body not to move or rotate by using these keywords, you must ensure its initial center-of-mass translational or angular velocity is 0.0. Otherwise the initial translational or angular momentum the body has will persist.

An xflag, yflag, or zflag set to *off* means turn off the component of force of torque in that dimension. A setting of *on* means turn on the component, which is the default. Which rigid body(s) the settings apply to is determined by the first argument of the *force* and *torque* keywords. It can be an integer M from 1 to Nbody, where Nbody is the number of rigid bodies defined. A wild-card asterisk can be used in place of, or in conjunction with, the M argument to set the flags for multiple rigid bodies. This takes the form "*" or "*n" or "n*" or "m*n". If N = the number of rigid bodies, then an asterisk with no numeric values means all bodies from 1 to N. A leading asterisk means all bodies from 1 to n (inclusive). A trailing asterisk means all bodies from n to N (inclusive). A middle asterisk means all types from m to n (inclusive). Note that you can use the *force* or *torque* keywords as many times as you like. If a particular rigid body has its component flags set multiple times, the settings from the final keyword are used.

**Note**

For computational efficiency, you may wish to turn off pairwise and bond interactions within each rigid body, as they no longer contribute to the motion. The *neigh_modify exclude* and *delete_bonds* commands are used to do this. If the rigid bodies have strongly overlapping atoms, you may need to turn off these interactions to avoid numerical problems due to large equal/opposite intra-body forces swamping the contribution of small inter-body forces.

For computational efficiency, you should typically define one fix rigid or fix rigid/small command which includes all the desired rigid bodies. LAMMPS will allow multiple rigid fixes to be defined, but it is more expensive.

---

The constituent particles within a rigid body can be point particles (the default in LAMMPS) or finite-size particles, such as spheres or ellipsoids or line segments or triangles. See the *atom_style sphere and ellipsoid and line and tri* commands for more details on these kinds of particles. Finite-size particles contribute differently to the moment of inertia of a rigid body than do point particles. Finite-size particles can also experience torque (e.g. due to *frictional granular interactions*) and have an orientation. These contributions are accounted for by these fixes.

Forces between particles within a body do not contribute to the external force or torque on the body. Thus for computational efficiency, you may wish to turn off pairwise and bond interactions between particles within each rigid body. The *neigh_modify exclude* and *delete_bonds* commands are used to do this. For finite-size particles this also means the particles can be highly overlapped when creating the rigid body.

---

The *rigid*, *rigid/nve*, *rigid/small*, and *rigid/small/nve* styles perform constant NVE time integration. They are referred to below as the 4 NVE rigid styles. The only difference is that the *rigid* and *rigid/small* styles use an integration technique based on Richardson iterations. The *rigid/nve* and *rigid/small/nve* styles uses the methods described in the paper by *Miller*, which are thought to provide better energy conservation than an iterative approach.

The *rigid/nvt* and *rigid/nvt/small* styles performs constant NVT integration using a Nose/Hoover thermostat with chains as described originally in *(Hoover)* and *(Martyna)*, which thermostats both the translational and rotational degrees of freedom of the rigid bodies. They are referred to below as the 2 NVT rigid styles. The rigid-body algorithm used by *rigid/nvt* is described in the paper by *Kamberaj*.

The *rigid/npt*, *rigid/nph*, *rigid/npt/small*, and *rigid/nph/small* styles perform constant NPT or NPH integration using a Nose/Hoover barostat with chains. They are referred to below as the 4 NPT and NPH rigid styles. For the NPT case, the same Nose/Hoover thermostat is also used as with *rigid/nvt* and *rigid/nvt/small*.

The barostat parameters are specified using one or more of the *iso*, *aniso*, *x*, *y*, *z* and *couple* keywords. These keywords give you the ability to specify 3 diagonal components of the external stress tensor, and to couple these components together so that the dimensions they represent are varied together during a constant-pressure simulation. The effects of these keywords are similar to those defined in *fix npt/nph*

**Note**

Currently the *rigid/npt*, *rigid/nph*, *rigid/npt/small*, and *rigid/nph/small* styles do not support triclinic (non-orthogonal) boxes.

The target pressures for each of the 6 components of the stress tensor can be specified independently via the *x*, *y*, *z* keywords, which correspond to the 3 simulation box dimensions. For each component, the external pressure or tensor component at each timestep is a ramped value during the run from *Pstart* to *Pstop*. If a target pressure is specified for a component, then the corresponding box dimension will change during a simulation. For example, if the *y* keyword is used, the y-box length will change. A box dimension will not change if that component is not specified, although you have the option to change that dimension via the *fix deform* command.

For all barostat keywords, the *Pdamp* parameter operates like the *Tdamp* parameter, determining the time scale on which pressure is relaxed. For example, a value of 10.0 means to relax the pressure in a timespan of (roughly) 10 time units (e.g. $\tau$ or fs or ps - see the *units* command).

Regardless of what atoms are in the fix group (the only atoms which are time integrated), a global pressure or stress tensor is computed for all atoms. Similarly, when the size of the simulation box is changed, all atoms are re-scaled to new positions, unless the keyword *dilate* is specified with a *dilate-group-ID* for a group that represents a subset of the atoms. This can be useful, for example, to leave the coordinates of atoms in a solid substrate unchanged and controlling the pressure of a surrounding fluid. Another example is a system consisting of rigid bodies and point particles where the barostat is only coupled with the rigid bodies. This option should be used with care, since it can be unphysical to dilate some atoms and not others, because it can introduce large, instantaneous displacements between a pair of atoms (one dilated, one not) that are far from the dilation origin.

The *couple* keyword allows two or three of the diagonal components of the pressure tensor to be "coupled" together. The value specified with the keyword determines which are coupled. For example, *xz* means the *Pxx* and *Pzz* components of the stress tensor are coupled. *Xyz* means all 3 diagonal components are coupled. Coupling means two things: the instantaneous stress will be computed as an average of the corresponding diagonal components, and the coupled box dimensions will be changed together in lockstep, meaning coupled dimensions will be dilated or contracted by the same percentage every timestep. The *Pstart*, *Pstop*, *Pdamp* parameters for any coupled dimensions must be identical. *Couple xyz* can be used for a 2d simulation; the *z* dimension is simply ignored.

The *iso* and *aniso* keywords are simply shortcuts that are equivalent to specifying several other keywords together.

The keyword *iso* means couple all 3 diagonal components together when pressure is computed (hydrostatic pressure), and dilate/contract the dimensions together. Using "iso Pstart Pstop Pdamp" is the same as specifying these 4 keywords:

```
x Pstart Pstop Pdamp
y Pstart Pstop Pdamp
z Pstart Pstop Pdamp
couple xyz
```

The keyword *aniso* means *x*, *y*, and *z* dimensions are controlled independently using the *Pxx*, *Pyy*, and *Pzz* components of the stress tensor as the driving forces, and the specified scalar external pressure. Using "aniso Pstart Pstop Pdamp" is the same as specifying these 4 keywords:

```
x Pstart Pstop Pdamp
y Pstart Pstop Pdamp
```

(continues on next page)

(continued from previous page)

```
z Pstart Pstop Pdamp
couple none
```

---

The keyword/value option pairs are used in the following ways.

The *reinit* keyword determines, whether the rigid body properties are re-initialized between run commands. With the option *yes* (the default) this is done, with the option *no* this is not done. Turning off the re-initialization can be helpful to protect rigid bodies against unphysical manipulations between runs or when properties cannot be easily re-computed (e.g. when read from a file). When using the *infile* keyword, the *reinit* option is automatically set to *no*.

The *langevin* and *temp* and *tparam* keywords perform thermostatting of the rigid bodies, altering both their translational and rotational degrees of freedom. What is meant by "temperature" of a collection of rigid bodies and how it can be monitored via the fix output is discussed below.

The *langevin* keyword applies a Langevin thermostat to the constant NVE time integration performed by any of the 4 NVE rigid styles: *rigid*, *rigid/nve*, *rigid/small*, *rigid/small/nve*. It cannot be used with the 2 NVT rigid styles: *rigid/nvt*, *rigid/small/nvt*. The desired temperature at each timestep is a ramped value during the run from *Tstart* to *Tstop*. The *Tdamp* parameter is specified in time units and determines how rapidly the temperature is relaxed. For example, a value of 100.0 means to relax the temperature in a timespan of (roughly) 100 time units ($\tau$ or fs or ps - see the *units* command). The random # *seed* must be a positive integer.

The way that Langevin thermostatting operates is explained on the *fix langevin* doc page. If you wish to simply viscously damp the rotational motion without thermostatting, you can set *Tstart* and *Tstop* to 0.0, which means only the viscous drag term in the Langevin thermostat will be applied. See the discussion on the *fix viscous* page for details.

**Note**

When the *langevin* keyword is used with fix rigid versus fix rigid/small, different dynamics will result for parallel runs. This is because of the way random numbers are used in the two cases. The dynamics for the two cases should be statistically similar, but will not be identical, even for a single timestep.

The *temp* and *tparam* keywords apply a Nose/Hoover thermostat to the NVT time integration performed by the 2 NVT rigid styles. They cannot be used with the 4 NVE rigid styles. The desired temperature at each timestep is a ramped value during the run from *Tstart* to *Tstop*. The *Tdamp* parameter is specified in time units and determines how rapidly the temperature is relaxed. For example, a value of 100.0 means to relax the temperature in a timespan of (roughly) 100 time units (tau or fs or ps - see the *units* command).

Nose/Hoover chains are used in conjunction with this thermostat. The *tparam* keyword can optionally be used to change the chain settings used. *Tchain* is the number of thermostats in the Nose Hoover chain. This value, along with *Tdamp* can be varied to dampen undesirable oscillations in temperature that can occur in a simulation. As a rule of thumb, increasing the chain length should lead to smaller oscillations. The keyword *pchain* specifies the number of thermostats in the chain thermostatting the barostat degrees of freedom.

**Note**

There are alternate ways to thermostat a system of rigid bodies. You can use *fix langevin* to treat the individual particles in the rigid bodies as effectively immersed in an implicit solvent, e.g. a Brownian dynamics model. For hybrid systems with both rigid bodies and solvent particles, you can thermostat only the solvent particles that surround one or more rigid bodies by appropriate choice of groups in the compute and fix commands for temperature and thermostatting. The solvent interactions with the rigid bodies should then effectively thermostat the rigid body temperature as well without use of the Langevin or Nose/Hoover options associated with the fix rigid commands.

---

The *mol* keyword can only be used with the *rigid/small* styles. It must be used when other commands, such as *fix deposit* or *fix pour*, add rigid bodies on-the-fly during a simulation. You specify a *template-ID* previously defined using the *molecule* command, which reads a file that defines the molecule. You must use the same *template-ID* that the other fix which is adding rigid bodies uses. The coordinates, atom types, atom diameters, center-of-mass, and moments of inertia can be specified in the molecule file. See the *molecule* command for details. The only settings required to be in this file are the coordinates and types of atoms in the molecule, in which case the molecule command calculates the other quantities itself.

Note that these other fixes create new rigid bodies, in addition to those defined initially by this fix via the *bodystyle* setting.

Also note that when using the *mol* keyword, extra restart information about all rigid bodies is written out whenever a restart file is written out. See the NOTE in the next section for details.

---

The *infile* keyword allows a file of rigid body attributes to be read in from a file, rather then having LAMMPS compute them. There are 5 such attributes: the total mass of the rigid body, its center-of-mass position, its 6 moments of inertia, its center-of-mass velocity, and the 3 image flags of the center-of-mass position. For rigid bodies consisting of point particles or non-overlapping finite-size particles, LAMMPS can compute these values accurately.

However, for rigid bodies consisting of finite-size particles which overlap each other, LAMMPS will ignore the overlaps when computing these 4 attributes, which means the dynamics of the bodies will be incorrect. The amount of error this induces depends on the amount of overlap. To avoid this issue, the values can be pre-computed (e.g. using Monte Carlo integration).

The format of the file is as follows. Note that the file does not have to list attributes for every rigid body integrated by fix rigid. Only bodies which the file specifies will have their computed attributes overridden. The file can contain initial blank lines or comment lines starting with "#" which are ignored. The first non-blank, non-comment line should list N = the number of lines to follow. The N successive lines contain the following information:

```
ID1 masstotal xcm ycm zcm ixx iyy izz ixy ixz iyz vxcm vycm vzcm lx ly lz ixcm iycm izcm
ID2 masstotal xcm ycm zcm ixx iyy izz ixy ixz iyz vxcm vycm vzcm lx ly lz ixcm iycm izcm
...
IDN masstotal xcm ycm zcm ixx iyy izz ixy ixz iyz vxcm vycm vzcm lx ly lz ixcm iycm izcm
```

The rigid body IDs are all positive integers. For the *single* bodystyle, only an ID of 1 can be used. For the *group* bodystyle, IDs from 1 to Ng can be used where Ng is the number of specified groups. For the *molecule* bodystyle, use the molecule ID for the atoms in a specific rigid body as the rigid body ID.

The masstotal and center-of-mass coordinates (xcm,ycm,zcm) are self-explanatory. The center-of-mass should be consistent with what is calculated for the position of the rigid body with all its atoms unwrapped by their respective image flags. If this produces a center-of-mass that is outside the simulation box, LAMMPS wraps it back into the box.

The 6 moments of inertia (ixx,iyy,izz,ixy,ixz,iyz) should be the values consistent with the current orientation of the rigid body around its center of mass. The values are with respect to the simulation box XYZ axes, not with respect to the principal axes of the rigid body itself. LAMMPS performs the latter calculation internally.

The (vxcm,vycm,vzcm) values are the velocity of the center of mass. The (lx,ly,lz) values are the angular momentum of the body. The (vxcm,vycm,vzcm) and (lx,ly,lz) values can simply be set to 0 if you wish the body to have no initial motion.

The (ixcm,iycm,izcm) values are the image flags of the center of mass of the body. For periodic dimensions, they specify which image of the simulation box the body is considered to be in. An image of 0 means it is inside the box as defined. A value of 2 means add 2 box lengths to get the true value. A value of -1 means subtract 1 box length to get the true value. LAMMPS updates these flags as the rigid bodies cross periodic boundaries during the simulation.

**Note**

If you use the *infile* or *mol* keywords and write restart files during a simulation, then each time a restart file is written, the fix also write an auxiliary restart file with the name rfile.rigid, where "rfile" is the name of the restart file, e.g. tmp.restart.10000 and tmp.restart.10000.rigid. This auxiliary file is in the same format described above. Thus it can be used in a new input script that restarts the run and re-specifies a rigid fix using an *infile* keyword and the appropriate filename. Note that the auxiliary file will contain one line for every rigid body, even if the original file only listed a subset of the rigid bodies.

If the system has rigid bodies with finite-size overlapping particles and the model uses the *fix gravity* command to apply a gravitational force to the rigid bodies, then the *gravity* keyword should be used in the following manner.

First, the group specified for the *fix gravity* command should not include any atoms in rigid bodies which have overlapping particles. It can be empty (see the *group empty* command) or only contain single particles not in rigid bodies, e.g. background particles.

Second, the *infile* keyword should be used to specify the total mass and other properties of the rigid bodies with overlaps, so that their dynamics will be modeled correctly, as explained above.

Third, the *gravity* keyword should be used the with the ID of the *fix gravity* command as its argument. The rigid fixes will access the gravity fix to extract the current direction of the gravity vector at each timestep (which can be static or dynamic). A gravity force will then be applied to each rigid body at its center-of-mass position using its total mass.

---

If you use a *temperature compute* with a group that includes particles in rigid bodies, the degrees-of-freedom removed by each rigid body are accounted for in the temperature (and pressure) computation, but only if the temperature group includes all the particles in a particular rigid body.

A 3d rigid body has 6 degrees of freedom (3 translational, 3 rotational), except for a collection of point particles lying on a straight line, which has only 5, e.g a dimer. A 2d rigid body has 3 degrees of freedom (2 translational, 1 rotational).

**Note**

You may wish to explicitly subtract additional degrees-of-freedom if you use the *force* and *torque* keywords to eliminate certain motions of one or more rigid bodies. LAMMPS does not do this automatically.

The rigid body contribution to the pressure of the system (virial) is also accounted for by this fix.

---

If your simulation is a hybrid model with a mixture of rigid bodies and non-rigid particles (e.g. solvent) there are several ways these rigid fixes can be used in tandem with *fix nve*, *fix nvt*, *fix npt*, and *fix nph*.

If you wish to perform NVE dynamics (no thermostatting or barostatting), use one of 4 NVE rigid styles to integrate the rigid bodies, and *fix nve* to integrate the non-rigid particles.

If you wish to perform NVT dynamics (thermostatting, but no barostatting), you can use one of the 2 NVT rigid styles for the rigid bodies, and any thermostatting fix for the non-rigid particles (*fix nvt*, *fix langevin*, *fix temp/berendsen*). You can also use one of the 4 NVE rigid styles for the rigid bodies and thermostat them using *fix langevin* on the group that contains all the particles in the rigid bodies. The net force added by *fix langevin* to each rigid body effectively thermostats its translational center-of-mass motion. Not sure how well it does at thermostatting its rotational motion.

If you wish to perform NPT or NPH dynamics (barostatting), you cannot use both *fix npt* and the NPT or NPH rigid styles. This is because there can only be one fix which monitors the global pressure and changes the simulation box dimensions. So you have 3 choices:

1. Use one of the 4 NPT or NPH styles for the rigid bodies. Use the *dilate* all option so that it will dilate the positions of the non-rigid particles as well. Use *fix nvt* (or any other thermostat) for the non-rigid particles.
2. Use *fix npt* for the group of non-rigid particles. Use the *dilate* all option so that it will dilate the center-of-mass positions of the rigid bodies as well. Use one of the 4 NVE or 2 NVT rigid styles for the rigid bodies.
3. Use *fix press/berendsen* to compute the pressure and change the box dimensions. Use one of the 4 NVE or 2 NVT rigid styles for the rigid bodies. Use *fix nvt* (or any other thermostat) for the non-rigid particles.

In all case, the rigid bodies and non-rigid particles both contribute to the global pressure and the box is scaled the same by any of the barostatting fixes.

You could even use the second and third options for a non-hybrid simulation consisting of only rigid bodies, assuming you give *fix npt* an empty group, though it's an odd thing to do. The barostatting fixes (*fix npt* and *fix press/berensen*) will monitor the pressure and change the box dimensions, but not time integrate any particles. The integration of the rigid bodies will be performed by fix rigid/nvt.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.226.4 Restart, fix_modify, output, run start/stop, minimize info

No information about the 4 NVE rigid styles is written to *binary restart files*. The exception is if the *infile* or *mol* keyword is used, in which case an auxiliary file is written out with rigid body information each time a restart file is written, as explained above for the *infile* keyword. For the 2 NVT rigid styles, the state of the Nose/Hoover thermostat is written to *binary restart files*. Ditto for the 4 NPT and NPH rigid styles, and the state of the Nose/Hoover barostat. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The *fix_modify* *temp* and *press* options are supported by the 4 NPT and NPH rigid styles to change the computes used to calculate the instantaneous pressure tensor. Note that the 2 NVT rigid fixes do not use any external compute to compute instantaneous temperature.

The *fix_modify* *bodyforces* option is supported by all rigid styles to set whether per-body forces and torques are computed early or late in a timestep, i.e. at the post-force stage or at the final-integrate stage or the timestep, respectively.

The cumulative energy change in the system imposed by the 6 NVT, NPT, NPH rigid fixes, via either thermostatting and/or barostatting, is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* doc page for details.

The 2 NVE rigid fixes compute a global scalar which can be accessed by various *output commands*. The scalar value calculated by these fixes is "intensive". The scalar is the current temperature of the collection of rigid bodies. This is averaged over all rigid bodies and their translational and rotational degrees of freedom. The translational energy of a rigid body is 1/2 m v^2, where m = total mass of the body and v = the velocity of its center of mass. The rotational energy of a rigid body is 1/2 I w^2, where I = the moment of inertia tensor of the body and w = its angular velocity. Degrees of freedom constrained by the *force* and *torque* keywords are removed from this calculation, but only for the *rigid* and *rigid/nve* fixes.

The 6 NVT, NPT, NPH rigid fixes compute a global scalar which can be accessed by various *output commands*. The scalar is the same cumulative energy change due to these fixes described above. The scalar value calculated by this fix is "extensive".

The *fix_modify virial* option is supported by these fixes to add the contribution due to the added forces on atoms to both the global pressure and per-atom stress of the system via the *compute pressure* and *compute stress/atom* commands. The former can be accessed by *thermodynamic output*. The default setting for this fix is *fix_modify virial yes*.

All of the *rigid* styles (but not the *rigid/small* styles) compute a global array of values which can be accessed by various *output commands*. Similar information about the bodies defined by the *rigid/small* styles can be accessed via the *compute rigid/local* command.

The number of rows in the array is equal to the number of rigid bodies. The number of columns is 15. Thus for each rigid body, 15 values are stored: the xyz coords of the center of mass (COM), the xyz components of the COM velocity, the xyz components of the force acting on the COM, the xyz components of the torque acting on the COM, and the xyz image flags of the COM.

The center of mass (COM) for each body is similar to unwrapped coordinates written to a dump file. It will always be inside (or slightly outside) the simulation box. The image flags have the same meaning as image flags for atom positions (see the "dump" command). This means you can calculate the unwrapped COM by applying the image flags to the COM, the same as when unwrapped coordinates are written to a dump file.

The force and torque values in the array are not affected by the *force* and *torque* keywords in the fix rigid command; they reflect values before any changes are made by those keywords.

The ordering of the rigid bodies (by row in the array) is as follows. For the *single* keyword there is just one rigid body. For the *molecule* keyword, the bodies are ordered by ascending molecule ID. For the *group* keyword, the list of group IDs determines the ordering of bodies.

The array values calculated by these fixes are "intensive", meaning they are independent of the number of atoms in the simulation.

No parameter of these fixes can be used with the *start/stop* keywords of the *run* command. These fixes are not invoked during *energy minimization*.

---

## 2.226.5 Restrictions

These fixes are all part of the RIGID package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Assigning a temperature via the *velocity create* command to a system with *rigid bodies* may not have the desired outcome for two reasons. First, the velocity command can be invoked before the rigid-body fix is invoked or initialized and the number of adjusted degrees of freedom (DOFs) is known. Thus it is not possible to compute the target temperature correctly. Second, the assigned velocities may be partially canceled when constraints are first enforced, leading to a different temperature than desired. A workaround for this is to perform a *run 0* command, which ensures all DOFs are accounted for properly, and then rescale the temperature to the desired value before performing a simulation. For example:

```
velocity all create 300.0 12345
run 0                          # temperature may not be 300K
velocity all scale 300.0       # now it should be
```

## 2.226.6 Related commands

*delete_bonds*, *neigh_modify* exclude, *fix shake*

### 2.226.7 Default

The option defaults are force * on on on and torque * on on on, meaning all rigid bodies are acted on by center-of-mass force and torque. Also Tchain = Pchain = 10, Titer = 1, Torder = 3, reinit = yes.

---

**(Hoover)** Hoover, Phys Rev A, 31, 1695 (1985).

**(Kamberaj)** Kamberaj, Low, Neal, J Chem Phys, 122, 224114 (2005).

**(Martyna)** Martyna, Klein, Tuckerman, J Chem Phys, 97, 2635 (1992); Martyna, Tuckerman, Tobias, Klein, Mol Phys, 87, 1117.

**(Miller)** Miller, Eleftheriou, Pattnaik, Ndirango, and Newns, J Chem Phys, 116, 8649 (2002).

**(Zhang)** Zhang, Glotzer, Nanoletters, 4, 1407-1413 (2004).

## 2.227 fix rigid/meso command

### 2.227.1 Syntax

```
fix ID group-ID rigid/meso bodystyle args keyword values ...
```

- ID, group-ID are documented in *fix* command
- rigid/meso = style name of this fix command
- bodystyle = *single* or *molecule* or *group*

```
single args = none
molecule args = none
custom args = i_propname or v_varname
  i_propname = an integer property defined via fix property/atom
  v_varname  = an atom-style or atomfile-style variable
group args = N groupID1 groupID2 ...
  N = # of groups
  groupID1, groupID2, ... = list of N group IDs
```

- zero or more keyword/value pairs may be appended
- keyword = *reinit* or *force* or *torque* or *infile*

```
reinit = yes or no
force values = M xflag yflag zflag
  M = which rigid body from 1-Nbody (see asterisk form below)
  xflag,yflag,zflag = off/on if component of center-of-mass force is active
torque values = M xflag yflag zflag
  M = which rigid body from 1-Nbody (see asterisk form below)
  xflag,yflag,zflag = off/on if component of center-of-mass torque is active
infile filename
  filename = file with per-body values of mass, center-of-mass, moments of inertia
```

### 2.227.2 Examples

```
fix 1 ellipsoid rigid/meso single
fix 1 rods      rigid/meso molecule
fix 1 spheres   rigid/meso single force 1 off off on
```

(continues on next page)

(continued from previous page)

```
fix 1 particles rigid/meso molecule force 1*5 off off off force 6*10 off off on
fix 2 spheres   rigid/meso group 3 sphere1 sphere2 sphere3 torque * off off off
```

### 2.227.3 Description

Treat one or more sets of mesoscopic SPH/SDPD particles as independent rigid bodies. This means that each timestep the total force and torque on each rigid body is computed as the sum of the forces and torques on its constituent particles. The coordinates and velocities of the particles in each body are then updated so that the body moves and rotates as a single entity using the methods described in the paper by *(Miller)*. Density and internal energy of the particles will also be updated. This is implemented by creating internal data structures for each rigid body and performing time integration on these data structures. Positions and velocities of the constituent particles are regenerated from the rigid body data structures in every time step. This restricts which operations and fixes can be applied to rigid bodies. See below for a detailed discussion.

The operation of this fix is exactly like that described by the *fix rigid/nve* command, except that particles' density, internal energy and extrapolated velocity are also updated.

**Note**

You should not update the particles in rigid bodies via other time-integration fixes (e.g. *fix sph*, *fix sph/stationary*), or you will have conflicting updates to positions and velocities resulting in unphysical behavior in most cases. When performing a hybrid simulation with some atoms in rigid bodies, and some not, a separate time integration fix like *fix sph* should be used for the non-rigid particles.

**Note**

These fixes are overkill if you simply want to hold a collection of particles stationary or have them move with a constant velocity. To hold particles stationary use *fix sph/stationary* instead. If you would like to move particles with a constant velocity use *fix meso/move*.

**Warning**

The aggregate properties of each rigid body are calculated at the start of a simulation run and are maintained in internal data structures. The properties include the position and velocity of the center-of-mass of the body, its moments of inertia, and its angular momentum. This is done using the properties of the constituent particles of the body at that point in time (or see the *infile* keyword option). Thereafter, changing these properties of individual particles in the body will have no effect on a rigid body's dynamics, unless they effect any computation of per-particle forces or torques. If the keyword *reinit* is set to *yes* (the default), the rigid body data structures will be recreated at the beginning of each *run* command; if the keyword *reinit* is set to *no*, the rigid body data structures will be built only at the very first *run* command and maintained for as long as the rigid fix is defined. For example, you might think you could displace the particles in a body or add a large velocity to each particle in a body to make it move in a desired direction before a second run is performed, using the *set* or *displace_atoms* or *velocity* commands. But these commands will not affect the internal attributes of the body unless *reinit* is set to *yes*. With *reinit* set to *no* (or using the *infile* option, which implies *reinit no*) the position and velocity of individual particles in the body will be reset when time integration starts again.

---

Each rigid body must have two or more particles. A particle can belong to at most one rigid body. Which particles are in which bodies can be defined via several options.

For bodystyle *single* the entire fix group of particles is treated as one rigid body.

For bodystyle *molecule*, particles are grouped into rigid bodies by their respective molecule IDs: each set of particles in the fix group with the same molecule ID is treated as a different rigid body. Note that particles with a molecule ID = 0 will be treated as a single rigid body. For a system with solvent (typically this is particles with molecule ID = 0) surrounding rigid bodies, this may not be what you want. Thus you should be careful to use a fix group that only includes particles you want to be part of rigid bodies.

Bodystyle *custom* is similar to bodystyle *molecule* except that it is more flexible in using other per-atom properties to define the sets of particles that form rigid bodies. An integer vector defined by the *fix property/atom* command can be used. Or an *atom-style or atomfile-style variable* can be used; the floating-point value produced by the variable is rounded to an integer. As with bodystyle *molecule*, each set of particles in the fix groups with the same integer value is treated as a different rigid body. Since fix property/atom vectors and atom-style variables produce values for all particles, you should be careful to use a fix group that only includes particles you want to be part of rigid bodies.

For bodystyle *group*, each of the listed groups is treated as a separate rigid body. Only particles that are also in the fix group are included in each rigid body.

**Note**

To compute the initial center-of-mass position and other properties of each rigid body, the image flags for each particle in the body are used to "unwrap" the particle coordinates. Thus you must ensure that these image flags are consistent so that the unwrapping creates a valid rigid body (one where the particles are close together) , particularly if the particles in a single rigid body straddle a periodic boundary. This means the input data file or restart file must define the image flags for each particle consistently or that you have used the *set* command to specify them correctly. If a dimension is non-periodic then the image flag of each particle must be 0 in that dimension, else an error is generated.

By default, each rigid body is acted on by other particles which induce an external force and torque on its center of mass, causing it to translate and rotate. Components of the external center-of-mass force and torque can be turned off by the *force* and *torque* keywords. This may be useful if you wish a body to rotate but not translate, or vice versa, or if you wish it to rotate or translate continuously unaffected by interactions with other particles. Note that if you expect a rigid body not to move or rotate by using these keywords, you must ensure its initial center-of-mass translational or angular velocity is 0.0. Otherwise the initial translational or angular momentum, the body has, will persist.

An xflag, yflag, or zflag set to *off* means turn off the component of force or torque in that dimension. A setting of *on* means turn on the component, which is the default. Which rigid body(s) the settings apply to is determined by the first argument of the *force* and *torque* keywords. It can be an integer M from 1 to Nbody, where Nbody is the number of rigid bodies defined. A wild-card asterisk can be used in place of, or in conjunction with, the M argument to set the flags for multiple rigid bodies. This takes the form "*" or "*n" or "n*" or "m*n". If N = the number of rigid bodies, then an asterisk with no numeric values means all bodies from 1 to N. A leading asterisk means all bodies from 1 to n (inclusive). A trailing asterisk means all bodies from n to N (inclusive). A middle asterisk means all bodies from m to n (inclusive). Note that you can use the *force* or *torque* keywords as many times as you like. If a particular rigid body has its component flags set multiple times, the settings from the final keyword are used.

For computational efficiency, you should typically define one fix rigid/meso command which includes all the desired rigid bodies. LAMMPS will allow multiple rigid/meso fixes to be defined, but it is more expensive.

---

The keyword/value option pairs are used in the following ways.

The *reinit* keyword determines, whether the rigid body properties are re-initialized between run commands. With the option *yes* (the default) this is done, with the option *no* this is not done. Turning off the re-initialization can be helpful to protect rigid bodies against unphysical manipulations between runs or when properties cannot be easily re-computed (e.g. when read from a file). When using the *infile* keyword, the *reinit* option is automatically set to *no*.

---

The *infile* keyword allows a file of rigid body attributes to be read in from a file, rather then having LAMMPS compute them. There are 5 such attributes: the total mass of the rigid body, its center-of-mass position, its 6 moments of inertia, its center-of-mass velocity, and the 3 image flags of the center-of-mass position. For rigid bodies consisting of point particles or non-overlapping finite-size particles, LAMMPS can compute these values accurately. However, for rigid bodies consisting of finite-size particles which overlap each other, LAMMPS will ignore the overlaps when computing these 4 attributes. The amount of error this induces depends on the amount of overlap. To avoid this issue, the values can be pre-computed (e.g. using Monte Carlo integration).

The format of the file is as follows. Note that the file does not have to list attributes for every rigid body integrated by fix rigid. Only bodies which the file specifies will have their computed attributes overridden. The file can contain initial blank lines or comment lines starting with "#" which are ignored. The first non-blank, non-comment line should list N = the number of lines to follow. The N successive lines contain the following information:

```
ID1 masstotal xcm ycm zcm ixx iyy izz ixy ixz iyz vxcm vycm vzcm lx ly lz ixcm iycm izcm
ID2 masstotal xcm ycm zcm ixx iyy izz ixy ixz iyz vxcm vycm vzcm lx ly lz ixcm iycm izcm
...
IDN masstotal xcm ycm zcm ixx iyy izz ixy ixz iyz vxcm vycm vzcm lx ly lz ixcm iycm izcm
```

The rigid body IDs are all positive integers. For the *single* bodystyle, only an ID of 1 can be used. For the *group* bodystyle, IDs from 1 to Ng can be used where Ng is the number of specified groups. For the *molecule* bodystyle, use the molecule ID for the atoms in a specific rigid body as the rigid body ID.

The masstotal and center-of-mass coordinates (xcm,ycm,zcm) are self-explanatory. The center-of-mass should be consistent with what is calculated for the position of the rigid body with all its atoms unwrapped by their respective image flags. If this produces a center-of-mass that is outside the simulation box, LAMMPS wraps it back into the box.

The 6 moments of inertia (ixx,iyy,izz,ixy,ixz,iyz) should be the values consistent with the current orientation of the rigid body around its center of mass. The values are with respect to the simulation box XYZ axes, not with respect to the principal axes of the rigid body itself. LAMMPS performs the latter calculation internally.

The (vxcm,vycm,vzcm) values are the velocity of the center of mass. The (lx,ly,lz) values are the angular momentum of the body. The (vxcm,vycm,vzcm) and (lx,ly,lz) values can simply be set to 0 if you wish the body to have no initial motion.

The (ixcm,iycm,izcm) values are the image flags of the center of mass of the body. For periodic dimensions, they specify which image of the simulation box the body is considered to be in. An image of 0 means it is inside the box as defined. A value of 2 means add 2 box lengths to get the true value. A value of -1 means subtract 1 box length to get the true value. LAMMPS updates these flags as the rigid bodies cross periodic boundaries during the simulation.

**Note**

If you use the *infile* keyword and write restart files during a simulation, then each time a restart file is written, the fix also write an auxiliary restart file with the name rfile.rigid, where "rfile" is the name of the restart file, e.g. tmp.restart.10000 and tmp.restart.10000.rigid. This auxiliary file is in the same format described above. Thus it can be used in a new input script that restarts the run and re-specifies a rigid fix using an *infile* keyword and the appropriate filename. Note that the auxiliary file will contain one line for every rigid body, even if the original file only listed a subset of the rigid bodies.

### 2.227.4 Restart, fix_modify, output, run start/stop, minimize info

No information is written to *binary restart files*. If the *infile* keyword is used, an auxiliary file is written out with rigid body information each time a restart file is written, as explained above for the *infile* keyword.

None of the *fix_modify* options are relevant to this fix.

This fix computes a global array of values which can be accessed by various *output commands*.

The number of rows in the array is equal to the number of rigid bodies. The number of columns is 28. Thus for each rigid body, 28 values are stored: the xyz coords of the center of mass (COM), the xyz components of the COM velocity, the xyz components of the force acting on the COM, the components of the 4-vector quaternion representing the orientation of the rigid body, the xyz components of the angular velocity of the body around its COM, the xyz components of the torque acting on the COM, the 3 principal components of the moment of inertia, the xyz components of the angular momentum of the body around its COM, and the xyz image flags of the COM.

The center of mass (COM) for each body is similar to unwrapped coordinates written to a dump file. It will always be inside (or slightly outside) the simulation box. The image flags have the same meaning as image flags for particle positions (see the "dump" command). This means you can calculate the unwrapped COM by applying the image flags to the COM, the same as when unwrapped coordinates are written to a dump file.

The force and torque values in the array are not affected by the *force* and *torque* keywords in the fix rigid command; they reflect values before any changes are made by those keywords.

The ordering of the rigid bodies (by row in the array) is as follows. For the *single* keyword there is just one rigid body. For the *molecule* keyword, the bodies are ordered by ascending molecule ID. For the *group* keyword, the list of group IDs determines the ordering of bodies.

The array values calculated by this fix are "intensive", meaning they are independent of the number of particles in the simulation.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

This fix is not invoked during *energy minimization*.

---

### 2.227.5 Restrictions

This fix is part of the DPD-SMOOTH package and also depends on the RIGID package. It is only enabled if LAMMPS was built with both packages. See the *Build package* page for more info.

This fix requires that atoms store density and internal energy as defined by the *atom_style sph* command.

All particles in the group must be mesoscopic SPH/SDPD particles.

Changed in version 29Aug2024.

This fix is incompatible with deformation controls that remap velocity, for instance the *remap v* option of *fix deform*.

### 2.227.6 Related commands

*fix meso/move*, *fix rigid*, *neigh_modify exclude*

### 2.227.7 Default

The option defaults are force * on on on and torque * on on on, meaning all rigid bodies are acted on by center-of-mass force and torque. Also reinit = yes.

---

**(Miller)** Miller, Eleftheriou, Pattnaik, Ndirango, and Newns, J Chem Phys, 116, 8649 (2002).

## 2.228 fix rx command

Accelerator Variants: *rx/kk*

### 2.228.1 Syntax

```
fix ID group-ID rx file localTemp matrix solver minSteps ...
```

- ID, group-ID are documented in *fix* command
- rx = style name of this fix command
- file = filename containing the reaction kinetic equations and Arrhenius parameters
- localTemp = *none,lucy* = no local temperature averaging or local temperature defined through Lucy weighting function
- matrix = *sparse, dense* format for the stoichiometric matrix
- solver = *lammps_rk4,rkf45* = rk4 is an explicit fourth order Runge-Kutta method; rkf45 is an adaptive fourth-order Runge-Kutta-Fehlberg method
- minSteps = # of steps for rk4 solver or minimum # of steps for rkf45 (rk4 or rkf45)
- maxSteps = maximum number of steps for the rkf45 solver (rkf45 only)
- relTol = relative tolerance for the rkf45 solver (rkf45 only)
- absTol = absolute tolerance for the rkf45 solver (rkf45 only)
- diag = Diagnostics frequency for the rkf45 solver (optional, rkf45 only)

### 2.228.2 Examples

```
fix 1 all rx kinetics.rx none dense lammps_rk4
fix 1 all rx kinetics.rx none sparse lammps_rk4 1
fix 1 all rx kinetics.rx lucy sparse lammps_rk4 10
fix 1 all rx kinetics.rx none dense rkf45 1 100 1e-6 1e-8
fix 1 all rx kinetics.rx none dense rkf45 1 100 1e-6 1e-8 -1
```

### 2.228.3 Description

Fix *rx* solves the reaction kinetic ODEs for a given reaction set that is defined within the file associated with this command.

For a general reaction such that

$$\nu_A A + \nu_B B \rightarrow \nu_C C$$

the reaction rate equation is defined to be of the form

$$r = k(T)[A]^{\nu_A}[B]^{\nu_B}$$

In the current implementation, the exponents are defined to be equal to the stoichiometric coefficients. A given reaction set consisting of $n$ reaction equations will contain a total of $m$ species. A set of $m$ ordinary differential equations (ODEs) that describe the change in concentration of a given species as a function of time are then constructed based on the $n$ reaction rate equations.

The ODE systems are solved over the full DPD timestep $dt$ using either a fourth order Runge-Kutta *rk4* method with a fixed step-size $h$, specified by the *lammps_rk4* keyword, or a fourth order Runge-Kutta-Fehlberg (rkf45) method with an adaptive step-size for $h$. The number of ODE steps per DPD timestep for the rk4 method is optionally specified immediately after the rk4 keyword. The ODE step-size is set as *dt/num_steps*. Smaller step-sizes tend to yield more accurate results but there is not control on the error. For error control, use the rkf45 ODE solver.

The rkf45 method adjusts the step-size so that the local truncation error is held within the specified absolute and relative tolerances. The initial step-size *h0* can be specified by the user or estimated internally. It is recommended that the user specify *h0* since this will generally reduced the number of ODE integration steps required. *h0* is defined as *dt / min_steps* if min_steps >= 1. If min_steps == 0, *h0* is estimated such that an explicit Euler method would likely produce an acceptable solution. This is generally overly conservative for the fourth-order method and users are advised to specify *h0* as some fraction of the DPD timestep. For small DPD timesteps, only one step may be necessary depending upon the tolerances. Note that more than min_steps ODE steps may be taken depending upon the ODE stiffness but no more than max_steps will be taken. If max_steps is reached, an error warning is printed and the simulation is stopped.

After each ODE step, the solution error *e* is tested and weighted using the absTol and relTol values. The error vector is weighted as *e* / (relTol * |*u*| + absTol) where *u* is the solution vector. If the norm of the error is <= 1, the solution is accepted, *h* is increased by a proportional amount, and the next ODE step is begun. Otherwise, *h* is shrunk and the ODE step is repeated.

Run-time diagnostics are available for the rkf45 ODE solver. The frequency (in timesteps) that diagnostics are reported is controlled by the last (optional) 12th argument. A negative frequency means that diagnostics are reported once at the end of each run. A positive value N means that the diagnostics are reported once per N timesteps.

The diagnostics report the average # of integrator steps and RHS function evaluations and run-time per ODE as well as the average/RMS/min/max per process. If the reporting frequency is 1, the RMS/min/max per ODE are also reported. The per ODE statistics can be used to adjust the tolerance and min/max step parameters. The statistics per MPI process can be useful to examine any load imbalance caused by the adaptive ODE solver. (Some DPD particles can take longer to solve than others. This can lead to an imbalance across the MPI processes.)

---

The filename specifies a file that contains the entire set of reaction kinetic equations and corresponding Arrhenius parameters. The format of this file is described below.

There is no restriction on the total number or reaction equations that are specified. The species names are arbitrary string names that are associated with the species concentrations, though the string names should be limited to what would be valid values for the names of *property/atom fixes*. Each species in a given reaction must be preceded by its stoichiometric coefficient. The only delimiters that are recognized between the species are either a + or = character. Spaces around these delimiters are **required**. The = character corresponds to an irreversible reaction. After specifying the reaction, the reaction rate constant is determined through the temperature dependent Arrhenius equation:

$$k = AT^n e^{\frac{-E_a}{k_B T}}$$

where $A$ is the Arrhenius factor in time units or concentration/time units, $n$ is the unitless exponent of the temperature dependence, and $E_a$ is the activation energy in energy units. The temperature dependence can be removed by specifying the exponent as zero.

The internal temperature of the coarse-grained particles can be used in constructing the reaction rate constants at every DPD timestep by specifying the keyword *none*. Alternatively, the keyword *lucy* can be specified to compute a local-average particle internal temperature for use in the reaction rate constant expressions. The local-average particle internal temperature is defined as:

$$\theta_i^{-1} = \frac{\sum_{j=1} \omega_{Lucy}\left(r_{ij}\right) \theta_j^{-1}}{\sum_{j=1} \omega_{Lucy}\left(r_{ij}\right)}$$

where the Lucy function is expressed as:

$$\omega_{Lucy}\left(r_{ij}\right) = \left(1 + \frac{3r_{ij}}{r_c}\right)\left(1 - \frac{r_{ij}}{r_c}\right)^3$$

The self-particle interaction is included in the above equation.

The stoichiometric coefficients for the reaction mechanism are stored in either a sparse or dense matrix format. The dense matrix should only be used for small reaction mechanisms. The sparse matrix should be used when there are many

reactions (e.g., more than 5). This allows the number of reactions and species to grow while keeping the computational cost tractable. The matrix format can be specified as using either the *sparse* or *dense* keywords. If all stoichiometric coefficients for a reaction are small integers (whole numbers <= 3), a fast exponential function is used. This can save significant computational time so users are encouraged to use integer coefficients where possible.

---

The format of a tabulated file is as follows (without the parenthesized comments):

```
# Rxn equations and parameters                                          (one or␣
↪more comment or blank lines)

1.0 hcn + 1.0 no2 = 1.0  no + 0.5 n2  + 0.5 h2 + 1.0 co   2.49E+01 0.0 1.34 # (rxn␣
↪equation, A, n, Ea)
1.0 hcn + 1.0  no = 1.0  co + 1.0 n2  + 0.5 h2            2.16E+00 0.0 1.52
...
1.0  no + 1.0  co = 0.5  n2 + 1.0 co2                     1.66E+06 0.0 0.69
```

A section begins with a non-blank line whose first character is not a "#"; blank lines or lines starting with "#" can be used as comments between sections.

Following a blank line, the next N lines list the N reaction equations. Each species within the reaction equation is specified through its stoichiometric coefficient and a species tag. Reactant species are specified on the left-hand side of the equation and product species are specified on the right-hand side of the equation. After specifying the reactant and product species, the final three arguments of each line represent the Arrhenius parameter *A*, the temperature exponent *n*, and the activation energy *Ea*.

Note that the species tags that are defined in the reaction equations are used by the *fix eos/table/rx* command to define the thermodynamic properties of each species. Furthermore, the number of species molecules (i.e., concentration) can be specified either with the *set* command using the "d_" prefix or by reading directly the concentrations from a data file. For the latter case, the *read_data* command with the fix keyword should be specified, where the fix-ID will be the "fix rx`ID with a <SPECIES">`_ suffix, e.g.

fix foo all rx reaction.file ... read_data data.dpd fix foo_SPECIES NULL Species

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

## 2.228.4 Restrictions

This command is part of the DPD-REACT package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This command also requires use of the *atom_style dpd* command.

This command can only be used with a constant energy or constant enthalpy DPD simulation.

### 2.228.5 Related commands

*fix eos/table/rx*, *fix shardlow*, *pair dpd/fdt/energy*

### 2.228.6 Default

none

## 2.229 fix saed/vtk command

### 2.229.1 Syntax

```
fix ID group-ID saed/vtk Nevery Nrepeat Nfreak c_ID attribute args ... keyword args ...
```

- ID, group-ID are documented in *fix* command
- saed/vtk = style name of this fix command
- Nevery = use input values every this many timesteps
- Nrepeat = # of times to use input values for calculating averages
- Nfreq = calculate averages every this many timesteps
- c_ID = saed compute ID

```
keyword = file or ave or start or file or overwrite:l
  ave args = one or running or window M
    one = output a new average value every Nfreq steps
    running = output cumulative average of all previous Nfreq steps
    window M = output average of M most recent Nfreq steps
  start args = Nstart
    Nstart = start averaging on this timestep
  file arg = filename
    filename = name of file to output time averages to
```

### 2.229.2 Examples

```
compute 1 all saed 0.0251 Al O Kmax 1.70 Zone 0 0 1 dR_Ewald 0.01 c 0.5 0.5 0.5
compute 2 all saed 0.0251 Ni Kmax 1.70 Zone 0 0 0 c 0.05 0.05 0.05 manual echo

fix 1 all saed/vtk 1 1 1 c_1 file Al2O3_001.saed
fix 2 all saed/vtk 1 1 1 c_2 file Ni_000.saed
```

### 2.229.3 Description

Time average computed intensities from *compute saed* and write output to a file in the third generation vtk image data format for visualization directly in parallelized visualization software packages like ParaView and VisIt. Note that if no time averaging is done, this command can be used as a convenient way to simply output diffraction intensities at a single snapshot.

To produce output in the image data vtk format ghost data is added outside the *Kmax* range assigned in the compute saed. The ghost data is assigned a value of -1 and can be removed setting a minimum isovolume of 0 within the visualization software. SAED images can be created by visualizing a spherical slice of the data that is centered at R_Ewald*[h k l]/norm([h k l]), where R_Ewald=1/lambda.

The group specified within this command is ignored. However, note that specified values may represent calculations performed by saed computes which store their own "group" definitions.

Fix saed/vtk is designed to work only with *compute saed* values, e.g.

```
compute 3 top saed 0.0251 Al O
fix saed/vtk 1 1 1 c_3 file Al2O3_001.saed
```

---

The *Nevery*, *Nrepeat*, and *Nfreq* arguments specify on what timesteps the input values will be used in order to contribute to the average. The final averaged quantities are generated on timesteps that are a multiple of *Nfreq*. The average is over *Nrepeat* quantities, computed in the preceding portion of the simulation every *Nevery* timesteps. *Nfreq* must be a multiple of *Nevery* and *Nevery* must be non-zero even if *Nrepeat* is 1. Also, the timesteps contributing to the average value cannot overlap, i.e. Nrepeat*Nevery can not exceed Nfreq.

For example, if Nevery=2, Nrepeat=6, and Nfreq=100, then values on timesteps 90,92,94,96,98,100 will be used to compute the final average on timestep 100. Similarly for timesteps 190,192,194,196,198,200 on timestep 200, etc. If Nrepeat=1 and Nfreq = 100, then no time averaging is done; values are simply generated on timesteps 100,200,etc.

---

The output for fix ave/time/saed is a file written with the third generation vtk image data formatting. The filename assigned by the *file* keyword is appended with _N.vtk where N is an index (0,1,2...) to account for multiple diffraction intensity outputs.

By default the header contains the following information (with example data):

```
# vtk DataFile Version 3.0 c_SAED
Image data set
ASCII
DATASET STRUCTURED_POINTS
DIMENSIONS 337 219 209
ASPECT_RATIO 0.00507953 0.00785161 0.00821458
ORIGIN -0.853361 -0.855826 -0.854316
POINT_DATA 15424827
SCALARS intensity float
LOOKUP_TABLE default
...data
```

In this example, kspace is sampled across a 337 x 219 x 209 point mesh where the mesh spacing is approximately 0.005, 0.007, and 0.008 inv(length) units in the k1, k2, and k3 directions, respectively. The data is shifted by -0.85, -0.85, -0.85 inv(length) units so that the origin will lie at 0, 0, 0. Here, 15,424,827 kspace points are sampled in total.

---

Additional optional keywords also affect the operation of this fix.

The *ave* keyword determines how the values produced every *Nfreq* steps are averaged with values produced on previous steps that were multiples of *Nfreq*, before they are accessed by another output command or written to a file.

If the *ave* setting is *one*, then the values produced on timesteps that are multiples of *Nfreq* are independent of each other; they are output as-is without further averaging.

If the *ave* setting is *running*, then the values produced on timesteps that are multiples of *Nfreq* are summed and averaged in a cumulative sense before being output. Each output value is thus the average of the value produced on that timestep with all preceding values. This running average begins when the fix is defined; it can only be restarted by deleting the fix via the *unfix* command, or by re-defining the fix by re-specifying it.

If the *ave* setting is *window*, then the values produced on timesteps that are multiples of *Nfreq* are summed and averaged within a moving "window" of time, so that the last M values are used to produce the output. E.g. if M = 3 and Nfreq

= 1000, then the output on step 10000 will be the average of the individual values on steps 8000,9000,10000. Outputs on early steps will average over less than M values if they are not available.

The *start* keyword specifies what timestep averaging will begin on. The default is step 0. Often input values can be 0.0 at time 0, so setting *start* to a larger value can avoid including a 0.0 in a running or windowed average.

The *file* keyword allows a filename to be specified. Every *Nfreq* steps, the vector of saed intensity data is written to a new file using the third generation vtk format. The base of each file is assigned by the *file* keyword and this string is appended with _N.vtk where N is an index (0,1,2...) to account for situations with multiple diffraction intensity outputs.

### 2.229.4 Restart, fix_modify, output, run start/stop, minimize info

This fix is part of the DIFFRACTION package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.229.5 Restrictions

The attributes for fix_saed_vtk must match the values assigned in the associated *compute_saed* command.

### 2.229.6 Related commands

*compute_saed*

### 2.229.7 Default

The option defaults are ave = one, start = 0, no file output.

---

**(Coleman)** Coleman, Spearot, Capolungo, MSMSE, 21, 055020 (2013).

## 2.230 fix set command

### 2.230.1 Syntax

```
fix ID group-ID set Nfreq rnflag set-args
```

- ID, group-ID are documented in *fix* command
- set = style name of this fix command
- Nfreq = reset per-atom properties every this many timesteps
- rnflag = 1 to reneighbor on next timestep, 0 to not
- set-args = identical to args for the *set* command

### 2.230.2 Examples

```
fix 10 all set 1 0 group all i_dump v_new
fix 10 all set 1 0 group all i_dump v_turnoff
```

### 2.230.3 Description

Added in version 12Jun2025.

Reset one or more properties of one or more atoms once every *Nfreq* steps during a simulation.

If the *rnflag* for reneighboring is set to 1, then a reneighboring will be triggered on the next timestep (since the fix set operation occurs at the end of the current timestep). This is important to do if this command changes per-atom properties that need to be communicated to ghost atoms. If this is not the case, an *rnflag* setting of 0 can be used; reneighboring will only be triggered on subsequent timesteps by the usual neighbor list criteria; see the *neigh_modify command*.

Here are two examples where an *rnflag* setting of 1 are needed. If a custom per-atom property is changed and the *fix property/atom* command to create the property used the *ghost yes* keyword. Or if per-atom charges are changed, all pair styles which compute Coulombic interactions require charge values for ghost atoms. In both these examples, the re-neighboring will trigger the changes in the owned atom properties to be immediately communicated to ghost atoms.

The arguments following *Nfreq* and *rnflag* are identical to those allowed for the *set* command, as in the examples above and below.

Note that the group-ID setting for this command is ignored. The syntax for the *set* arguments allows selection of which atoms have their properties reset.

This command can only be used to reset an atom property using a per-atom variable. This option in allowed by many, but not all, of the keyword/value pairs supported by the *set* command. The reason for this restriction is that if a per-atom variable is not used, this command will typically not change atom properties during the simulation.

The *set* command can be used with similar syntax to this command to reset atom properties once before or between simulations.

---

Here is an example of input script commands which will output atoms into a dump file only when their x-velocity crosses a threshold value *vthresh* for the first time. Their position and x-velocity will then be output every step for *twindow* timesteps.

```
variable        vthresh equal 2             # threshold velocity
variable        twindow equal 10            # dump for this many steps
#
# define custom property i_dump to store timestep threshold is crossed
#
fix             2 all property/atom i_dump
set             group all i_dump -1
#
# fix set command checks for threshold crossings every step
# resets i_dump from -1 to current timestep when crossing occurs
#
variable        start atom "vx > v_vthresh && i_dump == -1"
variable        new atom ternary(v_start,step,i_dump)
fix             3 all set 1 0 group all i_dump v_new
#
# dump command with thresh which enforces twindow
#
dump            1 all custom 1 tmp.dump id x y vx i_dump
variable        dumpflag atom "i_dump >= 0 && (step-i_dump) < v_twindow"
dump_modify     1 thresh v_dumpflag == 1
#
# run the simulation
```

(continues on next page)

(continued from previous page)

```
# final dump with all atom IDs which crossed threshold on which timestep
#
run             1000
write_dump      all custom tmp.dump.final id i_dump modify thresh i_dump >= 0
```

The tmp.dump.final file lists which atoms crossed the velocity threshold. This command will print the *twindow* timesteps when a specific atom ID (104 in this case) was output in the tmp.dump file:

```
% grep "^104 " tmp.dump
```

If these commands are used instead of the above, then an atom can cross the velocity threshold multiple times, and will be output for *twindow* timesteps each time. However the write_dump command is no longer useful.

```
variable        vthresh equal 2             # threshold velocity
variable        twindow equal 10            # dump for this many steps
#
# define custom property i_dump to store timestep threshold is crossed
#
fix             2 all property/atom i_dump
set             group all i_dump -1
#
# fix set command checks for threshold crossings every step
# resets i_dump from -1 to current timestep when crossing occurs
#
variable        start atom "vx > v_vthresh && i_dump == -1"
variable        turnon atom ternary(v_start,step,i_dump)
variable        stop atom "v_turnon >= 0 && (step-v_turnon) < v_twindow"
variable        turnoff atom ternary(v_stop,v_turnon,-1)
fix             3 all set 1 0 group all i_dump v_turnoff
#
# dump command with thresh which enforces twindow
#
dump            1 all custom 1 tmp.dump id x y vx i_dump
variable        dumpflag atom "i_dump >= 0 && (step-i_dump) < v_twindow"
dump_modify     1 thresh v_dumpflag == 1
#
# run the simulation
#
run             1000
```

### 2.230.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.230.5 Restrictions

none

### 2.230.6 Related commands

*set*

### 2.230.7 Default

none

## 2.231 fix setforce command

Accelerator Variants: *setforce/kk*

## 2.232 fix setforce/spin command

### 2.232.1 Syntax

```
fix ID group-ID setforce fx fy fz keyword value ...
```

- ID, group-ID are documented in *fix* command
- setforce = style name of this fix command
- fx,fy,fz = force component values
- any of fx,fy,fz can be a variable (see below)
- zero or more keyword/value pairs may be appended to args
- keyword = *region*

  ```
  region value = region-ID
    region-ID = ID of region atoms must be in to have added force
  ```

### 2.232.2 Examples

```
fix freeze indenter setforce 0.0 0.0 0.0
fix 2 edge setforce NULL 0.0 0.0
fix 1 edge setforce/spin 0.0 0.0 0.0
fix 2 edge setforce NULL 0.0 v_oscillate
```

### 2.232.3 Description

Set each component of force on each atom in the group to the specified values fx,fy,fz. This erases all previously computed forces on the atom, though additional fixes could add new forces. This command can be used to freeze certain atoms in the simulation by zeroing their force, either for running dynamics or performing an energy minimization. For dynamics, this assumes their initial velocity is also zero.

Any of the fx,fy,fz values can be specified as NULL which means do not alter the force component in that dimension.

Any of the 3 quantities defining the force components can be specified as an equal-style or atom-style *variable*, namely *fx*, *fy*, *fz*. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated each timestep, and its value used to determine the force component.

Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent force field.

Atom-style variables can specify the same formulas as equal-style variables but can also include per-atom values, such as atom coordinates. Thus it is easy to specify a spatially-dependent force field with optional time-dependence as well.

If the *region* keyword is used, the atom must also be in the specified geometric *region* in order to have force added to it.

---

Style *spin* suffix sets the components of the magnetic precession vectors instead of the mechanical forces. This also erases all previously computed magnetic precession vectors on the atom, though additional magnetic fixes could add new forces.

This command can be used to freeze the magnetic moment of certain atoms in the simulation by zeroing their precession vector.

All options defined above remain valid, they just apply to the magnetic precession vectors instead of the forces.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

**Note**

The region keyword is supported by Kokkos, but a Kokkos-enabled region must be used. See the region *region* command for more information.

---

### 2.232.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is setting the forces to the desired values; on all other levels, the force is set to 0.0 for the atoms in the fix group, so that setforce values are not counted multiple times. Default is to to override forces at the outermost level.

This fix computes a global 3-vector of forces, which can be accessed by various *output commands*. This is the total force on the group of atoms before the forces on individual atoms are changed by the fix. The vector values calculated by this fix are "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The forces due to this fix are imposed during an energy minimization, invoked by the *minimize* command, but you cannot set forces to any value besides zero when performing a minimization. Use the *fix addforce* command if you want to apply a non-zero force to atoms during a minimization.

### 2.232.5 Restrictions

Fix *setforce/spin* is part of the SPIN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.232.6 Related commands

*fix addforce*, *fix aveforce*

### 2.232.7 Default

none

## 2.233 fix settorque/atom command

### 2.233.1 Syntax

```
fix ID group-ID settorque/atom tx ty tz keyword value ...
```

- ID, group-ID are documented in *fix* command
- settorque/atom = style name of this fix command
- tx,ty,tz = torque component values
- any of tx,ty,tz can be a variable (see below)
- zero or more keyword/value pairs may be appended to args
- keyword = *region*

  *region* value = region-ID
    region-ID = ID of region atoms must be in to have added torque

### 2.233.2 Examples

```
fix freeze indenter settorque/atom 0.0 0.0 0.0
fix 2 edge settorque/atom NULL 0.0 0.0
fix 2 edge settorque/atom NULL 0.0 v_oscillate
```

### 2.233.3 Description

Added in version 10Dec2025.

This fix is intended to set the peratom torque of individual finite-size atoms in the fix group to the specified values. Unlike *fix addtorque/group*, it does not set a collective torque to a set of point particles.

Each component of torque on each atom in the group to the specified values *tx*, *ty*, *tz*. This erases all previously computed torques on the atom, though additional fixes could add new torques. This command can be used to freeze the rotation of certain atoms in the simulation by zeroing their torque, assuming their initial angular velocities are also zero.

Freezing both rotational and translational degrees of freedom can also be accomplished using *fix freeze*.

Any of the *tx*, *ty*, *tz* values can be specified as NULL which means do not alter the torque component in that dimension.

Any of the 3 quantities defining the torque components can be specified as an equal-style or atom-style *variable*, namely *tx*, *ty*, *tz*. If the value is a variable, it should be specified as *v_name*, where *name* is the variable name. In this case, the variable will be evaluated each timestep, and its value used to determine the torque component.

Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent torque field.

Atom-style variables can specify the same formulas as equal-style variables but can also include per-atom values, such as atom coordinates. Thus it is easy to specify a spatially-dependent torque field with optional time-dependence as well.

If the *region* keyword is used, the atom must also be in the specified geometric *region* in order to have torque added to it.

**Note**

This fix is not (currently) designed to be used with rigid fixes. While it will set the torque of all of the atoms in a rigid body as described above, there is not always an easy mapping between these peratom torques and the torque experienced by the body. The exception may be when it is used in conjunction with *fix setforce* to simultaneously zero forces and torques to freeze a rigid body.

---

### 2.233.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify* *respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is setting the torques to the desired values; on all other levels, the torque is set to 0.0 for the atoms in the fix group, so that settorque/atom values are not counted multiple times. Default is to to override torques at the outermost level.

This fix computes a global 3-vector of torques, which can be accessed by various *output commands*. This is the total torque on the group of atoms before the torques on individual atoms are changed by the fix. The vector values calculated by this fix are "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

### 2.233.5 Restrictions

Fix *settorque/atom* is part of the GRANULAR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.233.6 Related commands

*fix addforce*, *fix addtorque/atom*, *fix setforce*, *fix freeze*

### 2.233.7 Default

none

## 2.234 fix sgcmc command

### 2.234.1 Syntax

```
fix ID group-ID sgcmc every_nsteps swap_fraction temperature deltamu ...
```

- ID, group-ID are documented in *fix* command
- sgcmc = style name of this fix command

- every_nsteps = number of MD steps between MC cycles
- swap_fraction = fraction of a full MC cycle carried out at each call (a value of 1.0 will perform as many trial moves as there are atoms)
- temperature = temperature that enters Boltzmann factor in Metropolis criterion (usually the same as MD temperature)
- deltamu = *N-1* chemical potential differences $\mu_1 - \mu_2, \ldots, \mu_1 - \mu_N$ ($N$ is the number of atom types)
- Zero or more keyword/value pairs may be appended to fix definition line:

```
keyword = variance or randseed or window_moves or window_size
  variance kappa conc1 [conc2] ... [concN]
    kappa = variance constraint parameter
    c_2, c_3,..., c_N = N-1 target concentration fractions
  randseed N
    N = seed for pseudo random number generator
  window_moves N
    N = number of times sampling window is moved during one MC cycle
  window_size frac
    frac = size of sampling window (must be between 0.5 and 1.0)
  atomic/energy yes/no
    yes = use the atomic energy method to calculate energy changes
    no = use the default method to calculate energy changes
```

### 2.234.2 Examples

```
fix mc all sgcmc 50 0.1 400.0 -0.55
fix vc all sgcmc 20 0.2 700.0 -0.7 randseed 324234 variance 2000.0 0.05
fix 2  all sgcmc 20 0.1 700.0 -0.7 window_moves 20
```

### 2.234.3 Description

Added in version 22Dec2022.

This command allows to carry out parallel hybrid molecular dynamics/Monte Carlo (MD/MC) simulations using the algorithms described in *(Sadigh1)*. Simulations can be carried out in either the semi-grand canonical (SGC) or variance constrained semi-grand canonical (VC-SGC) ensemble *(Sadigh2)*. Only atom type swaps are performed by the SGCMC fix. Relaxations are accounted for by the molecular dynamics integration steps.

This fix can be used with standard multi-element EAM potentials (*pair styles eam/alloy or eam/fs*)

The SGCMC fix can handle Finnis/Sinclair type EAM potentials where $\rho(r)$ is atom-type specific, such that different elements can contribute differently to the total electron density at an atomic site depending on the identity of the element at that atomic site.

---

If this fix is applied, the regular MD simulation will be interrupted in defined intervals to carry out a fraction of a Monte Carlo (MC) cycle. The interval is set using the parameter *every_nsteps* which determines how many MD integrator steps are taken between subsequent calls to the MC routine.

It is possible to carry out pure lattice MC simulations by setting *every_nsteps* to 1 and not defining an integration fix such as NVE, NPT etc. In that case, the particles will not move and only the MC routine will be called to perform atom type swaps.

The parameter *swap_fraction* determines how many MC trial steps are carried out every time the MC routine is entered. It is measured in units of full MC cycles where one full cycle, *swap_fraction=1*, corresponds to as many MC trial steps as there are atoms.

---

The parameter *temperature* specifies the temperature that is used to evaluate the Metropolis acceptance criterion. While it usually should be set to the same value as the MD temperature there are cases when it can be useful to use two different values for at least part of the simulation, e.g., to speed up equilibration at low temperatures.

---

The parameter *deltamu* is used to set the chemical potential differences in the SGC MC algorithm (see Eq. 16 in *Sadigh1*). The *N-1* differences are defined as $\mu_1 - \mu_2, \ldots, \mu_1 - \mu_N$, where $N$ is the number of atom types.

---

The variance-constrained SGC MC algorithm is activated if the keyword *variance* is used. In that case the fix parameter *deltamu* determines the effective average constraint in the parallel VC-SGC MC algorithm (parameter $\delta\mu_0$ in Eq. (20) of *Sadigh1*). The parameter *kappa* specifies the variance constraint (see Eqs. (20-21) in *Sadigh1*). The parameter *conc* sets the *N-1* target atomic concentration fractions (parameter $c_0$ in Eqs. (20-21) of *Sadigh1*) $0 \leq c_2, \ldots, c_N \leq 1$, with $c_1 = 1 - \Sigma_{i=2}^{N} c_i$. When the simulation includes $N$ atom types (elements), *N-1* concentration values must be specified.

---

There are several technical parameters that can be set via optional flags.

*randseed* is expected to be a positive integer number and is used to initialize the random number generator on each processor.

*window_size* controls the size of the sampling window in a parallel MC simulation. The size has to lie between 0.5 and 1.0. Normally, this parameter should be left unspecified which instructs the code to choose the optimal window size automatically (see Sect. III.B and Figure 6 in *Sadigh1* for details).

The number of times the window is moved during a MC cycle is set using the parameter *window_moves* (see Sect. III.B in *Sadigh1* for details).

The *atomic/energy* keyword controls which method is used for calculating the energy change when atom types are swapped. A value of *no* uses the default method, see discussion below in Restrictions section. A value of *yes* uses the atomic energy method, if the method has been implemented for the LAMMPS energy model, otherwise LAMMPS will exit with an error message. So far this has only been implemented for EAM type potentials.

---

### 2.234.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to restart files.

The MC routine keeps track of the global concentration(s) as well as the number of accepted and rejected trial swaps during each MC step. These values are provided by the sgcmc fix in the form of a global vector that can be accessed by various *output commands* components of the vector represent the following quantities:

- 1 = The absolute number of accepted trial swaps during the last MC step
- 2 = The absolute number of rejected trial swaps during the last MC step
- 3 = Current global concentration *c_1* (= number of atoms of type 1 / total number of atoms)
- 4 = Current global concentration *c_2* (= number of atoms of type 2 / total number of atoms)
- ...
- N+2 = Current global concentration *c_N* (= number of atoms of type *N* / total number of atoms)

The vector values calculated by this fix are "intensive".

### 2.234.5 Restrictions

This fix is part of the MC package. It is only enabled if LAMMPS was built with that package. Since it also contains specific support for EAM potentials it also requires installing the MANYBODY package. See the *Build package* page for more info.

This fix style requires an *atom style* with per atom type masses.

The fix provides three methods for calculating the potential energy change due to atom type swaps. For EAM type potentials, the default method is a carefully optimized local energy change calculation that is part of the source code for this fix. It takes advantage of the specific computational and communication requirements of EAM. Customizing the local method to handle other energy models such as Tersoff has been done in earlier versions of this fix, but these cases are not supported in the public LAMMPS code. For all other LAMMPS energy models, the default method calculates the *total* potential energy of the system before and after each atom type swap. This method does not depend on the details of the energy model and so is guaranteed to be correct. It is also orders of magnitude slower than the custom EAM calculation. In addition, it can not be used with parallel execution i.e. only a single MPI process is allowed. The third method uses the *atomic/energy* keyword described above. This allows parallel execution and it is also a local calculation, making it only a bit slower than a fully-optimized local calculation. So far, this has been implemented for EAM type potentials. It is straightforward to extend this to other potentials, requiring adding an atomic energy method to the pair style.

---

### 2.234.6 Default

The optional parameters default to the following values:

- *randseed* = 324234
- *window_moves* = 8
- *window_size* = automatic
- *atomic/energy* = no

---

**(Sadigh1)** B. Sadigh, P. Erhart, A. Stukowski, A. Caro, E. Martinez, and L. Zepeda-Ruiz, Phys. Rev. B **85**, 184203 (2012)

**(Sadigh2)** B. Sadigh and P. Erhart, Phys. Rev. B **86**, 134204 (2012)

## 2.235 fix shake command

Accelerator Variants: *shake/kk*

## 2.236 fix rattle command

### 2.236.1 Syntax

```
fix ID group-ID style tol iter N constraint values ... [keyword args ...]
```

- ID, group-ID are documented in *fix* command
- style = shake or rattle = style name of this fix command
- tol = accuracy tolerance of SHAKE or RATTLE solution
- iter = max # of iterations in each SHAKE or RATTLE solution

- N = print SHAKE or RATTLE statistics every this many timesteps (0 = never)
- one or more constraint/value pairs are appended
- constraint = *b* or *a* or *t* or *m*

```
b values = one or more bond types (may use type labels)
a values = one or more angle types  (may use type labels)
t values = one or more atom types (may use type labels)
m value = one or more mass values
```

- zero or more keyword/value pairs may be appended
- keyword = *mol* or *kbond* or *store*

```
mol value = template-ID
  template-ID = ID of molecule template specified in a separate molecule command
kbond value = force constant
  force constant = force constant used to apply a restraint force when used during␣
↪minimization
store value = yes or no
```

## 2.236.2 Examples

```
fix 1 sub shake 0.0001 20 10 b 4 19 a 3 5 2
fix 1 sub shake 0.0001 20 10 t 5 6 m 1.0 a 31
fix 1 sub shake 0.0001 20 10 t 5 6 m 1.0 a 31 mol myMol
fix 1 sub shake 0.0001 20 10 t 5 6 m 1.0 a 31 store yes
fix 1 sub rattle 0.0001 20 10 t 5 6 m 1.0 a 31
fix 1 sub rattle 0.0001 20 10 t 5 6 m 1.0 a 31 mol myMol
```

## 2.236.3 Description

Apply bond and angle constraints to specified bonds and angles in the simulation by either the SHAKE or RATTLE algorithms. This typically enables a longer timestep. The SHAKE or RATTLE constraint algorithms, however, can *only* be applied during molecular dynamics runs.

Changed in version 15Sep2022.

These fixes may now also be used during minimization. In that case the constraints are *approximated* by strong harmonic restraints.

**SHAKE vs RATTLE:**

The SHAKE algorithm was invented for schemes such as standard Verlet timestepping, where only the coordinates are integrated and the velocities are approximated as finite differences to the trajectories (*Ryckaert et al. (1977)*). If the velocities are integrated explicitly, as with velocity Verlet which is what LAMMPS uses as an integration method, a second set of constraining forces is required in order to eliminate velocity components along the bonds (*Andersen (1983)*).

In order to formulate individual constraints for SHAKE and RATTLE, focus on a single molecule whose bonds are constrained. Let Ri and Vi be the position and velocity of atom $i$ at time $n$, for $i$ =1,...,$N$, where $N$ is the number of sites of our reference molecule. The distance vector between sites $i$ and $j$ is given by

$$\mathbf{r}_{ij}^{n+1} = \mathbf{r}_j^n - \mathbf{r}_i^n$$

The constraints can then be formulated as

$$\mathbf{r}_{ij}^{n+1} \cdot \mathbf{r}_{ij}^{n+1} = d_{ij}^2 \quad \text{and}$$
$$\mathbf{v}_{ij}^{n+1} \cdot \mathbf{r}_{ij}^{n+1} = 0$$

The SHAKE algorithm satisfies the first condition, i.e. the sites at time *n+1* will have the desired separations Dij immediately after the coordinates are integrated. If we also enforce the second condition, the velocity components along the bonds will vanish. RATTLE satisfies both conditions. As implemented in LAMMPS, *fix rattle* uses fix shake for satisfying the coordinate constraints. Therefore the settings and optional keywords are the same for both fixes, and all the information below about SHAKE is also relevant for RATTLE.

**SHAKE:**

Each timestep the specified bonds and angles are reset to their equilibrium lengths and angular values via the SHAKE algorithm (*Ryckaert et al. (1977)*). This is done by applying an additional constraint force so that the new positions preserve the desired atom separations. The equations for the additional force are solved via an iterative method that typically converges to an accurate solution in a few iterations. The desired tolerance (e.g. 1.0e-4 = 1 part in 10000) and maximum # of iterations are specified as arguments. Setting the N argument will print statistics to the screen and log file about regarding the lengths of bonds and angles that are being constrained. Small delta values mean SHAKE is doing a good job.

---

In LAMMPS, only small clusters of atoms can be constrained. This is so the constraint calculation for a cluster can be performed by a single processor, to enable good parallel performance. A cluster is defined as a central atom connected to others in the cluster by constrained bonds. LAMMPS allows for the following kinds of clusters to be constrained: one central atom bonded to 1 or 2 or 3 atoms, or one central atom bonded to 2 others and the angle between the three atoms also constrained. This means water molecules or CH2 or CH3 groups may be constrained, but not all the C-C backbone bonds of a long polymer chain.

The *b* constraint lists bond types that will be constrained. The *t* constraint lists atom types. All bonds connected to an atom of the specified type will be constrained. The *m* constraint lists atom masses. All bonds connected to atoms of the specified masses will be constrained (within a fudge factor of MASSDELTA specified in `src/RIGID/fix_shake.cpp`). The *a* constraint lists angle types. If both bonds in the angle are constrained then the angle will also be constrained if its type is in the list.

Changed in version 29Aug2024.

The types may be given as type labels *only* if there is no atom, bond, or angle type label named *b*, *a*, *t*, or *m* defined in the simulation. If that is the case, type labels cannot be used as constraint type index with these two fixes, because the type labels would be incorrectly treated as a new type of constraint instead. Thus, LAMMPS will print a warning and type label handling is disabled and numeric types must be used.

For all constraints, a particular bond is only constrained if *both* atoms in the bond are in the group specified with the SHAKE or RATTLE fix.

The degrees-of-freedom removed by SHAKE bonds and angles are accounted for in temperature and pressure computations. Similarly, the SHAKE or RATTLE contribution to the pressure of the system (virial) is also accounted for.

**Note**

The *fix shake* command works by using the current forces on atoms to calculate an additional constraint force which when added will leave the atoms in positions that satisfy the SHAKE constraints (e.g. bond length) after the next time integration step. If you define fixes (e.g. *fix efield*) that add additional forces to the atoms **after** *fix shake* operates, then those forces will not be taken into account, and the time integration will typically not fully satisfy the SHAKE constraints. The solution for this is to make sure that *fix shake* is defined in your input script **after** any other fixes which add or change forces (to atoms that *fix shake* operates on).

---

The *mol* keyword should be used when other commands, such as *fix deposit* or *fix pour*, add molecules on-the-fly during a simulation, and you wish to constrain the new molecules via SHAKE. You specify a *template-ID* previously defined

using the *molecule* command, which reads a file that defines the molecule. You must use the same *template-ID* that the command adding molecules uses. The coordinates, atom types, special bond restrictions, and SHAKE info can be specified in the molecule file. See the *molecule* command for details. The only settings required to be in this file (by this command) are the SHAKE info of atoms in the molecule.

The *kbond* keyword sets the restraint force constant when *fix shake* or fix rattle are used during minimization. In that case the constraint algorithms are *not* applied and restraint forces are used instead to maintain the geometries similar to the constraints. How well the geometries are maintained and how quickly a minimization converges, depends largely on the force constant *kbond*: larger values will reduce the deviation from the desired geometry, but can also lead to slower convergence of the minimization or lead to instabilities depending on the minimization algorithm requiring to reduce the value of *timestep*. Even though the restraints will not preserve the bond lengths and angles as closely as the constraints during the MD, they are generally close enough so that the constraints will be fulfilled to the desired accuracy within a few MD steps following the minimization. The default value for *kbond* depends on the *units* setting and is 1.0e9*k_B.

Added in version 10Dec2025.

The *store* keyword controls whether the fix stores the constraint (or restraint) forces as a per-atom property.

During an MD *run*, the constraint forces are the forces on atoms due to the constraints after a constrained position update. Applying the SHAKE constraint *minimizes* those forces. By using *store yes* the original constraint forces on all atoms can be accessed as a per-atom array of the fix.

During a *minimization*, restraint forces are added to the atoms to keep the constrained bonds and angles close to their initial values. By using *store yes* those added forces can be accessed as a per-atom array of the fix.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

**RATTLE:**

The velocity constraints lead to a linear system of equations which can be solved analytically. The implementation of the algorithm in LAMMPS closely follows (*Andersen (1983)*).

**Note**

The *fix rattle* command modifies forces and velocities and thus should be defined after all other integration fixes in your input script. If you define other fixes that modify velocities or forces after *fix rattle* operates, then *fix rattle* will not take them into account and the overall time integration will typically not satisfy the RATTLE constraints. You can check whether the constraints work correctly by setting the value of RATTLE_DEBUG in `src/RIGID/fix_rattle.cpp` to 1 and recompiling LAMMPS.

---

### 2.236.4 Restart, fix_modify, output, run start/stop, minimize info

No information about these fixes is written to *binary restart files*.

Both fix *shake* and fix *rattle* behave differently during a minimization in comparison to a molecular dynamics run:

- When used during a minimization, the SHAKE or RATTLE constraint algorithms themselves are **not** applied. Instead the constraints are replaced by harmonic restraint forces. The energy and virial contributions due to the restraint forces are tallied into global and per-atom accumulators. The total restraint energy is also accessible as a global scalar property of the fix.
- During molecular dynamics runs, however, the fixes do apply the requested SHAKE or RATTLE constraint algorithms.

  The *fix_modify virial* option is supported by these fixes to add the contribution due to the added constraint forces on atoms to both the global pressure and per-atom stress of the system via the *compute pressure* and *compute stress/atom* commands. The former can be accessed by *thermodynamic output*.

  The default setting for this fix is *fix_modify virial yes*. No global or per-atom quantities are stored by these fixes for access by various *output commands* during an MD run. No parameter of these fixes can be used with the *start/stop* keywords of the *run* command.

### 2.236.5 Restrictions

These fixes are part of the RIGID package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

For computational efficiency, there can only be one shake or rattle fix defined in a simulation.

If you use a tolerance that is too large or a max-iteration count that is too small, the constraints will not be enforced very strongly, which can lead to poor energy conservation. You can test for this in your system by running a constant NVE simulation with a particular set of SHAKE parameters and monitoring the energy versus time.

SHAKE or RATTLE *cannot* not be used to constrain an angle at 180 degrees (e.g. a linear CO2 molecule). This causes a divergence when solving the constraint equations numerically. You can use *fix rigid or fix rigid/small* instead to simulate rigid linear molecules.

When used during minimization choosing a too large value of the *kbond* can make minimization very inefficient and also cause stability problems with some minimization algorithms. Sometimes those can be avoided by reducing the *timestep*.

### 2.236.6 Related commands

*fix rigid*, *fix ehex*, *fix nve/manifold/rattle*

### 2.236.7 Default

kbond = 1.0e9*k_B, store = no

---

**(Ryckaert)** J.-P. Ryckaert, G. Ciccotti and H. J. C. Berendsen, J of Comp Phys, 23, 327-341 (1977).

**(Andersen)** H. Andersen, J of Comp Phys, 52, 24-34 (1983).

## 2.237 fix shardlow command

Accelerator Variants: *shardlow/kk*

### 2.237.1 Syntax

```
fix ID group-ID shardlow
```

- ID, group-ID are documented in *fix* command
- shardlow = style name of this fix command

### 2.237.2 Examples

```
fix 1 all shardlow
```

### 2.237.3 Description

Specifies that the Shardlow splitting algorithm (SSA) is to be used to integrate the DPD equations of motion. The SSA splits the integration into a stochastic and deterministic integration step. The fix *shardlow* performs the stochastic integration step and must be used in conjunction with a deterministic integrator (e.g. *fix nve* or *fix nph*). The stochastic integration of the dissipative and random forces is performed prior to the deterministic integration of the conservative force. Further details regarding the method are provided in *(Lisal)* and *(Larentzos1)*.

The fix *shardlow* must be used with the *pair_style dpd/fdt* or *pair_style dpd/fdt/energy* command to properly initialize the fluctuation-dissipation theorem parameter(s) sigma (and kappa, if necessary).

Note that numerous variants of DPD can be specified by choosing an appropriate combination of the integrator and *pair_style dpd/fdt* command. DPD under isothermal conditions can be specified by using fix *shardlow*, fix *nve* and pair_style *dpd/fdt*. DPD under isoenergetic conditions can be specified by using fix *shardlow*, fix *nve* and pair_style *dpd/fdt/energy*. DPD under isobaric conditions can be specified by using fix shardlow, fix *nph* and pair_style *dpd/fdt*. DPD under isoenthalpic conditions can be specified by using fix shardlow, fix *nph* and pair_style *dpd/fdt/energy*. Examples of each DPD variant are provided in the examples/PACKAGES/dpd-react directory.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.237.4 Restrictions

This command is part of the DPD-REACT package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix is currently limited to orthogonal simulation cell geometries.

This fix must be used with an additional fix that specifies time integration, e.g. *fix nve* or *fix nph*.

The Shardlow splitting algorithm requires the sizes of the subdomain lengths to be larger than twice the cutoff+skin. Generally, the domain decomposition is dependent on the number of processors requested.

### 2.237.5 Related commands

*pair_style dpd/fdt*, *fix eos/cv*

### 2.237.6 Default

none

---

**(Lisal)** M. Lisal, J.K. Brennan, J. Bonet Avalos, J. Chem. Phys., 135, 204105 (2011).

**(Larentzos1)** J.P. Larentzos, J.K. Brennan, J.D. Moore, M. Lisal and W.D. Mattson, Comput. Phys. Commun., 185, 1987-1998 (2014).

**(Larentzos2)** J.P. Larentzos, J.K. Brennan, J.D. Moore, and W.D. Mattson, ARL-TR-6863, U.S. Army Research Laboratory, Aberdeen Proving Ground, MD (2014).

## 2.238 fix smd command

### 2.238.1 Syntax

```
fix ID group-ID smd type values keyword values
```

- ID, group-ID are documented in *fix* command
- smd = style name of this fix command
- mode = *cvel* or *cfor* to select constant velocity or constant force SMD

  ```
  cvel values = K vel
    K = spring constant (force/distance units)
    vel = velocity of pulling (distance/time units)
  cfor values = force
    force = pulling force (force units)
  ```

- keyword = *tether* or *couple*

  ```
  tether values = x y z R0
    x,y,z = point to which spring is tethered
    R0 = distance of end of spring from tether point (distance units)
  couple values = group-ID2 x y z R0
    group-ID2 = 2nd group to couple to fix group with a spring
    x,y,z = direction of spring, automatically computed with 'auto'
    R0 = distance of end of spring (distance units)
  ```

### 2.238.2 Examples

```
fix  pull    cterm smd cvel 20.0 -0.00005 tether NULL NULL 100.0 0.0
fix  pull    cterm smd cvel 20.0 -0.0001 tether 25.0 25 25.0 0.0
fix  stretch cterm smd cvel 20.0  0.0001 couple nterm auto auto auto 0.0
fix  pull    cterm smd cfor  5.0 tether 25.0 25.0 25.0 0.0
```

### 2.238.3 Description

**Fix smd is unmaintained**

Please note that *fix smd* is unmaintained and has multiple known issues. We recommend to use the equivalent functionality in either *fix colvars* or *fix plumed* instead, which are both actively maintained.

This fix implements several options of steered MD (SMD) as reviewed in *(Izrailev)*, which allows to induce conformational changes in systems and to compute the potential of mean force (PMF) along the assumed reaction coordinate *(Park)* based on Jarzynski's equality *(Jarzynski)*. This fix borrows a lot from *fix spring* and *fix setforce*.

You can apply a moving spring force to a group of atoms (*tether* style) or between two groups of atoms (*couple* style). The spring can then be used in either constant velocity (*cvel*) mode or in constant force (*cfor*) mode to induce transitions in your systems. When running in *tether* style, you may need some way to fix some other part of the system (e.g. via *fix spring/self*)

The *tether* style attaches a spring between a point at a distance of R0 away from a fixed point *x,y,z* and the center of mass of the fix group of atoms. A restoring force of magnitude K (R - R0) Mi / M is applied to each atom in the group where *K* is the spring constant, Mi is the mass of the atom, and M is the total mass of all atoms in the group. Note that *K* thus represents the total force on the group of atoms, not a per-atom force.

In *cvel* mode the distance R is incremented or decremented monotonously according to the pulling (or pushing) velocity. In *cfor* mode a constant force is added and the actual distance in direction of the spring is recorded.

The *couple* style links two groups of atoms together. The first group is the fix group; the second is specified by group-ID2. The groups are coupled together by a spring that is at equilibrium when the two groups are displaced by a vector in direction *x,y,z* with respect to each other and at a distance R0 from that displacement. Note that *x,y,z* only provides a direction and will be internally normalized. But since it represents the *absolute* displacement of group-ID2 relative to the fix group, (1,1,0) is a different spring than (-1,-1,0). For each vector component, the displacement can be described with the *auto* parameter. In this case the direction is re-computed in every step, which can be useful for steering a local process where the whole object undergoes some other change. When the relative positions and distance between the two groups are not in equilibrium, the same spring force described above is applied to atoms in each of the two groups.

For both the *tether* and *couple* styles, any of the x,y,z values can be specified as NULL which means do not include that dimension in the distance calculation or force application.

For constant velocity pulling (*cvel* mode), the running integral over the pulling force in direction of the spring is recorded and can then later be used to compute the potential of mean force (PMF) by averaging over multiple independent trajectories along the same pulling path.

### 2.238.4 Restart, fix_modify, output, run start/stop, minimize info

The fix stores the direction of the spring, current pulling target distance and the running PMF to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The *fix_modify virial* option is supported by this fix to add the contribution due to the added forces on atoms to both the global pressure and per-atom stress of the system via the *compute pressure* and *compute stress/atom* commands. The former can be accessed by *thermodynamic output*. The default setting for this fix is *fix_modify virial no*.

The *fix_modify respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is adding its forces. Default is the outermost level.

This fix computes a vector list of 7 quantities, which can be accessed by various *output commands*. The quantities in the vector are in this order: the x-, y-, and z-component of the pulling force, the total force in direction of the pull, the equilibrium distance of the spring, the distance between the two reference points, and finally the accumulated PMF (the sum of pulling forces times displacement).

The force is the total force on the group of atoms by the spring. In the case of the *couple* style, it is the force on the fix group (group-ID) or the negative of the force on the second group (group-ID2). The vector values calculated by this

fix are "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.238.5 Restrictions

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.238.6 Related commands

*fix drag*, *fix spring*, *fix spring/self* , *fix spring/rg*, *fix colvars*, *fix plumed*

### 2.238.7 Default

none

---

**(Izrailev)** Izrailev, Stepaniants, Isralewitz, Kosztin, Lu, Molnar, Wriggers, Schulten. Computational Molecular Dynamics: Challenges, Methods, Ideas, volume 4 of Lecture Notes in Computational Science and Engineering, pp. 39-65. Springer-Verlag, Berlin, 1998.

**(Park)** Park, Schulten, J. Chem. Phys. 120 (13), 5946 (2004)

**(Jarzynski)** Jarzynski, Phys. Rev. Lett. 78, 2690 (1997)

## 2.239 fix smd/adjust_dt command

### 2.239.1 Syntax

```
fix ID group-ID smd/adjust_dt arg
```

- ID, group-ID are documented in *fix* command
- smd/adjust_dt = style name of this fix command
- arg = *s_fact*

  *s_fact* = safety factor

### 2.239.2 Examples

```
fix 1 all smd/adjust_dt 0.1
```

### 2.239.3 Description

The fix calculates a new stable time increment for use with the SMD time integrators.

The stable time increment is based on multiple conditions. For the SPH pair styles, a CFL criterion (Courant, Friedrichs & Lewy, 1928) is evaluated, which determines the speed of sound cannot propagate further than a typical spacing between particles within a single time step to ensure no information is lost. For the contact pair styles, a linear analysis of the pair potential determines a stable maximum time step.

This fix inquires the minimum stable time increment across all particles contained in the group for which this fix is defined. An additional safety factor *s_fact* is applied to the time increment.

See this PDF guide to use Smooth Mach Dynamics in LAMMPS.

### 2.239.4 Restart, fix_modify, output, run start/stop, minimize info

Currently, no part of MACHDYN supports restarting nor minimization.

### 2.239.5 Restrictions

This fix is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.239.6 Related commands

*smd/tlsph_dt*

### 2.239.7 Default

none

## 2.240 fix smd/integrate_tlsph command

### 2.240.1 Syntax

```
fix ID group-ID smd/integrate_tlsph keyword values
```

- ID, group-ID are documented in *fix* command
- smd/integrate_tlsph = style name of this fix command
- zero or more keyword/value pairs may be appended
- keyword = *limit_velocity*

```
limit_velocity value = max_vel
  max_vel = maximum allowed velocity
```

### 2.240.2 Examples

```
fix 1 all smd/integrate_tlsph
fix 1 all smd/integrate_tlsph limit_velocity 1000
```

### 2.240.3 Description

The fix performs explicit time integration for particles which interact according with the Total-Lagrangian SPH pair style.

See this PDF guide to using Smooth Mach Dynamics in LAMMPS.

The *limit_velocity* keyword will control the velocity, scaling the norm of the velocity vector to max_vel in case it exceeds this velocity limit.

### 2.240.4 Restart, fix_modify, output, run start/stop, minimize info

Currently, no part of MACHDYN supports restarting nor minimization. This fix has no outputs.

### 2.240.5 Restrictions

This fix is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Changed in version 29Aug2024.

This fix is incompatible with deformation controls that remap velocity, for instance the *remap v* option of *fix deform*.

### 2.240.6 Related commands

*smd/integrate_ulsph*

### 2.240.7 Default

none

## 2.241 fix smd/integrate_ulsph command

### 2.241.1 Syntax

```
fix ID group-ID smd/integrate_ulsph keyword
```

- ID, group-ID are documented in *fix* command
- smd/integrate_ulsph = style name of this fix command
- zero or more keyword/value pairs may be appended
- keyword = adjust_radius or limit_velocity

**adjust_radius values = adjust_radius_factor min_nn max_nn**
adjust_radius_factor = factor which scale the smooth/kernel radius min_nn = minimum number of neighbors max_nn = maximum number of neighbors

**limit_velocity values = max_velocity**
max_velocity = maximum allowed velocity.

### 2.241.2 Examples

```
fix 1 all smd/integrate_ulsph adjust_radius 1.02 25 50
fix 1 all smd/integrate_ulsph limit_velocity 1000
```

### 2.241.3 Description

The fix performs explicit time integration for particles which interact with the updated Lagrangian SPH pair style.

See this PDF guide to using Smooth Mach Dynamics in LAMMPS.

The *adjust_radius* keyword activates dynamic adjustment of the per-particle SPH smoothing kernel radius such that the number of neighbors per particles remains within the interval *min_nn* to *max_nn*. The parameter *adjust_radius_factor* determines the amount of adjustment per timestep. Typical values are *adjust_radius_factor* =1.02, *min_nn* =15, and *max_nn* =20.

The *limit_velocity* keyword will control the velocity, scaling the norm of the velocity vector to max_vel in case it exceeds this velocity limit.

### 2.241.4 Restart, fix_modify, output, run start/stop, minimize info

Currently, no part of MACHDYN supports restarting nor minimization. This fix has no outputs.

### 2.241.5 Restrictions

This fix is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Changed in version 29Aug2024.

This fix is incompatible with deformation controls that remap velocity, for instance the *remap v* option of *fix deform*.

### 2.241.6 Related commands

none

### 2.241.7 Default

none

## 2.242 fix smd/move_tri_surf command

### 2.242.1 Syntax

```
fix ID group-ID smd/move_tri_surf keyword
```

- ID, group-ID are documented in *fix* command
- smd/move_tri_surf keyword = style name of this fix command
- keyword = **LINEAR* or **WIGGLE* or **ROTATE*

```
*LINEAR args = Vx Vy Vz
   Vx,Vy,Vz = components of velocity vector (velocity units), any component can be␣
↪specified as NULL
*WIGGLE args = Vx Vy Vz max_travel
   vx,vy,vz = components of velocity vector (velocity units), any component can be␣
↪specified as NULL
   max_travel = wiggle amplitude
*ROTATE args = Px Py Pz Rx Ry Rz period
   Px,Py,Pz = origin point of axis of rotation (distance units)
   Rx,Ry,Rz = axis of rotation vector
   period = period of rotation (time units)
```

### 2.242.2 Examples

```
fix 1 tool smd/move_tri_surf *LINEAR 20 20 10
fix 2 tool smd/move_tri_surf *WIGGLE 20 20 10
fix 2 tool smd/move_tri_surf *ROTATE 0 0 0 5 2 1
```

### 2.242.3 Description

This fix applies only to rigid surfaces read from .STL files via fix *smd/wall_surface* . It updates position and velocity for the particles in the group each timestep without regard to forces on the particles. The rigid surfaces can thus be moved along simple trajectories during the simulation.

The **LINEAR* style moves particles with the specified constant velocity vector V = (Vx,Vy,Vz). This style also sets the velocity of each particle to V = (Vx,Vy,Vz).

The **WIGGLE* style moves particles in an oscillatory fashion. Particles are moved along (vx, vy, vz) with constant velocity until a displacement of max_travel is reached. Then, the velocity vector is reversed. This process is repeated.

The **ROTATE* style rotates particles around a rotation axis R = (Rx,Ry,Rz) that goes through a point P = (Px,Py,Pz). The period of the rotation is also specified. This style also sets the velocity of each particle to (omega cross Rperp) where omega is its angular velocity around the rotation axis and Rperp is a perpendicular vector from the rotation axis to the particle.

See this PDF guide to using Smooth Mach Dynamics in LAMMPS.

### 2.242.4 Restart, fix_modify, output, run start/stop, minimize info

Currently, no part of MACHDYN supports restarting nor minimization. This fix has no outputs.

### 2.242.5 Restrictions

This fix is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.242.6 Related commands

*smd/triangle_mesh_vertices*, *smd/wall_surface*

### 2.242.7 Default

none

## 2.243 fix smd/setvel command

### 2.243.1 Syntax

```
fix ID group-ID smd/setvel vx vy vz keyword value ...
```

- ID, group-ID are documented in *fix* command
- smd/setvel = style name of this fix command
- vx,vy,vz = velocity component values
- any of vx,vy,vz can be a variable (see below)
- zero or more keyword/value pairs may be appended to args
- keyword = *region*

  *region* value = region-ID
    region-ID = ID of region particles must be in to have their velocities set

### 2.243.2 Examples

```
fix top_velocity top_group smd/setvel 1.0 0.0 0.0
```

### 2.243.3 Description

Set each component of velocity on each particle in the group to the specified values vx,vy,vz, regardless of the forces acting on the particle. This command can be used to impose velocity boundary conditions.

Any of the vx,vy,vz values can be specified as NULL which means do not alter the velocity component in that dimension.

This fix is indented to be used together with a time integration fix.

Any of the 3 quantities defining the velocity components can be specified as an equal-style or atom-style *variable*, namely *vx*, *vy*, *vz*. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated each timestep, and its value used to determine the force component.

Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent velocity field.

Atom-style variables can specify the same formulas as equal-style variables but can also include per-atom values, such as atom coordinates. Thus it is easy to specify a spatially-dependent velocity field with optional time-dependence as well.

If the *region* keyword is used, the particle must also be in the specified geometric *region* in order to have its velocity set by this command.

---

### 2.243.4 Restart, fix_modify, output, run start/stop, minimize info

Currently, no part of MACHDYN supports restarting nor minimization None of the *fix_modify* options are relevant to this fix.

This fix computes a global 3-vector of forces, which can be accessed by various *output commands*. This is the total force on the group of atoms. The vector values calculated by this fix are "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

### 2.243.5 Restrictions

This fix is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.243.6 Related commands

none

### 2.243.7 Default

none

## 2.244 fix smd/wall_surface command

### 2.244.1 Syntax

```
fix ID group-ID smd/wall_surface arg type mol-ID
```

- ID, group-ID are documented in *fix* command
- smd/wall_surface = style name of this fix command
- arg = *file*

*file* = file name of a triangular mesh in stl format

- type = particle type to be given to the new particles created by this fix
- mol-ID = molecule-ID to be given to the new particles created by this fix (must be >= 65535)

### 2.244.2 Examples

```
fix stl_surf all smd/wall_surface tool.stl 2 65535
```

### 2.244.3 Description

This fix creates reads a triangulated surface from a file in .STL format. For each triangle, a new particle is created which stores the barycenter of the triangle and the vertex positions. The radius of the new particle is that of the minimum circle which encompasses the triangle vertices.

The triangulated surface can be used as a complex rigid wall via the *smd/tri_surface* pair style. It is possible to move the triangulated surface via the *smd/move_tri_surf* fix style.

Immediately after a .STL file has been read, the simulation needs to be run for 0 timesteps in order to properly register the new particles in the system. See the "funnel_flow" example in the MACHDYN examples directory.

See this PDF guide to use Smooth Mach Dynamics in LAMMPS.

### 2.244.4 Dump image info

Added in version 11Feb2026.

Fix *smd/wall_surface* supports the *fix* keyword of *dump image*. The fix will pass geometry information about the wall particles to *dump image* so that they be included in the rendered image.

The color of the wall mesh object is by default that of the first atom type when using color styles "type" or "element". With color style "const" the default value of "white" can be changed using *dump_modify fcolor*. The transparency is by default fully opaque and can be changed with *dump_modify ftrans*.

The *fflag1* setting of *dump image fix* determines whether the wall will be rendered as a set of connected triangles (1) or as a mesh of cylinders (2).

When rendering triangles, the *fflag2* setting is ignored. When using a mesh of cylinders, the *fflag2* setting determines the diameter of the cylinders.

### 2.244.5 Restart, fix_modify, output, run start/stop, minimize info

Currently, no part of MACHDYN supports restarting nor minimization. This fix has no outputs.

### 2.244.6 Restrictions

This fix is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The molecule ID given to the particles created by this fix have to be equal to or larger than 65535.

Within each .STL file, only a single triangulated object must be present, even though the STL format allows for the possibility of multiple objects in one file.

### 2.244.7 Related commands

*smd/triangle_mesh_vertices*, *smd/move_tri_surf* , *smd/tri_surface*, *fix surface/global*, *fix surface/local*

### 2.244.8 Default

none

## 2.245 fix sph command

### 2.245.1 Syntax

```
fix ID group-ID sph
```

- ID, group-ID are documented in *fix* command
- sph = style name of this fix command

### 2.245.2 Examples

```
fix 1 all sph
```

### 2.245.3 Description

Perform time integration to update position, velocity, internal energy and local density for atoms in the group each timestep. This fix is needed to time-integrate SPH systems where particles carry internal variables such as internal energy. SPH stands for Smoothed Particle Hydrodynamics.

See this PDF guide to using SPH in LAMMPS.

**Note**

Please note that the SPH PDF guide file has not been updated for many years and thus does not reflect the current *syntax* of the SPH package commands. For that please refer to the LAMMPS manual.

### 2.245.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.245.5 Restrictions

This fix is part of the SPH package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.245.6 Related commands

*fix sph/stationary*

### 2.245.7 Default

none

## 2.246 fix sph/stationary command

### 2.246.1 Syntax

```
fix ID group-ID sph/stationary
```

- ID, group-ID are documented in *fix* command
- sph = style name of this fix command

### 2.246.2 Examples

```
fix 1 boundary sph/stationary
```

### 2.246.3 Description

Perform time integration to update internal energy and local density, but not position or velocity for atoms in the group each timestep. This fix is needed for SPH simulations to correctly time-integrate fixed boundary particles which constrain a fluid to a given region in space. SPH stands for Smoothed Particle Hydrodynamics.

See this PDF guide to using SPH in LAMMPS.

**Note**

Please note that the SPH PDF guide file has not been updated for many years and thus does not reflect the current *syntax* of the SPH package commands. For that please refer to the LAMMPS manual.

### 2.246.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.246.5 Restrictions

This fix is part of the SPH package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.246.6 Related commands

*fix sph*

### 2.246.7 Default

none

# 2.247 fix spring command

Accelerator Variant: *spring/kk*

## 2.247.1 Syntax

```
fix ID group-ID spring keyword values
```

- ID, group-ID are documented in *fix* command
- spring = style name of this fix command
- keyword = *tether* or *couple*

```
tether values = K x y z R0
  K = spring constant (force/distance units)
  x,y,z = point to which spring is tethered
  R0 = equilibrium distance from tether point (distance units)
couple values = group-ID2 K x y z R0
  group-ID2 = 2nd group to couple to fix group with a spring
  K = spring constant (force/distance units)
  x,y,z = direction of spring
  R0 = equilibrium distance of spring (distance units)
```

## 2.247.2 Examples

```
fix pull ligand spring tether 50.0 0.0 0.0 0.0 0.0
fix pull ligand spring tether 50.0 0.0 0.0 0.0 5.0
fix pull ligand spring tether 50.0 NULL NULL 2.0 3.0
fix 5 bilayer1 spring couple bilayer2 100.0 NULL NULL 10.0 0.0
fix longitudinal pore spring couple ion 100.0 NULL NULL -20.0 0.0
fix radial pore spring couple ion 100.0 0.0 0.0 NULL 5.0
```

## 2.247.3 Description

Apply a spring force to a group of atoms or between two groups of atoms. This is useful for applying an umbrella force to a small molecule or lightly tethering a large group of atoms (e.g. all the solvent or a large molecule) to the center of the simulation box so that it does not wander away over the course of a long simulation. It can also be used to hold the centers of mass of two groups of atoms at a given distance or orientation with respect to each other.

The *tether* style attaches a spring between a fixed point *x,y,z* and the center of mass of the fix group of atoms. The equilibrium position of the spring is R0. At each timestep the distance R from the center of mass of the group of atoms to the tethering point is computed, taking account of wrap-around in a periodic simulation box. A restoring force of magnitude K (R - R0) Mi / M is applied to each atom in the group where *K* is the spring constant, Mi is the mass of the atom, and M is the total mass of all atoms in the group. Note that *K* thus represents the spring constant for the total force on the group of atoms, not for a spring applied to each atom.

The *couple* style links two groups of atoms together. The first group is the fix group; the second is specified by group-ID2. The groups are coupled together by a spring that is at equilibrium when the two groups are displaced by a vector *x,y,z* with respect to each other and at a distance R0 from that displacement. Note that *x,y,z* is the equilibrium displacement of group-ID2 relative to the fix group. Thus (1,1,0) is a different spring than (-1,-1,0). When the relative positions and distance between the two groups are not in equilibrium, the same spring force described above is applied to atoms in each of the two groups.

For both the *tether* and *couple* styles, any of the x,y,z values can be specified as NULL which means do not include that dimension in the distance calculation or force application.

The first example above pulls the ligand towards the point (0,0,0). The second example holds the ligand near the surface of a sphere of radius 5 around the point (0,0,0). The third example holds the ligand a distance 3 away from the z=2 plane (on either side).

The fourth example holds 2 bilayers a distance 10 apart in z. For the last two examples, imagine a pore (a slab of atoms with a cylindrical hole cut out) oriented with the pore axis along z, and an ion moving within the pore. The fifth example holds the ion a distance of -20 below the z = 0 center plane of the pore (umbrella sampling). The last example holds the ion a distance 5 away from the pore axis (assuming the center-of-mass of the pore in x,y is the pore axis).

**Note**

The center of mass of a group of atoms is calculated in "unwrapped" coordinates using atom image flags, which means that the group can straddle a periodic boundary. See the *dump* doc page for a discussion of unwrapped coordinates. It also means that a spring connecting two groups or a group and the tether point can cross a periodic boundary and its length be calculated correctly.

### 2.247.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify* *energy* option is supported by this fix to add the energy stored in the spring to the global potential energy of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify energy no*.

The *fix_modify* *respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is adding its forces. Default is the outermost level.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the spring energy = 0.5 * K * r^2.

This fix also computes global 4-vector which can be accessed by various *output commands*. The first 3 quantities in the vector are xyz components of the total force added to the group of atoms by the spring. In the case of the *couple* style, it is the force on the fix group (group-ID) or the negative of the force on the second group (group-ID2). The fourth quantity in the vector is the magnitude of the force added by the spring, as a positive value if (r-R0) > 0 and a negative value if (r-R0) < 0. This sign convention can be useful when using the spring force to compute a potential of mean force (PMF).

The scalar and vector values calculated by this fix are "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The forces due to this fix are imposed during an energy minimization, invoked by the *minimize* command.

**Note**

If you want the spring energy to be included in the total potential energy of the system (the quantity being minimized), you MUST enable the *fix_modify* *energy* option for this fix.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.247.5 Restrictions

none

### 2.247.6 Related commands

*fix drag*, *fix spring/self*, *fix spring/rg*, *fix smd*

### 2.247.7 Default

none

## 2.248 fix spring/chunk command

### 2.248.1 Syntax

```
fix ID group-ID spring/chunk K chunkID comID
```

- ID, group-ID are documented in *fix* command
- spring/chunk = style name of this fix command
- K = spring constant for each chunk (force/distance units)
- chunkID = ID of *compute chunk/atom* command
- comID = ID of *compute com/chunk* command

### 2.248.2 Examples

```
fix restrain all spring/chunk 100 chunkID comID
```

### 2.248.3 Description

Apply a spring force to the center-of-mass (COM) of chunks of atoms as defined by the *compute chunk/atom* command. Chunks can be molecules or spatial bins or other groupings of atoms. This is a way of tethering each chunk to its initial COM coordinates.

The *chunkID* is the ID of a compute chunk/atom command defined in the input script. It is used to define the chunks. The *comID* is the ID of a compute com/chunk command defined in the input script. It is used to compute the COMs of each chunk.

At the beginning of the first *run* or *minimize* command after this fix is defined, the initial COM of each chunk is calculated and stored as R0m, where M is the chunk number. Thereafter, at every timestep (or minimization iteration), the current COM of each chunk is calculated as Rm. A restoring force of magnitude K (Rm - R0m) Mi / Mm is applied to each atom in each chunk where $K$ is the specified spring constant, Mi is the mass of the atom, and Mm is the total mass of all atoms in the chunk. Note that $K$ thus represents the spring constant for the total force on each chunk of atoms, not for a spring applied to each atom.

### 2.248.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the locations of the initial per-chunk center of mass coordinates to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the fix continues in an uninterrupted fashion. Since this fix depends on an instance of *compute chunk/atom* it will check when reading the restart if the chunk still exists and will define the same number of chunks. The restart data is only applied when the number of chunks matches. Otherwise the center of mass coordinates are recomputed.

The *fix_modify energy* option is supported by this fix to add the energy stored in all the springs to the global potential energy of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify energy no*.

The *fix_modify respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is adding its forces. Default is the outermost level.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the energy of all the springs, i.e. 0.5 * K * r^2 per-spring.

The scalar value calculated by this fix is "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The forces due to this fix are imposed during an energy minimization, invoked by the *minimize* command.

**Note**

If you want the spring energies to be included in the total potential energy of the system (the quantity being minimized), you MUST enable the *fix_modify energy* option for this fix.

### 2.248.5 Restrictions

none

### 2.248.6 Related commands

*fix spring*, *fix spring/self* , *fix spring/rg*

### 2.248.7 Default

none

## 2.249 fix spring/rg command

### 2.249.1 Syntax

```
fix ID group-ID spring/rg K RG0
```

- ID, group-ID are documented in *fix* command
- spring/rg = style name of this fix command
- K = harmonic force constant (force/distance units)
- RG0 = target radius of gyration to constrain to (distance units)

```
if RG0 = NULL, use the current RG as the target value
```

### 2.249.2 Examples

```
fix 1 protein spring/rg 5.0 10.0
fix 2 micelle spring/rg 5.0 NULL
```

### 2.249.3 Description

Apply a harmonic restraining force to atoms in the group to affect their central moment about the center of mass (radius of gyration). This fix is useful to encourage a protein or polymer to fold/unfold and also when sampling along the radius of gyration as a reaction coordinate (i.e. for protein folding).

The radius of gyration is defined as RG in the first formula. The energy of the constraint and associated force on each atom is given by the second and third formulas, when the group is at a different RG than the target value RG0.

$$
\begin{aligned}
{R_G}^2 &= \frac{1}{M}\sum_i^N m_i \left(x_i - \frac{1}{M}\sum_j^N m_j x_j\right)^2 \\
E &= K\left(R_G - R_{G0}\right)^2 \\
F_i &= 2K\frac{m_i}{M}\left(1 - \frac{R_{G0}}{R_G}\right)\left(x_i - \frac{1}{M}\sum_j^N m_j x_j\right)
\end{aligned}
$$

The ($x_i$ - center-of-mass) term is computed taking into account periodic boundary conditions, $m_i$ is the mass of the atom, and $M$ is the mass of the entire group. Note that K is thus a force constant for the aggregate force on the group of atoms, not a per-atom force.

If $R_{G0}$ is specified as NULL, then the RG of the group is computed at the time the fix is specified, and that value is used as the target.

### 2.249.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the currently used reference RG ($R_{G0}$) to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the fix continues in an uninterrupted fashion.

None of the *fix_modify* options are relevant to this fix.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the reference radius of gyration $R_{G0}$ used by the fix. energy change due to this fix. The scalar value calculated by this fix is "intensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

The *fix_modify* *respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is adding its forces. Default is the outermost level.

### 2.249.5 Restrictions

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.249.6 Related commands

*fix spring*, *fix spring/self* *fix drag*, *fix smd*

### 2.249.7 Default

none

## 2.250 fix spring/self command

Accelerator Variant: *spring/self/kk*

### 2.250.1 Syntax

```
fix ID group-ID spring/self K dir
```

- ID, group-ID are documented in *fix* command
- spring/self = style name of this fix command
- K = spring constant (force/distance units), can be a variable (see below)
- dir = xyz, xy, xz, yz, x, y, or z (optional, default: xyz)

### 2.250.2 Examples

```
fix tether boundary-atoms spring/self 10.0
fix var all spring/self v_kvar
fix zrest  move spring/self 10.0 z
```

### 2.250.3 Description

Apply a spring force independently to each atom in the group to tether it to its initial position. The initial position for each atom is its location at the time the fix command was issued. At each timestep, the magnitude of the force on each atom is -Kr, where r is the displacement of the atom from its current position to its initial position. The distance r correctly takes into account any crossings of periodic boundary by the atom since it was in its initial position.

With the (optional) dir flag, one can select in which direction the spring force is applied. By default, the restraint is applied in all directions, but it can be limited to the xy-, xz-, yz-plane and the x-, y-, or z-direction, thus restraining the atoms to a line or a plane, respectively.

The force constant *k* can be specified as an equal-style or atom-style *variable*. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated each time step, and its value(s) will be used as force constant for the spring force.

Equal-style variables can specify formulas with various mathematical functions and include *thermo_style* command keywords for the simulation box parameters, time step, and elapsed time. Thus, it is easy to specify a time-dependent force field.

Atom-style variables can specify the same formulas as equal-style variables but can also include per-atom values, such as atom coordinates. Thus, it is easy to specify a spatially-dependent force field with optional time-dependence as well.

### 2.250.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the original coordinates of tethered atoms to *binary restart files*, so that the spring effect will be the same in a restarted simulation. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The *fix_modify energy* option is supported by this fix to add the energy stored in the per-atom springs to the global potential energy of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify energy no*.

The *fix_modify* *respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is adding its forces. Default is the outermost level.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is an energy which is the sum of the spring energy for each atom, where the per-atom energy is 0.5 * K * r^2. The scalar value calculated by this fix is "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The forces due to this fix are imposed during an energy minimization, invoked by the *minimize* command.

**Note**

If you want the per-atom spring energy to be included in the total potential energy of the system (the quantity being minimized), you MUST enable the *fix_modify* *energy* option for this fix.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.250.5 Restrictions

The KOKKOS version, *fix spring/self/kk* may only be used with a constant value of K, not a variable.

### 2.250.6 Related commands

*fix drag*, *fix spring*, *fix smd*, *fix spring/rg*

### 2.250.7 Default

none

## 2.251 fix srd command

### 2.251.1 Syntax

```
fix ID group-ID srd N groupbig-ID Tsrd hgrid seed keyword value ...
```

- ID, group-ID are documented in *fix* command
- srd = style name of this fix command
- N = reset SRD particle velocities every this many timesteps
- groupbig-ID = ID of group of large particles that SRDs interact with

- Tsrd = temperature of SRD particles (temperature units)
- hgrid = grid spacing for SRD grouping (distance units)
- seed = random # seed (positive integer)
- zero or more keyword/value pairs may be appended
- keyword = *lamda* or *collision* or *overlap* or *inside* or *exact* or *radius* or *bounce* or *search* or *cubic* or *shift* or *tstat* or *rescale*

```
lamda value = mean free path of SRD particles (distance units)
collision value = noslip or slip = collision model
overlap value = yes or no = whether big particles may overlap
inside value = error or warn or ignore = how SRD particles which end up inside a␣
↪big particle are treated
exact value = yes or no
radius value = rfactor = scale collision radius by this factor
bounce value = Nbounce = max # of collisions an SRD particle can undergo in one␣
↪timestep
search value = sgrid = grid spacing for collision partner searching (distance units)
cubic values = style tolerance
  style = error or warn
  tolerance = fractional difference allowed (0 <= tol <= 1)
shift values = flag shiftseed
  flag = yes or no or possible = SRD bin shifting for better statistics
    yes = perform bin shifting each time SRD velocities are rescaled
    no = no shifting
    possible = shift depending on mean free path and bin size
  shiftseed = random # seed (positive integer)
tstat value = yes or no = thermostat SRD particles or not
unbiased value = yes or no = use profile-unbiased thermostat or not
rescale value = yes or no or rotate or collide = rescaling of SRD velocities
  yes = rescale during velocity rotation and collisions
  no = no rescaling
  rotate = rescale during velocity rotation, but not collisions
  collide = rescale during collisions, but not velocity rotation
```

### 2.251.2 Examples

```
fix 1 all srd 10 NULL 1.0 1.0 482984
fix 1 srd srd 10 big 1.0 0.25 482984
fix 1 srd srd 10 big 0.5 0.25 482984 collision slip search 0.5
```

### 2.251.3 Description

Treat a group of particles as stochastic rotation dynamics (SRD) particles that serve as a background solvent when interacting with big (colloidal) particles in groupbig-ID. The SRD formalism is described in *(Hecht)*. The same methodology is also called multi-particle collision dynamics (MPCD) in the literature.

The key idea behind using SRD particles as a cheap coarse-grained solvent is that SRD particles do not interact with each other, but only with the solute particles, which in LAMMPS can be spheroids, ellipsoids, or line segments, or triangles, or rigid bodies containing multiple spheroids or ellipsoids or line segments or triangles. The collision and rotation properties of the model imbue the SRD particles with fluid-like properties, including an effective viscosity. Thus simulations with large solute particles can be run more quickly, to measure solute properties like diffusivity and viscosity in a background fluid. The usual LAMMPS fixes for such simulations, such as *fix deform*, *fix viscosity*, and *fix nvt/sllod*, can be used in conjunction with the SRD model.

These 3 papers give more details on how the SRD model is implemented in LAMMPS. *(Petersen)* describes pure SRD fluid systems. *(Bolintineanu1)* describes models where pure SRD fluids interact with boundary walls. *(Bolintineanu2)* describes mixture models where large colloidal particles are solvated by an SRD fluid. See the `examples/srd` directory for sample input scripts.

This fix does two things:

1. It advects the SRD particles, performing collisions between SRD and big particles or walls every timestep, imparting force and torque to the big particles. Collisions also change the position and velocity of SRD particles.
2. It resets the velocity distribution of SRD particles via random rotations every N timesteps.

SRD particles have a mass, temperature, characteristic timestep $dt_{SRD}$, and mean free path between collisions ($\lambda$). The fundamental equation relating these 4 quantities is

$$\lambda = dt_{SRD} \sqrt{\frac{k_B T_{SRD}}{m}}$$

The mass $m$ of SRD particles is set by the *mass* command elsewhere in the input script. The SRD timestep $dt_{SRD}$ is N times the step $dt$ defined by the *timestep* command. Big particles move in the normal way via a time integration *fix* with a short timestep dt. SRD particles advect with a large timestep $dt_{SRD} \geq dt$.

If the *lamda* keyword is not specified, the SRD temperature $T_{SRD}$ is used in the above formula to compute $\lambda$. If the *lamda* keyword is specified, then the *Tsrd* setting is ignored and the above equation is used to compute the SRD temperature.

The characteristic length scale for the SRD fluid is set by *hgrid* which is used to bin SRD particles for purposes of resetting their velocities. Normally hgrid is set to be 1/4 of the big particle diameter or smaller, to adequately resolve fluid properties around the big particles.

$\lambda$ cannot be smaller than 0.6 * hgrid, else an error is generated (unless the *shift* keyword is used, see below). The velocities of SRD particles are bounded by Vmax, which is set so that an SRD particle will not advect further than $D_{max} = 4\lambda$ in $dt_{SRD}$. This means that roughly speaking, $D_{max}$ should not be larger than a big particle diameter, else SRDs may pass through big particles without colliding. A warning is generated if this is the case.

Collisions between SRD particles and big particles or walls are modeled as a lightweight SRD point particle hitting a heavy big particle of given diameter or a wall at a point on its surface and bouncing off with a new velocity. The collision changes the momentum of the SRD particle. It imparts a force and torque to the big particle. It imparts a force to a wall. Static or moving SRD walls are setup via the *fix wall/srd* command. For the remainder of this doc page, a collision of an SRD particle with a wall can be viewed as a collision with a big particle of infinite radius and mass.

The *collision* keyword sets the style of collisions. The *slip* style means that the tangential component of the SRD particle momentum is preserved. Thus a force is imparted to a big particle, but no torque. The normal component of the new SRD velocity is sampled from a Gaussian distribution at temperature *Tsrd*.

For the *noslip* style, both the normal and tangential components of the new SRD velocity are sampled from a Gaussian distribution at temperature *Tsrd*. Additionally, a new tangential direction for the SRD velocity is chosen randomly. This collision style imparts torque to a big particle. Thus a time integrator *fix* that rotates the big particles appropriately should be used.

---

The *overlap* keyword should be set to *yes* if two (or more) big particles can ever overlap. This depends on the pair potential interaction used for big-big interactions, or could be the case if multiple big particles are held together as rigid bodies via the *fix rigid* command. If the *overlap* keyword is *no* and big particles do in fact overlap, then SRD/big collisions can generate an error if an SRD ends up inside two (or more) big particles at once. How this error is treated is determined by the *inside* keyword. Running with *overlap* set to *no* allows for faster collision checking, so it should only be set to *yes* if needed.

The *inside* keyword determines how a collision is treated if the computation determines that the timestep started with the SRD particle already inside a big particle. If the setting is *error* then this generates an error message and LAMMPS stops. If the setting is *warn* then this generates a warning message and the code continues. If the setting is *ignore* then

no message is generated. One of the output quantities logged by the fix (see below) tallies the number of such events, so it can be monitored. Note that once an SRD particle is inside a big particle, it may remain there for several steps until it drifts outside the big particle.

The *exact* keyword determines how accurately collisions are computed. A setting of *yes* computes the time and position of each collision as SRD and big particles move together. A setting of *no* estimates the position of each collision based on the end-of-timestep positions of the SRD and big particle. If *overlap* is set to yes, the setting of the *exact* keyword is ignored since time-accurate collisions are needed.

The *radius* keyword scales the effective size of big particles. If big particles will overlap as they undergo dynamics, then this keyword can be used to scale down their effective collision radius by an amount *rfactor*, so that SRD particle will only collide with one big particle at a time. For example, in a Lennard-Jones system at a temperature of 1.0 (in reduced LJ units), the minimum separation between two big particles is as small as about 0.88 sigma. Thus an *rfactor* value of 0.85 should prevent dual collisions.

The *bounce* keyword can be used to limit the maximum number of collisions an SRD particle undergoes in a single timestep as it bounces between nearby big particles. Note that if the limit is reached, the SRD can be left inside a big particle. A setting of 0 is the same as no limit.

---

There are 2 kinds of bins created and maintained when running an SRD simulation. The first are "SRD bins" which are used to bin SRD particles and reset their velocities, as discussed above. The second are "search bins" which are used to identify SRD/big particle collisions.

The *search* keyword can be used to choose a search bin size for identifying SRD/big particle collisions. The default is to use the *hgrid* parameter for SRD bins as the search bin size. Choosing a smaller or large value may be more efficient, depending on the problem. But, in a statistical sense, it should not change the simulation results.

The *cubic* keyword can be used to generate an error or warning when the bin size chosen by LAMMPS creates SRD bins that are non-cubic or different than the requested value of *hgrid* by a specified *tolerance*. Note that using non-cubic SRD bins can lead to undetermined behavior when rotating the velocities of SRD particles, hence LAMMPS tries to protect you from this problem.

LAMMPS attempts to set the SRD bin size to exactly *hgrid*. However, there must be an integer number of bins in each dimension of the simulation box. Thus the actual bin size will depend on the size and shape of the overall simulation box. The actual bin size is printed as part of the SRD output when a simulation begins.

If the actual bin size in non-cubic by an amount exceeding the tolerance, an error or warning is printed, depending on the style of the *cubic* keyword. Likewise, if the actual bin size differs from the requested *hgrid* value by an amount exceeding the tolerance, then an error or warning is printed. The *tolerance* is a fractional difference. E.g. a tolerance setting of 0.01 on the shape means that if the ratio of any 2 bin dimensions exceeds (1 +/- tolerance) then an error or warning is generated. Similarly, if the ratio of any bin dimension with *hgrid* exceeds (1 +/- tolerance), then an error or warning is generated.

**Note**

The fix srd command can be used with simulations where the size and/or shape of the simulation box changes. This can be due to non-periodic boundary conditions or the use of fixes such as the *fix deform* or *fix wall/srd* commands to impose a shear on an SRD fluid or an interaction with an external wall. If the box size changes then the size of SRD bins must be recalculated every reneighboring. This is not necessary if only the box shape changes. This re-binning is always done so as to fit an integer number of bins in the current box dimension, whether it be a fixed, shrink-wrapped, or periodic boundary, as set by the *boundary* command. If the box size or shape changes, then the size of the search bins must be recalculated every reneighboring. Note that changing the SRD bin size may alter the properties of the SRD fluid, such as its viscosity.

The *shift* keyword determines whether the coordinates of SRD particles are randomly shifted when binned for purposes of rotating their velocities. When no shifting is performed, SRD particles are binned and the velocity distribution of

the set of SRD particles in each bin is adjusted via a rotation operator. This is a statistically valid operation if SRD particles move sufficiently far between successive rotations. This is determined by their mean-free path $\lambda$. If $\lambda$ is less than 0.6 of the SRD bin size, then shifting is required. A shift means that all of the SRD particles are shifted by a vector whose coordinates are chosen randomly in the range [-1/2 bin size, 1/2 bin size]. Note that all particles are shifted by the same vector. The specified random number *shiftseed* is used to generate these vectors. This operation sufficiently randomizes which SRD particles are in the same bin, even if $\lambda$ is small.

If the *shift* flag is set to *no*, then no shifting is performed, but bin data will be communicated if bins overlap processor boundaries. An error will be generated if $\lambda < 0.6$ of the SRD bin size. If the *shift* flag is set to *possible*, then shifting is performed only if $\lambda < 0.6$ of the SRD bin size. A warning is generated to let you know this is occurring. If the *shift* flag is set to *yes* then shifting is performed regardless of the magnitude of $\lambda$. Note that the *shiftseed* is not used if the *shift* flag is set to *no*, but must still be specified. Note that shifting of SRD coordinates requires extra communication, hence it should not normally be enabled unless required.

The *tstat* keyword will thermostat the SRD particles to the specified *Tsrd*. This is done every N timesteps, during the velocity rotation operation, by rescaling the thermal velocities of particles in each SRD bin to the desired temperature. Note that collisions of SRD particles with big particles or walls have a thermostatting effect on the colliding particles, so it may not be necessary to thermostat the SRD particles on a bin by bin basis in that case.

Added in version 10Dec2025.

The *unbiased* keyword controls how the thermostat operates if there is a streaming velocity associated with the system, e.g. due to use of the *fix deform* command to perform a simulation undergoing shear. The default case, *no*, is profile-biased: velocities relative to the mean velocity of the bin are rescaled, and then the mean velocity of each bin is set to the position-dependent streaming velocity, based on the coordinates of the center of the SRD bin. This enforces a linear velocity profile. With *yes*, after rescaling, the mean velocity of the bin is not changed, which renders the thermostat profile-unbiased. Note that for streaming simulations, if no thermostatting is performed (the default), it may take a long time for the SRD fluid to come to equilibrium with a velocity profile that matches the simulation box deformation.

The *rescale* keyword enables rescaling of an SRD particle's velocity if it would travel more than 4 mean-free paths in an SRD timestep. If an SRD particle exceeds this velocity it is possible it will be lost when migrating to other processors or that collisions with big particles will be missed, either of which will generate errors. Thus the safest mode is to run with rescaling enabled. However rescaling removes kinetic energy from the system (the particle's velocity is reduced). The latter will not typically be a problem if thermostatting is enabled via the *tstat* keyword or if SRD collisions with big particles or walls effectively thermostat the system. If you wish to turn off rescaling (on is the default), e.g. for a pure SRD system with no thermostatting so that the temperature does not decline over time, the *rescale* keyword can be used. The *no* value turns rescaling off during collisions and the per-bin velocity rotation operation. The *collide* and *rotate* values turn it on for one of the operations and off for the other.

---

**Note**

This fix is normally used for simulations with a huge number of SRD particles relative to the number of big particles, e.g. 100 to 1. In this scenario, computations that involve only big particles (neighbor list creation, communication, time integration) can slow down dramatically due to the large number of background SRD particles.

Three other input script commands will largely overcome this effect, speeding up an SRD simulation by a significant amount. These are the *atom_modify first*, *neigh_modify include*, and *comm_modify group* commands. Each takes a group-ID as an argument, which in this case should be the group-ID of the big solute particles.

Additionally, when a *pair_style* for big/big particle interactions is specified, the *pair_coeff* command should be used to turn off big/SRD interactions, e.g. by setting their epsilon or cutoff length to 0.0.

The "delete_atoms overlap" command may be useful in setting up an SRD simulation to ensure there are no initial overlaps between big and SRD particles.

---

### 2.251.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix tabulates several SRD statistics which are stored in a vector of length 12, which can be accessed by various *output commands*. The vector values calculated by this fix are "intensive", meaning they do not scale with the size of the simulation. Technically, the first 8 do scale with the size of the simulation, but treating them as intensive means they are not scaled when printed as part of thermodynamic output.

These are the 12 quantities. All are values for the current timestep, except for quantity 5 and the last three, each of which are cumulative quantities since the beginning of the run.

(1) # of SRD/big collision checks performed

(2) # of SRDs which had a collision

(3) # of SRD/big collisions (including multiple bounces)

(4) # of SRD particles inside a big particle

(5) # of SRD particles whose velocity was rescaled to be < Vmax

(6) # of bins for collision searching

(7) # of bins for SRD velocity rotation

(8) # of bins in which SRD temperature was computed

(9) SRD temperature

(10) # of SRD particles which have undergone max # of bounces

(11) max # of bounces any SRD particle has had in a single step

(12) # of reneighborings due to SRD particles moving too far

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.251.5 Restrictions

This command can only be used if LAMMPS was built with the SRD package. See the *Build package* doc page for more info.

### 2.251.6 Related commands

*fix wall/srd*

### 2.251.7 Default

The option defaults are: *lamda* ($\lambda$) is inferred from *Tsrd*, collision = noslip, overlap = no, inside = error, exact = yes, radius = 1.0, bounce = 0, search = hgrid, cubic = error 0.01, shift = no, tstat = no, unbiased = no, and rescale = yes.

---

**(Hecht)** Hecht, Harting, Ihle, Herrmann, Phys Rev E, 72, 011408 (2005).

**(Petersen)** Petersen, Lechman, Plimpton, Grest, in' t Veld, Schunk, J Chem Phys, 132, 174106 (2010).

**(Bolintineanu1)** Bolintineanu, Lechman, Plimpton, Grest, Phys Rev E, 86, 066703 (2012).

**(Bolintineanu2)** Bolintineanu, Grest, Lechman, Pierce, Plimpton, Schunk, Comp Particle Mechanics, 1, 321-356 (2014).

# 2.252 fix store/force command

## 2.252.1 Syntax

```
fix ID group-ID store/force
```

- ID, group-ID are documented in *fix* command
- store/force = style name of this fix command

## 2.252.2 Examples

```
fix 1 all store/force
```

## 2.252.3 Description

Store the forces on atoms in the group at the point during each timestep when the fix is invoked, as described below. This is useful for storing forces before constraints or other boundary conditions are computed which modify the forces, so that unmodified forces can be *written to a dump file* or accessed by other *output commands* that use per-atom quantities.

This fix is invoked at the point in the velocity-Verlet timestepping immediately after *pair*, *bond*, *angle*, *dihedral*, *improper*, and *long-range* forces have been calculated. It is the point in the timestep when various fixes that compute constraint forces are calculated and potentially modify the force on each atom. Examples of such fixes are *fix shake*, *fix wall*, and *fix indent*.

**Note**

The order in which various fixes are applied which operate at the same point during the timestep, is the same as the order they are specified in the input script. Thus normally, if you want to store per-atom forces due to force field interactions, before constraints are applied, you should list this fix first within that set of fixes, i.e. before other fixes that apply constraints. However, if you wish to include certain constraints (e.g. fix shake) in the stored force, then it could be specified after some fixes and before others.

## 2.252.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix produces a per-atom array which can be accessed by various *output commands*. The number of columns for each atom is 3, and the columns store the x,y,z forces on each atom. The per-atom values be accessed on any timestep.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

## 2.252.5 Restrictions

none

## 2.252.6 Related commands

*fix store_state*

### 2.252.7 Default

none

## 2.253 fix store/state command

### 2.253.1 Syntax

```
fix ID group-ID store/state N input1 input2 ... keyword value ...
```

- ID, group-ID are documented in *fix* command
- store/state = style name of this fix command
- N = store atom attributes every N steps, N = 0 for initial store only
- input = one or more atom attributes

```
possible attributes = id, mol, type, mass,
                      x, y, z, xs, ys, zs, xu, yu, zu, xsu, ysu, zsu, ix, iy, iz,
                      vx, vy, vz, fx, fy, fz,
                      q, mux, muy, muz, mu,
                      radius, diameter, omegax, omegay, omegaz,
                      angmomx, angmomy, angmomz, tqx, tqy, tqz,
                      c_ID, c_ID[I], f_ID, f_ID[I], v_name,
                      d_name, i_name, i2_name[I], d2_name[I],
```

```
id = atom ID
mol = molecule ID
type = atom type
mass = atom mass
x,y,z = unscaled atom coordinates
xs,ys,zs = scaled atom coordinates
xu,yu,zu = unwrapped atom coordinates
xsu,ysu,zsu = scaled unwrapped atom coordinates
ix,iy,iz = box image that the atom is in
vx,vy,vz = atom velocities
fx,fy,fz = forces on atoms
q = atom charge
mux,muy,muz = orientation of dipolar atom
mu = magnitued of dipole moment of atom
radius,diameter = radius.diameter of spherical particle
omegax,omegay,omegaz = angular velocity of spherical particle
angmomx,angmomy,angmomz = angular momentum of aspherical particle
tqx,tqy,tqz = torque on finite-size particles
c_ID = per-atom vector calculated by a compute with ID
c_ID[I] = Ith column of per-atom array calculated by a compute with ID
f_ID = per-atom vector calculated by a fix with ID
f_ID[I] = Ith column of per-atom array calculated by a fix with ID
v_name = per-atom vector calculated by an atom-style variable with name
i_name = custom integer vector with name
d_name = custom floating point vector with name
i2_name[I] = Ith column of custom integer array with name
d2_name[I] = Ith column of custom floating-point array with name
```

- zero or more keyword/value pairs may be appended

- keyword = *com* or *history* or *thresh*

```
com value = yes or no
history values = Nevery Nrepeat Nfreq
  Nevery = accumulate atom attributes once every this many steps
  Nrepeat = # of times to accumulate atom attributes
  Nfreq = make stored atom attributes (history) available every this many steps
thresh args = variable operator value
  variable = equal style or compatible variable reference
  operator = "<" or "<=" or ">" or ">=" or "==" or "!=" or "|^"
  value = numeric value to compare to
```

### 2.253.2 Examples

```
fix 1 all store/state 0 x y z
fix 1 all store/state 0 xu yu zu com yes
fix 2 all store/state 1000 vx vy vz
fix 2 all store/state 0 vx vy vz history 5 100 0
fix 2 all store/state 0 vx vy vz history 5 20 1000
fix 1 all store/state 0 x y z thresh v_dist < 5.0
```

### 2.253.3 Description

Changed in version 4Jul2026: The *history* keyword was added.

Define a fix that stores attributes for each atom in the group either once or for multiple recent timesteps by use of the optional *history* keyword.

If the optional keyword *history* is not used, then atom attributes are stored once. If $N > 0$, then the stored attributes will be updated once every $N$ steps. If $N = 0$, then the attributes are stored when the fix is defined (see the following Note) and never changed. The latter is a way of archiving an atom attribute for future use in a calculation or output.

**Note**

Only atom attributes specified by keywords like *xu* or *vy* or *radius* are initially stored immediately at the point in your input script when the fix is defined. Attributes specified by a compute, fix, or variable are not initially stored until the first run following the fix definition begins. This is because calculating those attributes may require quantities that are not defined in between runs.

If the optional keyword *history* is used, then the $N$ keyword must be specified as 0, and attributes are stored for multiple timesteps. This enables calculations involving the recent history of each atom. The Nevery, Nrepeat, and Nfreq settings determine when the history is stored and made available for calculations. There are 2 use cases, for Nfreq = 0 and Nfreq > 0. In both cases, attributes are stored Nrepeat times, once every Nevery steps. Nevery >= 1, Nrepeat > 1, and Nfreq = a multiple of Nevery (when Nfreq > 0) are required.

If Nfreq = 0, then the Nrepeat attributes are updated continuously and the attributes are accessible on any timestep which is a multiple of Nevery. For example, this command in the Examples section above:

```
fix 2 all store/state 0 vx vy vz history 5 100 0
```

will continuously store attributes once every 5 timesteps repeated 100 times (including the current timestep). On step 500, each atom stores attributes for steps 5,10,15, ... 500. On step 600, each atom stores attributes for steps 105,105,110, ... 600. For timesteps < 495, less than 100 attributes are necessarily stored.

If Nfreq is non-zero, it must be a multiple of Nevery, and the attributes are accessible only on timesteps which are a multiple of Nfreq. This can be useful to limit storage of atom attributes to only the time windows when the attributes are needed. For example, this command in the Examples section above:

```
fix 2 all store/state 0 vx vy vz history 5 20 1000
```

will store attributes once every 5 timesteps repeated 20 times (including the current timestep), but only preceding timesteps which are multiples of 1000. On step 1000, each atom stores attributes for steps 905,910,915, ... 1000. On step 2000, each atom stores attributes for steps 1905,1910,1915, ... 2000. Between timesteps 1000 to 1905, no attributes are stored, which can be more efficient if those attributes are not needed.

**Warning**

Specifying a large value for *Nrepeat* may require significant extra memory. Since the attribute values need to persist with each atom as it migrates to a new processor, it may also slow down a simulation.

---

The list of possible attributes is the same as that used by the *dump custom* command, which describes their meaning.

The requested values are stored in a per-atom vector or array as discussed below. Zeroes are stored for atoms not in the specified group.

If the *com* keyword is set to *yes* then the *xu*, *yu*, and *zu* inputs store the position of each atom relative to the center-of-mass of the group of atoms, instead of storing the absolute position.

Added in version 10Dec2025.

If the *thresh* keyword is used, data is only stored on steps where also the threshold condition following the keyword is met. The first argument *must* be a *variable* reference to an equal-style or compatible variable. The second argument is a logical operator and the third argument is a number. The choice of operators listed above are the usual comparison operators. The XOR operation (exclusive or) is also included as "|^". In this context, XOR means that if either the variable or the value is 0.0 and the other is non-zero, then the result is "true" and the threshold criterion is met. Otherwise it is not met.

The *thresh* keyword may be used *exactly once*. If a more complex threshold condition is needed or access to data from computes, fixes, or similar, this can be realized within the expression used when defining the referenced variable.

### 2.253.4 Restart, fix_modify, output, run start/stop, minimize info

If the optional *history* keyword is not used, this fix writes the per-atom values it stores to *binary restart files*, so that the values can be restored when a simulation is restarted. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

**Warning**

To enable the stored per-atom info to be restored from the restart file, this fix command has to be specified **exactly** the same way as before. LAMMPS will only check whether a fix is of the same style and has the same fix ID and in case of a match will then try to initialize the fix with the data stored in the binary restart file. If the fix store/state command does not match exactly (for example, if the list of values does not match), data can be corrupted or LAMMPS may crash. If the new fix uses the optional *history* keyword, the data stored in the restart file will be discarded.

If the *history* keyword is used, no information about this fix is written to *binary restart files*. Instead, you should simply specify a new fix in the restarted run to begin new accumulation of per-atom history.

None of the *fix_modify* options are relevant to this fix.

If the optional *history* keyword is not used, this fix produces a per-atom vector if a single input is specified. Or a per-atom array if multiple inputs are specified, where the number of columns for each atom is the number of inputs. These can be accessed by various *output commands*. These per-atom values can be accessed on any timestep (see the discussion of Noutput below).

If the *history* keyword is used, no per-atom vector or array is produced. Instead, the fix provides access to its multiple timesteps of stored per-atom history via its *extract()* method. The *compute property/atom* command is an example of a command which does this.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.253.5 Restrictions

none

### 2.253.6 Related commands

*dump custom*, *compute property/atom*, *fix property/atom*, *variable*

### 2.253.7 Default

The option default is com = no.

## 2.254 fix surface/global command

### 2.254.1 Syntax

```
fix ID group-ID surface/global input args input args ... model args model args ...␣
↪keyword value ...
```

- ID, group-ID are documented in *fix* command
- surface/global = style name of this fix command
- input = one or more input keywords can be specified

```
input args = source source-args
  source = mol or stl
    mol arg = template-ID
      template-ID = ID of molecule template specified in a separate molecule␣
↪command, which defines a set of triangles or lines
    stl args = stype smol stlfile
      stype = numeric type assigned to all triangles in STL file
      smol = numeric molecule ID assigned to all triangles in STL file
      stlfile = STL filename which defines a set of triangles
```

- model = one or more model keywords can be specified

```
model args = ptype stype fstyle fstyle-args
  ptype = numeric particle type (see asterisk form below)
  stype = numeric surface type (see asterisk form below)
  fstyle =  hooke or hooke/history or hertz/history or granular
    hooke or hooke/history or hertz/history args = Kn Kt gamma_n gamma_t xmu␣
↪dampflag (limit_damping)
```

```
        Kn = elastic constant for normal particle repulsion (force/distance units␣
↪or pressure units - see discussion below)
        Kt = elastic constant for tangential contact (force/distance units or␣
↪pressure units - see discussion below)
        gamma_n = damping coefficient for collisions in normal direction (1/time␣
↪units or 1/time-distance units - see discussion below)
        gamma_t = damping coefficient for collisions in tangential direction (1/
↪time units or 1/time-distance units - see discussion below)
        xmu = static yield criterion (unitless value between 0.0 and 1.0e4)
        dampflag = 0 or 1 if tangential damping force is excluded or included
        limit_damping = optional argument to prevent attractive interactions
    granular args = same syntax for args as pair_coeff command of pair_style␣
↪granular
```

- zero or more keyword/value pairs may be appended
- keyword = *smaxtype* or *smaxmol* or *flat* or *neighbor* or *temperature*

```
smaxtype value = smaxtype
  smaxtype = maximum surface type allowed
smaxmol value = smaxmol
  smaxmol = maximum surface molecule ID allowed
flat value = maxangle
  maxangle = maximum angle (degrees) between a pair of connected triangles/lines␣
↪for a flat connection
temperature value = Tsurf
  Tsurf = surface temperature (degrees Kelvin), required if model with heat is used
neighbor value = nsq or bin
  nsq = check all surfaces against particles for contacts
  bin = create a bin list to find contacts
```

### 2.254.2 Examples

```
molecule lines surf.line
fix 1 all surface/global input mol linesurf model 1 1 hooke 4000.0 NULL 100.0 NULL 0.5 1
fix 1 all surface/global input stl 1 1 object.stl model * 1 hooke/history 4000.0 NULL␣
↪100.0 NULL 0.5 1
```

### 2.254.3 Description

Added in version 4Jul2026.

Enable granular surfaces to be used as boundary conditions on particles in a granular simulation. Granular surfaces are defined as a set of triangles (3d) or a set of line segments (2d).

The *Howto granular surfaces* doc page gives an overview of granular surfaces of two types, *global* and *local*, with guidelines for how to use them.

This command is used for models with *global* surfaces. The *fix surface/local* command is used for models with *local* surfaces. As explained on the *Howto granular surfaces* doc page, *global* surfaces are most appropriate when there is a modest number of them. Each surface (triangle/line) can be of any size, even as large as a dimension of the simulation box. A copy of the list of *global* surfaces is stored by each processor.

*Global* triangles or line segments are not stored as particles and distributed across processors. Rather, they are stored by this fix and each processor stores a copy of all of them. This fix also computes forces between the global surfaces and all the particles.

*Global* surfaces can be defined in 2 ways, which correspond to the 2 options listed above for the *source* argument of the *input* keyword:

- via a molecule file(s), read by the *molecule* command
- via an STL file(s), read by this command

If triangles or lines were previously read in by the *molecule* command, the *source* argument of the *input* keyword is *mol* and its *template-ID* argument is the molecule template ID used with the *molecule* command. Note that a *molecule* command can read and assign several molecule files to the same template-ID. Each molecule file must define triangles or lines, not atoms. For multiple molecule files, the set of triangles or lines defined by this input option will be the union of the triangles and lines from all the molecule files. Note that each line/triangle in a molecule file is assigned a type and molecule ID.

An STL (stereolithography) file defines a set of triangles. For use with this command, the *source* argument of the *input* keyword is *stl*. The *stype* argument is the numeric type assigned to all the triangles from the file. Note that STL files do not contain types or other flags for each triangle. The *smol* argument is the numeric molecule ID assigned to all triangles in the file. The *stlfile* argument is the name of the STL file. It can be in text or binary format; this command auto-detects the format. One global triangle is created for each triangle in the STL file(s). Note that STL files cannot be used for 2d simulations since they only define triangles.

This Wikipedia page describes the format of both text and binary STL files. Binary STL files can be converted to ASCII for editing via the stl_bin2txt tool in the lammps/tools directory. Examples of text-based STL files are included in the examples/gransurf directory.

Note that this command allows for multiple uses of the *input* keyword, each with a *source* argument as either *mol* or *stl*. The surfaces used by this command are the union of the triangles and lines from all the input keywords.

Once all the global triangle/line particles are defined, this command calculates their connectivity. Two triangles are "connected" if they are in the same molecule and have a single corner point in common or an edge in common (2 corner points). Two line segments are "connected" if they are in the same molecule and they have an end point in common. More technical details on connectivity and its significance for granular surface simulations is given on *Howto granular surfaces* doc page. In brief, a pair of connected surfaces interact with a particle which contacts both of them simultaneously according to a set of rules which are designed to generate physically sensible forces on the particle.

Note that there is no requirement that all the surfaces be connected to one another. The surfaces can represent the surface of one or more independent objects. Particles in the specified group-ID interact with the surface when they are close enough to overlap (touch) one or more individual triangles or lines. Both sides of a triangle or line interact with particles. Thus a surface can be infinitely thin, e.g. the blade of a mixer. See the *Howto granular surfaces* doc page for restrictions on the geometry of a collection of triangles or lines.

The nature of individual surface/particle interactions are determined by the *model* keyword. Each use of the model keyword is applied to one or more particle types interacting with one or more surface types. The *ptype* argument is the particle type, *stype* is the surface type.

Either *ptype* and *stype* can be specified as a single numeric value. Or a wildcard asterisk can be used to specify multiple particle or surface types. This takes the form "*" or "*n" or "n*" or "m*n". If *N* is the number of particle or surface types, then an asterisk with no numeric values means all types from 1 to *N*. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to *N* (inclusive). A middle asterisk means all types from m to n (inclusive).

The model keywords must specify interactions for each particle type interacting with each surface type, otherwise an error is flagged. If use of the model keywords specifies an individual particle/surface type pair more than once, then the final specification is used.

The number of particle types is the number of atom types in the system. The number of surface types is determined by the maximum surface type in any of files read by the *input* keyword(s) or by the optional *smaxtype* keyword. The latter can be useful if the *fix_modify type/region* command (described below) is used to assign new types to surfaces after they are read in. As for particles, there is no requirement that triangles/lines exist for every surface type.

The *fstyle* argument (for force style) can be any of the styles defined by the *pair_style gran/** or the more general *pair_style granular* commands. Currently the options are *hooke*, *hooke/history*, or *hertz/history* for the former, and *granular* with all the possible options of the associated *pair_coeff* command for the latter. The equation for the force between a triangle/line and a particle touching it is the same as the corresponding equation on the *pair_style gran/** and *pair_style granular* doc pages, in the limit of one of the two particles going to infinite radius and mass (flat surface). Specifically, *delta* = *radius* - *r* = overlap of particle with triangle/line, *m_eff* = mass of particle, and the effective radius of contact = *RiRj/Ri+Rj* is set to the radius of the particle. See the *Howto granular surfaces* page for information on how overlaps and normal vectors are calculated based on the geometry of the surface and when friction is transferred between lines/triangles.

The parameters *Kn*, *Kt*, *gamma_n*, *gamma_t*, *xmu*, *dampflag*, and the optional keyword *limit_damping* have the same meaning and units as those specified with the *pair_style gran/** commands. This means a NULL can be used for either *Kt* or *gamma_t* as described on that page. If a NULL is used for *Kt*, then a default value is used where *Kt* = 2/7 *Kn*. If a NULL is used for *gamma_t*, then a default value is used where *gamma_t* = 1/2 *gamma_n*.

Note that the fix surface/global command can be used multiple times though it is not typically necessary to do so. Note that if it is used multiple times, the surfaces defined by the different commands will NOT be "connected" to each other in the manner described above or on the *Howto granular surfaces* doc page.

---

These are the optional keywords and values.

The *smaxtype* keyword sets the maximum value of a surface type which can be used. By default, this is the maximum type of any surface defined by the *input* keyword(s). If the *fix_modify type/region* command (described below) will be used later to change a surface type to a larger value than the default, then the *smaxtype* keyword can allow this.

The *smaxmol* keyword sets the maximum value of a surface molecule ID which can be used. By default, this is the maximum molecule ID of any surface defined by the *input* keyword(s). If the *fix_modify mol/region* command (described below) will be used later to change a surface type to a larger value than the default, then the *smaxmol* keyword can allow this.

The *flat* keyword sets a *maxangle* threshold for the angle (in degrees) between two connected surfaces (triangles or line segments) which will be treated as "flat" by the particle/surface interaction models. A flat connection means a single force will be applied to the particle even if it is contact with both surfaces simultaneously. See the *Howto granular surfaces* doc page for more details. The default for *maxangle* is one degree.

The *neighbor* keyword sets which algorithm is used build a neighbor list between surfaces and atoms. Akin to the *neighbor* command, the *bin* style (the default) spatially bins atoms and surfaces and constructs a neighbor list by checking atoms and surfaces in nearby bins. The *nsq* style instead loops through all atoms and then surfaces. Users are encouraged to test both options as, depending on the size and number of surfaces, one option may be faster than the other. As a rule of thumb, the *bin* option should become faster as the number of surfaces grow. However, if there are large number of surfaces, *fix surface/local* may also be more performant.

The *temperature* keyword is required if any of the granular models used includes a heat model which depends on the surface temperature. Otherwise it is ignored. Its *Tsurf* value is the temperature of the surface in degrees Kelvin.

---

### 2.254.4 Dump image info

Added in version 4Jul2026.

This wall fix supports the *fix* keyword of *dump image*. The fix will pass geometry information about the surface to *dump image* so that it will be included in the rendered image. For *2d systems*, the "surface" will be a collection of line segments displayed as cylinders. For *3d systems*, the surface will be a collection of triangles that are displayed as either as a wireframe, or triangles, or both.

The color of the wall is by default that of the atom type associated with the triangles using the color styles "type" or "element". With color style "const" the default value of "white" can be changed using *dump_modify fcolor*. The transparency is by default fully opaque and can be changed with *dump_modify ftrans*.

For 2d systems, *fflag1* sets the radius of the rendered cylinders.

For 3d systems, *fflag1* selects the render style (1 = triangles, 2 = wireframe, 3 = both), and *fflag2* sets the diameter of the wireframe.

---

### 2.254.5 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

This fix defines three new keywords for the *fix_modify* command, *move*, *type/region*, and *mol/region*. Because they are specific to this command, they are only described here, not on the *fix_modify* doc page. All keywords can be used multiple times. In the description that follows, a surface means a triangle (3d) or line segment (2d).

The *move* keyword can be used to make all or a subset of the surfaces move in a prescribed manner, similar to the *fix move* command. The *type/region* keyword can be used to change the types of surfaces which are within a geometric region. The *mol/region* keyword similarly changes the molecule ID of surfaces. Their syntax is as follows:

```
fix_modify fix-ID keyword values ...
```

- fix-ID = ID of the fix to modify
- keyword (specific to this fix) = *move* or *type/region* or *mol/region*

```
move values = smol mstyle args
   smol =  numeric surface molecule ID(s) (see comma-separated, asterisk form below)
   mstyle = none or linear or wiggle or rotate or transrot or variable
    none args = none
    linear args = Vx Vy Vz
    wiggle args = Ax Ay Az period
    rotate args = Px Py Pz Rx Ry Rz period
    transrot args = Vx Vy Vz Px Py Pz Rx Ry Rz period
    variable args = v_dx v_dy v_dz v_vx v_vy v_vz
type/region values = stype region-ID
  stype = numeric surface type
  region-ID = ID of a region previously defined by the region command
mol/region values = smol region-ID
  smol = numeric surface molecule ID
  region-ID = ID of a region previously defined by the region command
```

The *smol* argument can specify one or more surface molecule IDs. It must specify all the surface molecule IDs within a connected object(s). If an object is composed of surfaces of 2 or more molecules, it is an error to use the *move* keyword and not specify all those molecule IDs, since this would break the connections. Note that LAMMPS does NOT check that this requirement is met. It is likewise an error to use the *move* keyword multiple times to induce motion which overlaps surfaces in ways that violate the surface geometry restrictions explained on the *Howto granular surfaces* doc page. Again, LAMMPS does NOT check that this requirement is met.

The general format of *smol* is sm,sm,...,sm where one or more *sm* sub-arguments are separated by commas. A single *sm* sub-argument is either a single numeric value or contains a wildcard asterisk. The asterisk is used in place of or in conjunction with numeric arguments to specify multiple molecule values. This takes the form "*" or "*n" or "n*" or "m*n". If *N* is the number of molecules, then an asterisk with no numeric values means all molecules from 1 to *N*. A leading asterisk means all molecules from 1 to n (inclusive). A trailing asterisk means all molecules from n to *N* (inclusive). A middle asterisk means all molecules from m to n (inclusive).

The *mstyle* argument is one of the listed styles above. The *none* style turns off motion which was previously enabled, e.g. stops the rotation of an object. Again, the list of surface molecule IDs must include all the surfaces in a connected object. The other *move* styles and their effects on motion are the same as those defined by the *fix move* command. Their arguments are also the same as those documented by the *fix move* command.

The move *variable* style for this command is more limited than for the *fix move* command. Only an equal-style variable can be used, as defined by the *variable*. Atom-style variables cannot be used. Also, if both the displacement and velocity variables for a particular x,y,z component are specified as NULL, then no change is made to those position or velocity components of an individual triangle/line, which is different than the explanation given by the *fix move* command for individual particles.

Note that for *local* surfaces (see the *fix surface/local* doc page) the same motion operations can be performed using the *fix move* command with a group-ID defined by the *group* which includes the appropriate particle types for triangle and line-segment particles.

The *type/region* keyword can be used to re-assign surface types to surfaces after they have been initialized by the *input* keyword. This is most useful for STL triangles since STL files do not allow for assignment of types to individual triangles.

The *mol/region* keyword can be used to re-assign surface molecule IDs to surfaces after they have been initialized by the *input* keyword. This is most useful for STL triangles since STL files do not allow for assignment of molecules to individual triangles.

The *smol* argument is a single numeric value, which must be between 1 and maxmol inclusive. Maxmol is either the maximum molecule ID of all surfaces read in by the *input* keyword or the setting of the optional *smaxmol* keyword. The *region-ID* is the ID of a geometric region defined by the *region* command. Note that regions can be defined as either the inside or the outside of a geometric object, such as a sphere or block. The geometric center point of a triangle or line segment is used to determine where a surface is in the region or not. If it is, its molecule ID is reset to *smol*.

Examples for the various keywords are as follows:

```
fix_modify 1 move 2 rotate 0 0 0 0 0 1 25
fix_modify 1 move 1,3*5,8* rotate 0 0 0 0 0 1 25
fix_modify 1 type/region 3 myBlock
```

No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

## 2.254.6 Restrictions

This fix is part of the GRANSURF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Molecule IDs are not currently used by granular surface interactions, though they may be in the future. They are intended to be assigned uniquely to each inter-connected set of triangles/lines, as if each object were a "molecule". However, this is not required, and LAMMPS does not check that this is the case. LAMMPS will issue a warning if a set of inter-connected triangles/lines do not all have the same molecule ID, in case this was not intentional.

## 2.254.7 Related commands

*fix surface/local*, *pair_style surf/granular*, *fix smd/wall_surface*

## 2.254.8 Default

The keyword defaults are smaxtype (mol) = max type (molecule ID) of all surfaces defined by the input keyword(s), flat = one degree, neighbor = bin, temperature = none.

# 2.255 fix surface/local command

## 2.255.1 Syntax

```
fix ID group-ID surface/local input args input args ... keyword value ...
```

- ID, group-ID are documented in *fix* command
- surface/local = style name of this fix command
- input = zero or more input keywords can be specified

```
input args = source source-args
  source = mol or stl
     mol arg = template-ID
        template-ID = ID of molecule template specified in a
        separate molecule command, which defines a set of triangles or lines
     stl args = stype smol stlfile
        stype = numeric type assigned to all triangles in STL file
        smol = numeric molecule ID assigned to all triangles in STL file
        stlfile = STL filename which defines a set of triangles
```

- zero or more keyword/value pairs may be appended
- keyword = *flat*

```
flat value = maxangle
  maxangle = maximum angle (degrees) between a pair of connected triangles/lines␣
↪for a flat connection
```

## 2.255.2 Examples

```
read_data data.tris
fix 1 all surface/local

molecule tris surf.tri
fix 1 all surface/local input mol tris

fix 1 all surface/local input stl 1 1 surf.tri.stl
```

## 2.255.3 Description

Added in version 4Jul2026.

Enable granular surfaces to be used as boundary conditions on particles in a granular simulation. Granular surfaces are defined as a set of triangles (for 3d models) or a set of line segments (for 2d models).

The *Howto granular surfaces* doc page gives an overview of granular surfaces of two types, *global* and *local*, and gives guidelines for how they should be defined.

This command is used for models with *local* surfaces. The *fix surface/global* command is used for models with *global* surfaces. As explained on the *Howto granular surfaces* doc page, *local* surfaces are most appropriate when there is a large number of them. They should typically use surfaces (triangles/lines) whose size is no more than a few times larger than the spherical particles used in a granular model. There can be as many surfaces as needed to describe one or more physical surfaces at high resolution.

*Local* triangles or line segments are stored as triangle or line particles and are distributed across processors in the same manner as other particles, based on which processor's sub-domain the center of the triangle or line segment is inside

of. This is in contrast to *global* triangles or line segments, which are defined by the *fix surface/global* command and stored by the fix itself.

*Local* triangles/lines can be defined in 3 ways. The latter two correspond to the 2 options listed above for the *source* argument of the *input* keyword:

- via a data file, read by the *read_data* command
- via a molecule file(s), read by the *molecule* command
- via an STL file(s), read by this command

If triangles/lines were previously read in by the *read_data* command, then distributed triangles or lines already exist. As explained on the *Howto granular surfaces* doc page, these are "particles" as defined by the *atom_style tri or line* command, typically as a sub-style of the *atom_style hybrid* command.

Use of the *input* keyword adds new triangle/line particles to the system.

If triangles or lines were previously read in by the *molecule* command, the *source* keyword of the *input* keyword is *mol*. Its *template-ID* argument is the molecule template ID used with the *molecule* command. Note that a *molecule* command can read and assign several molecule files to the same template-ID. Each molecule file must define triangles or lines, not atoms. For multiple molecule files, the set of triangle or line particles defined by this input option will be the union of the triangles and lines from all the molecule files. Note that each line/triangle in a molecule file is assigned a type and molecule ID.

An STL (stereolithography) file defines a set of triangles. For use with this command, the *source* argument of the *input* keyword is *stl*. The *stype* argument is the numeric type assigned to all the triangles from the file. Note that STL files do not contain types or other flags for each triangle. The *smol* argument is the numeric molecule ID assigned to all triangles in the file. The *stlfile* argument is the name of the STL file. It can be in text or binary format; this command auto-detects the format. One triangle particle is created for each triangle in the STL file(s). Note that STL files cannot be used for 2d simulations since they only define triangles.

This Wikipedia page describes the format of both text and binary STL files. Binary STL files can be converted to ASCII for editing with the stl_bin2txt tool in the lammps/tools directory. Examples of text STL files are included in the examples/gransurf directory.

Note that this command allows for pre-defined triangle/line particles read in by the *read_data* command as well as multiple uses of the *input* keyword, each with a *source* argument as either *mol* or *stl*. The number of triangle/line particles operated on by this command will be the union of those already read by the *read_data* command and those specified by the *input* keywords.

Once all the distributed triangle/line particles are defined, this command calculates their connectivity. Two triangles are "connected" if they are in the same molecule and have a single corner point in common or an edge in common (2 corner points). Two line segments are "connected" if they are in the same molecule and they have an end point in common. More technical details on connectivity and its significance for granular surface simulations is given on *Howto granular surfaces* doc page. In brief, a pair of connected surfaces interact with a particle which contacts both of them simultaneously according to a set of rules which are designed to generate physically sensible forces on the particle.

Note that there is no requirement that all the surfaces be connected to one another. The surfaces can represent the surface of one or more independent objects. Particles interact with the surface when they are close enough to overlap (touch) one or more individual triangles or lines. Both sides of a triangle or line interact with particles. Thus a surface can be infinitely thin, e.g. the blade of a mixer. See the *Howto granular surfaces* doc page for restrictions on the geometry of a collection of triangles or lines.

The nature of individual surface/particle interactions are determined by the *pair_coeff* command which specifies interaction parameters for each pair of particle types. It is thus important to specify different types for granular particles and granular surfaces (triangle/line particles). Typically a granular simulation with local surfaces uses the *pair_style hybrid* command so that multiple sub-styles can be defined by the *pair_coeff* command, one or more for particle/particle interactions, and one or more for particle/surface interactions. Similar to the *fix surface/global* command this allows multiple particle/surface granular interaction models to be used, based on the surface particle types.

Note that although this fix does not store any triangles or line segments, it does store the connectivity information for the set of surface particles it operates on. It also communicates the connectivity info associated with surface particles when they migrate to new processors.

---

These are the optional keywords and values.

The *flat* keyword sets a *maxangle* threshold for the angle (in degrees) between two connected surfaces (triangles or line segments) which will be treated as "flat" by the particle/surface interaction models. A flat connection means a single force will be applied to the particle even if it is contact with both surfaces simultaneously. See the *Howto granular surfaces* doc page for more details. The default for *maxangle* is one degree.

Note that the *move* keyword used by the *fix surface/global* command (via *fix modify*) is not defined for this command. Instead, similar kinds of motion can be prescribed for *local* surfaces, using the *fix move* command directly since the *local* triangles or line segments are particles.

Also mote that the *smaxtype* and *smaxmol* keywords used by the *fix surface/global* command (also via *fix modify*) are not defined by this command. This is because local triangles and lines are already particles and their type is limited by the maximum number of particle types and their molecule IDs are limited by the standard limitations on the number of molecules.

### 2.255.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

---

### 2.255.5 Restrictions

This fix is part of the GRANSURF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.255.6 Related commands

*fix surface/global*, *pair_style surf/granular*, *fix smd/wall_surface*

### 2.255.7 Default

none

## 2.256 fix temp/berendsen command

Accelerator Variants: *temp/berendsen/kk*

### 2.256.1 Syntax

```
fix ID group-ID temp/berendsen Tstart Tstop Tdamp
```

- ID, group-ID are documented in *fix* command
- temp/berendsen = style name of this fix command
- Tstart,Tstop = desired temperature at start/end of run

```
Tstart can be a variable (see below)
```

- Tdamp = temperature damping parameter (time units)

### 2.256.2 Examples

```
fix 1 all temp/berendsen 300.0 300.0 100.0
```

### 2.256.3 Description

Reset the temperature of a group of atoms by using a Berendsen thermostat *(Berendsen)*, which rescales their velocities every timestep.

The thermostat is applied to only the translational degrees of freedom for the particles, which is an important consideration for finite-size particles which have rotational degrees of freedom are being thermostatted with this fix. The translational degrees of freedom can also have a bias velocity removed from them before thermostatting takes place; see the description below.

The desired temperature at each timestep is a ramped value during the run from *Tstart* to *Tstop*. The *Tdamp* parameter is specified in time units and determines how rapidly the temperature is relaxed. For example, a value of 100.0 means to relax the temperature in a timespan of (roughly) 100 time units (tau or fs or ps - see the *units* command).

*Tstart* can be specified as an equal-style *variable*. In this case, the *Tstop* setting is ignored. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated each timestep, and its value used to determine the target temperature.

**Note**

This thermostat will generate an error if the current temperature is zero at the end of a timestep. It cannot rescale a zero temperature.

Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent temperature.

**Note**

Unlike the *fix nvt* command which performs Nose/Hoover thermostatting AND time integration, this fix does NOT perform time integration. It only modifies velocities to effect thermostatting. Thus you must use a separate time integration fix, like *fix nve* to actually update the positions of atoms using the modified velocities. Likewise, this fix should not normally be used on atoms that also have their temperature controlled by another fix - e.g. by *fix nvt* or *fix langevin* commands.

See the *Howto thermostat* page for a discussion of different ways to compute temperature and perform thermostatting.

This fix computes a temperature each timestep. To do this, the fix creates its own compute of style “temp”, as if this command had been issued:

```
compute fix-ID_temp group-ID temp
```

See the *compute temp* command for details. Note that the ID of the new compute is the fix-ID + underscore + “temp”, and the group for the new compute is the same as the fix group.

Note that this is NOT the compute used by thermodynamic output (see the *thermo_style* command) with ID = *thermo_temp*. This means you can change the attributes of this fix's temperature (e.g. its degrees-of-freedom) via the *compute_modify* command or print this temperature during thermodynamic output via the *thermo_style custom* command using the appropriate compute-ID. It also means that changing attributes of *thermo_temp* will have no effect on this fix.

Like other fixes that perform thermostatting, this fix can be used with *compute commands* that remove a "bias" from the atom velocities. E.g. to apply the thermostat only to atoms within a spatial *region*, or to remove the center-of-mass velocity from a group of atoms, or to remove the x-component of velocity from the calculation.

This is not done by default, but only if the *fix_modify* command is used to assign a temperature compute to this fix that includes such a bias term. See the doc pages for individual *compute temp commands* to determine which ones include a bias. In this case, the thermostat works in the following manner: bias is removed from each atom, thermostatting is performed on the remaining thermal degrees of freedom, and the bias is added back in.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.256.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the cumulative global energy change to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the fix continues in an uninterrupted fashion.

The *fix_modify temp* option is supported by this fix. You can use it to assign a temperature *compute* you have defined to this fix which will be used in its thermostatting procedure, as described above. For consistency, the group used by this fix and by the compute should be the same.

The cumulative energy change in the system imposed by this fix is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* doc page for details.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the same cumulative energy change due to this fix described in the previous paragraph. The scalar value calculated by this fix is "extensive".

This fix can ramp its target temperature over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.256.5 Restrictions

This fix can be used with dynamic groups as defined by the *group* command. Likewise it can be used with groups to which atoms are added or deleted over time, e.g. a deposition simulation. However, the conservation properties of the thermostat and barostat are defined for systems with a static set of atoms. You may observe odd behavior if the atoms in a group vary dramatically over time or the atom count becomes very small.

### 2.256.6 Related commands

*fix nve*, *fix nvt*, *fix temp/rescale*, *fix langevin*, *fix_modify*, *compute temp*, *fix press/berendsen*

### 2.256.7 Default

none

---

**(Berendsen)** Berendsen, Postma, van Gunsteren, DiNola, Haak, J Chem Phys, 81, 3684 (1984).

## 2.257 fix temp/csvr command

## 2.258 fix temp/csld command

### 2.258.1 Syntax

```
fix ID group-ID temp/csvr Tstart Tstop Tdamp seed
fix ID group-ID temp/csld Tstart Tstop Tdamp seed
```

- ID, group-ID are documented in *fix* command
- temp/csvr or temp/csld = style name of this fix command
- Tstart,Tstop = desired temperature at start/end of run

  ```
  Tstart can be a variable (see below)
  ```

- Tdamp = temperature damping parameter (time units)
- seed = random number seed to use for white noise (positive integer)

### 2.258.2 Examples

```
fix 1 all temp/csvr 300.0 300.0 100.0 54324
fix 1 all temp/csld 100.0 300.0 10.0 123321
```

### 2.258.3 Description

Adjust the temperature with a canonical sampling thermostat that uses global velocity rescaling with Hamiltonian dynamics (*temp/csvr*) *(Bussi1)*, or Langevin dynamics (*temp/csld*) *(Bussi2)*. In the case of *temp/csvr* the thermostat is similar to the empirical Berendsen thermostat in *temp/berendsen*, but chooses the actual scaling factor from a suitably chosen (gaussian) distribution rather than having it determined from the time constant directly. In the case of *temp/csld* the velocities are updated to a linear combination of the current velocities with a gaussian distribution of velocities at the desired temperature. Both thermostats are applied every timestep.

The thermostat is applied to only the translational degrees of freedom for the particles, which is an important consideration for finite-size particles which have rotational degrees of freedom are being thermostatted with these fixes. The translational degrees of freedom can also have a bias velocity removed from them before thermostatting takes place; see the description below.

The desired temperature at each timestep is a ramped value during the run from *Tstart* to *Tstop*. The *Tdamp* parameter is specified in time units and determines how rapidly the temperature is relaxed. For example, a value of 100.0 means to relax the temperature in a timespan of (roughly) 100 time units (tau or fs or ps - see the *units* command).

*Tstart* can be specified as an equal-style *variable*. In this case, the *Tstop* setting is ignored. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated each timestep, and its value used to determine the target temperature.

Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent temperature.

**Note**

Unlike the *fix nvt* command which performs Nose/Hoover thermostatting AND time integration, these fixes do NOT perform time integration. They only modify velocities to effect thermostatting. Thus you must use a separate time integration fix, like *fix nve* to actually update the positions of atoms using the modified velocities. Likewise, these fixes should not normally be used on atoms that also have their temperature controlled by another fix - e.g. by *fix nvt* or *fix langevin* commands.

See the *Howto thermostat* page for a discussion of different ways to compute temperature and perform thermostatting.

These fixes compute a temperature each timestep. To do this, the fix creates its own compute of style "temp", as if this command had been issued:

```
compute fix-ID_temp group-ID temp
```

See the *compute temp* command for details. Note that the ID of the new compute is the fix-ID + underscore + "temp", and the group for the new compute is the same as the fix group.

Note that this is NOT the compute used by thermodynamic output (see the *thermo_style* command) with ID = *thermo_temp*. This means you can change the attributes of this fix's temperature (e.g. its degrees-of-freedom) via the *compute_modify* command or print this temperature during thermodynamic output via the *thermo_style custom* command using the appropriate compute-ID. It also means that changing attributes of *thermo_temp* will have no effect on this fix.

Like other fixes that perform thermostatting, this fix can be used with *compute commands* that remove a "bias" from the atom velocities. E.g. to apply the thermostat only to atoms within a spatial *region*, or to remove the center-of-mass velocity from a group of atoms, or to remove the x-component of velocity from the calculation.

This is not done by default, but only if the *fix_modify* command is used to assign a temperature compute to this fix that includes such a bias term. See the doc pages for individual *compute temp commands* to determine which ones include a bias. In this case, the thermostat works in the following manner: bias is removed from each atom, thermostatting is performed on the remaining thermal degrees of freedom, and the bias is added back in.

An important feature of these thermostats is that they have an associated effective energy that is a constant of motion. The effective energy is the total energy (kinetic + potential) plus the accumulated kinetic energy changes due to the thermostat. The latter quantity is the global scalar computed by these fixes. This feature is useful to check the integration of the equations of motion against discretization errors. In other words, the conservation of the effective energy can be used to choose an appropriate integration *timestep*. This is similar to the usual paradigm of checking the conservation of the total energy in the microcanonical ensemble.

---

### 2.258.4 Restart, fix_modify, output, run start/stop, minimize info

These fixes write the cumulative global energy change and the random number generator states to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the selected fix continues in an uninterrupted fashion. The random number generator state can only be restored when the number of processors remains unchanged from what is recorded in the restart file.

The *fix_modify* *temp* option is supported by these fixes. You can use it to assign a temperature *compute* you have defined to these fixes which will be used in its thermostatting procedure, as described above. For consistency, the group used by these fixes and by the compute should be the same.

The cumulative energy change in the system imposed by these fixes is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* doc page for details.

These fixes compute a global scalar which can be accessed by various *output commands*. The scalar is the same cumulative energy change due to this fix described in the previous paragraph. The scalar value calculated by this fix is "extensive".

These fixes can ramp their target temperature over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

These fixes are not invoked during *energy minimization*.

### 2.258.5 Restrictions

Fix *temp/csld* is not compatible with *fix shake*.

These fixes are part of the EXTRA-FIX package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

These fixes can be used with dynamic groups as defined by the *group* command. Likewise it can be used with groups to which atoms are added or deleted over time, e.g. a deposition simulation. However, the conservation properties of the thermostat and barostat are defined for systems with a static set of atoms. You may observe odd behavior if the atoms in a group vary dramatically over time or the atom count becomes very small.

### 2.258.6 Related commands

*fix nve*, *fix nvt*, *fix temp/rescale*, *fix langevin*, *fix_modify*, *compute temp*, *fix temp/berendsen*

### 2.258.7 Default

none

---

**(Bussi1)** Bussi, Donadio and Parrinello, J. Chem. Phys. 126, 014101(2007)

**(Bussi2)** Bussi and Parrinello, Phys. Rev. E 75, 056707 (2007)

## 2.259 fix temp/rescale command

Accelerator Variants: *temp/rescale/kk*

### 2.259.1 Syntax

```
fix ID group-ID temp/rescale N Tstart Tstop window fraction
```

- ID, group-ID are documented in *fix* command
- temp/rescale = style name of this fix command
- N = perform rescaling every N steps
- Tstart,Tstop = desired temperature at start/end of run (temperature units)

```
Tstart can be a variable (see below)
```

- window = only rescale if temperature is outside this window (temperature units)
- fraction = rescale to target temperature by this fraction

### 2.259.2 Examples

```
fix 3 flow temp/rescale 100 1.0 1.1 0.02 0.5
fix 3 boundary temp/rescale 1 1.0 1.5 0.05 1.0
fix 3 boundary temp/rescale 1 1.0 1.5 0.05 1.0
```

### 2.259.3 Description

Reset the temperature of a group of atoms by explicitly rescaling their velocities.

The rescaling is applied to only the translational degrees of freedom for the particles, which is an important consideration if finite-size particles which have rotational degrees of freedom are being thermostatted with this fix. The translational degrees of freedom can also have a bias velocity removed from them before thermostatting takes place; see the description below.

Rescaling is performed every N timesteps. The target temperature is a ramped value between the *Tstart* and *Tstop* temperatures at the beginning and end of the run.

**Note**

This thermostat will generate an error if the current temperature is zero at the end of a timestep it is invoked on. It cannot rescale a zero temperature.

*Tstart* can be specified as an equal-style *variable*. In this case, the *Tstop* setting is ignored. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated each timestep, and its value used to determine the target temperature.

Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent temperature.

Rescaling is only performed if the difference between the current and desired temperatures is greater than the *window* value. The amount of rescaling that is applied is a *fraction* (from 0.0 to 1.0) of the difference between the actual and desired temperature. E.g. if *fraction* = 1.0, the temperature is reset to exactly the desired value.

**Note**

Unlike the *fix nvt* command which performs Nose/Hoover thermostatting AND time integration, this fix does NOT perform time integration. It only modifies velocities to effect thermostatting. Thus you must use a separate time integration fix, like *fix nve* to actually update the positions of atoms using the modified velocities. Likewise, this fix should not normally be used on atoms that also have their temperature controlled by another fix - e.g. by *fix nvt* or *fix langevin* commands.

See the *Howto thermostat* page for a discussion of different ways to compute temperature and perform thermostatting.

This fix computes a temperature each timestep. To do this, the fix creates its own compute of style “temp”, as if one of this command had been issued:

```
compute fix-ID_temp group-ID temp
```

See the *compute temp* for details. Note that the ID of the new compute is the fix-ID + underscore + "temp", and the group for the new compute is the same as the fix group.

Note that this is NOT the compute used by thermodynamic output (see the *thermo_style* command) with ID = *thermo_temp*. This means you can change the attributes of this fix's temperature (e.g. its degrees-of-freedom) via the *compute_modify* command or print this temperature during thermodynamic output via the *thermo_style custom* command using the appropriate compute-ID. It also means that changing attributes of *thermo_temp* will have no effect on this fix.

Like other fixes that perform thermostatting, this fix can be used with *compute commands* that remove a "bias" from the atom velocities. E.g. to apply the thermostat only to atoms within a spatial *region*, or to remove the center-of-mass velocity from a group of atoms, or to remove the x-component of velocity from the calculation.

This is not done by default, but only if the *fix_modify* command is used to assign a temperature compute to this fix that includes such a bias term. See the doc pages for individual *compute temp commands* to determine which ones include a bias. In this case, the thermostat works in the following manner: bias is removed from each atom, thermostatting is performed on the remaining thermal degrees of freedom, and the bias is added back in.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.259.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the cumulative global energy change to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the fix continues in an uninterrupted fashion.

The *fix_modify* *temp* option is supported by this fix. You can use it to assign a temperature *compute* you have defined to this fix which will be used in its thermostatting procedure, as described above. For consistency, the group used by this fix and by the compute should be the same.

The cumulative energy change in the system imposed by this fix is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* doc page for details.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the same cumulative energy change due to this fix described in the previous paragraph. The scalar value calculated by this fix is "extensive".

This fix can ramp its target temperature over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.259.5 Restrictions

none

### 2.259.6 Related commands

*fix langevin*, *fix nvt*, *fix_modify*

### 2.259.7 Default

none

## 2.260 fix temp/rescale/eff command

### 2.260.1 Syntax

```
fix ID group-ID temp/rescale/eff N Tstart Tstop window fraction
```

- ID, group-ID are documented in *fix* command
- temp/rescale/eff = style name of this fix command
- N = perform rescaling every N steps
- Tstart,Tstop = desired temperature at start/end of run (temperature units)
- window = only rescale if temperature is outside this window (temperature units)
- fraction = rescale to target temperature by this fraction

### 2.260.2 Examples

```
fix 3 flow temp/rescale/eff 10 1.0 100.0 0.02 1.0
```

### 2.260.3 Description

Reset the temperature of a group of nuclei and electrons in the *electron force field* model by explicitly rescaling their velocities.

The operation of this fix is exactly like that described by the *fix temp/rescale* command, except that the rescaling is also applied to the radial electron velocity for electron particles.

### 2.260.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify* *temp* option is supported by this fix. You can use it to assign a temperature *compute* you have defined to this fix which will be used in its thermostatting procedure, as described above. For consistency, the group used by this fix and by the compute should be the same.

The cumulative energy change in the system imposed by this fix is included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* doc page for details.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the same cumulative energy change due to this fix described in the previous paragraph. The scalar value calculated by this fix is "extensive".

This fix can ramp its target temperature over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.260.5 Restrictions

This fix is part of the EFF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.260.6 Related commands

*fix langevin/eff*, *fix nvt/eff*, *fix_modify*, *fix temp rescale*,

### 2.260.7 Default

none

## 2.261 fix tfmc command

### 2.261.1 Syntax

```
fix ID group-ID tfmc Delta Temp seed keyword value
```

- ID, group-ID are documented in *fix* command
- tfmc = style name of this fix command
- Delta = maximal displacement length (distance units)
- Temp = imposed temperature of the system
- seed = random number seed (positive integer)
- zero or more keyword/arg pairs may be appended
- keyword = *com* or *rot*

```
com args = xflag yflag zflag
  xflag,yflag,zflag = 0/1 to exclude/include each dimension
rot args = none
```

### 2.261.2 Examples

```
fix 1 all tfmc 0.1 1000.0 159345
fix 1 all tfmc 0.05 600.0 658943 com 1 1 0
fix 1 all tfmc 0.1 750.0 387068 com 1 1 1 rot
```

### 2.261.3 Description

Perform uniform-acceptance force-bias Monte Carlo (fbMC) simulations, using the time-stamped force-bias Monte Carlo (tfMC) algorithm described in *(Mees)* and *(Bal)*.

One successful use case of force-bias Monte Carlo methods is that they can be used to extend the time scale of atomistic simulations, in particular when long time scale relaxation effects must be considered; some interesting examples are given in the review by *(Neyts)*. An example of a typical use case would be the modelling of chemical vapor deposition (CVD) processes on a surface, in which impacts by gas-phase species can be performed using MD, but subsequent relaxation of the surface is too slow to be done using MD only. Using tfMC can allow for a much faster relaxation of the surface, so that higher fluxes can be used, effectively extending the time scale of the simulation. (Such an alternating simulation approach could be set up using a *loop*.)

The initial version of tfMC algorithm in *(Mees)* contained an estimation of the effective time scale of such a simulation, but it was later shown that the speed-up one can gain from a tfMC simulation is system- and process-dependent, ranging

from none to several orders of magnitude. In general, solid-state processes such as (re)crystallization or growth can be accelerated by up to two or three orders of magnitude, whereas diffusion in the liquid phase is not accelerated at all. The observed pseudodynamics when using the tfMC method is not the actual dynamics one would obtain using MD, but the relative importance of processes can match the actual relative dynamics of the system quite well, provided *Delta* is chosen with care. Thus, the system's equilibrium is reached faster than in MD, along a path that is generally roughly similar to a typical MD simulation (but not necessarily so). See *(Bal)* for details.

Each step, all atoms in the selected group are displaced using the stochastic tfMC algorithm, which is designed to sample the canonical (NVT) ensemble at the temperature *Temp*. Although tfMC is a Monte Carlo algorithm and thus strictly speaking does not perform time integration, it is similar in the sense that it uses the forces on all atoms in order to update their positions. Therefore, it is implemented as a time integration fix, and no other fixes of this type (such as *fix nve*) should be used at the same time. Because velocities do not play a role in this kind of Monte Carlo simulations, instantaneous temperatures as calculated by *temperature computes* or *thermodynamic output* have no meaning: the only relevant temperature is the sampling temperature *Temp*. Similarly, performing tfMC simulations does not require setting a *timestep* and the *simulated time* as calculated by LAMMPS is meaningless.

The critical parameter determining the success of a tfMC simulation is *Delta*, the maximal displacement length of the lightest element in the system: the larger it is, the longer the effective time scale of the simulation will be (there is an approximately quadratic dependence). However, *Delta* must also be chosen sufficiently small in order to comply with detailed balance; in general values between 5 and 10 % of the nearest neighbor distance are found to be a good choice. For a more extensive discussion with specific examples, please refer to *(Bal)*, which also describes how the code calculates element-specific maximal displacements from *Delta*, based on the fourth root of their mass.

Because of the uncorrelated movements of the atoms, the center-of-mass of the fix group will not necessarily be stationary, just like its orientation. When the *com* keyword is used, all atom positions will be shifted (after every tfMC iteration) in order to fix the position of the center-of-mass along the included directions, by setting the corresponding flag to 1. The *rot* keyword does the same for the rotational component of the tfMC displacements after every iteration.

**Note**

the *com* and *rot* keywords should not be used if an external force is acting on the specified fix group, along the included directions. This can be either a true external force (e.g. through *fix wall*) or forces due to the interaction with atoms not included in the fix group. This is because in such cases, translations or rotations of the fix group could be induced by these external forces, and removing them will lead to a violation of detailed balance.

### 2.261.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

None of the *fix_modify* options are relevant to this fix.

This fix is not invoked during *energy minimization*.

### 2.261.5 Restrictions

This fix is part of the MC package. It is only enabled if LAMMPS was built with that package. See the *Build package* doc page for more info.

This fix is not compatible with *fix shake*.

### 2.261.6 Related commands

*fix gcmc*, *fix nvt*

### 2.261.7 Default

The option default is com = 0 0 0

---

**(Bal)** K. M Bal and E. C. Neyts, J. Chem. Phys. 141, 204104 (2014).

**(Mees)** M. J. Mees, G. Pourtois, E. C. Neyts, B. J. Thijsse, and A. Stesmans, Phys. Rev. B 85, 134301 (2012).

**(Neyts)** E. C. Neyts and A. Bogaerts, Theor. Chem. Acc. 132, 1320 (2013).

## 2.262 fix tgnvt/drude command

## 2.263 fix tgnpt/drude command

### 2.263.1 Syntax

```
fix ID group-ID style_name keyword values ...
```

- ID, group-ID are documented in *fix* command
- style_name = *tgnvt/drude* or *tgnpt/drude*
- one or more keyword/values pairs may be appended

```
keyword = temp iso or aniso or tri or x or y or z or xy or yz or xz or couple or␣
↪tchain or pchain or mtk or tloop or ploop or nreset or scalexy or scaleyz or␣
↪scalexz or flip or fixedpoint
  temp values = Tstart Tstop Tdamp Tdrude Tdamp_drude
    Tstart, Tstop = external temperature at start/end of run (temperature units)
    Tdamp = temperature damping parameter (time units)
    Tdrude = desired temperature of Drude oscillators (temperature units)
    Tdamp_drude = temperature damping parameter for Drude oscillators (time units)
  iso or aniso or tri values = Pstart Pstop Pdamp
    Pstart,Pstop = scalar external pressure at start/end of run (pressure units)
    Pdamp = pressure damping parameter (time units)
  x or y or z or xy or yz or xz values = Pstart Pstop Pdamp
    Pstart,Pstop = external stress tensor component at start/end of run (pressure␣
↪units)
    Pdamp = stress damping parameter (time units)
  couple = none or xyz or xy or yz or xz
  tchain value = N
    N = length of thermostat chain (1 = single thermostat)
  pchain value = N
    N length of thermostat chain on barostat (0 = no thermostat)
  mtk value = yes or no = add in MTK adjustment term or not
  tloop value = M
    M = number of sub-cycles to perform on thermostat
  ploop value = M
    M = number of sub-cycles to perform on barostat thermostat
  nreset value = reset reference cell every this many timesteps
  scalexy value = yes or no = scale xy with ly
```

```
scaleyz value = yes or no = scale yz with lz
scalexz value = yes or no = scale xz with lz
flip value = yes or no = allow or disallow box flips when it becomes highly skewed
fixedpoint values = x y z
  x,y,z = perform barostat dilation/contraction around this point (distance units)
```

### 2.263.2 Examples

```
comm_modify vel yes
fix 1 all tgnvt/drude temp 300.0 300.0 100.0 1.0 20.0
fix 1 water tgnpt/drude temp 300.0 300.0 100.0 1.0 20.0 iso 0.0 0.0 1000.0
fix 2 jello tgnpt/drude temp 300.0 300.0 100.0 1.0 20.0 tri 5.0 5.0 1000.0
fix 2 ice tgnpt/drude temp 250.0 250.0 100.0 1.0 20.0 x 1.0 1.0 0.5 y 2.0 2.0 0.5 z 3.0␣
↪3.0 0.5 yz 0.1 0.1 0.5 xz 0.2 0.2 0.5 xy 0.3 0.3 0.5 nreset 1000
```

Example input scripts available: examples/PACKAGES/drude

### 2.263.3 Description

These commands are variants of the Nose-Hoover fix styles *fix nvt* and *fix npt* for thermalized Drude polarizable models. They apply temperature-grouped Nose-Hoover thermostat (TGNH) proposed by *(Son)*. When there are fast vibrational modes with frequencies close to Drude oscillators (e.g. double bonds or out-of-plane torsions), this thermostat can provide better kinetic energy equipartitioning.

The difference between TGNH and the original Nose-Hoover thermostat is that, TGNH separates the kinetic energy of the group into three contributions: molecular center of mass (COM) motion, motion of COM of atom-Drude pairs or non-polarizable atoms relative to molecular COM, and relative motion of atom-Drude pairs. An independent Nose-Hoover chain is applied to each type of motion. The temperatures for these three types of motion are denoted as molecular translational temperature ($T_{\mathrm{M}}$), real atomic temperature ($T_{\mathrm{R}}$) and Drude temperature ($T_{\mathrm{D}}$), which are defined in terms of their associated degrees of freedom (DOF):

$$T_{\mathrm{M}} = \frac{\Sigma_i^{N_{\mathrm{mol}}} M_i V_i^2}{3\left(N_{\mathrm{mol}} - \frac{N_{\mathrm{mol}}}{N_{\mathrm{mol,sys}}}\right) k_{\mathrm{B}}}$$

$$T_{\mathrm{R}} = \frac{\Sigma_i^{N_{\mathrm{real}}} m_i (v_i - v_{M,i})^2}{(N_{\mathrm{DOF}} - 3N_{\mathrm{mol}} + 3\frac{N_{\mathrm{mol}}}{N_{\mathrm{mol,sys}}} - 3N_{\mathrm{drude}}) k_{\mathrm{B}}}$$

$$T_{\mathrm{D}} = \frac{\Sigma_i^{N_{\mathrm{drude}}} m_i' v_i'^2}{3N_{\mathrm{drude}} k_{\mathrm{B}}}$$

Here $N_{\mathrm{mol}}$ and $N_{\mathrm{mol,sys}}$ are the numbers of molecules in the group and in the whole system, respectively. $N_{\mathrm{real}}$ is the number of atom-Drude pairs and non-polarizable atoms in the group. $N_{\mathrm{drude}}$ is the number of Drude particles in the group. $N_{\mathrm{DOF}}$ is the DOF of the group. $M_i$ and $V_i$ are the mass and the COM velocity of the i-th molecule. $m_i$ is the mass of the i-th atom-Drude pair or non-polarizable atom. $v_i$ is the velocity of COM of i-th atom-Drude pair or non-polarizable atom. $v_{M,i}$ is the COM velocity of the molecule the i-th atom-Drude pair or non-polarizable atom belongs to. $m_i'$ and $v_i'$ are the reduced mass and the relative velocity of the i-th atom-Drude pair.

**Note**

These fixes require that each atom knows whether it is a Drude particle or not. You must therefore use the *fix drude* command to specify the Drude status of each atom type.

Because the TGNH thermostat thermostats the molecular COM motion, all atoms belonging to the same molecule must be in the same group. That is, these fixes can not be applied to a subset of a molecule.

For this fix to act correctly, ghost atoms need to know their velocity. You must use the *comm_modify* command to enable this.

These fixes assume that the translational DOF of the whole system is removed. It is therefore recommended to invoke *fix momentum* command so that the $T_{\mathrm{M}}$ is calculated correctly.

---

The thermostat parameters are specified using the *temp* keyword. The thermostat is applied to only the translational DOF for the particles. The translational DOF can also have a bias velocity removed before thermostatting takes place; see the description below. The desired temperature for molecular and real atomic motion is a ramped value during the run from *Tstart* to *Tstop*. The *Tdamp* parameter is specified in time units and determines how rapidly the temperature is relaxed. For example, a value of 10.0 means to relax the temperature in a timespan of (roughly) 10 time units (e.g. $\tau$ or fs or ps - see the *units* command). The parameter *Tdrude* is the desired temperature for Drude motion at each timestep. Similar to *Tdamp*, the *Tdamp_drude* parameter determines the relaxation speed for Drude motion. Fix group are the only ones whose velocities and positions are updated by the velocity/position update portion of the integration. Other thermostat-related keywords are *tchain* and *tloop*, which are detailed in *fix nvt*.

**Note**

A Nose-Hoover thermostat will not work well for arbitrary values of *Tdamp*. If *Tdamp* is too small, the temperature can fluctuate wildly; if it is too large, the temperature will take a very long time to equilibrate. A good choice for many models is a *Tdamp* of around 100 timesteps. A smaller *Tdamp_drude* value would be required to maintain Drude motion at low temperature.

```
fix 1 all nvt temp 300.0 300.0 $(100.0*dt) 1.0 $(20.0*dt)
```

---

The barostat parameters for fix style *tgnpt/drude* is specified using one or more of the *iso*, *aniso*, *tri*, *x*, *y*, *z*, *xy*, *xz*, *yz*, and *couple* keywords. These keywords give you the ability to specify all 6 components of an external stress tensor, and to couple various of these components together so that the dimensions they represent are varied together during a constant-pressure simulation. Other barostat-related keywords are *pchain*, *mtk*, *ploop*, *nreset*, *scalexy*, *scaleyz*, *scalexz*, *flip*and *fixedpoint*. The meaning of barostat parameters are detailed in *fix npt*.

Regardless of what atoms are in the fix group (the only atoms which are time integrated), a global pressure or stress tensor is computed for all atoms. Similarly, when the size of the simulation box is changed, all atoms are re-scaled to new positions.

**Note**

Unlike the *fix temp/berendsen* command which performs thermostatting but NO time integration, these fixes perform thermostatting/barostatting AND time integration. Thus you should not use any other time integration fix, such as *fix nve* on atoms to which this fix is applied. Likewise, these fixes should not be used on atoms that also have their temperature controlled by another fix - e.g. by *fix langevin/drude* command.

See the *Howto thermostat* and *Howto barostat* doc pages for a discussion of different ways to compute temperature and perform thermostatting and barostatting.

---

Like other fixes that perform thermostatting, this fix can be used with *compute commands* that remove a "bias" from the atom velocities. E.g. to apply the thermostat only to atoms within a spatial *region*, or to remove the center-of-mass velocity from a group of atoms, or to remove the x-component of velocity from the calculation.

This is not done by default, but only if the *fix_modify* command is used to assign a temperature compute to this fix that includes such a bias term. See the doc pages for individual *compute temp commands* to determine which ones include a bias. In this case, the thermostat works in the following manner: bias is removed from each atom, thermostatting is performed on the remaining thermal degrees of freedom, and the bias is added back in.

**Note**

However, not all temperature compute commands are valid to be used with these fixes. Precisely, only temperature compute that does not modify the DOF of the group can be used. E.g. *compute temp/ramp* and *compute viscosity/cos* compute the kinetic energy after remove a velocity gradient without affecting the DOF of the group, then they can be invoked in this way. In contrast, *compute temp/partial* may remove the DOF at one or more dimensions, therefore it cannot be used with these fixes.

---

### 2.263.4 Restart, fix_modify, output, run start/stop, minimize info

These fixes writes the state of all the thermostat and barostat variables to *binary restart files*. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The *fix_modify* *temp* and *press* options are supported by these fixes. You can use them to assign a *compute* you have defined to this fix which will be used in its thermostatting or barostatting procedure, as described above. If you do this, note that the kinetic energy derived from the compute temperature should be consistent with the virial term computed using all atoms for the pressure. LAMMPS will warn you if you choose to compute temperature on a subset of atoms.

**Note**

If both the *temp* and *press* keywords are used in a single thermo_modify command (or in two separate commands), then the order in which the keywords are specified is important. Note that a *pressure compute* defines its own temperature compute as an argument when it is specified. The *temp* keyword will override this (for the pressure compute being used by fix npt), but only if the *temp* keyword comes after the *press* keyword. If the *temp* keyword comes before the *press* keyword, then the new pressure compute specified by the *press* keyword will be unaffected by the *temp* setting.

The cumulative energy change in the system imposed by these fixes, due to thermostatting and/or barostatting, are included in the *thermodynamic output* keywords *ecouple* and *econserve*. See the *thermo_style* page for details.

These fixes compute a global scalar which can be accessed by various *output commands*. The scalar is the same cumulative energy change due to this fix described in the previous paragraph. The scalar value calculated by this fix is "extensive".

These fixes also compute a global vector of quantities, which can be accessed by various *output commands*. The vector values are "intensive". The vector stores the three temperatures $T_{\mathrm{M}}$, $T_{\mathrm{R}}$ and $T_{\mathrm{D}}$.

These fixes can ramp their external temperature and pressure over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

These fixes are not invoked during *energy minimization*.

---

### 2.263.5 Restrictions

These fixes are only available when LAMMPS was built with the DRUDE package. These fixes cannot be used with dynamic groups as defined by the *group* command. These fixes cannot be used in 2D simulations.

*X*, *y*, *z* cannot be barostatted if the associated dimension is not periodic. *Xy*, *xz*, and *yz* can only be barostatted if the simulation domain is triclinic and the second dimension in the keyword (*y* dimension in *xy*) is periodic. The *create_box*, *read data*, and *read_restart* commands specify whether the simulation box is orthogonal or non-orthogonal (triclinic) and explain the meaning of the xy,xz,yz tilt factors.

For the *temp* keyword, the final *Tstop* cannot be 0.0 since it would make the external T = 0.0 at some timestep during the simulation which is not allowed in the Nose/Hoover formulation.

The *scaleyz yes*, *scalexz yes*, and *scalexy yes* options can only be used if the second dimension in the keyword is periodic, and if the tilt factor is not coupled to the barostat via keywords *tri*, *yz*, *xz*, and *xy*.

### 2.263.6 Related commands

*fix drude*, *fix nvt*, *fix_npt*, *fix_modify*

### 2.263.7 Default

The keyword defaults are tchain = 3, pchain = 3, mtk = yes, tloop = 1, ploop = 1, nreset = 0, couple = none, flip = yes, scaleyz = scalexz = scalexy = yes if periodic in second dimension and not coupled to barostat, otherwise no.

---

**(Son)** Son, McDaniel, Cui and Yethiraj, J Phys Chem Lett, 10, 7523 (2019).

## 2.264 fix thermal/conductivity command

### 2.264.1 Syntax

```
fix ID group-ID thermal/conductivity N edim Nbin keyword value ...
```

- ID, group-ID are documented in *fix* command
- thermal/conductivity = style name of this fix command
- N = perform kinetic energy exchange every N steps
- edim = *x* or *y* or *z* = direction of kinetic energy transfer
- Nbin = # of layers in edim direction (must be even number)
- zero or more keyword/value pairs may be appended
- keyword = *swap*

  *swap* value = Nswap = number of swaps to perform every N steps

### 2.264.2 Examples

```
fix 1 all thermal/conductivity 100 z 20
fix 1 all thermal/conductivity 50 z 20 swap 2
```

### 2.264.3 Description

Use the Muller-Plathe algorithm described in *this paper* to exchange kinetic energy between two particles in different regions of the simulation box every N steps. This induces a temperature gradient in the system. As described below this enables the thermal conductivity of a material to be calculated. This algorithm is sometimes called a reverse non-equilibrium MD (reverse NEMD) approach to computing thermal conductivity. This is because the usual NEMD approach is to impose a temperature gradient on the system and measure the response as the resulting heat flux. In the Muller-Plathe method, the heat flux is imposed, and the temperature gradient is the system's response.

See the *compute heat/flux* command for details on how to compute thermal conductivity in an alternate way, via the Green-Kubo formalism.

The simulation box is divided into *Nbin* layers in the *edim* direction, where the layer 1 is at the low end of that dimension and the layer *Nbin* is at the high end. Every N steps, Nswap pairs of atoms are chosen in the following manner. Only atoms in the fix group are considered. The hottest Nswap atoms in layer 1 are selected. Similarly, the coldest Nswap atoms in the "middle" layer (see below) are selected. The two sets of Nswap atoms are paired up and their velocities are exchanged. This effectively swaps their kinetic energies, assuming their masses are the same. If the masses are different, an exchange of velocities relative to center of mass motion of the two atoms is performed, to conserve kinetic energy. Over time, this induces a temperature gradient in the system which can be measured using commands such as the following, which writes the temperature profile (assuming z = edim) to the file tmp.profile:

```
compute   ke all ke/atom
variable  temp atom c_ke/1.5
compute   layers all chunk/atom bin/1d z lower 0.05 units reduced
fix       3 all ave/chunk 10 100 1000 layers v_temp file tmp.profile
```

Note that by default, Nswap = 1, though this can be changed by the optional *swap* keyword. Setting this parameter appropriately, in conjunction with the swap rate N, allows the heat flux to be adjusted across a wide range of values, and the kinetic energy to be exchanged in large chunks or more smoothly.

The "middle" layer for velocity swapping is defined as the *Nbin*/2 + 1 layer. Thus if *Nbin* = 20, the two swapping layers are 1 and 11. This should lead to a symmetric temperature profile since the two layers are separated by the same distance in both directions in a periodic sense. This is why *Nbin* is restricted to being an even number.

As described below, the total kinetic energy transferred by these swaps is computed by the fix and can be output. Dividing this quantity by time and the cross-sectional area of the simulation box yields a heat flux. The ratio of heat flux to the slope of the temperature profile is proportional to the thermal conductivity of the fluid, in appropriate units. See the *Muller-Plathe paper* for details.

**Note**

If your system is periodic in the direction of the heat flux, then the flux is going in 2 directions. This means the effective heat flux in one direction is reduced by a factor of 2. You will see this in the equations for thermal conductivity (kappa) in the Muller-Plathe paper. LAMMPS is simply tallying kinetic energy which does not account for whether or not your system is periodic; you must use the value appropriately to yield a kappa for your system.

> **Note**
>
> After equilibration, if the temperature gradient you observe is not linear, then you are likely swapping energy too frequently and are not in a regime of linear response. In this case you cannot accurately infer a thermal conductivity and should try increasing the Nevery parameter.

### 2.264.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the cumulative kinetic energy transferred between the bottom and middle of the simulation box (in the *edim* direction) is stored as a scalar quantity by this fix. This quantity is zeroed when the fix is defined and accumulates thereafter, once every N steps. The units of the quantity are energy; see the *units* command for details. The scalar value calculated by this fix is "intensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.264.5 Restrictions

Swaps conserve both momentum and kinetic energy, even if the masses of the swapped atoms are not equal. Thus you should not need to thermostat the system. If you do use a thermostat, you may want to apply it only to the non-swapped dimensions (other than *vdim*).

LAMMPS does not check, but you should not use this fix to swap the kinetic energy of atoms that are in constrained molecules, e.g. via *fix shake* or *fix rigid*. This is because application of the constraints will alter the amount of transferred momentum. You should, however, be able to use flexible molecules. See the *Zhang paper* for a discussion and results of this idea.

When running a simulation with large, massive particles or molecules in a background solvent, you may want to only exchange kinetic energy between solvent particles.

### 2.264.6 Related commands

*fix ehex*, *fix heat*, *fix ave/chunk*, *fix viscosity*, *compute heat/flux*

### 2.264.7 Default

The option defaults are swap = 1.

---

**(Muller-Plathe)** Muller-Plathe, J Chem Phys, 106, 6082 (1997).

**(Zhang)** Zhang, Lussetti, de Souza, Muller-Plathe, J Phys Chem B, 109, 15060-15067 (2005).

## 2.265 fix ti/spring command

### 2.265.1 Syntax

```
fix ID group-ID ti/spring k t_s t_eq keyword value ...
```

- ID, group-ID are documented in *fix* command
- ti/spring = style name of this fix command

- k = spring constant (force/distance units)
- t_eq = number of steps for the equilibration procedure
- t_s = number of steps for the switching procedure
- zero or more keyword/value pairs may be appended to args
- keyword = *function*

```
function value = function-ID
  function-ID = ID of the switching function (1 or 2)
```

### 2.265.2 Example

```
fix 1 all ti/spring 50.0 2000 1000 function 2
```

### 2.265.3 Description

This fix allows you to compute the free energy of crystalline solids by performing a nonequilibrium thermodynamic integration between the solid of interest and an Einstein crystal. A detailed explanation of how to use this command and choose its parameters for optimal performance and accuracy is given in the paper by *Freitas*. The paper also presents a short summary of the theory of nonequilibrium thermodynamic integration.

The thermodynamic integration procedure is performed by rescaling the force on each atom. Given an atomic configuration the force (F) on each atom is given by

$$F = \left(1 - \lambda\right) F_{\text{solid}} + \lambda F_{\text{harm}}$$

where F_solid is the force that acts on an atom due to an interatomic potential (*e.g.* EAM potential), F_harm is the force due to the Einstein crystal harmonic spring, and lambda is the coupling parameter of the thermodynamic integration. An Einstein crystal is a solid where each atom is attached to its equilibrium position by a harmonic spring with spring constant $k$. With this fix a spring force is applied independently to each atom in the group defined by the fix to tether it to its initial position. The initial position of each atom is its position at the time the fix command was issued.

The fix acts as follows: during the first *t_eq* steps after the fix is defined the value of lambda is zero. This is the period to equilibrate the system in the lambda = 0 state. After this the value of lambda changes dynamically during the simulation from 0 to 1 according to the function defined using the keyword *function* (described below), this switching from lambda from 0 to 1 is done in *t_s* steps. Then comes the second equilibration period of *t_eq* to equilibrate the system in the lambda = 1 state. After that, the switching back to the lambda = 0 state is made using *t_s* timesteps and following the same switching function. After this period the value of lambda is kept equal to zero and the fix has no other effect on the dynamics of the system.

The processes described above is known as nonequilibrium thermodynamic integration and is has been shown (*Freitas*) to present a much superior efficiency when compared to standard equilibrium methods. The reason why the switching it is made in both directions (potential to Einstein crystal and back) is to eliminate the dissipated heat due to the nonequilibrium process. Further details about nonequilibrium thermodynamic integration and its implementation in LAMMPS is available in *Freitas*.

The *function* keyword allows the use of two different lambda paths. Option *1* results in a constant rate of change of lambda with time:

$$\lambda(\tau) = \tau$$

where $\tau$ is the scaled time variable *t/t_s*. The option *2* performs the lambda switching at a rate defined by the following switching function

$$\lambda(\tau) = \tau^5 \left(70\tau^4 - 315\tau^3 + 540\tau^2 - 420\tau + 126\right)$$

This function has zero slope as lambda approaches its extreme values (0 and 1), according to *de Koning* this results in smaller fluctuations on the integral to be computed on the thermodynamic integration. The use of option *2* is recommended since it results in better accuracy and less dissipation without any increase in computational resources cost.

**Note**

As described in *Freitas*, it is important to keep the center-of-mass fixed during the thermodynamic integration. A nonzero total velocity will result in divergences during the integration due to the fact that the atoms are 'attached' to their equilibrium positions by the Einstein crystal. Check the option *zero* of *fix langevin* and *velocity*. The use of the Nose-Hoover thermostat (*fix nvt*) is *NOT* recommended due to its well documented issues with the canonical sampling of harmonic degrees of freedom (notice that the *chain* option will *NOT* solve this problem). The Langevin thermostat (*fix langevin*) correctly thermostats the system and we advise its usage with ti/spring command.

### 2.265.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the original coordinates of tethered atoms to *binary restart files*, so that the spring effect will be the same in a restarted simulation. See the *read restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

The *fix_modify energy* option is supported by this fix to add the energy stored in the per-atom springs to the global potential energy of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify energy no*.

This fix computes a global scalar and a global vector quantities which can be accessed by various *output commands*. The scalar is an energy which is the sum of the spring energy for each atom, where the per-atom energy is $0.5 \cdot k \cdot r^2$. The vector stores 2 values. The first value is the coupling parameter lambda. The second value is the derivative of lambda with respect to the integer timestep $s$, i.e. $\frac{d\lambda}{ds}$. In order to obtain $\frac{d\lambda}{dt}$, where t is simulation time, this 2nd value needs to be divided by the timestep size (e.g. 0.5 fs). The scalar and vector values calculated by this fix are "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The forces due to this fix are imposed during an energy minimization, invoked by the *minimize* command.

**Note**

If you want the per-atom spring energy to be included in the total potential energy of the system (the quantity being minimized), you MUST enable the *fix modify energy* option for this fix.

### 2.265.5 Related commands

*fix spring*, *fix adapt*

### 2.265.6 Restrictions

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.265.7 Default

The keyword default is function = 1.

---

**(Freitas)** Freitas, Asta, and de Koning, Computational Materials Science, 112, 333 (2016).

**(de Koning)** de Koning and Antonelli, Phys Rev E, 53, 465 (1996).

# 2.266 fix tmd command

## 2.266.1 Syntax

```
fix ID group-ID tmd rho_final file1 N file2
```

- ID, group-ID are documented in *fix* command
- tmd = style name of this fix command
- rho_final = desired value of rho at the end of the run (distance units)
- file1 = filename to read target structure from
- N = dump TMD statistics every this many timesteps, 0 = no dump
- file2 = filename to write TMD statistics to (only needed if N > 0)

## 2.266.2 Examples

```
fix 1 all nve
fix 2 tmdatoms tmd 1.0 target_file 100 tmd_dump_file
```

## 2.266.3 Description

Perform targeted molecular dynamics (TMD) on a group of atoms. A holonomic constraint is used to force the atoms to move towards (or away from) the target configuration. The parameter "rho" is monotonically decreased (or increased) from its initial value to rho_final at the end of the run.

Rho has distance units and is a measure of the root-mean-squared distance (RMSD) between the current configuration of the atoms in the group and the target coordinates listed in file1. Thus a value of rho_final = 0.0 means move the atoms all the way to the final structure during the course of the run.

The target file1 can be ASCII text or a compressed text file (detected by a *supported compression format suffix*). The format of the target file1 is as follows:

```
0.0 25.0 xlo xhi
0.0 25.0 ylo yhi
0.0 25.0 zlo zhi
125     24.97311   1.69005     23.46956 0 0 -1
126     1.94691    2.79640     1.92799  1 0 0
127     0.15906    3.46099     0.79121  1 0 0
...
```

The first 3 lines may or may not be needed, depending on the format of the atoms to follow. If image flags are included with the atoms, the first 3 lo/hi lines **must** appear in the file. If image flags are not included, the first 3 lines **must not** appear. The 3 lines contain the simulation box dimensions for the atom coordinates, in the same format as in a LAMMPS data file (see the *read_data* command).

The remaining lines each contain an atom ID and its target x,y,z coordinates. The atom lines (all or none of them) can optionally be followed by 3 integer values: nx,ny,nz.For periodic dimensions, they specify which image of the box the atom is considered to be in, i.e. a value of N (positive or negative) means add N times the box length to the coordinate to get the true value. Those 3 integers either must be given for all atoms or none.

The atom lines can be listed in any order, but every atom in the group must be listed in the file. Atoms not in the fix group may also be listed; they will be ignored.

Comments starting with '#' and empty lines may be included as well.

TMD statistics are written to file2 every N timesteps, unless N is specified as 0, which means no statistics.

The atoms in the fix tmd group should be integrated (via a fix nve, nvt, npt) along with other atoms in the system.

Restarts can be used with a fix tmd command. For example, imagine a 10000 timestep run with a rho_initial = 11 and a rho_final = 1. If a restart file was written after 2000 time steps, then the configuration in the file would have a rho value of 9. A new 8000 time step run could be performed with the same rho_final = 1 to complete the conformational change at the same transition rate. Note that for restarted runs, the name of the TMD statistics file should be changed to prevent it being overwritten.

For more information about TMD, see *(Schlitter1)* and *(Schlitter2)*.

### 2.266.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*.

This fix can ramp its rho parameter over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

This fix is not invoked during *energy minimization*.

### 2.266.5 Restrictions

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

All TMD fixes must be listed in the input script after all integrator fixes (nve, nvt, npt) are applied. This ensures that atoms are moved before their positions are corrected to comply with the constraint.

Atoms that have a TMD fix applied should not be part of a group to which a SHAKE fix is applied. This is because LAMMPS assumes there are not multiple competing holonomic constraints applied to the same atoms.

To read compressed target files, you must compile LAMMPS with the `-DLAMMPS_GZIP` option. See the *Build settings* doc page for details.

### 2.266.6 Related commands

none

### 2.266.7 Default

none

---

**(Schlitter1)** Schlitter, Swegat, Mulders, "Distance-type reaction coordinates for modelling activated processes", J Molecular Modeling, 7, 171-177 (2001).

**(Schlitter2)** Schlitter and Klahn, "The free energy of a reaction coordinate at multiple constraints: a concise formulation", Molecular Physics, 101, 3439-3443 (2003).

## 2.267 fix ttm command

## 2.268 fix ttm/grid command

## 2.269 fix ttm/mod command

## 2.270 fix ttm/thermal command

### 2.270.1 Syntax

```
fix ID group-ID ttm seed C_e rho_e kappa_e gamma_p gamma_s v_0 Nx Ny Nz keyword value ...
fix ID group-ID ttm/mod seed init_file Nx Ny Nz keyword value ...
fix ID group-ID ttm/thermal seed properties_file Nx Ny Nz keyword value ...
```

- ID, group-ID are documented in *fix* command
- style = *ttm* or *ttm/grid* or *ttm/mod* or *ttm/thermal*
- seed = random number seed to use for white noise (positive integer)
- remaining arguments for fix ttm or fix ttm/grid

```
C_e  = electronic specific heat (energy/(electron*temperature) units)
rho_e = electronic density (electrons/volume units)
kappa_e = electronic thermal conductivity (energy/(time*distance*temperature) units)
gamma_p = friction coefficient due to electron-ion interactions (mass/time units)
gamma_s = friction coefficient due to electronic stopping (mass/time units)
v_0 = electronic stopping critical velocity (velocity units)
Nx = number of thermal solve grid points in the x-direction (positive integer)
Ny = number of thermal solve grid points in the y-direction (positive integer)
Nz = number of thermal solve grid points in the z-direction (positive integer)
```

- remaining arguments for fix ttm/mod:

```
init_file = file with the parameters to TTM
Nx = number of thermal solve grid points in the x-direction (positive integer)
Ny = number of thermal solve grid points in the y-direction (positive integer)
Nz = number of thermal solve grid points in the z-direction (positive integer)
```

- remaining arguments for fix ttm/thermal:

```
properties_file = file with grid based TTM properties
Nx = number of thermal solve grid points in the x-direction (positive integer)
Ny = number of thermal solve grid points in the y-direction (positive integer)
Nz = number of thermal solve grid points in the z-direction (positive integer)
```

- zero or more keyword/value(s) pairs may be appended
- keyword = *set* or *infile* or *outfile*

```
set value = Tinit
  Tinit = initial electronic temperature at all grid points (temperature units)
infile value = file.in with grid values for electronic temperatures
outfile values = Nout file.out
  Nout = dump grid temperatures every this many timesteps
  file.out = filename to write grid temperatures to
```

- fix ttm/thermal supports an additional keyword: *source*

  *source* value = source
    source = volumetric heating term applied to electrons (energy/(time*volume) units)

### 2.270.2 Examples

```
fix 2 all ttm 699489 1.0 1.0 10 0.1 0.0 2.0 1 12 1 infile initial outfile 1000 T.out
fix 3 all ttm/grid 123456 1.0 1.0 1.0 1.0 1.0 5.0 5 5 5 infile Te.in
fix 4 all ttm/mod 34277 parameters.txt 5 5 5 infile T_init outfile 10 T_out
fix 5 all ttm/thermal 11111 properties.in 10 10 10 source 0.1 infile temps.in outfile 10␣
↪temps.out
```

Example input scripts using these commands can be found in examples/ttm.

### 2.270.3 Description

Use a two-temperature model (TTM) to represent heat transfer through and between electronic and atomic subsystems. LAMMPS models the atomic subsystem as usual with a molecular dynamics model and the classical force field specified by the user. The electronic subsystem is modeled as a continuum, or a background "gas", on a regular grid which overlays the simulation domain. Energy can be transferred spatially within the grid representing the electrons. Energy can also be transferred between the electronic and atomic subsystems. The algorithm underlying this fix was derived by D. M. Duffy and A. M. Rutherford and is discussed in two J Physics: Condensed Matter papers: *(Duffy)* and *(Rutherford)*. They used this algorithm in cascade simulations where a primary knock-on atom (PKA) was initialized with a high velocity to simulate a radiation event.

The description in this subsection applies to all 4 fix styles: *ttm*, *ttm/grid*, *ttm/mod*, and *ttm/thermal*.

Fix *ttm/grid* distributes the regular grid across processors consistent with the subdomains of atoms owned by each processor, but is otherwise identical to fix ttm. Note that fix *ttm* stores a copy of the grid on each processor, which is acceptable when the overall grid is reasonably small. For larger grids you should use fix *ttm/grid* instead.

Fix *ttm/mod* adds options to account for external heat sources (e.g. at a surface) and for specifying parameters that allow the electronic heat capacity to depend strongly on electronic temperature. It is more expensive computationally than fix *ttm* because it treats the thermal diffusion equation as non-linear. More details on fix *ttm/mod* are given below.

Added in version 10Dec2025.

Fix *ttm/thermal* allows for electronic properties to be assigned independently to each TTM grid point and supports external heat sources to the electronic subsystem. More details on fix *ttm/thermal* are given below.

Heat transfer between the electronic and atomic subsystems is carried out via an inhomogeneous Langevin thermostat. Only atoms in the fix group contribute to and are affected by this heat transfer.

This thermostatting differs from the regular Langevin thermostat (*fix langevin*) in three important ways. First, the Langevin thermostat is applied uniformly to all atoms in the user-specified group for a single target temperature, whereas the TTM fixes apply Langevin thermostatting locally to atoms within the volumes represented by the user-specified grid points with a target temperature specific to that grid point. Second, the Langevin thermostat couples the temperature of the atoms to an infinite heat reservoir, whereas the heat reservoir for the TTM fixes is finite and represents the local electrons. Third, the TTM fixes allow users to specify not just one friction coefficient, but rather two independent friction coefficients: one for the electron-ion interactions (*gamma_p*), and one for electron stopping (*gamma_s*).

When the friction coefficient due to electron stopping, *gamma_s*, is non-zero, electron stopping effects are included for atoms moving faster than the electron stopping critical velocity, *v_0*. For further details about this algorithm, see *(Duffy)* and *(Rutherford)*.

Energy transport within the electronic subsystem is solved according to the heat diffusion equation with added source

terms for heat transfer between the subsystems:

$$C_e \rho_e \frac{\partial T_e}{\partial t} = \bigtriangledown(\kappa_e \bigtriangledown T_e) - g_p(T_e - T_a) + g_s T'_a$$

where $C_e$ is the specific heat, $\rho_e$ is the density, $\kappa_e$ is the thermal conductivity, $T$ is temperature, the "e" and "a" subscripts represent electronic and atomic subsystems respectively, $g_p$ is the coupling constant for the electron-ion interaction, and $g_s$ is the electron stopping coupling parameter. $C_e$, $\rho_e$, and $\kappa_e$ are specified as parameters to the fix *ttm* or *ttm/grid*. The other quantities are derived. The form of the heat diffusion equation used here is almost the same as that in equation 6 of *(Duffy)*, with the exception that the electronic density is explicitly represented, rather than being part of the specific heat parameter.

Currently, the TTM fixes assume that none of the user-supplied parameters will vary with temperature. Note that *(Duffy)* used a tanh() functional form for the temperature dependence of the electronic specific heat, but ignored temperature dependencies of any of the other parameters. See more discussion below for fix *ttm/mod*.

**Note**

These fixes do not perform time integration of the atoms in the fix group, they only rescale their velocities. Thus a time integration fix such as *fix nve* should be used in conjunction with these fixes. These fixes should not normally be used on atoms that have their temperature controlled by another thermostatting fix, e.g. *fix nvt* or *fix langevin*.

**Note**

These fixes require use of an orthogonal 3d simulation box with periodic boundary conditions in all dimensions. They also require that the size and shape of the simulation box do not vary dynamically, e.g. due to use of the *fix npt* command. Likewise, the size/shape of processor subdomains cannot vary due to dynamic load-balancing via use of the *fix balance* command. It is possible however to load balance before the simulation starts using the *balance* command, so that each processor has a different size subdomain.

Periodic boundary conditions are also used in the heat equation solve for the electronic subsystem. This varies from the approach of *(Rutherford)* where the atomic subsystem was embedded within a larger continuum representation of the electronic subsystem.

The *set* keyword specifies a *Tinit* temperature value to initialize the value stored on all grid points. By default the temperatures are all zero when the grid is created.

The *infile* keyword specifies an input file of electronic temperatures for each grid point to be read in to initialize the grid, as an alternative to using the *set* keyword.

The input file is a text file which may have comments starting with the '#' character. Each line contains four numeric columns: ix,iy,iz,Temperature. Empty or comment-only lines will be ignored. The number of lines must be equal to the number of user-specified grid points (Nx by Ny by Nz). The ix,iy,iz are grid point indices ranging from 1 to Nxyz inclusive in each dimension. The lines can appear in any order. For example, the initial electronic temperatures on a 1 by 2 by 3 grid could be specified in the file as follows:

```
# UNITS: metal COMMENT: initial electron temperature
1 1 1 1.0
1 1 2 1.0
1 1 3 1.0
1 2 1 2.0
1 2 2 2.0
1 2 3 2.0
```

where the electronic temperatures along the y=0 plane have been set to 1.0, and the electronic temperatures along the y=1 plane have been set to 2.0. If all the grid point values are not specified, LAMMPS will generate an error. LAMMPS will check if a "UNITS:" tag is in the first line and stop with an error, if there is a mismatch with the current units used.

**Note**

The electronic temperature at each grid point must be a non-zero positive value, both initially, and as the temperature evolves over time. Thus you must use either the *set* or *infile* keyword or be restarting a simulation that used this fix previously.

The *outfile* keyword has 2 values. The first value *Nout* triggers output of the electronic temperatures for each grid point every Nout timesteps. The second value is the filename for output, which will be suffixed by the timestep. The format of each output file is exactly the same as the input temperature file. It will contain a comment in the first line reporting the date the file was created, the LAMMPS units setting in use, grid size and the current timestep.

**Note**

The fix ttm/grid command does not support the *outfile* keyword. Instead you can use the *dump grid* command to output the electronic temperature on the distributed grid to a dump file or the *restart* command which creates a file specific to this fix which the *read restart* command reads. The file has the same format as the file the *infile* option reads.

For the fix ttm, fix ttm/mod, and fix ttm/thermal commands, the corresponding atomic temperature for atoms in each grid cell can be computed and output by the *fix ave/chunk* command using the *compute chunk/atom* command to create a 3d array of chunks consistent with the grid used by this fix.

For the fix ttm/grid command the same thing can be done using the *fix ave/grid* command and its per-grid values can be output via the *dump grid* command.

---

**Additional details for fix ttm/mod**

Fix *ttm/mod* uses the heat diffusion equation with possible external heat sources (e.g. laser heating in ablation simulations):

$$C_e\rho_e\frac{\partial T_e}{\partial t} = \bigtriangledown(\kappa_e \bigtriangledown T_e) - g_p(T_e - T_a) + g_s T_a' + \theta(x - x_{surface})I_0 \exp(-x/l_{skin})$$

where $\theta$ is the Heaviside step function, $I_0$ is the (absorbed) laser pulse intensity for ablation simulations, $l_{skin}$ is the depth of the skin-layer, and all other designations have the same meaning as in the former equation. The duration of the pulse is set by the parameter *tau* in the *init_file*.

Fix *ttm/mod* also allows users to specify the dependencies of $C_e$ and $\kappa_e$ on the electronic temperature. The specific heat is expressed as

$$C_e = C_0 + (a_0 + a_1 X + a_2 X^2 + a_3 X^3 + a_4 X^4)\exp(-(AX)^2)$$

where $X = \frac{T_e}{1000}$, and the thermal conductivity is defined as $\kappa_e = D_e \cdot rho_e \cdot C_e$, where $D_e$ is the thermal diffusion coefficient.

Electronic pressure effects are included in the TTM model to account for the blast force acting on ions because of electronic pressure gradient (see *(Chen)*, *(Norman)*). The total force acting on an ion is:

$$\vec{F}_i = -\partial U/\partial \vec{r}_i + \vec{F}_{langevin} - \nabla P_e/n_{ion}$$

where $F_{langevin}$ is a force from Langevin thermostat simulating electron-phonon coupling, and $\nabla P_e/n_{ion}$ is the electron blast force.

The electronic pressure is taken to be $P_e = B \cdot rho_e \cdot C_e \cdot T_e$

The current fix *ttm/mod* implementation allows TTM simulations with a vacuum. The vacuum region is defined as the grid cells with zero electronic temperature. The numerical scheme does not allow energy exchange with such cells. Since the material can expand to previously unoccupied region in some simulations, the vacuum border can be allowed to move. It is controlled by the *surface_movement* parameter in the *init_file*. If it is set to 1, then "vacuum" cells can be changed to "electron-filled" cells with the temperature *T_e_min* if atoms move into them (currently only implemented for the case of 1-dimensional motion of a flat surface normal to the X axis). The initial locations of the interfaces of the electron density to the vacuum can be set in the *init_file* via *lsurface* and *rsurface* parameters. In this case, electronic pressure gradient is calculated as

$$\nabla_x P_e = \left[ \frac{C_e T_e(x) \lambda}{(x+\lambda)^2} + \frac{x}{x+\lambda} \frac{(C_e T_e)_{x+\Delta x} - (C_e T_e)_x}{\Delta x} \right]$$

where $\lambda$ is the electron mean free path (see *(Norman)*, *(Pisarev)*)

**Note**

The *lsurface* and *rsurface* parameters always define the range of electronic grid cells in the x-direction over which the electron-ion (Langevin) coupling and any external (laser) source term are applied, regardless of the *surface_movement* setting. Only atoms in grid cells whose x grid index *ix* satisfies *lsurface* <= *ix* < *rsurface* are coupled to the electronic subsystem; atoms in cells outside this range exchange no energy with the electrons. Therefore, to model a bulk system with full periodicity and no vacuum (i.e. to reproduce the behavior of fix *ttm*), you must set *lsurface* = 0 and *rsurface* = *Nx* so that the coupling acts on the entire simulation box. Setting *rsurface* to a smaller value couples only a sub-slab of the grid cells and will make energy transfer between the two subsystems appear much too slow. Fix *ttm/mod* prints a warning when this coupling region does not span the whole box (with *surface_movement* disabled).

The fix *ttm/mod* parameter file *init_file* has the following syntax. Every line with an odd number is considered as a comment and ignored. The lines with the even numbers are treated as follows:

```
a_0, energy/(temperature*electron) units
a_1, energy/(temperature^2*electron) units
a_2, energy/(temperature^3*electron) units
a_3, energy/(temperature^4*electron) units
a_4, energy/(temperature^5*electron) units
C_0, energy/(temperature*electron) units
A, 1/temperature units
rho_e, electrons/volume units
D_e, length^2/time units
gamma_p, mass/time units
gamma_s, mass/time units
v_0, length/time units
I_0, energy/(time*length^2) units
lsurface, lower x grid index of the coupled (electron) region (positive integer; see␣
↪note above)
rsurface, upper x grid index of the coupled (electron) region (positive integer; see␣
↪note above)
l_skin, electron grid units (positive integer)
tau, time units
B, dimensionless
lambda, length units
```

```
n_ion, ions/volume units
surface_movement: 0 to disable tracking of surface motion, 1 to enable
T_e_min, temperature units
```

---

**Additional details for fix ttm/thermal**

Fix *ttm/thermal* uses the heat diffusion equation with possible external heat sources (e.g. inductive heating). The effects of electron stopping have been removed:

$$C_{\text{vol}} \frac{\partial T_e}{\partial t} = \bigtriangledown(\kappa_{\text{eff}} \bigtriangledown T_e) - g_p(T_e - T_a) + \eta s$$

where $s$ is the applied heating power density and $\eta$ is the absorption efficiency (0-1) defined for each ttm grid cell in the *properties.in* file. Also note that compared to the original *fix ttm*, it uses use a volumetric specific heat, $C_{\text{vol}}$ , which represents the product of $C_e \rho_e$.

*(Baer)* defined $\kappa_{\text{eff}}$ as an effective electronic thermal conductivity when two adjacent TTM cells (denoted by the subscripts $a$ and $b$) have different conductivities as:

$$\kappa_{\text{eff}} = \frac{2\kappa_a \kappa_b}{\kappa_a + \kappa_b}$$

The current fix *ttm/thermal* implementation allows TTM simulations with TTM cells that do not contain electrons (vacuum or insulators). Similar to *ttm/mod*, the absence of electrons is defined as the grid cells with zero electronic temperature.The numerical scheme does not allow energy exchange with such cells.

The fix *ttm/thermal* parameter file *properties_file* uses a similar syntax as the keyword *infile*. The file is read in line by line and each ttm cell's properties are set. Comment lines are allowed and each line should have properties listed in the order:

```
ix iy iz C_vol kappa_e gamma_p eta
```

the grid must match the one declared in the fix and all grid points must have all properties set or *ttm/thermal* will exit with an error.

---

## 2.270.4 Restart, fix_modify, output, run start/stop, minimize info

The fix ttm, fix ttm/mod, and fix ttm/thermal commands write the state of the electronic subsystem and the energy exchange between the subsystems to *binary restart files*. The fix ttm/grid command does not yet support writing of its distributed grid to a restart file.

See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion. Note that the restart script must define the same size grid as the original script.

The fix ttm/grid command also outputs an auxiliary file each time a restart file is written, with the electron temperatures for each grid cell. The format of this file is the same as that read by the *infile* option explained above. The filename is the same as the restart filename with ".ttm" appended. This auxiliary file can be read in for a restarted run by using the *infile* option for the fix ttm/grid command, following the *read_restart* command.

None of the *fix_modify* options are relevant to these fixes.

These fixes compute 2 output quantities stored in a vector of length 2, which can be accessed by various *output commands*. The first quantity is the total energy of the electronic subsystem. The second quantity is the energy transferred from the electronic to the atomic subsystem on that timestep. Note that the velocity verlet integrator applies the fix ttm forces to the atomic subsystem as two half-step velocity updates: one on the current timestep and one on the subsequent timestep. Consequently, the change in the atomic subsystem energy is lagged by half a timestep relative to the change

in the electronic subsystem energy. As a result of this, users may notice slight fluctuations in the sum of the atomic and electronic subsystem energies reported at the end of the timestep.

The vector values calculated are "extensive".

The fix ttm/grid command also outputs a per-grid vector which stores the electron temperature for each grid cell in temperature *units*. which can be accessed by various *output commands*. The length of the vector (distributed across all processors) is Nx * Ny * Nz. For access by other commands, the name of the single grid produced by fix ttm/grid is "grid". The name of its per-grid data is "data".

No parameter of the fixes can be used with the *start/stop* keywords of the *run* command. The fixes are not invoked during *energy minimization*.

### 2.270.5 Restrictions

All these fixes are part of the EXTRA-FIX package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

As mentioned above, these fixes require 3d simulations and orthogonal simulation boxes periodic in all 3 dimensions.

These fixes used a random number generator to Langevin thermostat the electron temperature. This means you will not get identical answers when running on different numbers of processors or when restarting a simulation (even on the same number of processors). However, in a statistical sense, simulations on different processor counts and restarted simulation should produce results which are statistically the same.

### 2.270.6 Related commands

*fix langevin*, *fix dt/reset*

### 2.270.7 Default

none

---

**(Duffy)** D M Duffy and A M Rutherford, J. Phys.: Condens. Matter, 19, 016207-016218 (2007).

**(Rutherford)** A M Rutherford and D M Duffy, J. Phys.: Condens. Matter, 19, 496201-496210 (2007).

**(Chen)** J Chen, D Tzou and J Beraun, Int. J. Heat Mass Transfer, 49, 307-316 (2006).

**(Norman)** G E Norman, S V Starikov, V V Stegailov et al., Contrib. Plasma Phys., 53, 129-139 (2013).

**(Pisarev)** V V Pisarev and S V Starikov, J. Phys.: Condens. Matter, 26, 475401 (2014).

**(Baer)** B Baer and D G Walker, J. Mol. Model, 31, 220 (2025)

## 2.271 fix tune/kspace command

### 2.271.1 Syntax

```
fix ID group-ID tune/kspace N
```

- ID, group-ID are documented in *fix* command
- tune/kspace = style name of this fix command
- N = invoke this fix every N steps

### 2.271.2 Examples

```
fix 2 all tune/kspace 100
```

### 2.271.3 Description

This fix tests each kspace style (Ewald, PPPM, and MSM), and automatically selects the fastest style to use for the remainder of the run. If the fastest style is Ewald or PPPM, the fix also adjusts the Coulombic cutoff towards optimal speed. Future versions of this fix will automatically select other kspace parameters to use for maximum simulation speed. The kspace parameters may include the style, cutoff, grid points in each direction, order, Ewald parameter, MSM parallelization cut-point, MPI tasks to use, etc.

The rationale for this fix is to provide the user with as-fast-as-possible simulations that include long-range electrostatics (kspace) while meeting the user-prescribed accuracy requirement. A simple heuristic could never capture the optimal combination of parameters for every possible run-time scenario. But by performing short tests of various kspace parameter sets, this fix allows parameters to be tailored specifically to the user's machine, MPI ranks, use of threading or accelerators, the simulated system, and the simulation details. In addition, it is possible that parameters could be evolved with the simulation on-the-fly, which is useful for systems that are dynamically evolving (e.g. changes in box size/shape or number of particles).

When this fix is invoked, LAMMPS will perform short timed tests of various parameter sets to determine the optimal parameters. Tests are performed on-the-fly, with a new test initialized every N steps. N should be chosen large enough so that adequate CPU time lapses between tests, thereby providing statistically significant timings. But N should not be chosen to be so large that an unfortunate parameter set test takes an inordinate amount of wall time to complete. An N of 100 for most problems seems reasonable. Once an optimal parameter set is found, that set is used for the remainder of the run.

This fix uses heuristics to guide its selection of parameter sets to test, but the actual timed results will be used to decide which set to use in the simulation.

It is not necessary to discard trajectories produced using sub-optimal parameter sets, or a mix of various parameter sets, since the user-prescribed accuracy will have been maintained throughout. However, some users may prefer to use this fix only to discover the optimal parameter set for a given setup that can then be used on subsequent production runs.

This fix starts with kspace parameters that are set by the user with the *kspace_style* and *kspace_modify* commands. The prescribed accuracy will be maintained by this fix throughout the simulation.

None of the *fix_modify* options are relevant to this fix.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.271.4 Restrictions

This fix is part of the KSPACE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Do not set "neigh_modify once yes" or else this fix will never be called. Reneighboring is required.

This fix is not compatible with a hybrid pair style, long-range dispersion, TIP4P water support, or long-range point dipole support.

### 2.271.5 Related commands

*kspace_style*, *boundary kspace_modify*, *pair_style lj/cut/coul/long*, *pair_style lj/charmm/coul/long*, *pair_style lj/long*, *pair_style lj/long/coul/long*, *pair_style buck/coul/long*

### 2.271.6 Default

## 2.272 fix vector command

### 2.272.1 Syntax

```
fix ID group-ID vector Nevery value1 value2 ... keyword args ...
```

- ID, group-ID are documented in *fix* command
- vector = style name of this fix command
- Nevery = use input values every this many timesteps
- one or more input values can be listed
- value = c_ID, c_ID[N], f_ID, f_ID[N], v_name

```
c_ID = global scalar calculated by a compute with ID
c_ID[I] = Ith component of global vector calculated by a compute with ID
f_ID = global scalar calculated by a fix with ID
f_ID[I] = Ith component of global vector calculated by a fix with ID
v_name = value calculated by an equal-style variable with name
v_name[I] = Ith component of vector-style variable with name
```

- zero or more keyword/args pairs may be appended
- keyword = *nmax*

  *nmax* length = set maximal length of vector to <length>

### 2.272.2 Examples

```
fix 1 all vector 100 c_myTemp
fix 1 all vector 5 c_myTemp v_integral
fix 1 all vector 50 c_myTemp nmax 200
```

### 2.272.3 Description

Use one or more global values as inputs every few timesteps, and simply store them as a sequence. For a single specified value, the values are stored as a global vector of growing length. For multiple specified values, they are stored as rows in a global array, whose number of rows is growing. The resulting vector or array can be used by other *output commands*.

The optional *nmax* keyword can be used to restrict the length of the vector to the given *length* value. Once the restricted vector is filled, the oldest entry will be discarded when a entry is added.

One way to to use this command is to accumulate a vector that is numerically integrated using the *variable trap()* function. For example, the velocity auto-correlation function (VACF) can be integrated, to yield a diffusion coefficient, as follows:

```
compute         2 all vacf
fix             5 all vector 1 c_2[4]
variable        diff equal dt*trap(f_5)
thermo_style    custom step v_diff
```

The group specified with this command is ignored. However, note that specified values may represent calculations performed by computes and fixes which store their own "group" definitions.

Each listed value can be the result of a *compute* or *fix* or the evaluation of an equal-style or vector-style *variable*. In each case, the compute, fix, or variable must produce a global quantity, not a per-atom or local quantity. And the global quantity must be a scalar, not a vector or array.

*Computes* that produce global quantities are those which do not have the word *atom* in their style name. Only a few *fixes* produce global quantities. See the doc pages for individual fixes for info on which ones produce such values. *Variables* of style *equal* or *vector* are the only ones that can be used with this fix. Variables of style *atom* cannot be used, since they produce per-atom values.

The *Nevery* argument specifies on what timesteps the input values will be used in order to be stored. Only timesteps that are a multiple of *Nevery*, including timestep 0, will contribute values.

**Note**

If *Nevery* is a small number and the simulation runs for many steps, the accumulated vector or array can become very large and thus consume a lot of memory. The implementation limit is about 2 billion entries. Using the *nmax* keyword mentioned above can avoid that by limiting the size of the vector.

Note that if you perform multiple runs, using the "pre no" option of the *run* command to avoid initialization on subsequent runs, then you need to use the *stop* keyword with the first *run* command with a timestep value that encompasses all the runs. This is so that the vector or array stored by this fix can be allocated to a sufficient size.

---

If a value begins with "c_", a compute ID must follow which has been previously defined in the input script. If no bracketed term is appended, the global scalar calculated by the compute is used. If a bracketed term is appended, the Ith element of the global vector calculated by the compute is used.

Note that there is a *compute reduce* command which can sum per-atom quantities into a global scalar or vector which can thus be accessed by fix vector. Or it can be a compute defined not in your input script, but by *thermodynamic output* or other fixes such as *fix nvt* or *fix temp/rescale*. See the doc pages for these commands which give the IDs of these computes. Users can also write code for their own compute styles and *add them to LAMMPS*.

If a value begins with "f_", a fix ID must follow which has been previously defined in the input script. If no bracketed term is appended, the global scalar calculated by the fix is used. If a bracketed term is appended, the Ith element of the global vector calculated by the fix is used.

Note that some fixes only produce their values on certain timesteps, which must be compatible with *Nevery*, else an error will result. Users can also write code for their own fix styles and *add them to LAMMPS*.

If a value begins with "v_", a variable name must follow which has been previously defined in the input script. An equal-style or vector-style variable can be referenced; the latter requires a bracketed term to specify the Ith element of the vector calculated by the variable. See the *variable* command for details. Note that variables of style *equal* and *vector* define a formula which can reference individual atom properties or thermodynamic keywords, or they can invoke other computes, fixes, or variables when they are evaluated, so this is a very general means of specifying quantities to be stored by fix vector.

---

### 2.272.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix produces a global vector or global array which can be accessed by various *output commands*. The values can only be accessed on timesteps that are multiples of *Nevery*.

A vector is produced if only a single input value is specified. An array is produced if multiple input values are specified. The length of the vector or the number of rows in the array grows by 1 every *Nevery* timesteps.

If the fix produces a vector, then the entire vector will be either "intensive" or "extensive", depending on whether the values stored in the vector are "intensive" or "extensive". If the fix produces an array, then all elements in the array must be the same, either "intensive" or "extensive". If a compute or fix provides the value stored, then the compute or fix determines whether the value is intensive or extensive; see the page for that compute or fix for further info. Values produced by a variable are treated as intensive.

This fix can allocate storage for stored values accumulated over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this. If using the *run pre no* command option, this is required to allow the fix to allocate sufficient storage for stored values.

This fix is not invoked during *energy minimization*.

### 2.272.5 Restrictions

none

### 2.272.6 Related commands

*compute*, *variable*

### 2.272.7 Defaults

The default value of *nmax* is deduced from the number of steps in a run (or multiple runs when using the *start* and *stop* keywords of the *run command*) divided by the choice of *Nevery* plus 1.

## 2.273 fix viscosity command

### 2.273.1 Syntax

```
fix ID group-ID viscosity N vdim pdim Nbin keyword value ...
```

- ID, group-ID are documented in *fix* command
- viscosity = style name of this fix command
- N = perform momentum exchange every N steps
- vdim = *x* or *y* or *z* = which momentum component to exchange
- pdim = *x* or *y* or *z* = direction of momentum transfer
- Nbin = # of layers in pdim direction (must be even number)
- zero or more keyword/value pairs may be appended
- keyword = *swap* or *vtarget*

  ```
  swap value = Nswap = number of swaps to perform every N steps
  vtarget value = V or INF = target velocity of swap partners (velocity units)
  ```

### 2.273.2 Examples

```
fix 1 all viscosity 100 x z 20
fix 1 all viscosity 50 x z 20 swap 2 vtarget 1.5
```

### 2.273.3 Description

Use the Muller-Plathe algorithm described in *this paper* to exchange momenta between two particles in different regions of the simulation box every N steps. This induces a shear velocity profile in the system. As described below this enables a viscosity of the fluid to be calculated. This algorithm is sometimes called a reverse non-equilibrium MD (reverse NEMD) approach to computing viscosity. This is because the usual NEMD approach is to impose a shear velocity profile on the system and measure the response via an off-diagonal component of the stress tensor, which is proportional to the momentum flux. In the Muller-Plathe method, the momentum flux is imposed, and the shear velocity profile is the system's response.

The simulation box is divided into *Nbin* layers in the *pdim* direction, where the layer 1 is at the low end of that dimension and the layer *Nbin* is at the high end. Every N steps, Nswap pairs of atoms are chosen in the following manner. Only atoms in the fix group are considered. Nswap atoms in layer 1 with positive velocity components in the *vdim* direction closest to the target value *V* are selected. Similarly, Nswap atoms in the "middle" layer (see below) with negative velocity components in the *vdim* direction closest to the negative of the target value *V* are selected. The two sets of Nswap atoms are paired up and their *vdim* momenta components are swapped within each pair. This resets their velocities, typically in opposite directions. Over time, this induces a shear velocity profile in the system which can be measured using commands such as the following, which writes the profile to the file tmp.profile:

```
compute layers all chunk/atom bin/1d z lower 0.05 units reduced
fix f1 all ave/chunk 100 10 1000 layers vx file tmp.profile
```

Note that by default, Nswap = 1 and vtarget = INF, though this can be changed by the optional *swap* and *vtarget* keywords. When vtarget = INF, one or more atoms with the most positive and negative velocity components are selected. Setting these parameters appropriately, in conjunction with the swap rate N, allows the momentum flux rate to be adjusted across a wide range of values, and the momenta to be exchanged in large chunks or more smoothly.

The "middle" layer for momenta swapping is defined as the *Nbin*/2 + 1 layer. Thus if *Nbin* = 20, the two swapping layers are 1 and 11. This should lead to a symmetric velocity profile since the two layers are separated by the same distance in both directions in a periodic sense. This is why *Nbin* is restricted to being an even number.

As described below, the total momentum transferred by these velocity swaps is computed by the fix and can be output. Dividing this quantity by time and the cross-sectional area of the simulation box yields a momentum flux. The ratio of momentum flux to the slope of the shear velocity profile is proportional to the viscosity of the fluid, in appropriate units. See the *Muller-Plathe paper* for details.

**Note**

If your system is periodic in the direction of the momentum flux, then the flux is going in 2 directions. This means the effective momentum flux in one direction is reduced by a factor of 2. You will see this in the equations for viscosity in the Muller-Plathe paper. LAMMPS is simply tallying momentum which does not account for whether or not your system is periodic; you must use the value appropriately to yield a viscosity for your system.

**Note**

After equilibration, if the velocity profile you observe is not linear, then you are likely swapping momentum too frequently and are not in a regime of linear response. In this case you cannot accurately infer a viscosity and should try increasing the Nevery parameter.

An alternative method for calculating a viscosity is to run a NEMD simulation, as described on the *Howto nemd* doc page. NEMD simulations deform the simulation box via the *fix deform* command.

Some features or combination of settings in LAMMPS do not support non-orthogonal boxes. Using fix viscosity keeps the box orthogonal; thus it does not suffer from these limitations.

### 2.273.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix computes a global scalar which can be accessed by various *output commands*. The scalar is the cumulative momentum transferred between the bottom and middle of the simulation box (in the *pdim* direction) is stored as a scalar quantity by this fix. This quantity is zeroed when the fix is defined and accumulates thereafter, once every N steps. The units of the quantity are momentum = mass*velocity. The scalar value calculated by this fix is "intensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.273.5 Restrictions

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Swaps conserve both momentum and kinetic energy, even if the masses of the swapped atoms are not equal. Thus you should not need to thermostat the system. If you do use a thermostat, you may want to apply it only to the non-swapped dimensions (other than *vdim*).

LAMMPS does not check, but you should not use this fix to swap velocities of atoms that are in constrained molecules, e.g. via *fix shake* or *fix rigid*. This is because application of the constraints will alter the amount of transferred momentum. You should, however, be able to use flexible molecules. See the *Maginn paper* for an example of using this algorithm in a computation of alcohol molecule properties.

When running a simulation with large, massive particles or molecules in a background solvent, you may want to only exchange momenta between solvent particles.

### 2.273.6 Related commands

*fix ave/chunk*, *fix thermal/conductivity*

### 2.273.7 Default

The option defaults are swap = 1 and vtarget = INF.

---

**(Muller-Plathe)** Muller-Plathe, Phys Rev E, 59, 4894-4898 (1999).

**(Maginn)** Kelkar, Rafferty, Maginn, Siepmann, Fluid Phase Equilibria, 260, 218-231 (2007).

## 2.274 fix viscous command

Accelerator Variants: *viscous/kk*

### 2.274.1 Syntax

```
fix ID group-ID viscous gamma keyword values ...
```

- ID, group-ID are documented in *fix* command
- viscous = style name of this fix command
- gamma = damping coefficient (force/velocity units)
- zero or more keyword/value pairs may be appended

```
keyword = scale
  scale values = type ratio
    type = atom type (1-N)
    ratio = factor to scale the damping coefficient by
```

### 2.274.2 Examples

```
fix 1 flow viscous 0.1
fix 1 damp viscous 0.5 scale 3 2.5
```

### 2.274.3 Description

Add a viscous damping force to atoms in the group that is proportional to the velocity of the atom. The added force can be thought of as a frictional interaction with implicit solvent, i.e. the no-slip Stokes drag on a spherical particle. In granular simulations this can be useful for draining the kinetic energy from the system in a controlled fashion. If used without additional thermostatting (to add kinetic energy to the system), it has the effect of slowly (or rapidly) freezing the system; hence it can also be used as a simple energy minimization technique.

The damping force $F_i$ is given by $F_i = -\gamma v_i$. The larger the coefficient, the faster the kinetic energy is reduced. If the optional keyword *scale* is used, $\gamma$ can scaled up or down by the specified factor for atoms of that type. It can be used multiple times to adjust $\gamma$ for several atom types.

**Note**

You should specify gamma in force/velocity units. This is not the same as mass/time units, at least for some of the LAMMPS *units* options like "real" or "metal" that are not self-consistent.

In a Brownian dynamics context, $\gamma = \frac{k_B T}{D}$, where $k_B$ = Boltzmann's constant, $T$ = temperature, and $D$ = particle diffusion coefficient. $D$ can be written as $\frac{k_B T}{3\pi\eta d}$, where $\eta$ = dynamic viscosity of the frictional fluid and d = diameter of particle. This means $\gamma = 3\pi\eta d$, and thus is proportional to the viscosity of the fluid and the particle diameter.

In the current implementation, rather than have the user specify a viscosity, $\gamma$ is specified directly in force/velocity units. If needed, $\gamma$ can be adjusted for atoms of different sizes (i.e. $\sigma$) by using the *scale* keyword.

Note that Brownian dynamics models also typically include a randomized force term to thermostat the system at a chosen temperature. The *fix langevin* command does this. It has the same viscous damping term as fix viscous and adds a random force to each atom. The random force term is proportional to the square root of the chosen thermostatting temperature. Thus if you use fix langevin with a target $T = 0$, its random force term is zero, and you are essentially performing the same operation as fix viscous. Also note that the gamma of fix viscous is related to the damping parameter of *fix langevin*, however the former is specified in units of force/velocity and the latter in units of time, so that it can more easily be used as a thermostat.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 2.274.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The *fix_modify* *respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is modifying forces. Default is the outermost level.

The forces due to this fix are imposed during an energy minimization, invoked by the *minimize* command. This fix should only be used with damped dynamics minimizers that allow for non-conservative forces. See the *min_style* command for details.

### 2.274.5 Restrictions

none

### 2.274.6 Related commands

*fix langevin*, *fix viscous/sphere*, *fix damping/cundall*

### 2.274.7 Default

none

## 2.275 fix viscous/nonlinear command

### 2.275.1 Syntax

```
fix ID group-ID viscous/nonlinear rho_fluid mu_fluid keyword values ...
```

- ID, group-ID are documented in *fix* command
- viscous/nonlinear = style name of this fix command
- rho_fluid = mass density of the surrounding fluid (mass/volume units)
- mu_fluid = dynamic viscosity of the surrounding fluid (pressure*time units)
- zero or more keyword/value pairs may be appended

```
keyword = velocity
  velocity values = Vx Vy Vz
    Vx,Vy,Vz = components of the (uniform) fluid velocity (velocity units)
```

### 2.275.2 Examples

```
fix drag all viscous/nonlinear 1.2 1.8e-5
fix drag flow viscous/nonlinear 1.2 1.8e-5 velocity 0.0 0.0 0.4
```

### 2.275.3 Description

Added in version 4Jul2026.

Add a nonlinear (Reynolds-number dependent) drag force to each finite-size spherical particle in the group, modeling the interaction with a uniform background fluid (e.g. an upward gas stream). Unlike *fix viscous*, which applies a drag force strictly proportional to the particle velocity (Stokes drag), this fix uses the standard drag-coefficient relation with the Schiller-Naumann correlation, which is accurate over a much wider range of particle Reynolds numbers.

The drag force on particle $i$ is

$$\vec{F}_i = -\frac{1}{2}\, C_d\, \rho_f\, \pi r_i^2\, |\vec{v}_{rel}|\, \vec{v}_{rel}$$

where $r_i$ is the particle radius, $\rho_f$ is the fluid mass density, $\vec{v}_{rel} = \vec{v}_i - \vec{v}_f$ is the particle velocity relative to the fluid, and the drag coefficient $C_d$ follows the Schiller-Naumann correlation

$$C_d = \frac{24}{Re}\left(1 + 0.15\, Re^{0.687}\right), \qquad Re = \frac{\rho_f\, |\vec{v}_{rel}|\, (2r_i)}{\mu_f}$$

with $Re$ the particle Reynolds number based on the diameter $2r_i$, the fluid density $\rho_f$, and the dynamic viscosity of the fluid $\mu_f$. In the low-Reynolds-number limit ($Re \to 0$) the correlation reduces to $C_d = 24/Re$ and the force becomes the Stokes drag $\vec{F}_i = -6\pi\mu_f r_i \vec{v}_{rel}$.

By default the fluid is at rest. The optional *velocity* keyword sets a uniform fluid velocity $\vec{v}_f$, so the drag is computed from the particle velocity relative to the moving fluid. This fix only applies a drag force; buoyancy and gravity (if desired) must be added separately, e.g. with *fix gravity*.

### 2.275.4 Restrictions

This fix is part of the GRANULAR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix requires that atoms store a radius as defined by the *atom_style sphere* command.

### 2.275.5 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The *fix_modify* *respa* option is supported by this fix. This allows one to set at which level of the *r-RESPA* integrator the fix is modifying forces. Default is the outermost level.

The forces due to this fix are imposed during an energy minimization, invoked by the *minimize* command.

### 2.275.6 Related commands

*fix viscous*, *fix viscous/sphere*, *fix gravity*

### 2.275.7 Default

The fluid velocity is (0,0,0).

## 2.276 fix viscous/sphere command

Accelerator Variant: *viscous/sphere/kk*

### 2.276.1 Syntax

```
fix ID group-ID viscous/sphere gamma keyword values ...
```

- ID, group-ID are documented in *fix* command
- viscous/sphere = style name of this fix command
- gamma = damping coefficient (torque/angular velocity units)
- zero or more keyword/value pairs may be appended

```
keyword = scale
  scale values = type ratio or v_name
    type = atom type (1-N)
    ratio = factor to scale the damping coefficients by
    v_name = reference to atom style variable name
```

### 2.276.2 Examples

```
fix 1 flow viscous/sphere 0.1
fix 1 damp viscous/sphere 0.5 scale 3 2.5
fix 1 damp viscous/sphere 0.5 scale v_radscale
```

### 2.276.3 Description

Add a viscous damping torque to finite-size spherical particles in the group that is proportional to the angular velocity of the atom. In granular simulations this can be useful for draining the rotational kinetic energy from the system in a controlled fashion. If used without additional thermostatting (to add kinetic energy to the system), it has the effect of slowly (or rapidly) freezing the system; hence it can also be used as a simple energy minimization technique.

The damping torque $T_i$ is given by $T_i = -\gamma\omega_i$. The larger the coefficient, the faster the rotational kinetic energy is reduced.

If the optional keyword *scale* is used, $\gamma$ can be scaled up or down by the specified factor for atoms. This factor can be set for different atom types and thus the *scale* keyword used multiple times followed by the atom type and the associated scale factor. Alternately the scaling factor can be computed for each atom (e.g. based on its radius) by using an *atom-style variable*.

**Note**

You should specify gamma in torque/angular velocity units. This is not the same as mass/time units, at least for some of the LAMMPS *units* options like "real" or "metal" that are not self-consistent.

In the current implementation, rather than have the user specify a viscosity, $\gamma$ is specified directly in torque/angular velocity units. If needed, $\gamma$ can be adjusted for atoms of different sizes (i.e. $\sigma$) by using the *scale* keyword.

---

### 2.276.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The *fix_modify* *respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is modifying torques. Default is the outermost level.

The torques due to this fix are imposed during an energy minimization, invoked by the *minimize* command. This fix should only be used with damped dynamics minimizers that allow for non-conservative forces. See the *min_style* command for details.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.276.5 Restrictions

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This fix requires that atoms store torque and angular velocity (omega) and a radius as defined by the *atom_style sphere* command.

All particles in the group must be finite-size spheres. They cannot be point particles.

### 2.276.6 Related commands

*fix viscous*, *fix damping/cundall*

### 2.276.7 Default

none

## 2.277 fix wall/lj93 command

Accelerator Variants: *wall/lj93/kk*

## 2.278 fix wall/lj126 command

Accelerator Variants: *wall/lj126/kk*

## 2.279 fix wall/lj1043 command

Accelerator Variants: *wall/lj1043/kk*

## 2.280 fix wall/colloid command

## 2.281 fix wall/harmonic command

Accelerator Variants: *wall/harmonic/kk*

## 2.282 fix wall/harmonic/outside command

## 2.283 fix wall/lepton command

## 2.284 fix wall/morse command

Accelerator Variants: *wall/morse/kk*

## 2.285 fix wall/table command

### 2.285.1 Syntax

```
fix ID group-ID style [tabstyle] [N] face args ... keyword value ...
```

- ID, group-ID are documented in *fix* command
- style = *wall/lj93* or *wall/lj126* or *wall/lj1043* or *wall/colloid* or *wall/harmonic* or *wall/harmonic/outside* or *wall/lepton* or *wall/morse* or *wall/table*
- tabstyle = *linear* or *spline* = method of table interpolation (only applies to *wall/table*)
- N = use N values in *linear* or *spline* interpolation (only applies to *wall/table*)
- one or more face/arg pairs may be appended
- face = *xlo* or *xhi* or *ylo* or *yhi* or *zlo* or *zhi*
- args for styles *lj93* or *lj126* or *lj1043* or *colloid* or *harmonic*

  ```
  args = coord epsilon sigma cutoff
  coord = position of wall = EDGE or constant or variable
    EDGE = current lo or hi edge of simulation box
    constant = number like 0.0 or -30.0 (distance units)
    variable = equal-style variable like v_x or v_wiggle
  epsilon = strength factor for wall-particle interaction (energy or energy/distance^
  ↪2 units)
    epsilon can be a variable (see below)
  sigma = size factor for wall-particle interaction (distance units)
    sigma can be a variable (see below)
  cutoff = distance from wall at which wall-particle interactions are cut off␣
  ↪(distance units)
  ```

- args for style *lepton*

  ```
  args = coord expression cutoff
  coord = position of wall = EDGE or constant or variable
    EDGE = current lo or hi edge of simulation box
    constant = number like 0.0 or -30.0 (distance units)
    variable = equal-style variable like v_x or v_wiggle
  expression = Lepton expression for the potential  (energy units)
  cutoff = distance from wall at which wall-particle interactions are cut off␣
  ↪(distance units)
  ```

- args for style *morse*

  ```
  args = coord D_0 alpha r_0 cutoff
  coord = position of wall = EDGE or constant or variable
    EDGE = current lo or hi edge of simulation box
  ```

```
    constant = number like 0.0 or -30.0 (distance units)
    variable = equal-style variable like v_x or v_wiggle
  D_0 = depth of the potential (energy units)
    D_0 can be a variable (see below)
  alpha = width factor for wall-particle interaction (1/distance units)
    alpha can be a variable (see below)
  r_0 = distance of the potential minimum from the face of region (distance units)
    r_0 can be a variable (see below)
  cutoff = distance from wall at which wall-particle interactions are cut off␣
  ↪(distance units)
```

- args for style *table*

```
args = coord filename keyword cutoff
coord = position of wall = EDGE or constant or variable
  EDGE = current lo or hi edge of simulation box
  constant = number like 0.0 or -30.0 (distance units)
  variable = equal-style variable like v_x or v_wiggle
filename = file containing tabulated energy and force values
keyword = section identifier to select a specific table in table file
cutoff = distance from wall at which wall-particle interactions are cut off␣
↪(distance units)
```

- zero or more keyword/value pairs may be appended
- keyword = *units* or *fld* or *pbc*

```
units value = lattice or box
  lattice = the wall position is defined in lattice units
  box = the wall position is defined in simulation box units
fld value = yes or no
  yes = invoke the wall constraint to be compatible with implicit FLD
  no = invoke the wall constraint in the normal way
pbc value = yes or no
  yes = allow periodic boundary in a wall dimension
  no = require non-perioidic boundaries in any wall dimension
```

### 2.285.2 Examples

```
fix wallhi all wall/lj93 xlo -1.0 1.0 1.0 2.5 units box
fix wallhi all wall/lj93 xhi EDGE 1.0 1.0 2.5
fix wallhi all wall/harmonic xhi EDGE 100.0 0.0 4.0 units box
fix wallhi all wall/morse xhi EDGE 1.0 1.0 1.0 2.5 units box
fix wallhi all wall/lj126 v_wiggle 23.2 1.0 1.0 2.5
fix zwalls all wall/colloid zlo 0.0 1.0 1.0 0.858 zhi 40.0 1.0 1.0 0.858
fix xwall mobile wall/table spline 200 EDGE -5.0 walltab.dat HARMONIC 4.0
fix xwalls mobile wall/lepton xlo -5.0 "k*(r-rc)^2;k=100.0" 4.0 xhi 5.0 "k*(r-rc)^2;
↪k=100.0" 4.0
```

### 2.285.3 Description

Bound the simulation domain on one or more of its faces with a flat wall that interacts with the atoms in the group by generating a force on the atom in a direction perpendicular to the wall. The energy of wall-particle interactions depends on the style.

For style *wall/lj93*, the energy E is given by the 9-3 Lennard-Jones potential:

$$E = \varepsilon \left[ \frac{2}{15} \left( \frac{\sigma}{r} \right)^9 - \left( \frac{\sigma}{r} \right)^3 \right] \qquad r < r_c$$

For style *wall/lj126*, the energy E is given by the 12-6 Lennard-Jones potential:

$$E = 4\varepsilon \left[ \left( \frac{\sigma}{r} \right)^{12} - \left( \frac{\sigma}{r} \right)^6 \right] \qquad r < r_c$$

For style *wall/lj1043*, the energy E is given by the 10-4-3 Lennard-Jones potential:

$$E = 2\pi\varepsilon \left[ \frac{2}{5} \left( \frac{\sigma}{r} \right)^{10} - \left( \frac{\sigma}{r} \right)^4 - \frac{\sqrt{(2)}\sigma^3}{3\left(r + \left(0.61/\sqrt{(2)}\right)\sigma\right)^3} \right] \qquad r < r_c$$

For style *wall/colloid*, the energy E is given by an integrated form of the *pair_style colloid* potential:

$$\begin{aligned} E =& \varepsilon \left[ \frac{\sigma^6}{7560} \left( \frac{6R - D}{D^7} + \frac{D + 8R}{(D + 2R)^7} \right) \right. \\ & \left. - \frac{1}{6} \left( \frac{2R(D + R) + D(D + 2R)\left[\ln D - \ln(D + 2R)\right]}{D(D + 2R)} \right) \right] \qquad r < r_c \end{aligned}$$

For style *wall/harmonic*, the energy E is given by a repulsive-only harmonic spring potential:

$$E = \varepsilon \quad (r - r_c)^2 \qquad r < r_c$$

For style *wall/harmonic/outside*, the energy E is given by an attractive-only harmonic spring potential for selected atoms group passing outside the wall placed at $w_0$ up to a cutoff distance $r_c$: .. math:

```
E = \epsilon \quad (r - w_0)^2 \qquad r < r_c
```

For style *wall/morse*, the energy E is given by a Morse potential:

$$E = D_0 \left[ e^{-2\alpha(r - r_0)} - 2e^{-\alpha(r - r_0)} \right] \qquad r < r_c$$

For style *wall/harmonic/outside*, the formulation avoids the presence of repulsive bands inside a controlled volume preventing the definition of a precise volumetric concentration (see example scripts for 2D visualization). Instead, the particles are allowed to go outside by a small amount, getting a finer definition of the concentration inside the controlled volume, as employed for the CMC determination in Barraud et al *(Barraud)*.

Added in version 28Mar2023.

For style *wall/lepton*, the energy E is provided as an Lepton expression string using "r" as the distance variable. The Lepton library, that the *wall/lepton* style interfaces with, evaluates this expression string at run time to compute the wall-particle energy. It also creates an analytical representation of the first derivative of this expression with respect to "r" and then uses that to compute the force between the wall and atoms in the fix group. The Lepton expression must be either enclosed in quotes or must not contain any whitespace so that LAMMPS recognizes it as a single keyword.

Optionally, the expression may use "rc" to refer to the cutoff distance for the given wall. Further constants in the expression can be defined in the same string as additional expressions separated by semicolons. The expression "k*(r-rc)^2;k=100.0" represents a repulsive-only harmonic spring as in fix *wall/harmonic* or *wall/harmonic/outside* with a force constant $K$ (same as $\varepsilon$ above) of 100 energy units. More details on the Lepton expression strings are given below.

Added in version 28Mar2023.

For style *wall/table*, the energy E and forces are determined from interpolation tables listed in one or more files as a function of distance. The interpolation tables are used to evaluate energy and forces between particles and the wall similar to how analytic formulas are used for the other wall styles.

The interpolation tables are created as a pre-computation by fitting cubic splines to the file values and interpolating energy and force values at each of *N* distances. During a simulation, the tables are used to interpolate energy and force values as needed for each wall and particle separated by a distance *R*. The interpolation is done in one of two styles: *linear* or *spline*.

For the *linear* style, the distance *R* is used to find the 2 surrounding table values from which an energy or force is computed by linear interpolation.

For the *spline* style, cubic spline coefficients are computed and stored for each of the *N* values in the table, one set of splines for energy, another for force. Note that these splines are different than the ones used to pre-compute the *N* values. Those splines were fit to the *Nfile* values in the tabulated file, where often *Nfile* < *N*. The distance *R* is used to find the appropriate set of spline coefficients which are used to evaluate a cubic polynomial which computes the energy or force.

For each wall a filename and a keyword must be provided as in the examples above. The filename specifies a file containing tabulated energy and force values. The keyword specifies a section of the file. The format of this file is described below.

In all cases, *r* is the distance from the particle to the wall at position *coord*, and $r_c$ is the *cutoff* distance at which the particle and wall no longer interact. The energy of the wall potential is shifted so that the wall-particle interaction energy is 0.0 at the cutoff distance.

Up to 6 walls or faces can be specified in a single command: *xlo*, *xhi*, *ylo*, *yhi*, *zlo*, *zhi*. A *lo* face interacts with particles near the lower side of the simulation box in that dimension. A *hi* face interacts with particles near the upper side of the simulation box in that dimension.

The position of each wall can be specified in one of 3 ways: as the EDGE of the simulation box, as a constant value, or as a variable. If EDGE is used, then the corresponding boundary of the current simulation box is used. If a numeric constant is specified then the wall is placed at that position in the appropriate dimension (x, y, or z). In both the EDGE and constant cases, the wall will never move. If the wall position is a variable, it should be specified as v_name, where name is an *equal-style variable* name. In this case the variable is evaluated each timestep and the result becomes the current position of the reflecting wall. Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent wall position. See examples below.

For the *wall/lj93* and *wall/lj126* and *wall/lj1043* styles, $\varepsilon$ and $\sigma$ are the usual Lennard-Jones parameters, which determine the strength and size of the particle as it interacts with the wall. Epsilon has energy units. Note that this $\varepsilon$ and $\sigma$ may be different than any $\varepsilon$ or $\sigma$ values defined for a pair style that computes particle-particle interactions.

The *wall/lj93* interaction is derived by integrating over a 3d half-lattice of Lennard-Jones 12/6 particles. The *wall/lj126* interaction is effectively a harder, more repulsive wall interaction. The *wall/lj1043* interaction is yet a different form of wall interaction, described in Magda et al in *(Magda)*.

For the *wall/colloid* style, *R* is the radius of the colloid particle, *D* is the distance from the surface of the colloid particle to the wall (r-R), and $\sigma$ is the size of a constituent LJ particle inside the colloid particle and wall. Note that the cutoff distance Rc in this case is the distance from the colloid particle center to the wall. The prefactor $\varepsilon$ can be thought of as an effective Hamaker constant with energy units for the strength of the colloid-wall interaction. More specifically, the $\varepsilon$ prefactor is $4\pi^2\rho_{wall}\rho_{colloid}\varepsilon\sigma^6$, where $\varepsilon$ and $\sigma$ are the LJ parameters for the constituent LJ particles. $\rho_{wall}$ and $\rho_{colloid}$ are the number density of the constituent particles, in the wall and colloid respectively, in units of 1/volume.

The *wall/colloid* interaction is derived by integrating over constituent LJ particles of size $\sigma$ within the colloid particle and a 3d half-lattice of Lennard-Jones 12/6 particles of size $\sigma$ in the wall. As mentioned in the preceding paragraph, the density of particles in the wall and colloid can be different, as specified by the $\varepsilon$ prefactor.

For the *wall/harmonic* and *wall/harmonic/outside* style, $\varepsilon$ is effectively the spring constant K, and has units (energy/distance^2). The input parameter $\sigma$ is ignored. The minimum energy position of the harmonic spring is at the *cutoff*. This is a repulsive-only spring since the interaction is truncated at the *cutoff*

For the *wall/morse* style, the three parameters are in this order: $D_0$ the depth of the potential, $\alpha$ the width parameter, and $r_0$ the location of the minimum. $D_0$ has energy units, $\alpha$ inverse distance units, and $r_0$ distance units.

For any wall that supports them, the $\epsilon$ and/or $\sigma$ and/or $\alpha$ parameter can be specified as an *equal-style variable*, in which case it should be specified as v_name, where name is the variable name. As with a variable wall position, the variable is evaluated each timestep and the result becomes the current epsilon or sigma of the wall. Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent wall interaction.

**Note**

For all of the styles (except wall/harmonic/outside), you must ensure that r is always > 0 for all particles in the group, or LAMMPS will generate an error. This means you cannot start your simulation with particles at the wall position *coord* (r = 0) or with particles on the wrong side of the wall (r < 0). For the *wall/lj93* and *wall/lj126* styles, the energy of the wall/particle interaction (and hence the force on the particle) blows up as r -> 0. The *wall/colloid* style is even more restrictive, since the energy blows up as D = r-R -> 0. This means the finite-size particles of radius R must be a distance larger than R from the wall position *coord*. The *harmonic* style is a softer potential and does not blow up as r -> 0, but you must use a large enough $\epsilon$ that particles always reamin on the correct side of the wall (r > 0).

The *units* keyword determines the meaning of the distance units used to define a wall position, but only when a numeric constant or variable is used. It is not relevant when EDGE is used to specify a face position. In the variable case, the variable is assumed to produce a value compatible with the *units* setting you specify.

A *box* value selects standard distance units as defined by the *units* command, e.g. Angstroms for units = real or metal. A *lattice* value means the distance units are in lattice spacings. The *lattice* command must have been previously used to define the lattice spacings.

The *fld* keyword can be used with a *yes* setting to invoke the wall constraint before pairwise interactions are computed. This allows an implicit FLD model using *pair_style lubricateU* to include the wall force in its calculations. If the setting is *no*, wall forces are imposed after pairwise interactions, in the usual manner.

The *pbc* keyword can be used with a *yes* setting to allow walls to be specified in a periodic dimension. See the *boundary* command for options on simulation box boundaries. The default for *pbc* is *no*, which means the system must be non-periodic when using a wall. But you may wish to use a periodic box. E.g. to allow some particles to interact with the wall via the fix group-ID, and others to pass through it and wrap around a periodic box. In this case you should ensure that the wall is sufficiently far enough away from the box boundary. If you do not, then particles may interact with both the wall and with periodic images on the other side of the box, which is probably not what you want.

---

Here are examples of variable definitions that move the wall position in a time-dependent fashion using equal-style *variables*. The wall interaction parameters (epsilon, sigma) could be varied with additional variable definitions.

```
variable ramp equal ramp(0,10)
fix 1 all wall xlo v_ramp 1.0 1.0 2.5

variable linear equal vdisplace(0,20)
fix 1 all wall xlo v_linear 1.0 1.0 2.5

variable wiggle equal swiggle(0.0,5.0,3.0)
fix 1 all wall xlo v_wiggle 1.0 1.0 2.5

variable wiggle equal cwiggle(0.0,5.0,3.0)
fix 1 all wall xlo v_wiggle 1.0 1.0 2.5
```

The *ramp(lo,hi)* function adjusts the wall position linearly from *lo* to *hi* over the course of a run. The *vdisplace(c0,velocity)* function does something similar using the equation *position = c0 + velocity*delta*, where *delta* is the elapsed time.

The *swiggle(c0,A,period)* function causes the wall position to oscillate sinusoidally according to this equation, where *omega = 2 PI / period*:

```
position = c0 + A sin(omega*delta)
```

The *cwiggle(c0,A,period)* function causes the wall position to oscillate sinusoidally according to this equation, which will have an initial wall velocity of 0.0, and thus may impose a gentler perturbation on the particles:

```
position = c0 + A (1 - cos(omega*delta))
```

---

### 2.285.4 Lepton expression syntax and features

Lepton supports the following operators in expressions:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| + | Add | - | Subtract | * | Multiply | / | Divide | ^ | Power |

The following mathematical functions are available:

| | | | |
|---|---|---|---|
| sqrt(x) | Square root | exp(x) | Exponential |
| log(x) | Natural logarithm | sin(x) | Sine (angle in radians) |
| cos(x) | Cosine (angle in radians) | sec(x) | Secant (angle in radians) |
| csc(x) | Cosecant (angle in radians) | tan(x) | Tangent (angle in radians) |
| cot(x) | Cotangent (angle in radians) | asin(x) | Inverse sine (in radians) |
| acos(x) | Inverse cosine (in radians) | atan(x) | Inverse tangent (in radians) |
| sinh(x) | Hyperbolic sine | cosh(x) | Hyperbolic cosine |
| tanh(x) | Hyperbolic tangent | erf(x) | Error function |
| erfc(x) | Complementary Error function | abs(x) | Absolute value |
| min(x,y) | Minimum of two values | max(x,y) | Maximum of two values |
| delta(x) | delta(x) is 1 for $x = 0$, otherwise 0 | step(x) | step(x) is 0 for $x < 0$, otherwise 1 |

Numbers may be given in either decimal or exponential form. All of the following are valid numbers: *5*, *-3.1*, *1e6*, and *3.12e-2*.

As an extension to the standard Lepton syntax, it is also possible to use LAMMPS *variables* in the format "v_name". Before evaluating the expression, "v_name" will be replaced with the value of the variable "name". This is compatible with all kinds of scalar variables, but not with vectors, arrays, local, or per-atom variables. If necessary, a custom scalar variable needs to be defined that can access the desired (single) item from a non-scalar variable. As an example, the following lines will instruct LAMMPS to ramp the force constant for a harmonic bond from 100.0 to 200.0 during the next run:

```
variable fconst equal ramp(100.0, 200)
bond_style lepton
bond_coeff 1 1.5 "v_fconst * (r^2)"
```

An expression may be followed by definitions for intermediate values that appear in the expression. A semicolon ";" is used as a delimiter between value definitions. For example, the expression:

```
a^2+a*b+b^2; a=a1+a2; b=b1+b2
```

is exactly equivalent to

```
(a1+a2)^2+(a1+a2)*(b1+b2)+(b1+b2)^2
```

The definition of an intermediate value may itself involve other intermediate values. Whitespace and quotation characters (’’ and ‘’’’) are ignored. All uses of a value must appear *before* that value’s definition. For efficiency reasons, the expression string is parsed, optimized, and then stored in an internal, pre-parsed representation for evaluation.

Evaluating a Lepton expression is typically between 2.5 and 5 times slower than the corresponding compiled and optimized C++ code. If additional speed or GPU acceleration (via GPU or KOKKOS) is required, the interaction can be represented as a table. Suitable table files can be created either internally using the *pair_write* or *bond_write* command or through the Python scripts in the *tools/tabulate* folder.

---

### 2.285.5 Table file format

Suitable tables for use with fix *wall/table* can be created by the Python code in the `tools/tabulate` folder of the LAMMPS source code distribution.

The format of a tabulated file is as follows (without the parenthesized comments):

```
# Tabulated wall potential UNITS: real

HARMONIC                        (keyword is the first text on a line)
N 100 FP 200 200
                                (blank line)
1  0.04    1568.16    792.00   (index, distance to wall, energy, force)
2  0.08    1536.64    784.00
3  0.12    1505.44    776.00
...
99  3.96      0.16      8.00
100  4.00     0         0
```

A section begins with a non-blank line whose first character is not a “#”; blank lines or lines starting with “#” can be used as comments between sections. The first line begins with a keyword which identifies the section. The line can contain additional text, but the initial text must match the argument specified in the fix *wall/table* command. The next line lists (in any order) one or more parameters for the table. Each parameter is a keyword followed by one or more numeric values.

The parameter “N” is required and its value is the number of table entries that follow. Note that this may be different than the *N* specified in the fix *wall/table* command. Let Ntable = *N* in the fix command, and Nfile = “N” in the tabulated file. What LAMMPS does is a preliminary interpolation by creating splines using the Nfile tabulated values as nodal points. It uses these to interpolate as needed to generate energy and force values at Ntable different points. The resulting tables of length Ntable are then used as described above, when computing energy and force for wall-particle interactions. This means that if you want the interpolation tables of length Ntable to match exactly what is in the tabulated file (with effectively no preliminary interpolation), you should set Ntable = Nfile.

---

### 2.285.6 Dump image info

Added in version 11Feb2026.

These wall fixes support the *fix* keyword of *dump image*. The fixes will pass geometry information about the walls to *dump image* so that the walls will be included in the rendered image. Please note, that for *2d systems*, a wall rendered as a plane would be invisible and it is thus rendered as a cylinder.

The color of the wall is by default that of the first atom type when using color styles “type” or “element”. With color style “const” the default value of “white” can be changed using *dump_modify fcolor*. The transparency is by default fully opaque and can be changed with *dump_modify ftrans*.

For 2d systems, the *fflag1* setting determines whether the cylinder representing the wall is capped with a sphere at the ends: 0 means no caps, 1 means the lower end is capped, 2 means the upper end is capped, and 3 means both ends are capped. The *fflag2* setting allows to adjust the radius of the rendered cylinder. It should be set to a value > 0 or the cylinder will not be visible since the diameter is set internally to zero due to lack of a suitable heuristic for deriving a meaningful diameter for all types of walls and unit settings.

For 3d systems, both *fflag1* and *fflag2* are ignored.

---

### 2.285.7 Restart, fix_modify, output, run start/stop, minimize info

No information about these fixes are written to *binary restart files*.

The *fix_modify* *energy* option is supported by these fixes to add the energy of interaction between atoms and all the specified walls to the global potential energy of the system as part of *thermodynamic output*. The default setting for these fixes is *fix_modify energy no*.

The *fix_modify* *virial* option is supported by these fixes to add the contribution due to the interaction between atoms and all the specified walls to both the global pressure and per-atom stress of the system via the *compute pressure* and *compute stress/atom* commands. The former can be accessed by *thermodynamic output*. The default setting for these fixes is *fix_modify virial no*.

The *fix_modify* *respa* option is supported by these fixes. This allows to set at which level of the *r-RESPA* integrator a fix is adding its forces. Default is the outermost level.

These fixes compute a global scalar energy and a global vector of forces, which can be accessed by various *output commands*. Note that the scalar energy is the sum of interactions with all defined walls. If you want the energy on a per-wall basis, you need to use multiple fix wall commands. The length of the vector is equal to the number of walls defined by the fix. Each vector value is the normal force on a specific wall. Note that an outward force on a wall will be a negative value for *lo* walls and a positive value for *hi* walls. The scalar and vector values calculated by this fix are "extensive".

No parameter of these fixes can be used with the *start/stop* keywords of the *run* command.

The forces due to these fixes *are* imposed during an energy minimization, invoked by the *minimize* command.

> **Note**
>
> If you want the atom/wall interaction energy to be included in the total potential energy of the system (the quantity being minimized), you MUST enable the *fix_modify* *energy* option for this fix.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.285.8 Restrictions

Fix *wall/lepton* is part of the LEPTON package and only enabled if LAMMPS was built with this package. See the *Build package* page for more info.

### 2.285.9 Related commands

*fix wall/reflect*, *fix wall/gran*, *fix wall/region*

### 2.285.10 Default

The option defaults units = lattice, fld = no, and pbc = no.

---

**(Magda)** Magda, Tirrell, Davis, J Chem Phys, 83, 1888-1901 (1985); erratum in JCP 84, 2901 (1986).

**(Barraud)** Barraud, Dalmazzone, Moureta, De Bruin, Creton, Pasquier, Lachet and Nieto-Draghi, Langmuir, 41 (11), 7272-7282 (2025).

## 2.286 fix wall/body/polygon command

### 2.286.1 Syntax

```
fix ID group-ID wall/body/polygon k_n c_n c_t wallstyle args keyword values ...
```

- ID, group-ID are documented in *fix* command
- wall/body/polygon = style name of this fix command
- k_n = normal repulsion strength (force/distance or pressure units)
- c_n = normal damping coefficient (force/distance or pressure units)
- c_t = tangential damping coefficient (force/distance or pressure units)
- wallstyle = *xplane* or *yplane*
- args = list of arguments for a particular style

  ```
  xplane or yplane args = lo hi
    lo,hi = position of lower and upper plane (distance units), either can be NULL)
  ```

- zero or more keyword/value pairs may be appended to args
- keyword = *wiggle*

  ```
  wiggle values = dim amplitude period
    dim = x or y or z
    amplitude = size of oscillation (distance units)
    period = time of oscillation (time units)
  ```

### 2.286.2 Examples

```
fix 1 all wall/body/polygon 1000.0 20.0 5.0 xplane -10.0 10.0
```

### 2.286.3 Description

This fix is for use with 2d models of body particles of style *rounded/polygon*. It bounds the simulation domain with wall(s). All particles in the group interact with the wall when they are close enough to touch it. The nature of the interaction between the wall and the polygon particles is the same as that between the polygon particles themselves, which is similar to a Hookean potential. See the *Howto body* page for more details on using body particles.

The parameters *k_n*, *c_n*, *c_t* have the same meaning and units as those specified with the *pair_style body/rounded/polygon* command.

The *wallstyle* is planar and allows to specify a pair of walls in x- and y direction each. Wall positions are given by *lo* and *hi*. Either of the values can be specified as NULL if a single wall per dimension is desired. Optionally, the wall can be moving, if the *wiggle* keyword is appended.

For the *wiggle* keyword, the wall oscillates sinusoidally, similar to the oscillations of particles which can be specified by the *fix move* command. This is useful in packing simulations of particles. The arguments to the *wiggle* keyword specify a dimension for the motion, as well as its *amplitude* and *period*. Note that if the dimension is in the plane of the wall, this is effectively a shearing motion. If the dimension is perpendicular to the wall, it is more of a shaking motion.

Each timestep, the position of a wiggled wall in the appropriate *dim* is set according to this equation:

```
position = coord + A - A cos (omega * delta)
```

where *coord* is the specified initial position of the wall, *A* is the *amplitude*, *omega* is 2 PI / *period*, and *delta* is the time elapsed since the fix was specified. The velocity of the wall is set to the derivative of this expression.

---

### 2.286.4 Dump image info

Added in version 11Feb2026.

This fix supports the *fix* keyword of *dump image*. The fix will pass geometry information about the walls to *dump image* so that the walls will be included in the rendered image. Please note, that for *2d systems*, a wall rendered as a plane would be invisible and it is thus rendered as a cylinder.

The color of the wall is by default that of the first atom type when using color styles "type" or "element". With color style "const" the default value of "white" can be changed using *dump_modify fcolor*. The transparency is by default fully opaque and can be changed with *dump_modify ftrans*.

The *fflag1* setting determines whether the cylinder representing the wall is capped with a sphere at the ends: 0 means no caps, 1 means the lower end is capped, 2 means the upper end is capped, and 3 means both ends are capped. The *fflag2* setting allows to set the radius of the rendered cylinders.

---

### 2.286.5 Restart, fix_modify, output, run start/stop, minimize info

None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.286.6 Restrictions

This fix is part of the BODY package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Any dimension (xy) that has a wall must be non-periodic.

### 2.286.7 Related commands

*atom_style body*, *pair_style body/rounded/polygon*

### 2.286.8 Default

none

## 2.287 fix wall/body/polyhedron command

### 2.287.1 Syntax

```
fix ID group-ID wall/body/polyhedron k_n c_n c_t wallstyle args keyword values ...
```

- ID, group-ID are documented in *fix* command
- wall/body/polyhedron = style name of this fix command
- k_n = normal repulsion strength (force/distance units or pressure units - see discussion below)
- c_n = normal damping coefficient (force/distance units or pressure units - see discussion below)
- c_t = tangential damping coefficient (force/distance units or pressure units - see discussion below)
- wallstyle = *xplane* or *yplane* or *zplane*
- args = list of arguments for a particular style

  *xplane* or *yplane* or *zplane* args = lo hi
    lo,hi = position of lower and upper plane (distance units), either can be NULL)

- zero or more keyword/value pairs may be appended to args
- keyword = *wiggle*

  *wiggle* values = dim amplitude period
    dim = *x* or *y* or *z*
    amplitude = size of oscillation (distance units)
    period = time of oscillation (time units)

### 2.287.2 Examples

```
fix 1 all wall/body/polyhedron 1000.0 20.0 5.0 xplane -10.0 10.0
```

### 2.287.3 Description

This fix is for use with 3d models of body particles of style *rounded/polyhedron*. It bounds the simulation domain with wall(s). All particles in the group interact with the wall when they are close enough to touch it. The nature of the interaction between the wall and the polygon particles is the same as that between the polygon particles themselves, which is similar to a Hookean potential. See the *Howto body* page for more details on using body particles.

The parameters *k_n*, *c_n*, *c_t* have the same meaning and units as those specified with the *pair_style body/rounded/polyhedron* command.

The *wallstyle* can be planar or cylindrical. The 3 planar options specify a pair of walls in a dimension. Wall positions are given by *lo* and *hi*. Either of the values can be specified as NULL if a single wall is desired.

Optionally, the wall can be moving, if the *wiggle* keyword is appended.

For the *wiggle* keyword, the wall oscillates sinusoidally, similar to the oscillations of particles which can be specified by the *fix move* command. This is useful in packing simulations of particles. The arguments to the *wiggle* keyword specify a dimension for the motion, as well as its *amplitude* and *period*. Note that if the dimension is in the plane of the wall, this is effectively a shearing motion. If the dimension is perpendicular to the wall, it is more of a shaking motion.

Each timestep, the position of a wiggled wall in the appropriate *dim* is set according to this equation:

```
position = coord + A - A cos (omega * delta)
```

where *coord* is the specified initial position of the wall, *A* is the *amplitude*, *omega* is 2 PI / *period*, and *delta* is the time elapsed since the fix was specified. The velocity of the wall is set to the derivative of this expression.

---

### 2.287.4 Dump image info

Added in version 11Feb2026.

This fix supports the *fix* keyword of *dump image*. The fix will pass geometry information about *xplane*, *yplane*, and *zplane* style walls to *dump image* so that the walls will be included in the rendered image. Please note, that for *2d systems*, a wall rendered as a plane would be invisible and it is thus rendered as a cylinder.

The color of the wall is by default that of the first atom type when using color styles "type" or "element". With color style "const" the default value of "white" can be changed using *dump_modify fcolor*. The transparency is by default fully opaque and can be changed globally with *dump_modify ftrans*.

For 2d systems, the *fflag1* setting determines whether the cylinder representing the wall is capped with a sphere at the ends: 0 means no caps, 1 means the lower end is capped, 2 means the upper end is capped, and 3 means both ends are capped. The *fflag2* setting allows to set the radius of the rendered cylinders.

For 3d systems, both *fflag1* and *fflag2* are ignored.

---

### 2.287.5 Restart, fix_modify, output, run start/stop, minimize info

None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.287.6 Restrictions

This fix is part of the BODY package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Any dimension (xyz) that has a wall must be non-periodic.

### 2.287.7 Related commands

*atom_style body*, *pair_style body/rounded/polyhedron*

### 2.287.8 Default

none

# 2.288 fix wall/ees command

# 2.289 fix wall/region/ees command

## 2.289.1 Syntax

```
fix ID group-ID style args
```

- ID, group-ID are documented in *fix* command
- style = *wall/ees* or *wall/region/ees*

```
args for style wall/ees: one or more face parameters groups may be appended
face = xlo or xhi or ylo or yhi or zlo or zhi
parameters = coord epsilon sigma cutoff
  coord = position of wall = EDGE or constant or variable
    EDGE = current lo or hi edge of simulation box
    constant = number like 0.0 or -30.0 (distance units)
    variable = equal-style variable like v_x or v_wiggle
  epsilon = strength factor for wall-particle interaction (energy or energy/
↪distance^2 units)
    epsilon can be a variable (see below)
  sigma = size factor for wall-particle interaction (distance units)
    sigma can be a variable (see below)
  cutoff = distance from wall at which wall-particle interaction is cut off␣
↪(distance units)
```

```
args for style wall/region/ees: region-ID epsilon sigma cutoff
  region-ID = region whose boundary will act as wall
  epsilon = strength factor for wall-particle interaction (energy or energy/
↪distance^2 units)
  sigma = size factor for wall-particle interaction (distance units)
  cutoff = distance from wall at which wall-particle interaction is cut off␣
↪(distance units)
```

## 2.289.2 Examples

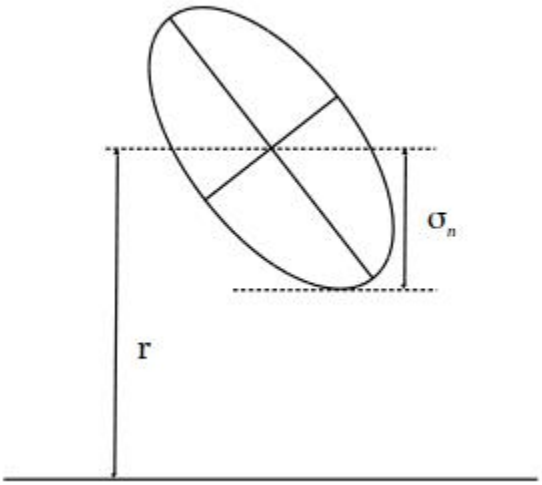
```
fix wallhi all wall/ees xlo -1.0 1.0 1.0 2.5 units box
fix wallhi all wall/ees xhi EDGE 1.0 1.0 2.5
fix wallhi all wall/ees v_wiggle 23.2 1.0 1.0 2.5
fix zwalls all wall/ees zlo 0.0 1.0 1.0 0.858 zhi 40.0 1.0 1.0 0.858

fix ees_cube all wall/region/ees myCube 1.0 1.0 2.5
```

## 2.289.3 Description

Fix *wall/ees* bounds the simulation domain on one or more of its faces with a flat wall that interacts with the ellipsoidal atoms in the group by generating a force on the atom in a direction perpendicular to the wall and a torque parallel with the wall. The energy of wall-particle interactions E is given by:

$$E = \varepsilon \left[ \frac{2\sigma_{LJ}^{12}\left(7r^5 + 14r^3\sigma_n^2 + 3r\sigma_n^4\right)}{945\left(r^2 - \sigma_n^2\right)^7} - \frac{\sigma_{LJ}^6\left(2r\sigma_n^3 + \sigma_n^2\left(r^2 - \sigma_n^2\right)\log\left[\frac{r-\sigma_n}{r+\sigma_n}\right]\right)}{12\sigma_n^5\left(r^2 - \sigma_n^2\right)} \right] \qquad \sigma_n < r < r_c$$

Introduced by Babadi and Ejtehadi in *(Babadi2)*. Here, $r$ is the distance from the particle to the wall at position *coord*, and Rc is the *cutoff* distance at which the particle and wall no longer interact. Also, $\sigma_n$ is the distance between center of ellipsoid and the nearest point of its surface to the wall as shown below.




Details of using this command and specifications are the same as fix/wall command. You can also find an example in USER/ees/ under examples/ directory.

The prefactor $\varepsilon$ can be thought of as an effective Hamaker constant with energy units for the strength of the ellipsoid-wall interaction. More specifically, the $\varepsilon$ prefactor is

$$8\pi^2 \quad \rho_{wall} \quad \rho_{ellipsoid} \quad \varepsilon \quad \sigma_a \quad \sigma_b \quad \sigma_c$$

where $\varepsilon$ is the LJ energy parameter for the constituent LJ particles and $\sigma_a$, $\sigma_b$, and $\sigma_c$ are the radii of the ellipsoidal particles. $\rho_{wall}$ and $\rho_{ellipsoid}$ are the number density of the constituent particles, in the wall and ellipsoid respectively, in units of 1/volume.

**Note**

You must ensure that r is always bigger than $\sigma_n$ for all particles in the group, or LAMMPS will generate an error. This means you cannot start your simulation with particles touching the wall position *coord* ($r = \sigma_n$) or with particles penetrating the wall ($0 =< r < \sigma_n$) or with particles on the wrong side of the wall ($r < 0$).

Fix *wall/region/ees* treats the surface of the geometric region defined by the *region-ID* as a bounding wall which interacts with nearby ellipsoidal particles according to the EES potential introduced above.

Other details of this command are the same as for the *fix wall/region* command. One may also find an example of using this fix in the examples/PACKAGES/ees/ directory.

---

## 2.289.4 Restart, fix_modify, output, run start/stop, minimize info

No information about these fixes are written to *binary restart files*.

The *fix_modify energy* option is supported by these fixes to add the energy of interaction between atoms and all the specified walls or region wall to the global potential energy of the system as part of *thermodynamic output*. The default settings for these fixes are *fix_modify energy no*.

The *fix_modify* *respa* option is supported by these fixes. This allows to set at which level of the *r-RESPA* integrator the fix is adding its forces. Default is the outermost level.

These fixes computes a global scalar and a global vector of forces, which can be accessed by various *output commands*. See the *fix wall* command for a description of the scalar and vector.

No parameter of these fixes can be used with the *start/stop* keywords of the *run* command.

The forces due to these fixes are imposed during an energy minimization, invoked by the *minimize* command.

**Note**

If you want the atom/wall interaction energy to be included in the total potential energy of the system (the quantity being minimized), you MUST enable the *fix_modify* *energy* option for this fix.

---

## 2.289.5 Dump image info

Added in version 11Feb2026.

Fix *wall/ees* fix supports the *fix* keyword of *dump image*. The fix will pass geometry information about the walls to *dump image* so that the walls will be included in the rendered image. Please note, that for *2d systems*, a wall rendered as a plane would be invisible and it is thus rendered as a cylinder. Fix *wall/ees/region* does **not** support graphics info, but the region can be visualized with the *region* keyword of *dump image*.

The color of the wall is by default that of the first atom type when using color styles "type" or "element". With color style "const" the default value of "white" can be changed using *dump_modify fcolor*. The transparency is by default fully opaque and can be changed with *dump_modify ftrans*.

For 2d systems, the *fflag1* setting determines whether the cylinder representing the wall is capped with a sphere at the ends: 0 means no caps, 1 means the lower end is capped, 2 means the upper end is capped, and 3 means both ends are capped. The *fflag2* setting allows to adjust the radius of the rendered cylinder. It should be set to a value > 0 or the cylinder will not be visible since the diameter is set internally to zero due to lack of a suitable heuristic for deriving a meaningful diameter for all types of walls and unit settings.

For 3d systems, both *fflag1* and *fflag2* are ignored.

---

## 2.289.6 Restrictions

These fixes are part of the EXTRA-FIX package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

These fixes requires that atoms be ellipsoids as defined by the *atom_style ellipsoid* command.

## 2.289.7 Related commands

*fix wall*, *pair resquared*

## 2.289.8 Default

none

---

**(Babadi2)** Babadi and Ejtehadi, EPL, 77 (2007) 23002.

# 2.290 fix wall/flow command

Accelerator Variants: *wall/flow/kk*

## 2.290.1 Syntax

```
fix ID group-ID wall/flow axis vflow T seed N coords ... keyword value
```

- ID, group-ID are documented in *fix* command
- wall/flow = style name of this fix command
- axis = flow axis ($x$, $y$, or $z$)
- vflow = generated flow velocity in *axis* direction (velocity units)
- T = flow temperature (temperature units)
- seed = random seed for stochasticity (positive integer)
- N = number of walls
- coords = list of N wall positions along the *axis* direction in ascending order (distance units)
- zero or more keyword/value pairs may be appended
- keyword = *units*

```
units value = lattice or box
  lattice = wall positions are defined in lattice units
  box = the wall positions are defined in simulation box units
```

## 2.290.2 Examples

```
fix 1 all wall/flow x 0.4 1.5 593894 4 2.0 4.0 6.0 8.0
```

## 2.290.3 Description

Added in version 17Apr2024.

This fix implements flow boundary conditions (FBC) introduced in *(Pavlov1)* and *(Pavlov2)*. The goal is to generate a stationary flow with a shifted Maxwell velocity distribution:

$$f_a(v_a) \propto \exp\left(-\frac{m(v_a - v_{\text{flow}})^2}{2kBT}\right)$$

where $v_a$ is the component of velocity along the specified *axis* argument (a = x,y,z), $v_{\text{flow}}$ is the flow velocity specified as the *vflow* argument, $T$ is the specified flow temperature, $m$ is the particle mass, and $kB$ is the Boltzmann constant.

This is achieved by defining a series of $N$ transparent walls along the flow *axis* direction. Each wall is at the specified position listed in the *coords* argument. Note that an additional transparent wall is defined by the code at the boundary of the (periodic) simulation domain in the *axis* direction. So there are effectively N+1 walls.

Each time a particle in the specified group passes through one of the transparent walls, its velocity is re-assigned. Particles not in the group do not interact with the wall. This can be used, for example, to add obstacles composed of atoms, or to simulate a solution of complex molecules in a one-atom liquid (note that the fix has been tested for one-atom systems only).

Conceptually, the velocity re-assignment represents creation of a new particle within the system with simultaneous removal of the particle which passed through the wall. The velocity components in directions parallel to the wall

are re-assigned according to the standard Maxwell velocity distribution for the specified temperature $T$. The velocity component perpendicular to the wall is re-assigned according to the shifted Maxwell distribution defined above:

$$f_{\text{a generated}}(v_a) \propto v_a f_a(v_a)$$

It can be shown that for an ideal-gas scenario this procedure makes the velocity distribution of particles between walls exactly as desired.

Since in most cases simulated systems are not an ideal gas, multiple walls can be defined, since a single wall may not be sufficient for maintaining a stationary flow without "congestion" which can manifest itself as regions in the flow with increased particle density located upstream from static obstacles.

For the same reason, the actual temperature and velocity of the generated flow may differ from what is requested. The degree of discrepancy is determined by how different from an ideal gas the simulated system is. Therefore, a calibration procedure may be required for such a system as described in *(Pavlov)*.

Note that the interactions between particles on different sides of a transparent wall are not disabled or neglected. Likewise particle positions are not altered by the velocity reassignment. This removes the need to modify the force field to work correctly in cases when a particle is close to a wall.

For example, if particle positions were uniformly redistributed across the surface of a wall, two particles could end up too close to each other, potentially causing the simulation to explode. However due to this compromise, some collective phenomena such as regions with increased/decreased density or collective movements are not fully removed when particles cross a wall. This unwanted consequence can also be potentially mitigated by using more multiple walls.

**Note**

When the specified flow has a high velocity, a lost atoms error can occur (see *error messages*). If this happens, you should ensure the checks for neighbor list rebuilds, set via the *neigh_modify* command, are as conservative as possible (every timestep if needed). Those are the default settings.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.290.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

None of the *fix_modify* options are relevant to this fix.

No global or per-atom quantities are stored by this fix for access by various *output commands*.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

This fix is not invoked during *energy minimization*.

### 2.290.5 Restrictions

Fix *wall_flow* is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Flow boundary conditions should not be used with rigid bodies such as those defined by a "fix rigid" command.

This fix can only be used with periodic boundary conditions along the flow axis. The size of the box in this direction must not change. Also, the fix is designed to work only in an orthogonal simulation box.

### 2.290.6 Related commands

*fix wall/reflect* command

### 2.290.7 Default

The default for the units keyword is lattice.

---

**(Pavlov1)** Pavlov, Kolotinskii, Stegailov, "GPU-Based Molecular Dynamics of Turbulent Liquid Flows with OpenMM", Proceedings of PPAM-2022, LNCS (Springer), vol. 13826, pp. 346-358 (2023)

**(Pavlov2)** Pavlov, Galigerov, Kolotinskii, Nikolskiy, Stegailov, "GPU-based Molecular Dynamics of Fluid Flows: Reaching for Turbulence", Int. J. High Perf. Comp. Appl., (2024)

## 2.291 fix wall/gran command

Accelerator Variants: *wall/gran/kk*

### 2.291.1 Syntax

```
fix ID group-ID wall/gran fstyle fstyle_params wallstyle args keyword values ...
```

- ID, group-ID are documented in *fix* command
- wall/gran = style name of this fix command
- fstyle = style of force interactions between particles and wall

  ```
  possible choices: hooke, hooke/history, hertz/history, granular
  ```

- fstyle_params = parameters associated with force interaction style

  ```
  For hooke, hooke/history, and hertz/history, fstyle_params are:
       Kn = elastic constant for normal particle repulsion (force/distance units or␣
  ↪pressure units - see discussion below)
       Kt = elastic constant for tangential contact (force/distance units or␣
  ↪pressure units - see discussion below)
       gamma_n = damping coefficient for collisions in normal direction (1/time␣
  ↪units or 1/time-distance units - see discussion below)
       gamma_t = damping coefficient for collisions in tangential direction (1/time␣
  ↪units or 1/time-distance units - see discussion below)
       xmu = static yield criterion (unitless value between 0.0 and 1.0e4)
       dampflag = 0 or 1 if tangential damping force is excluded or included
       optional keyword = limit_damping, limit damping to prevent attractive␣
  ↪interaction
  ```

```
For granular, fstyle_params are set using the same syntax as for the pair_coeff␣
↪command of pair_style granular
```

- wallstyle = *xplane* or *yplane* or *zplane*
- args = list of arguments for a particular style

```
xplane or yplane or zplane args = lo hi
  lo,hi = position of lower and upper plane (distance units), either can be NULL)
```

- zero or more keyword/value pairs may be appended to args
- keyword = *wiggle* or *shear* or *contacts* or *temperature*

```
 wiggle values = dim amplitude period
   dim = x or y or z
   amplitude = size of oscillation (distance units)
   period = time of oscillation (time units)
 shear values = dim vshear
   dim = x or y or z
   vshear = magnitude of shear velocity (velocity units)
contacts value = none
   generate contact information for each particle
temperature value = temperature
   specify temperature of wall
```

### 2.291.2 Examples

```
fix 1 all wall/gran hooke  200000.0 NULL 50.0 NULL 0.5 0 xplane -10.0 10.0
fix 1 all wall/gran hooke/history 200000.0 NULL 50.0 NULL 0.5 0 zplane 0.0 NULL
fix 3 all wall/gran granular hooke 1000.0 50.0 tangential linear_nohistory 1.0 0.4␣
↪damping velocity region myBox
fix 4 all wall/gran granular jkr 1e5 1500.0 0.3 10.0 tangential mindlin NULL 1.0 0.5␣
↪rolling sds 500.0 200.0 0.5 twisting marshall region myCone
fix 5 all wall/gran granular dmt 1e5 0.2 0.3 10.0 tangential mindlin NULL 1.0 0.5␣
↪rolling sds 500.0 200.0 0.5 twisting marshall damping tsuji heat 10 region myCone␣
↪temperature 1.0
fix 6 all wall/gran hooke  200000.0 NULL 50.0 NULL 0.5 0 xplane -10.0 10.0 contacts
```

### 2.291.3 Description

Bound the simulation domain of a granular system with a frictional wall. All particles in the group interact with the wall when they are close enough to touch it.

The nature of the wall/particle interactions are determined by the *fstyle* setting. It can be any of the styles defined by the *pair_style gran/** or the more general *pair_style granular* commands. Currently the options are *hooke*, *hooke/history*, or *hertz/history* for the former, and *granular* with all the possible options of the associated *pair_coeff* command for the latter. The equation for the force between the wall and particles touching it is the same as the corresponding equation on the *pair_style gran/** and *pair_style granular* doc pages, in the limit of one of the two particles going to infinite radius and mass (flat wall). Specifically, delta = radius - r = overlap of particle with wall, m_eff = mass of particle, and the effective radius of contact = RiRj/Ri+Rj is set to the radius of the particle.

The parameters *Kn*, *Kt*, *gamma_n*, *gamma_t*, *xmu*, *dampflag*, and the optional keyword *limit_damping* have the same meaning and units as those specified with the *pair_style gran/** commands. This means a NULL can be used for either *Kt* or *gamma_t* as described on that page. If a NULL is used for *Kt*, then a default value is used where *Kt* = 2/7 *Kn*. If a NULL is used for *gamma_t*, then a default value is used where *gamma_t* = 1/2 *gamma_n*.

Changed in version 4Jul2026.

**Note**

In versions of LAMMPS between 28Mar23 and Jun26, the gran/hooke tangential friction calculation accidentally included an extra factor of the contact radius. The granular models or KOKKOS versions were not affected.

All the model choices for cohesion, tangential friction, rolling friction and twisting friction supported by the *pair_style granular* through its *pair_coeff* command are also supported for walls. These are discussed in greater detail on the doc page for *pair_style granular*.

**Note**

When *fstyle granular* is specified, the associated *fstyle_params* are taken as those for a wall/particle interaction. For example, for the *hertz/material* normal contact model with $E = 960$ and $\nu = 0.2$, the effective Young's modulus for a wall/particle interaction is computed as $E_{eff} = \frac{960}{2(1-0.2^2)} = 500$. Any pair coefficients defined by *pair_style granular* are not taken into consideration. To model different wall/particle interactions for particles of different material types, the user may define multiple fix wall/gran commands operating on separate groups (e.g. based on particle type) each with a different wall/particle effective Young's modulus.

Note that you can choose a different force styles and/or different values for the wall/particle coefficients than for particle/particle interactions. E.g. if you wish to model the wall as a different material.

**Note**

As discussed on the page for *pair_style gran/**, versions of LAMMPS before 9Jan09 used a different equation for Hertzian interactions. This means Hertzian wall/particle interactions have also changed. They now include a sqrt(radius) term which was not present before. Also the previous versions used Kn and Kt from the pairwise interaction and hardwired dampflag to 1, rather than letting them be specified directly. This means you can set the values of the wall/particle coefficients appropriately in the current code to reproduce the results of a previous Hertzian monodisperse calculation. For example, for the common case of a monodisperse system with particles of diameter 1, Kn, Kt, gamma_n, and gamma_s should be set sqrt(2.0) larger than they were previously.

The effective mass *m_eff* in the formulas listed on the *pair_style granular* page is the mass of the particle for particle/wall interactions (mass of wall is infinite). If the particle is part of a rigid body, its mass is replaced by the mass of the rigid body in those formulas. This is determined by searching for a *fix rigid* command (or its variants).

The *wallstyle* must one of the three planar options. They specify a pair of walls in a dimension. Wall positions are given by *lo* and *hi*. Either of the values can be specified as NULL if a single wall is desired.

Deprecated since version 11Feb2026.

The *zcylinder* wallstyle has been removed. Pleas use *fix wall/gran/region* instead.

Optionally, the wall can be moving, if the *wiggle* or *shear* keywords are appended. Both keywords cannot be used together.

For the *wiggle* keyword, the wall oscillates sinusoidally, similar to the oscillations of particles which can be specified by the *fix move* command. This is useful in packing simulations of granular particles. The arguments to the *wiggle* keyword specify a dimension for the motion, as well as its *amplitude* and *period*. Note that if the dimension is in the plane of the wall, this is effectively a shearing motion. If the dimension is perpendicular to the wall, it is more of a shaking motion.

Each timestep, the position of a wiggled wall in the appropriate *dim* is set according to this equation:

```
position = coord + A - A cos (omega * delta)
```

where *coord* is the specified initial position of the wall, *A* is the *amplitude*, *omega* is 2 PI / *period*, and *delta* is the time elapsed since the fix was specified. The velocity of the wall is set to the derivative of this expression.

For the *shear* keyword, the wall moves continuously in the specified dimension with velocity *vshear*. The dimension must be tangential to walls with a planar *wallstyle*, e.g. in the *y* or *z* directions for an *xplane* wall. A dimension of *x* or *y* means the cylinder is spinning around the z-axis, either in the clockwise direction for *vshear* > 0 or counter-clockwise for *vshear* < 0. In this case, *vshear* is the tangential velocity of the wall at whatever *radius* has been defined.

The *temperature* keyword is used to assign a temperature to the wall. The following value can either be a numeric value or an equal-style *variable*. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated each timestep, and its value used to determine the temperature. This option must be used in conjunction with a heat conduction model defined in *pair_style granular*, *fix property/atom* to store temperature and a heat flow, and *fix heat/flow* to integrate heat flow.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.291.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the shear friction state of atoms interacting with the wall to *binary restart files*, so that a simulation can continue correctly if granular potentials with shear "history" effects are being used. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

If the `contacts` option is used, this fix generates a per-atom array with at least 8 columns as output, containing the contact information for owned particles (nlocal on each processor). All columns in this per-atom array will be zero if no contact has occurred. The first 8 values of these columns are listed in the following table.

| Index | Value | Units |
|---|---|---|
| 1 | 1.0 if particle is in contact with wall, 0.0 otherwise | |
| 2 | Force $f_x$ exerted by the wall | force units |
| 3 | Force $f_y$ exerted by the wall | force units |
| 4 | Force $f_z$ exerted by the wall | force units |
| 5 | $x$-coordinate of contact point on wall | distance units |
| 6 | $y$-coordinate of contact point on wall | distance units |
| 7 | $z$-coordinate of contact point on wall | distance units |
| 8 | Radius $r$ of atom | distance units |

If a granular sub-model calculates additional contact information (e.g. the heat sub-models calculate the amount of heat exchanged), these quantities are appended to the end of this array. First, any extra values from the normal sub-model are appended followed by the damping, tangential, rolling, twisting, then heat models. See the descriptions of granular sub-models in the *pair granular* page for information on any extra quantities.

None of the *fix_modify* options are relevant to this fix. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

---

### 2.291.5 Dump image info

Added in version 11Feb2026.

This fix supports the *fix* keyword of *dump image*. The fix will pass geometry information about *xplane*, *yplane*, and *zplane* style walls to *dump image* so that the walls will be included in the rendered image. Please note, that for *2d systems*, a wall rendered as a plane would be invisible and it is thus rendered as a cylinder.

The color of the wall is by default that of the first atom type when using color styles "type" or "element". With color style "const" the default value of "white" can be changed using *dump_modify fcolor*. The transparency is by default fully opaque and can be changed globally with *dump_modify ftrans*.

For 2d systems, the *fflag1* setting determines whether the cylinder representing the wall is capped with a sphere at the ends: 0 means no caps, 1 means the lower end is capped, 2 means the upper end is capped, and 3 means both ends are capped. The *fflag2* setting allows to set the radius of the rendered cylinders.

For 3d systems, both *fflag1* and *fflag2* are ignored.

---

### 2.291.6 Restrictions

This fix is part of the GRANULAR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Any dimension (xyz) that has a granular wall must be non-periodic.

### 2.291.7 Related commands

*fix move*, *fix wall/gran/region*, *pair_style gran/** *pair_style granular*

### 2.291.8 Default

none

## 2.292 fix wall/gran/region command

### 2.292.1 Syntax

```
fix ID group-ID wall/gran/region fstyle fstyle_params wallstyle regionID keyword values .
↪..
```

- ID, group-ID are documented in *fix* command
- wall/region = style name of this fix command
- fstyle = style of force interactions between particles and wall

  ```
  possible choices: hooke, hooke/history, hertz/history, granular
  ```

- fstyle_params = parameters associated with force interaction style

```
For hooke, hooke/history, and hertz/history, fstyle_params are:
      Kn = elastic constant for normal particle repulsion (force/distance units or␣
↪pressure units - see discussion below)
      Kt = elastic constant for tangential contact (force/distance units or␣
↪pressure units - see discussion below)
      gamma_n = damping coefficient for collisions in normal direction (1/time␣
↪units or 1/time-distance units - see discussion below)
      gamma_t = damping coefficient for collisions in tangential direction (1/time␣
↪units or 1/time-distance units - see discussion below)
      xmu = static yield criterion (unitless value between 0.0 and 1.0e4)
      dampflag = 0 or 1 if tangential damping force is excluded or included

For granular, fstyle_params are set using the same syntax as for the pair_coeff␣
↪command of pair_style granular
```

- wallstyle = region (see *fix wall/gran* for options for other kinds of walls)
- region-ID = region whose boundary will act as wall
- keyword = *contacts* or *temperature*

```
contacts value = none
   generate contact information for each particle
temperature value = temperature
   specify temperature of wall
```

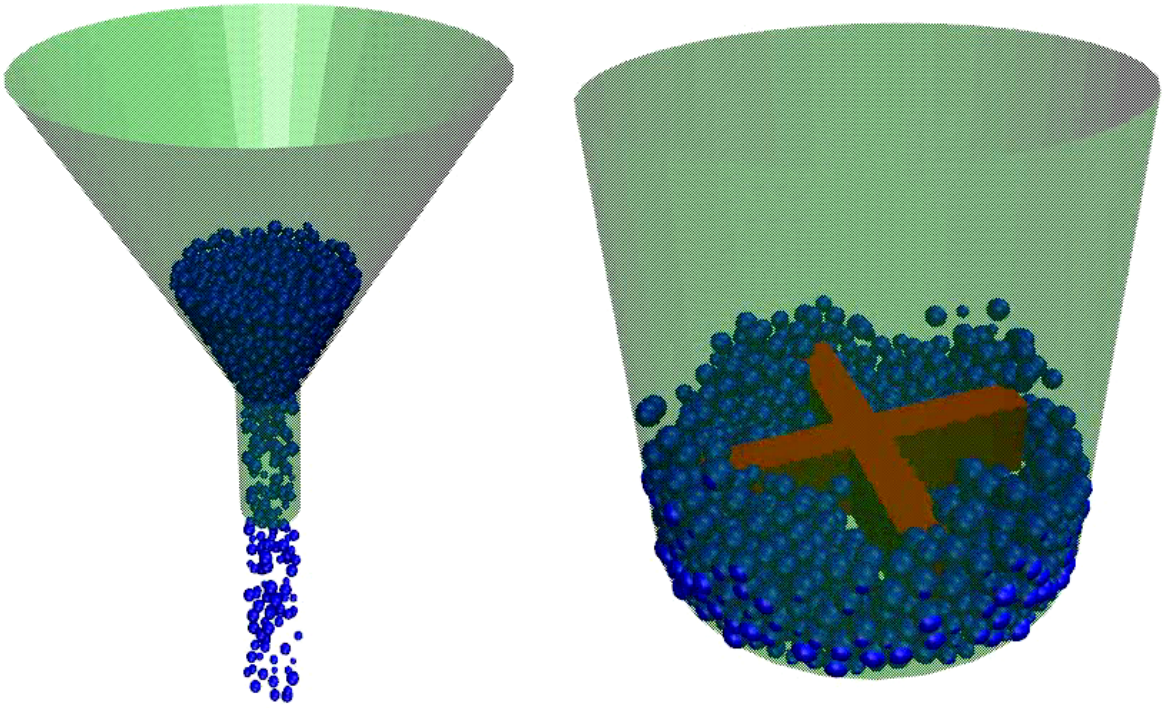
### 2.292.2 Examples

```
fix wall all wall/gran/region hooke/history 1000.0 200.0 200.0 100.0 0.5 1 region myCone
fix 3 all wall/gran/region granular hooke 1000.0 50.0 tangential linear_nohistory 1.0 0.
↪4 damping velocity region myBox
fix 4 all wall/gran/region granular jkr 1e5 1500.0 0.3 10.0 tangential mindlin NULL 1.0␣
↪0.5 rolling sds 500.0 200.0 0.5 twisting marshall region myCone
fix 5 all wall/gran/region granular dmt 1e5 0.2 0.3 10.0 tangential mindlin NULL 1.0 0.5␣
↪rolling sds 500.0 200.0 0.5 twisting marshall damping tsuji region myCone
fix wall all wall/gran/region hooke/history 1000.0 200.0 200.0 100.0 0.5 1 region myCone␣
↪contacts
```

### 2.292.3 Description

Treat the surface of the geometric region defined by the *region-ID* as a bounding frictional wall which interacts with nearby finite-size granular particles when they are close enough to touch the wall. See the *fix wall/region* and *fix wall/gran* commands for related kinds of walls for non-granular particles and simpler wall geometries, respectively.

Here are snapshots of example models using this command. Corresponding input scripts can be found in examples/granregion. Movies of these simulations are here on the Movies page of the LAMMPS website.



The distance between a particle and the region boundary is the distance to the nearest point on the region surface. The force the wall exerts on the particle is along the direction between that point and the particle center, which is the direction normal to the surface at that point. Note that if the region surface is comprised of multiple "faces", then each face can exert a force on the particle if it is close enough. E.g. for *region_style block*, a particle in the interior, near a corner of the block, could feel wall forces from 1, 2, or 3 faces of the block.

Regions are defined using the *region* command. Note that the region volume can be interior or exterior to the bounding surface, which will determine in which direction the surface interacts with particles, i.e. the direction of the surface normal. The exception to this is if one or more *open* options are specified for the region command, in which case particles interact with both the interior and exterior surfaces of regions.

Regions can either be primitive shapes (block, sphere, cylinder, etc) or combinations of primitive shapes specified via the *union* or *intersect* region styles. These latter styles can be used to construct particle containers with complex shapes.

Regions can also move dynamically via the *region* command keywords (move) and *rotate*, or change their shape by use of variables as inputs to the *region* command. If such a region is used with this fix, then the region surface will move in time in the corresponding manner.

**Note**

As discussed on the *region* command doc page, regions in LAMMPS do not get wrapped across periodic boundaries. It is up to you to ensure that the region location with respect to periodic or non-periodic boundaries is specified appropriately via the *region* and *boundary* commands when using a region as a wall that bounds particle motion.

**Note**

For primitive regions with sharp corners and/or edges (e.g. a block or cylinder), wall/particle forces are computed accurately for both interior and exterior regions. For *union* and *intersect* regions, additional sharp corners and edges may be present due to the intersection of the surfaces of 2 or more primitive volumes. These corners and edges can

be of two types: concave or convex. Concave points/edges are like the corners of a cube as seen by particles in the interior of a cube. Wall/particle forces around these features are computed correctly. Convex points/edges are like the corners of a cube as seen by particles exterior to the cube, i.e. the points jut into the volume where particles are present. LAMMPS does NOT compute the location of these convex points directly, and hence wall/particle forces in the cutoff volume around these points suffer from inaccuracies. The basic problem is that the outward normal of the surface is not continuous at these points. This can cause particles to feel no force (they don't "see" the wall) when in one location, then move a distance epsilon, and suddenly feel a large force because they now "see" the wall. In a worst-case scenario, this can blow particles out of the simulation box. Thus, as a general rule you should not use the fix wall/gran/region command with *union* or *interesect* regions that have convex points or edges resulting from the union/intersection (convex points/edges in the union/intersection due to a single sub-region are still OK).

**Note**

Similarly, you should not define *union* or *intersert* regions for use with this command that share an overlapping common face that is part of the overall outer boundary (interior boundary is OK), even if the face is smooth. E.g. two regions of style block in a *union* region, where the two blocks overlap on one or more of their faces. This is because LAMMPS discards points that are part of multiple sub-regions when calculating wall/particle interactions, to avoid double-counting the interaction. Having two coincident faces could cause the face to become invisible to the particles. The solution is to make the two faces differ by epsilon in their position.

The nature of the wall/particle interactions are determined by the *fstyle* setting. It can be any of the styles defined by the *pair_style gran/** or the more general *pair_style granular* commands. Currently the options are *hooke*, *hooke/history*, or *hertz/history* for the former, and *granular* with all the possible options of the associated *pair_coeff* command for the latter. The equation for the force between the wall and particles touching it is the same as the corresponding equation on the *pair_style gran/** and *pair_style granular* doc pages, but the effective radius is calculated using the radius of the particle and the radius of curvature of the wall at the contact point.

Specifically, delta = radius - r = overlap of particle with wall, m_eff = mass of particle, and RiRj/Ri+Rj is the effective radius, with Rj replaced by the radius of curvature of the wall at the contact point. The radius of curvature can be negative for a concave wall section, e.g. the interior of cylinder. For a flat wall, delta = radius - r = overlap of particle with wall, m_eff = mass of particle, and the effective radius of contact is just the radius of the particle.

The parameters *Kn*, *Kt*, *gamma_n*, *gamma_t*, *xmu*, *dampflag*, and the optional keyword *limit_damping* have the same meaning and units as those specified with the *pair_style gran/** commands. This means a NULL can be used for either *Kt* or *gamma_t* as described on that page. If a NULL is used for *Kt*, then a default value is used where *Kt* = 2/7 *Kn*. If a NULL is used for *gamma_t*, then a default value is used where *gamma_t* = 1/2 *gamma_n*.

All the model choices for cohesion, tangential friction, rolling friction and twisting friction supported by the *pair_style granular* through its *pair_coeff* command are also supported for walls. These are discussed in greater detail on the doc page for *pair_style granular*.

Note that you can choose a different force styles and/or different values for the 6 wall/particle coefficients than for particle/particle interactions. E.g. if you wish to model the wall as a different material.

The *temperature* keyword is used to assign a temperature to the wall. The following value can either be a numeric value or an equal-style *variable*. If the value is a variable, it should be specified as v_name, where name is the variable name. In this case, the variable will be evaluated each timestep, and its value used to determine the temperature. This option must be used in conjunction with a heat conduction model defined in *pair_style granular*, *fix property/atom* to store temperature and a heat flow, and *fix heat/flow* to integrate heat flow.

### 2.292.4 Restart, fix_modify, output, run start/stop, minimize info

Similar to *fix wall/gran* command, this fix writes the shear friction state of atoms interacting with the wall to *binary restart files*, so that a simulation can continue correctly if granular potentials with shear "history" effects are being used. This fix also includes info about a moving region in the restart file. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

**Note**

Information about region definitions is NOT included in restart files, as discussed on the *read_restart* doc page. So you must re-define your region and if it is a moving region, define its motion attributes in a way that is consistent with the simulation that wrote the restart file. In particular, if you want to change the region motion attributes (e.g. its velocity), then you should ensure the position/orientation of the region at the initial restart timestep is the same as it was on the timestep the restart file was written. If this is not possible, you may need to ignore info in the restart file by defining a new fix wall/gran/region command in your restart script, e.g. with a different fix ID. Or if you want to keep the shear history info but discard the region motion information, you can use the same fix ID for fix wall/gran/region, but assign it a region with a different region ID.

If the `contacts` option is used, this fix generates a per-atom array with at least 8 columns as output, containing the contact information for owned particles (nlocal on each processor). All columns in this per-atom array will be zero if no contact has occurred. The first 8 values of these columns are listed in the following table.

| Index | Value | Units |
|---|---|---|
| 1 | 1.0 if particle is in contact with wall, 0.0 otherwise | |
| 2 | Force $f_x$ exerted by the wall | force units |
| 3 | Force $f_y$ exerted by the wall | force units |
| 4 | Force $f_z$ exerted by the wall | force units |
| 5 | $x$-coordinate of contact point on wall | distance units |
| 6 | $y$-coordinate of contact point on wall | distance units |
| 7 | $z$-coordinate of contact point on wall | distance units |
| 8 | Radius $r$ of atom | distance units |

If a granular sub-model calculates additional contact information (e.g. the heat sub-models calculate the amount of heat exchanged), these quantities are appended to the end of this array. First, any extra values from the normal sub-model are appended followed by the damping, tangential, rolling, twisting, then heat models. See the descriptions of granular sub-models in the *pair granular* page for information on any extra quantities.

None of the *fix_modify* options are relevant to this fix. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.292.5 Dump image info

This fix does **not** support the *fix* keyword of the *dump image* command. Instead the region used by the fix can be visualized using the *region* keyword of *dump image*.

### 2.292.6 Restrictions

This fix is part of the GRANULAR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.292.7 Related commands

*fix_move*, *fix wall/gran*, *fix wall/region*, *pair_style granular*, *region*

### 2.292.8 Default

none

## 2.293 fix wall/piston command

### 2.293.1 Syntax

```
fix ID group-ID wall/piston face ... keyword value ...
```

- ID, group-ID are documented in *fix* command
- wall/piston = style name of this fix command
- face = *zlo*
- zero or more keyword/value pairs may be appended
- keyword = *pos* or *vel* or *ramp* or *temp* or *units*

```
pos args = z
  z = z coordinate at which the piston begins (distance units)
vel args = vz
  vz = final velocity of the piston (velocity units)
ramp = use a linear velocity ramp from 0 to vz
temp args = target damp seed extent
  target = target velocity for region immediately ahead of the piston
  damp = damping parameter (time units)
  seed = random number seed for langevin kicks
  extent = extent of thermostatted region (distance units)
units value = lattice or box
  lattice = the wall position is defined in lattice units
  box = the wall position is defined in simulation box units
```

### 2.293.2 Examples

```
fix xwalls all wall/piston zlo
fix walls all wall/piston zlo pos 1.0 vel 10.0 units box
fix top all wall/piston zlo vel 10.0 ramp
```

### 2.293.3 Description

Bound the simulation with a moving wall which reflect particles in the specified group and drive the system with an effective infinite-mass piston capable of driving shock waves.

A momentum mirror technique is used, which means that if an atom (or the wall) moves such that an atom is outside the wall on a timestep by a distance delta (e.g. due to *fix nve*), then it is put back inside the face by the same delta, and the velocity relative to the moving wall is flipped in z. For instance, a stationary particle hit with a piston wall with velocity vz, will end the timestep with a velocity of 2*vz.

Currently the *face* keyword can only be *zlo*. This creates a piston moving in the positive z direction. Particles with z coordinate less than the wall position are reflected to a z coordinate greater than the wall position. If the piston velocity is vpz and the particle velocity before reflection is vzi, the particle velocity after reflection is -vzi + 2*vpz.

The initial position of the wall can be specified by the *pos* keyword.

The final velocity of the wall can be specified by the *vel* keyword

The *ramp* keyword will cause the wall/piston to adjust the velocity linearly from zero velocity to *vel* over the course of the run. If the *ramp* keyword is omitted then the wall/piston moves at a constant velocity defined by *vel*.

The *temp* keyword will cause the region immediately in front of the wall/piston to be thermostatted with a Langevin thermostat. This region moves with the piston. The damping and kicking are measured in the reference frame of the piston. So, a temperature of zero would mean all particles were moving at exactly the speed of the wall/piston.

The *units* keyword determines the meaning of the distance units used to define a wall position, but only when a numeric constant is used.

A *box* value selects standard distance units as defined by the *units* command, e.g. Angstroms for units = real or metal. A *lattice* value means the distance units are in lattice spacings. The *lattice* command must have been previously used to define the lattice spacings.

---

### 2.293.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.293.5 Restrictions

This fix style is part of the SHOCK package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The face that has the wall/piston must be boundary type 's' (shrink-wrapped). The opposing face can be any boundary type other than periodic.

A wall/piston should not be used with rigid bodies such as those defined by a "fix rigid" command. This is because the wall/piston displaces atoms directly rather than exerting a force on them.

### 2.293.6 Related commands

*fix wall/reflect* command, *fix append/atoms* command

### 2.293.7 Default

The keyword defaults are pos = 0, vel = 0, units = lattice.

## 2.294 fix wall/reflect command

Accelerator Variants: *wall/reflect/kk*

### 2.294.1 Syntax

```
fix ID group-ID wall/reflect face arg ... keyword value ...
```

- ID, group-ID are documented in *fix* command
- wall/reflect = style name of this fix command
- one or more face/arg pairs may be appended

- face = *xlo* or *xhi* or *ylo* or *yhi* or *zlo* or *zhi*

  ```
  arg = EDGE or constant or variable
    EDGE = current lo edge of simulation box
    constant = number like 0.0 or 30.0 (distance units)
    variable = equal-style variable like v_x or v_wiggle
  ```

- zero or more keyword/value pairs may be appended
- keyword = *units*

  ```
  units value = lattice or box
    lattice = the wall position is defined in lattice units
    box = the wall position is defined in simulation box units
  ```

### 2.294.2 Examples

```
fix xwalls all wall/reflect xlo EDGE xhi EDGE
fix walls all wall/reflect xlo 0.0 ylo 10.0 units box
fix top all wall/reflect zhi v_pressdown
```

### 2.294.3 Description

Bound the simulation with one or more walls which reflect particles in the specified group when they attempt to move through them.

Reflection means that if an atom moves outside the wall on a timestep by a distance delta (e.g. due to *fix nve*), then it is put back inside the face by the same delta, and the sign of the corresponding component of its velocity is flipped.

When used in conjunction with *fix nve* and *run_style verlet*, the resultant time-integration algorithm is equivalent to the primitive splitting algorithm (PSA) described by *Bond*. Because each reflection event divides the corresponding timestep asymmetrically, energy conservation is only satisfied to O(dt), rather than to O(dt^2) as it would be for velocity-Verlet integration without reflective walls.

Up to 6 walls or faces can be specified in a single command: *xlo*, *xhi*, *ylo*, *yhi*, *zlo*, *zhi*. A *lo* face reflects particles that move to a coordinate less than the wall position, back in the *hi* direction. A *hi* face reflects particles that move to a coordinate higher than the wall position, back in the *lo* direction.

The position of each wall can be specified in one of 3 ways: as the EDGE of the simulation box, as a constant value, or as a variable. If EDGE is used, then the corresponding boundary of the current simulation box is used. If a numeric constant is specified then the wall is placed at that position in the appropriate dimension (x, y, or z). In both the EDGE and constant cases, the wall will never move. If the wall position is a variable, it should be specified as v_name, where name is an *equal-style variable* name. In this case the variable is evaluated each timestep and the result becomes the current position of the reflecting wall. Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent wall position.

The *units* keyword determines the meaning of the distance units used to define a wall position, but only when a numeric constant or variable is used. It is not relevant when EDGE is used to specify a face position. In the variable case, the variable is assumed to produce a value compatible with the *units* setting you specify.

A *box* value selects standard distance units as defined by the *units* command, e.g. Angstroms for units = real or metal. A *lattice* value means the distance units are in lattice spacings. The *lattice* command must have been previously used to define the lattice spacings.

---

Here are examples of variable definitions that move the wall position in a time-dependent fashion using equal-style *variables*.

```
variable ramp equal ramp(0,10)
fix 1 all wall/reflect xlo v_ramp

variable linear equal vdisplace(0,20)
fix 1 all wall/reflect xlo v_linear

variable wiggle equal swiggle(0.0,5.0,3.0)
fix 1 all wall/reflect xlo v_wiggle

variable wiggle equal cwiggle(0.0,5.0,3.0)
fix 1 all wall/reflect xlo v_wiggle
```

The *ramp(lo,hi)* function adjusts the wall position linearly from *lo* to *hi* over the course of a run. The *vdisplace(c0,velocity)* function does something similar using the equation *position = c0 + velocity*delta*, where *delta* is the elapsed time.

The *swiggle(c0,A,period)* function causes the wall position to oscillate sinusoidally according to this equation, where *omega = 2 PI / period*:

```
position = c0 + A sin(omega*delta)
```

The *cwiggle(c0,A,period)* function causes the wall position to oscillate sinusoidally according to this equation, which will have an initial wall velocity of 0.0, and thus may impose a gentler perturbation on the particles:

```
position = c0 + A (1 - cos(omega*delta))
```

---

### 2.294.4 Dump image info

Added in version 11Feb2026.

These wall fixes support the *fix* keyword of *dump image*. The fixes will pass geometry information about the walls to *dump image* so that the walls will be included in the rendered image. Please note, that for *2d systems*, a wall rendered as a plane would be invisible and it is thus rendered as a cylinder.

The color of the wall is by default that of the first atom type when using color styles "type" or "element". With color style "const" the default value of "white" can be changed using *dump_modify fcolor*. The transparency is by default fully opaque and can be changed globally with *dump_modify ftrans*.

For 2d systems, the *fflag1* setting determines whether the cylinder representing the wall is capped with a sphere at the ends: 0 means no caps, 1 means the lower end is capped, 2 means the upper end is capped, and 3 means both ends are capped. The *fflag2* setting allows to adjust the radius of the rendered cylinder. It should be set to a value > 0 or the cylinder will not be visible since the diameter is set internally to zero due to lack of a suitable heuristic for deriving a meaningful diameter for all types of walls and unit settings.

For 3d systems, both *fflag1* and *fflag2* are ignored.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 2.294.5 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix. No global or per-atom quantities are stored by this fix for access by various *output commands*. No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.294.6 Restrictions

Any dimension (xyz) that has a reflecting wall must be non-periodic.

A reflecting wall should not be used with rigid bodies such as those defined by a "fix rigid" command. This is because the wall/reflect displaces atoms directly rather than exerts a force on them. For rigid bodies, use a soft wall instead, such as *fix wall/lj93*. LAMMPS will flag the use of a rigid fix with fix wall/reflect with a warning, but will not generate an error.

### 2.294.7 Related commands

*fix wall/lj93*, *fix oneway*

### 2.294.8 Default

The default for the units keyword is lattice.

**(Bond)** Bond and Leimkuhler, SIAM J Sci Comput, 30, p 134 (2007).

## 2.295 fix wall/reflect/stochastic command

### 2.295.1 Syntax

```
fix ID group-ID wall/reflect/stochastic rstyle seed face args ... keyword value ...
```

- ID, group-ID are documented in *fix* command
- wall/reflect/stochastic = style name of this fix command
- rstyle = diffusive or maxwell or ccl
- seed = random seed for stochasticity (positive integer)
- one or more face/args pairs may be appended
- face = *xlo* or *xhi* or *ylo* or *yhi* or *zlo* or *zhi*

```
args = pos temp velx vely velz accomx accomy accomz
  pos = EDGE or constant
    EDGE = current lo or hi edge of simulation box
    constant = number like 0.0 or 30.0 (distance units)
  temp = wall temperature (temperature units)
  velx,vely,velz = wall velocity in x,y,z directions (velocity units)
  accomx,accomy,accomz = accommodation coeffs in x,y,z directions (unitless)
    not specified for rstyle = diffusive
    single accom coeff specified for rstyle maxwell
    all 3 coeffs specified for rstyle cll
```

- zero or more keyword/value pairs may be appended
- keyword = *units*

```
units value = lattice or box
  lattice = the wall position is defined in lattice units
  box = the wall position is defined in simulation box units
```

### 2.295.2 Examples

```
fix zwalls all wall/reflect/stochastic diffusive 23424 zlo EDGE 300 0.1 0.1 0 zhi EDGE␣
↪200 0.1 0.1 0
fix ywalls all wall/reflect/stochastic maxwell 345533 ylo 5.0 300 0.1 0.0 0.0 0.8 yhi 10.
↪0 300 0.1 0.0 0.0 0.8
fix xwalls all wall/reflect/stochastic cercignanilampis 2308 xlo 0.0 300 0.0 0.1 0.9 0.8␣
↪0.7 xhi EDGE 300 0.0 0.1 0 0.9 0.8 0.7 units box
```

### 2.295.3 Description

Bound the simulation with one or more walls which reflect particles in the specified group when they attempt to move through them.

Reflection means that if an atom moves outside the wall on a timestep (e.g. due to the *fix nve* command), then it is put back inside the wall with a changed velocity.

This fix models treats the wall as a moving solid boundary with a finite temperature, which can exchange energy with particles that collide with it. This is different than the simpler *fix wall/reflect* command which models mirror reflection. For this fix, the post collision velocity of each particle is treated stochastically. The randomness can come from many sources: thermal motion of the wall atoms, surface roughness, etc. Three stochastic reflection models are currently implemented.

For rstyle *diffusive*, particles are reflected diffusively. Their velocity distribution corresponds to an equilibrium distribution of particles at the wall temperature. No accommodation coefficients are specified.

For rstyle *maxwell*, particle reflection is Maxwellian which means partially diffusive and partially specular (*Maxwell*). A single accommodation coeff is specified which must be between 0.0 and 1.0 inclusive. It determines the fraction of the collision which is diffusive versus specular. An accommodation coefficient of 1.0 is fully diffusive; a coefficient of 0.0 is fully specular.

For rstyle *cll*, particle collisions are computed by the Cercignani/Lampis model. See *CL* and *To* for details. Three accommodations coefficient are specified. Each must be between 0.0 and 1.0 inclusive. Two are velocity accommodation coefficients; one is a normal kinetic energy accommodation. The normal coeff is the one corresponding to the normal of the wall itself. For example if the wall is *ylo* or *yhi*, *accomx* and *accomz* are the tangential velocity accommodation coefficients, and *accomy* is the normal kinetic energy accommodation coefficient.

The optional *units* keyword determines the distance units used to define a wall position. A *box* value selects standard distance units as defined by the *units* command, e.g. Angstroms for units = real or metal. A *lattice* value means the distance units are in lattice spacings. The *lattice* command must have been previously used to define the lattice spacings.

---

### 2.295.4 Dump image info

Added in version 11Feb2026.

This wall fix supports the *fix* keyword of *dump image*. The fix will pass geometry information about the walls to *dump image* so that the walls will be included in the rendered image. Please note, that for *2d systems*, a wall rendered as a plane would be invisible and it is thus rendered as a cylinder.

The color of the wall is by default that of the first atom type when using color styles "type" or "element". With color style "const" the default value of "white" can be changed using *dump_modify fcolor*. The transparency is by default fully opaque and can be changed with *dump_modify ftrans*.

The *fflag1* setting and the *fflag2* setting of *dump image fix* are only relevant for 2d systems. The *fflag1* setting determines whether the cylinder is capped with a sphere at the ends: 0 means no caps, 1 means the lower end is capped, 2 means the upper end is capped, and 3 means both ends are capped. The *fflag2* setting allows to adjust the radius of the rendered cylinder. It should be set to a value > 0 or the cylinder will not be visible since the diameter is set internally to zero due to lack of a suitable heuristic for deriving a meaningful diameter for all types of walls and unit settings.

---

### 2.295.5 Restrictions

This fix has the same limitations as the *fix wall/reflect* command. Any dimension (xyz) that has a wall must be non-periodic. It should not be used with rigid bodies such as those defined by the *fix rigid* command. The wall velocity must lie on the same plane as the wall itself.

This fix is part of the EXTRA-FIX package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 2.295.6 Related commands

*fix wall/reflect*

### 2.295.7 Default

The default for the units keyword is lattice.

---

**(Maxwell)** J.C. Maxwell, Philos. Tans. Royal Soc. London, 157: 49-88 (1867).

**(Cercignani)** C. Cercignani and M. Lampis. Trans. Theory Stat. Phys. 1, 2, 101 (1971).

**(To)** Q.D. To, V.H. Vu, G. Lauriat, and C. Leonard. J. Math. Phys. 56, 103101 (2015).

## 2.296 fix wall/region command

Accelerator Variants: *wall/region/kk*

### 2.296.1 Syntax

```
fix ID group-ID wall/region region-ID style args ... cutoff
```

- ID, group-ID are documented in *fix* command
- wall/region = style name of this fix command
- region-ID = region whose boundary will act as wall
- style = *lj93* or *lj126* or *lj1043* or *colloid* or *harmonic* or *morse*
- args for styles *lj93* or *lj126* or *lj1043* or *colloid* or *harmonic* =

  ```
  epsilon = strength factor for wall-particle interaction (energy or energy/distance^
  →2 units)
  sigma = size factor for wall-particle interaction (distance units)
  ```

- args for style *morse* =

```
D_0 = depth of the potential (energy units)
alpha = width parameter (1/distance units)
r_0 = distance of the potential minimum from wall position (distance units)
```

- cutoff = distance from wall at which wall-particle interaction is cut off (distance units)

### 2.296.2 Examples

```
fix wall all wall/region mySphere lj93 1.0 1.0 2.5
fix wall all wall/region mySphere harmonic 1.0 0.0 2.5
fix wall all wall/region box_top morse 1.0 1.0 1.5 3.0
```

### 2.296.3 Description

Treat the surface of the geometric region defined by the *region-ID* as a bounding wall which interacts with nearby particles according to the specified style.

The distance between a particle and the surface is the distance to the nearest point on the surface and the force the wall exerts on the particle is along the direction between that point and the particle, which is the direction normal to the surface at that point. Note that if the region surface is comprised of multiple "faces", then each face can exert a force on the particle if it is close enough. E.g. for *region_style block*, a particle in the interior, near a corner of the block, could feel wall forces from 1, 2, or 3 faces of the block.

Regions are defined using the *region* command. Note that the region volume can be interior or exterior to the bounding surface, which will determine in which direction the surface interacts with particles, i.e. the direction of the surface normal. The surface of the region only exerts forces on particles "inside" the region; if a particle is "outside" the region it will generate an error, because it has moved through the wall.

Regions can either be primitive shapes (block, sphere, cylinder, etc) or combinations of primitive shapes specified via the *union* or *intersect* region styles. These latter styles can be used to construct particle containers with complex shapes. Regions can also change over time via the *region* command keywords (move) and *rotate*. If such a region is used with this fix, then the of region surface will move over time in the corresponding manner.

**Note**

As discussed on the *region* command doc page, regions in LAMMPS do not get wrapped across periodic boundaries. It is up to you to ensure that periodic or non-periodic boundaries are specified appropriately via the *boundary* command when using a region as a wall that bounds particle motion. This also means that if you embed a region in your simulation box and want it to repulse particles from its surface (using the "side out" option in the *region* command), that its repulsive force will not be felt across a periodic boundary.

**Note**

For primitive regions with sharp corners and/or edges (e.g. a block or cylinder), wall/particle forces are computed accurately for both interior and exterior regions. For *union* and *intersect* regions, additional sharp corners and edges may be present due to the intersection of the surfaces of 2 or more primitive volumes. These corners and edges can be of two types: concave or convex. Concave points/edges are like the corners of a cube as seen by particles in the interior of a cube. Wall/particle forces around these features are computed correctly. Convex points/edges are like the corners of a cube as seen by particles exterior to the cube, i.e. the points jut into the volume where particles are present. LAMMPS does NOT compute the location of these convex points directly, and hence wall/particle forces in the cutoff volume around these points suffer from inaccuracies. The basic problem is that the outward normal of the surface is not continuous at these points. This can cause particles to feel no force (they don't "see" the wall)

when in one location, then move a distance epsilon, and suddenly feel a large force because they now "see" the wall. In a worst-case scenario, this can blow particles out of the simulation box. Thus, as a general rule you should not use the fix wall/gran/region command with *union* or *interesect* regions that have convex points or edges resulting from the union/intersection (convex points/edges in the union/intersection due to a single sub-region are still OK).

**Note**

Similarly, you should not define *union* or *intersert* regions for use with this command that share an overlapping common face that is part of the overall outer boundary (interior boundary is OK), even if the face is smooth. E.g. two regions of style block in a *union* region, where the two blocks overlap on one or more of their faces. This is because LAMMPS discards points that are part of multiple sub-regions when calculating wall/particle interactions, to avoid double-counting the interaction. Having two coincident faces could cause the face to become invisible to the particles. The solution is to make the two faces differ by epsilon in their position.

The energy of wall-particle interactions depends on the specified style.

For style *lj93*, the energy E is given by the 9/3 potential:

$$E = \varepsilon \left[ \frac{2}{15} \left( \frac{\sigma}{r} \right)^9 - \left( \frac{\sigma}{r} \right)^3 \right] \qquad r < r_c$$

For style *lj126*, the energy E is given by the 12/6 potential:

$$E = 4\varepsilon \left[ \left( \frac{\sigma}{r} \right)^{12} - \left( \frac{\sigma}{r} \right)^6 \right] \qquad r < r_c$$

For style *wall/lj1043*, the energy E is given by the 10/4/3 potential:

$$E = 2\pi\varepsilon \left[ \frac{2}{5} \left( \frac{\sigma}{r} \right)^{10} - \left( \frac{\sigma}{r} \right)^4 - \frac{\sqrt{(2)}\sigma^3}{3\left(r + \left(0.61/\sqrt{(2)}\right)\sigma\right)^3} \right] \qquad r < r_c$$

For style *colloid*, the energy E is given by an integrated form of the *pair_style colloid* potential:

$$\begin{aligned} E =& \varepsilon \left[ \frac{\sigma^6}{7560} \left( \frac{6R - D}{D^7} + \frac{D + 8R}{(D + 2R)^7} \right) \right. \\ & \left. - \frac{1}{6} \left( \frac{2R(D + R) + D(D + 2R)\left[\ln D - \ln(D + 2R)\right]}{D(D + 2R)} \right) \right] \qquad r < r_c \end{aligned}$$

For style *wall/harmonic*, the energy E is given by a harmonic spring potential (the distance parameter is ignored):

$$E = \varepsilon \quad (r - r_c)^2 \qquad r < r_c$$

For style *wall/morse*, the energy E is given by the Morse potential:

$$E = D_0 \left[ e^{-2\alpha(r - r_0)} - 2e^{-\alpha(r - r_0)} \right] \qquad r < r_c$$

Unlike other styles, this requires three parameters ($D_0$, $\alpha$, and $r_0$ in this order) instead of two like for the other wall styles.

In all cases, $r$ is the distance from the particle to the region surface, and Rc is the *cutoff* distance at which the particle and surface no longer interact. The cutoff is always the last argument. The energy of the wall potential is shifted so that the wall-particle interaction energy is 0.0 at the cutoff distance.

For a full description of these wall styles, see fix_style *wall*

### 2.296.4 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*.

The *fix_modify energy* option is supported by this fix to add the energy of interaction between atoms and the region wall to the global potential energy of the system as part of *thermodynamic output*. The default setting for this fix is *fix_modify energy no*.

The *fix_modify virial* option is supported by this fix to add the contribution due to the interaction between atoms and the region wall to both the global pressure and per-atom stress of the system via the *compute pressure* and *compute stress/atom* commands. The former can be accessed by *thermodynamic output*. The default setting for this fix is *fix_modify virial no*.

The *fix_modify respa* option is supported by this fix. This allows to set at which level of the *r-RESPA* integrator the fix is adding its forces. Default is the outermost level.

This fix computes a global scalar energy and a global 3-length vector of forces, which can be accessed by various *output commands*. The scalar energy is the sum of energy interactions for all particles interacting with the wall represented by the region surface. The 3 vector quantities are the x,y,z components of the total force acting on the wall due to the particles. The scalar and vector values calculated by this fix are "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command.

The forces due to this fix are imposed during an energy minimization, invoked by the *minimize* command.

**Note**

If you want the atom/wall interaction energy to be included in the total potential energy of the system (the quantity being minimized), you MUST enable the *fix_modify energy* option for this fix.

### 2.296.5 Dump image info

This fix does **not** support the *fix* keyword of the *dump image* command. Instead the region used by the fix can be visualized using the *region* keyword of *dump image*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 2.296.6 Restrictions

none

### 2.296.7 Related commands

*fix wall/lj93*, *fix wall/lj126*, *fix wall/lj1043*, *fix wall/colloid*, *fix wall/harmonic*, *fix wall/gran*

### 2.296.8 Default

none

## 2.297 fix wall/srd command

### 2.297.1 Syntax

```
fix ID group-ID wall/srd face arg ... keyword value ...
```

- ID, group-ID are documented in *fix* command
- wall/srd = style name of this fix command
- one or more face/arg pairs may be appended
- face = *xlo* or *xhi* or *ylo* or *yhi* or *zlo* or *zhi*

  ```
  xlo,ylo,zlo arg = EDGE or constant or variable
    EDGE = current lo edge of simulation box
    constant = number like 0.0 or -30.0 (distance units)
    variable = equal-style variable like v_x or v_wiggle
  xhi,yhi,zhi arg = EDGE or constant or variable
    EDGE = current hi edge of simulation box
    constant = number like 50.0 or 100.3 (distance units)
    variable = equal-style variable like v_x or v_wiggle
  ```

- zero or more keyword/value pairs may be appended
- keyword = *units*

  ```
  units value = lattice or box
    lattice = the wall position is defined in lattice units
    box = the wall position is defined in simulation box units
  ```

### 2.297.2 Examples

```
fix xwalls all wall/srd xlo EDGE xhi EDGE
fix walls all wall/srd xlo 0.0 ylo 10.0 units box
fix top all wall/srd zhi v_pressdown
```

### 2.297.3 Description

Bound the simulation with one or more walls which interact with stochastic reaction dynamics (SRD) particles as slip (smooth) or no-slip (rough) flat surfaces. The wall interaction is actually invoked via the *fix srd* command, only on the group of SRD particles it defines, so the group setting for the fix wall/srd command is ignored.

A particle/wall collision occurs if an SRD particle moves outside the wall on a timestep. This alters the position and velocity of the SRD particle and imparts a force to the wall.

The *collision* and *Tsrd* settings specified via the *fix srd* command affect the SRD/wall collisions. A *slip* setting for the *collision* keyword means that the tangential component of the SRD particle momentum is preserved. Thus only a normal force is imparted to the wall. The normal component of the new SRD velocity is sampled from a Gaussian distribution at temperature *Tsrd*.

For a *noslip* setting of the *collision* keyword, both the normal and tangential components of the new SRD velocity are sampled from a Gaussian distribution at temperature *Tsrd*. Additionally, a new tangential direction for the SRD velocity is chosen randomly. This collision style imparts both a normal and tangential force to the wall.

Up to 6 walls or faces can be specified in a single command: *xlo*, *xhi*, *ylo*, *yhi*, *zlo*, *zhi*. A *lo* face reflects particles that move to a coordinate less than the wall position, back in the *hi* direction. A *hi* face reflects particles that move to a coordinate higher than the wall position, back in the *lo* direction.

The position of each wall can be specified in one of 3 ways: as the EDGE of the simulation box, as a constant value, or as a variable. If EDGE is used, then the corresponding boundary of the current simulation box is used. If a numeric constant is specified then the wall is placed at that position in the appropriate dimension (x, y, or z). In both the EDGE and constant cases, the wall will never move. If the wall position is a variable, it should be specified as v_name, where name is an *equal-style variable* name. In this case the variable is evaluated each timestep and the result becomes the current position of the reflecting wall. Equal-style variables can specify formulas with various mathematical functions, and include *thermo_style* command keywords for the simulation box parameters and timestep and elapsed time. Thus it is easy to specify a time-dependent wall position.

**Note**

Because the trajectory of the SRD particle is tracked as it collides with the wall, you must ensure that r = distance of the particle from the wall, is always > 0 for SRD particles, or LAMMPS will generate an error. This means you cannot start your simulation with SRD particles at the wall position *coord* (r = 0) or with particles on the wrong side of the wall (r < 0).

**Note**

If you have 2 or more walls that come together at an edge or corner (e.g. walls in the x and y dimensions), then be sure to set the *overlap* keyword to *yes* in the *fix srd* command, since the walls effectively overlap when SRD particles collide with them. LAMMPS will issue a warning if you do not do this.

**Note**

The walls of this fix only interact with SRD particles, as defined by the *fix srd* command. If you are simulating a mixture containing other kinds of particles, then you should typically use *another wall command* to act on the other particles. Since SRD particles will be colliding both with the walls and the other particles, it is important to ensure that the other particle's finite extent does not overlap an SRD wall. If you do not do this, you may generate errors when SRD particles end up "inside" another particle or a wall at the beginning of a collision step.

The *units* keyword determines the meaning of the distance units used to define a wall position, but only when a numeric constant is used. It is not relevant when EDGE or a variable is used to specify a face position.

A *box* value selects standard distance units as defined by the *units* command, e.g. Angstroms for units = real or metal. A *lattice* value means the distance units are in lattice spacings. The *lattice* command must have been previously used to define the lattice spacings.

---

Here are examples of variable definitions that move the wall position in a time-dependent fashion using equal-style *variables*.

```
variable ramp equal ramp(0,10)
fix 1 all wall/srd xlo v_ramp
```

(continues on next page)

(continued from previous page)

```
variable linear equal vdisplace(0,20)
fix 1 all wall/srd xlo v_linear

variable wiggle equal swiggle(0.0,5.0,3.0)
fix 1 all wall/srd xlo v_wiggle

variable wiggle equal cwiggle(0.0,5.0,3.0)
fix 1 all wall/srd xlo v_wiggle
```

The *ramp(lo,hi)* function adjusts the wall position linearly from *lo* to *hi* over the course of a run. The *vdisplace(c0,velocity)* function does something similar using the equation *position = c0 + velocity*delta*, where *delta* is the elapsed time.

The *swiggle(c0,A,period)* function causes the wall position to oscillate sinusoidally according to this equation, where *omega = 2 PI / period*:

```
position = c0 + A sin(omega*delta)
```

The *cwiggle(c0,A,period)* function causes the wall position to oscillate sinusoidally according to this equation, which will have an initial wall velocity of 0.0, and thus may impose a gentler perturbation on the particles:

```
position = c0 + A (1 - cos(omega*delta))
```

---

### 2.297.4 Dump image info

Added in version 11Feb2026.

Fix *wall/srd* supports the *fix* keyword of *dump image*. The fix will pass geometry information about the walls to *dump image* so that the walls will be included in the rendered image. Please note, that for *2d systems*, a wall rendered as a plane would be invisible and it is thus rendered as a cylinder.

The color of the wall is by default that of the first atom type when using color styles "type" or "element". With color style "const" the default value of "white" can be changed using *dump_modify fcolor*. The transparency is by default fully opaque and can be changed with *dump_modify ftrans*.

For 2d systems, the *fflag1* setting determines whether the cylinder representing the wall is capped with a sphere at the ends: 0 means no caps, 1 means the lower end is capped, 2 means the upper end is capped, and 3 means both ends are capped. The *fflag2* setting allows to adjust the radius of the rendered cylinder. It should be set to a value > 0 or the cylinder will not be visible since the diameter is set internally to zero due to lack of a suitable heuristic for deriving a meaningful diameter for all types of walls and unit settings.

For 3d systems, both *fflag1* and *fflag2* are ignored.

---

### 2.297.5 Restart, fix_modify, output, run start/stop, minimize info

No information about this fix is written to *binary restart files*. None of the *fix_modify* options are relevant to this fix.

This fix computes a global array of values which can be accessed by various *output commands*. The number of rows in the array is equal to the number of walls defined by the fix. The number of columns is 3, for the x,y,z components of force on each wall.

Note that an outward normal force on a wall will be a negative value for *lo* walls and a positive value for *hi* walls. The array values calculated by this fix are "extensive".

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.297.6 Restrictions

Any dimension (xyz) that has an SRD wall must be non-periodic.

### 2.297.7 Related commands

*fix srd*

### 2.297.8 Default

none

## 2.298 fix widom command

### 2.298.1 Syntax

```
fix ID group-ID widom N M type seed T keyword values ...
```

- ID, group-ID are documented in *fix* command
- widom = style name of this fix command
- N = invoke this fix every N steps
- M = number of Widom insertions to attempt every N steps
- type = atom type (1-Ntypes or type label) for inserted atoms (must be 0 if mol keyword used)
- seed = random # seed (positive integer)
- T = temperature of the system (temperature units)
- zero or more keyword/value pairs may be appended to args

```
keyword = mol, region, full_energy, charge, intra_energy
  mol value = template-ID
    template-ID = ID of molecule template specified in a separate molecule command
  region value = region-ID
    region-ID = ID of region where Widom insertions are allowed
  full_energy = compute the entire system energy when performing Widom insertions
  charge value = charge of inserted atoms (charge units)
  intra_energy value = intramolecular energy (energy units)
```

### 2.298.2 Examples

```
fix 2 gas widom 1 50000 1 19494 2.0
fix 3 water widom 1000 100 0 29494 300.0 mol h2omol full_energy

labelmap atom 1 Li
fix 2 ion widom 1 50000 Li 19494 2.0
```

### 2.298.3 Description

This fix performs Widom insertions of atoms or molecules at the given temperature as discussed in *(Frenkel)*. Specific uses include computation of Henry constants of small molecules in microporous materials or amorphous systems.

Every N timesteps the fix attempts M number of Widom insertions of atoms or molecules.

If the *mol* keyword is used, only molecule insertions are performed. Conversely, if the *mol* keyword is not used, only atom insertions are performed.

This command may optionally use the *region* keyword to define an insertion volume. The specified region must have been previously defined with a *region* command. It must be defined with side = *in*. Insertion attempts occur only within the specified region. For non-rectangular regions, random trial points are generated within the rectangular bounding box until a point is found that lies inside the region. If no valid point is generated after 1000 trials, no insertion is performed. If an attempted insertion places the atom or molecule center-of-mass outside the specified region, a new attempted insertion is generated. This process is repeated until the atom or molecule center-of-mass is inside the specified region.

Note that neighbor lists are re-built every timestep that this fix is invoked, so you should not set N to be too small. See the *neighbor* command for details.

When an atom or molecule is to be inserted, its coordinates are chosen at a random position within the current simulation cell or region. Relative coordinates for atoms in a molecule are taken from the template molecule provided by the user. The center of mass of the molecule is placed at the insertion point. The orientation of the molecule is chosen at random by rotating about this point.

Individual atoms are inserted, unless the *mol* keyword is used. It specifies a *template-ID* previously defined using the *molecule* command, which reads a file that defines the molecule. The coordinates, atom types, charges, etc., as well as any bonding and special neighbor information for the molecule can be specified in the molecule file. See the *molecule* command for details. The only settings required to be in this file are the coordinates and types of atoms in the molecule.

Note that fix widom does not use configurational bias MC or any other kind of sampling of intramolecular degrees of freedom. Inserted molecules can have different orientations, but they will all have the same intramolecular configuration, which was specified in the molecule command input.

For atoms, inserted particles have the specified atom type. For molecules, they use the same atom types as in the template molecule supplied by the user.

The excess chemical potential mu_ex is defined as:

$$\mu_{ex} = -kT \ln(< \exp(-(U_{N+1} - U_N)/k_B T) >)$$

where $k_B$ is the Boltzmann constant, $T$ is the user-specified temperature, $U_N$ and $U_{N+1}$ is the potential energy of the system with $N$ and $N + 1$ particles.

The *full_energy* option means that the fix calculates the total potential energy of the entire simulated system, instead of just the energy of the part that is changed. By default, this option is off, in which case only partial energies are computed to determine the energy difference due to the proposed change.

The *full_energy* option is needed for systems with complicated potential energy calculations, including the following:

- long-range electrostatics (kspace)
- many-body pair styles
- hybrid pair styles
- eam pair styles
- tail corrections
- need to include potential energy contributions from other fixes

In these cases, LAMMPS will automatically apply the *full_energy* keyword and issue a warning message.

When the *mol* keyword is used, the *full_energy* option also includes the intramolecular energy of inserted and deleted molecules, whereas this energy is not included when *full_energy* is not used. If this is not desired, the *intra_energy* keyword can be used to define an amount of energy that is subtracted from the final energy when a molecule is inserted, and subtracted from the initial energy when a molecule is deleted. For molecules that have a non-zero intramolecular energy, this will ensure roughly the same behavior whether or not the *full_energy* option is used.

Some fixes have an associated potential energy. Examples of such fixes include: *efield*, *gravity*, *addforce*, *restrain*, and *wall fixes*. For that energy to be included in the total potential energy of the system (the quantity used when performing Widom insertions), you MUST enable the *fix_modify energy* option for that fix. The doc pages for individual *fix* commands specify if this should be done.

Use the *charge* option to insert atoms with a user-specified point charge. Note that doing so will cause the system to become non-neutral. LAMMPS issues a warning when using long-range electrostatics (kspace) with non-neutral systems. See the *compute group/group* documentation for more details about simulating non-neutral systems with kspace on.

### 2.298.4 Restart, fix_modify, output, run start/stop, minimize info

This fix writes the state of the fix to *binary restart files*. This includes information about the random number generator seed, the next timestep for Widom insertions etc. See the *read_restart* command for info on how to re-specify a fix in an input script that reads a restart file, so that the operation of the fix continues in an uninterrupted fashion.

> **Note**
>
> For this to work correctly, the timestep must **not** be changed after reading the restart with *reset_timestep*. The fix will try to detect it and stop with an error.

None of the *fix_modify* options are relevant to this fix.

This fix computes a global vector of length 3, which can be accessed by various *output commands*. The vector values are the following global cumulative quantities:

1. average excess chemical potential on each timestep
2. average difference in potential energy on each timestep
3. volume of the insertion region

The vector values calculated by this fix are “intensive”.

No parameter of this fix can be used with the *start/stop* keywords of the *run* command. This fix is not invoked during *energy minimization*.

### 2.298.5 Restrictions

This fix is part of the MC package. It is only enabled if LAMMPS was built with that package. See the *Build package* doc page for more info.

Do not set “neigh_modify once yes” or else this fix will never be called. Reneighboring is **required**.

This fix style requires an *atom style* with per atom type masses.

Can be run in parallel, but some aspects of the insertion procedure will not scale well in parallel. Only usable for 3D simulations.

### 2.298.6 Related commands

*fix gcmc fix gemc*, *fix atom/swap*, *neighbor*, *fix deposit*, *fix evaporate*,

### 2.298.7 Default

The option defaults are mol = no, intra_energy = 0.0 and full_energy = no, except for the situations where full_energy is required, as listed above.

---

**(Frenkel)** Frenkel and Smit, Understanding Molecular Simulation, Academic Press, London, 2002.

CHAPTER

THREE

# COMPUTE STYLES

## 3.1 compute ackland/atom command

Accelerator Variants: *ackland/atom/omp*

### 3.1.1 Syntax

```
compute ID group-ID ackland/atom keyword/value
```

- ID, group-ID are documented in *compute* command
- ackland/atom = style name of this compute command
- zero or more keyword/value pairs may be appended
- keyword = *legacy*

  ```
  legacy args = yes or no = use (yes) or do not use (no) legacy Ackland algorithm␣
  ↪implementation
  ```

### 3.1.2 Examples

```
compute 1 all ackland/atom
compute 1 all ackland/atom legacy yes
```

### 3.1.3 Description

Defines a computation that calculates the local lattice structure according to the formulation given in *(Ackland)*. Historically, LAMMPS had two, slightly different implementations of the algorithm from the paper. With the *legacy* keyword, it is possible to switch between the pre-2015 (*legacy yes*) and post-2015 implementation (*legacy no*). The post-2015 variant is the default.

In contrast to the *centro-symmetry parameter* this method is stable against temperature boost, because it is based not on the distance between particles but the angles. Therefore statistical fluctuations are averaged out a little more. A comparison with the Common Neighbor Analysis metric is made in the paper.

The result is a number which is mapped to the following different lattice structures:

- 0 = UNKNOWN
- 1 = BCC
- 2 = FCC
- 3 = HCP

- 4 = ICO

The neighbor list needed to compute this quantity is constructed each time the calculation is performed (i.e. each time a snapshot of atoms is dumped). Thus it can be inefficient to compute/dump this quantity too frequently or to have multiple compute/dump commands, each of which computes this quantity.-

### 3.1.4 Output info

This compute calculates a per-atom vector, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 3.1.5 Restrictions

This compute is part of the EXTRA-COMPUTE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The per-atom vector values will be unitless since they are the integers defined above.

### 3.1.6 Related commands

*compute centro/atom*

### 3.1.7 Default

The keyword *legacy* defaults to *no*.

---

**(Ackland)** Ackland, Jones, Phys Rev B, 73, 054104 (2006).

## 3.2 compute adf command

### 3.2.1 Syntax

```
compute ID group-ID adf Nbin itype1 jtype1 ktype1 Rjinner1 Rjouter1 Rkinner1 Rkouter1 ...
```

- ID, group-ID are documented in *compute* command
- adf = style name of this compute command
- Nbin = number of ADF bins
- itypeN = central atom type for Nth ADF histogram (see asterisk form below)

- jtypeN = J atom type for Nth ADF histogram (see asterisk form below)
- ktypeN = K atom type for Nth ADF histogram (see asterisk form below)
- RjinnerN = inner radius of J atom shell for Nth ADF histogram (distance units)
- RjouterN = outer radius of J atom shell for Nth ADF histogram (distance units)
- RkinnerN = inner radius of K atom shell for Nth ADF histogram (distance units)
- RkouterN = outer radius of K atom shell for Nth ADF histogram (distance units)
- zero or one keyword/value pairs may be appended
- keyword = *ordinate*

```
ordinate value = degree or radian or cosine
  Choose the ordinate parameter for the histogram
```

### 3.2.2 Examples

```
compute 1 fluid adf 32 1 1 1 0.0 1.2 0.0 1.2 &
                       1 1 2 0.0 1.2 0.0 1.5 &
                       1 2 2 0.0 1.5 0.0 1.5 &
                       2 1 1 0.0 1.2 0.0 1.2 &
                       2 1 2 0.0 1.5 2.0 3.5 &
                       2 2 2 2.0 3.5 2.0 3.5
compute 1 fluid adf 32 1*2 1*2 1*2 0.5 3.5
compute 1 fluid adf 32
```

### 3.2.3 Description

Define a computation that calculates one or more angular distribution functions (ADF) for a group of particles. Each ADF is calculated in histogram form by measuring the angle formed by a central atom and two neighbor atoms and binning these angles into *Nbin* bins. Only neighbors for which *Rinner* < *R* < *Router* are counted, where *Rinner* and *Router* are specified separately for the first and second neighbor atom in each requested ADF.

**Note**

If you have a bonded system, then the settings of *special_bonds* command can remove pairwise interactions between atoms in the same bond, angle, or dihedral. This is the default setting for the *special_bonds* command, and means those pairwise interactions do not appear in the neighbor list. Because this fix uses a neighbor list, it also means those pairs will not be included in the ADF. This does not apply when using long-range coulomb interactions (*coul/long*, *coul/msm*, *coul/wolf* or similar. One way to get around this would be to set special_bond scaling factors to very tiny numbers that are not exactly zero (e.g. 1.0e-50). Another workaround is to write a dump file, and use the *rerun* command to compute the ADF for snapshots in the dump file. The rerun script can use a *special_bonds* command that includes all pairs in the neighbor list.

**Note**

If you request any outer cutoff *Router* > force cutoff, or if no pair style is defined, e.g. the *rerun* command is being used to post-process a dump file of snapshots you must ensure ghost atom information out to the largest value of *Router* + *skin* is communicated, via the *comm_modify cutoff* command, else the ADF computation cannot be performed, and LAMMPS will give an error message. The *skin* value is what is specified with the *neighbor* command.

The *itypeN*,*jtypeN*,*ktypeN* settings can be specified in one of two ways. An explicit numeric value can be used, as in the first example above. Or a wild-card asterisk can be used to specify a range of atom types as in the second example above. This takes the form "*" or "*n" or "n*" or "m*n". If N = the number of atom types, then an asterisk with no numeric values means all types from 1 to N. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to N (inclusive). A middle asterisk means all types from m to n (inclusive).

If *itypeN*, *jtypeN*, and *ktypeN* are single values, as in the first example above, this means that the ADF is computed where atoms of type *itypeN* are the central atom, and neighbor atoms of type *jtypeN* and *ktypeN* are forming the angle. If any of *itypeN*, *jtypeN*, or *ktypeN* represent a range of values via the wild-card asterisk, as in the second example above, this means that the ADF is computed where atoms of any of the range of types represented by *itypeN* are the central atom, and the angle is formed by two neighbors, one neighbor in the range of types represented by *jtypeN* and another neighbor in the range of types represented by *ktypeN*.

If no *itypeN*, *jtypeN*, *ktypeN* settings are specified, then LAMMPS will generate a single ADF for all atoms in the group. The inner cutoff is set to zero and the outer cutoff is set to the force cutoff. If no pair_style is specified, there is no force cutoff and LAMMPS will give an error message. Note that in most cases, generating an ADF for all atoms is not a good thing. Such an ADF is both uninformative and extremely expensive to compute. For example, with liquid water with a 10 A force cutoff, there are 80,000 angles per atom. In addition, most of the interesting angular structure occurs for neighbors that are the closest to the central atom, involving just a few dozen angles.

Angles for each ADF are generated by double-looping over the list of neighbors of each central atom I, just as they would be in the force calculation for a three-body potential such as *Stillinger-Weber*. The angle formed by central atom I and neighbor atoms J and K is included in an ADF if the following criteria are met:

- atoms I,J,K are all in the specified compute group
- the distance between atoms I,J is between Rjinner and Rjouter
- the distance between atoms I,K is between Rkinner and Rkouter
- the type of the I atom matches itypeN (one or a range of types)
- atoms I,J,K are distinct
- the type of the J atom matches jtypeN (one or a range of types)
- the type of the K atom matches ktypeN (one or a range of types)

Each unique angle satisfying the above criteria is counted only once, regardless of whether either or both of the neighbor atoms making up the angle appear in both the J and K lists. It is OK if a particular angle is included in more than one individual histogram, due to the way the *itypeN*, *jtypeN*, *ktypeN* arguments are specified.

The first ADF value for a bin is calculated from the histogram count by dividing by the total number of triples satisfying the criteria, so that the integral of the ADF w.r.t. angle is 1, i.e. the ADF is a probability density function.

The second ADF value is reported as a cumulative sum of all bins up to the current bins, averaged over atoms of type *itypeN*. It represents the number of angles per central atom with angle less than or equal to the angle of the current bin, analogous to the coordination number radial distribution function.

The *ordinate* optional keyword determines whether the bins are of uniform angular size from zero to 180 (*degree*), zero to Pi (*radian*), or the cosine of the angle uniform in the range [-1,1] (*cosine*). *cosine* has the advantage of eliminating the *acos()* function call, which speeds up the compute by 2-3x, and it is also preferred on physical grounds, because the for uniformly distributed particles in 3D, the angular probability density w.r.t dtheta is sin(theta)/2, while for d(cos(theta)), it is 1/2, Regardless of which ordinate is chosen, the first column of ADF values is normalized w.r.t. the range of that ordinate, so that the integral is 1.

The simplest way to output the results of the compute adf calculation to a file is to use the *fix ave/time* command, for example:

```
compute myADF all adf 32 2 2 2 0.5 3.5 0.5 3.5
fix 1 all ave/time 100 1 100 c_myADF[*] file tmp.adf mode vector
```

### 3.2.4 Output info

This compute calculates a global array with the number of rows = *Nbins* and the number of columns = $1 + 2 \times$ *Ntriples*, where *Ntriples* is the number of I,J,K triples specified. The first column has the bin coordinate (angle-related ordinate at midpoint of bin). Each subsequent column has the two ADF values for a specific set of (*itypeN*,*jtypeN*,*ktypeN*) interactions, as described above. These values can be used by any command that uses a global values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The array values calculated by this compute are all "intensive".

The first column of array values is the angle-related ordinate, either the angle in degrees or radians, or the cosine of the angle. Each subsequent pair of columns gives the first and second kinds of ADF for a specific set of (*itypeN*,*jtypeN*,*ktypeN*). The values in the first ADF column are normalized numbers $\geq 0.0$, whose integral w.r.t. the ordinate is 1, i.e. the first ADF is a normalized probability distribution. The values in the second ADF column are also numbers $\geq 0.0$. They are the cumulative density distribution of angles per atom. By definition, this ADF is monotonically increasing from zero to a maximum value equal to the average total number of angles per atom satisfying the ADF criteria.

### 3.2.5 Restrictions

This compute is part of the EXTRA-COMPUTE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

By default, the ADF is not computed for distances longer than the largest force cutoff, since the neighbor list creation will only contain pairs up to that distance (plus neighbor list skin). If you use outer cutoffs larger than that, you must use *neighbor style 'bin' or 'nsq'*.

If you want an ADF for a larger outer cutoff, you can also use the *rerun* command to post-process a dump file, use *pair style zero* and set the force cutoff to be larger in the rerun script. Note that in the rerun context, the force cutoff is arbitrary and with pair style zero you are not computing any forces, and since you are not running dynamics you are not changing the model that generated the trajectory.

The ADF is not computed for neighbors outside the force cutoff, since processors (in parallel) don't know about atom coordinates for atoms further away than that distance. If you want an ADF for larger distances, you can use the *rerun* command to post-process a dump file and set the cutoff for the potential to be longer in the rerun script. Note that in the rerun context, the force cutoff is arbitrary, since you are not running dynamics and thus are not changing your model.

### 3.2.6 Related commands

*compute rdf*, *fix ave/time*, *compute_modify*

### 3.2.7 Default

The keyword default is ordinate = degree.

## 3.3 compute angle command

### 3.3.1 Syntax

```
compute ID group-ID angle
```

- ID, group-ID are documented in *compute* command
- angle = style name of this compute command

### 3.3.2 Examples

```
compute 1 all angle
```

### 3.3.3 Description

Define a computation that extracts the angle energy calculated by each of the angle sub-styles used in the *angle_style hybrid* command. These values are made accessible for output or further processing by other commands. The group specified for this command is ignored.

This compute is useful when using *angle_style hybrid* if you want to know the portion of the total energy contributed by one or more of the hybrid sub-styles.

### 3.3.4 Output info

This compute calculates a global vector of length $N$, where $N$ is the number of sub_styles defined by the *angle_style hybrid* command, which can be accessed by indices 1 through $N$. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The vector values are "extensive" and will be in energy *units*.

### 3.3.5 Restrictions

none

### 3.3.6 Related commands

*compute pe*, *compute pair*

### 3.3.7 Default

none

## 3.4 compute angle/local command

### 3.4.1 Syntax

```
compute ID group-ID angle/local value1 value2 ... keyword args ...
```

- ID, group-ID are documented in *compute* command
- angle/local = style name of this compute command
- one or more values may be appended
- value = *theta* or *eng* or *v_name*

  *theta* = tabulate angles
  *eng* = tabulate angle energies
  *v_name* = equal-style variable with name (see below)

- zero or more keyword/args pairs may be appended
- keyword = *set*

```
set args = theta name
  theta = only currently allowed arg
  name = name of variable to set with theta
```

### 3.4.2 Examples

```
compute 1 all angle/local theta
compute 1 all angle/local eng theta
compute 1 all angle/local theta v_cos set theta t
```

### 3.4.3 Description

Define a computation that calculates properties of individual angle interactions. The number of datums generated, aggregated across all processors, equals the number of angles in the system, modified by the group parameter as explained below.

The value *theta* is the angle for the three atoms in the interaction.

The value *eng* is the interaction energy for the angle.

The value *v_name* can be used together with the *set* keyword to compute a user-specified function of the angle theta. The *name* specified for the *v_name* value is the name of an *equal-style variable* which should evaluate a formula based on a variable which will store the angle theta. This other variable must be an *internal-style variable* specified by the *set* keyword. It is an internal-style variable, because this command resets its value directly. The internal-style variable does not need to be defined in the input script (though it can be); if it is not defined, then the *set* option creates an *internal-style variable* with the specified name.

Note that the value of theta for each angle which is stored in the internal variable is in radians, not degrees.

As an example, these commands can be added to the bench/in.rhodo script to compute the cosine and cosine-squared of every angle in the system and output the statistics in various ways:

```
variable cos equal cos(v_t)
variable cossq equal cos(v_t)*cos(v_t)

compute 1 all property/local aatom1 aatom2 aatom3 atype
compute 2 all angle/local eng theta v_cos v_cossq set theta t
dump 1 all local 100 tmp.dump c_1[*] c_2[*]

compute 3 all reduce ave c_2[*] inputs local
thermo_style custom step temp press c_3[*]

fix 10 all ave/histo 10 10 100 -1 1 20 c_2[3] mode vector file tmp.histo
```

The *dump local* command will output the potential energy ($\phi$), the angle ($\theta$), $\cos(\theta)$, and $\cos^2(\theta)$ for every angle $\theta$ in the system. The *thermo_style* command will print the average of those quantities via the *compute reduce* command with thermo output. And the *fix ave/histo* command will histogram the $\cos(\theta)$ values and write them to a file.

---

The local data stored by this command is generated by looping over all the atoms owned on a processor and their angles. An angle will only be included if all three atoms in the angle are in the specified compute group. Any angles that have been broken (see the *angle_style* command) by setting their angle type to 0 are not included. Angles that have been turned off (see the *fix shake* or *delete_bonds* commands) by setting their angle type negative are written into the file, but their energy will be 0.0.

Note that as atoms migrate from processor to processor, there will be no consistent ordering of the entries within the local vector or array from one timestep to the next. The only consistency that is guaranteed is that the ordering on a

particular timestep will be the same for local vectors or arrays generated by other compute commands. For example, angle output from the *compute property/local* command can be combined with data from this command and output by the *dump local* command in a consistent way.

Here is an example of how to do this:

```
compute 1 all property/local atype aatom1 aatom2 aatom3
compute 2 all angle/local theta eng
dump 1 all local 1000 tmp.dump index c_1[1] c_1[2] c_1[3] c_1[4] c_2[1] c_2[2]
```

### 3.4.4 Output info

This compute calculates a local vector or local array depending on the number of values. The length of the vector or number of rows in the array is the number of angles. If a single value is specified, a local vector is produced. If two or more values are specified, a local array is produced where the number of columns = the number of values. The vector or array can be accessed by any command that uses local values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The output for *theta* will be in degrees. The output for *eng* will be in energy *units*.

### 3.4.5 Restrictions

none

### 3.4.6 Related commands

*dump local*, *compute property/local*

### 3.4.7 Default

none

## 3.5 compute angmom/chunk command

### 3.5.1 Syntax

```
compute ID group-ID angmom/chunk chunkID
```

- ID, group-ID are documented in *compute* command
- angmom/chunk = style name of this compute command
- chunkID = ID of *compute chunk/atom* command

### 3.5.2 Examples

```
compute 1 fluid angmom/chunk molchunk
```

### 3.5.3 Description

Define a computation that calculates the angular momentum of multiple chunks of atoms.

In LAMMPS, chunks are collections of atoms defined by a *compute chunk/atom* command, which assigns each atom to a single chunk (or no chunk). The ID for this command is specified as chunkID. For example, a single chunk could be the atoms in a molecule or atoms in a spatial bin. See the *compute chunk/atom* and *Howto chunk* doc pages for details of how chunks can be defined and examples of how they can be used to measure properties of a system.

This compute calculates the 3 components of the angular momentum vector for each chunk, due to the velocity/momentum of the individual atoms in the chunk around the center-of-mass of the chunk. The calculation includes all effects due to atoms passing through periodic boundaries.

Note that only atoms in the specified group contribute to the calculation. The *compute chunk/atom* command defines its own group; atoms will have a chunk ID = 0 if they are not in that group, signifying they are not assigned to a chunk, and will thus also not contribute to this calculation. You can specify the "all" group for this command if you simply want to include atoms with non-zero chunk IDs.

**Note**

The coordinates of an atom contribute to the chunk's angular momentum in "unwrapped" form, by using the image flags associated with each atom. See the *dump custom* command for a discussion of "unwrapped" coordinates. See the Atoms section of the *read_data* command for a discussion of image flags and how they are set for each atom. You can reset the image flags (e.g. to 0) before invoking this compute by using the *set image* command.

The simplest way to output the results of the compute angmom/chunk calculation to a file is to use the *fix ave/time* command, for example:

```
compute cc1 all chunk/atom molecule
compute myChunk all angmom/chunk cc1
fix 1 all ave/time 100 1 100 c_myChunk[*] file tmp.out mode vector
```

### 3.5.4 Output info

This compute calculates a global array where the number of rows = the number of chunks *Nchunk* as calculated by the specified *compute chunk/atom* command. The number of columns = 3 for the three ($x$, $y$, $z$) components of the angular momentum for each chunk. These values can be accessed by any command that uses global array values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The array values are "intensive". The array values will be in mass-velocity-distance *units*.

### 3.5.5 Restrictions

none

### 3.5.6 Related commands

*variable angmom() function*

### 3.5.7 Default

none

## 3.6 compute ave/sphere/atom command

Accelerator Variants: *ave/sphere/atom/kk*, *ave/sphere/atom/omp*

### 3.6.1 Syntax

```
compute ID group-ID ave/sphere/atom keyword values ...
```

- ID, group-ID are documented in *compute* command

- ave/sphere/atom = style name of this compute command
- one or more keyword/value pairs may be appended

```
keyword = cutoff
  cutoff value = distance cutoff
```

### 3.6.2 Examples

```
compute 1 all ave/sphere/atom

compute 1 all ave/sphere/atom cutoff 5.0
comm_modify cutoff 5.0
```

### 3.6.3 Description

Added in version 7Jan2022.

Define a computation that calculates the local mass density and temperature for each atom based on its neighbors inside a spherical cutoff. If an atom has $M$ neighbors, then its local mass density is calculated as the sum of its mass and its $M$ neighbor masses, divided by the volume of the cutoff sphere (or circle in 2d). The local temperature of the atom is calculated as the temperature of the collection of $M + 1$ atoms, after subtracting the center-of-mass velocity of the $M + 1$ atoms from each of the $M + 1$ atom's velocities. This is effectively the thermal velocity of the neighborhood of the central atom, similar to *compute temp/com*.

The optional keyword *cutoff* defines the distance cutoff used when searching for neighbors. The default value is the cutoff specified by the pair style. If no pair style is defined, then a cutoff must be defined using this keyword. If the specified cutoff is larger than that of the pair_style plus neighbor skin (or no pair style is defined), the *comm_modify cutoff* option must also be set to match that of the *cutoff* keyword.

The neighbor list needed to compute this quantity is constructed each time the calculation is performed (i.e. each time a snapshot of atoms is dumped). Thus it can be inefficient to compute/dump this quantity too frequently.

**Note**

If you have a bonded system, then the settings of *special_bonds* command can remove pairwise interactions between atoms in the same bond, angle, or dihedral. This is the default setting for the *special_bonds* command, and means those pairwise interactions do not appear in the neighbor list. Because this compute uses the neighbor list, it also means those pairs will not be included in the order parameter. This difficulty can be circumvented by writing a dump file, and using the *rerun* command to compute the order parameter for snapshots in the dump file. The rerun script can use a *special_bonds* command that includes all pairs in the neighbor list.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 3.6.4 Output info

This compute calculates a per-atom array with two columns: mass density in density *units* and temperature in temperature *units*.

These values can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

### 3.6.5 Restrictions

This compute is part of the EXTRA-COMPUTE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This compute requires *neighbor styles 'bin' or 'nsq'*.

### 3.6.6 Related commands

*comm_modify*

### 3.6.7 Default

The option defaults are *cutoff* = pair style cutoff.

## 3.7 compute basal/atom command

### 3.7.1 Syntax

```
compute ID group-ID basal/atom
```

- ID, group-ID are documented in *compute* command
- basal/atom = style name of this compute command

### 3.7.2 Examples

```
compute 1 all basal/atom
```

### 3.7.3 Description

Defines a computation that calculates the hexagonal close-packed "c" lattice vector for each atom in the group. It does this by calculating the normal unit vector to the basal plane for each atom. The results enable efficient identification and characterization of twins and grains in hexagonal close-packed structures.

The output of the compute is thus the 3 components of a unit vector associated with each atom. The components are set to 0.0 for atoms not in the group.

Details of the calculation are given in *(Barrett)*.

The neighbor list needed to compute this quantity is constructed each time the calculation is performed (i.e. each time a snapshot of atoms is dumped). Thus it can be inefficient to compute/dump this quantity too frequently or to have multiple compute/dump commands, each of which computes this quantity.

An example input script that uses this compute is provided in examples/PACKAGES/basal.

### 3.7.4 Output info

This compute calculates a per-atom array with three columns, which can be accessed by indices 1–3 by any command that uses per-atom values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

The per-atom vector values are unitless since the three columns represent components of a unit vector.

### 3.7.5 Restrictions

This compute is part of the EXTRA-COMPUTE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The output of this compute will be meaningless unless the atoms are on (or near) hcp lattice sites, since the calculation assumes a well-defined basal plane.

### 3.7.6 Related commands

*compute centro/atom*, *compute ackland/atom*

### 3.7.7 Default

none

---

**(Barrett)** Barrett, Tschopp, El Kadiri, Scripta Mat. 66, p.666 (2012).

## 3.8 compute body/local command

### 3.8.1 Syntax

```
compute ID group-ID body/local input1 input2 ...
```

- ID, group-ID are documented in *compute* command
- body/local = style name of this compute command
- one or more keywords may be appended
- keyword = *id* or *type* or *integer*

  ```
  id = atom ID of the body particle
  type = atom type of the body particle
  integer = 1,2,3,etc = index of fields defined by body style
  ```

### 3.8.2 Examples

```
compute 1 all body/local type 1 2 3
compute 1 all body/local 3 6
```

### 3.8.3 Description

Define a computation that calculates properties of individual body sub-particles. The number of data generated, aggregated across all processors, equals the number of body sub-particles plus the number of non-body particles in the system, modified by the group parameter as explained below. See the *Howto body* page for more details on using body particles.

The local data stored by this command is generated by looping over all the atoms. An atom will only be included if it is in the group. If the atom is a body particle, then its $N$ sub-particles will be looped over, and it will contribute $N$ data to the count of data. If it is not a body particle, it will contribute 1 datum.

For both body particles and non-body particles, the *id* keyword will store the ID of the particle.

For both body particles and non-body particles, the *type* keyword will store the type of the particle.

The *integer* keywords mean different things for body and non-body particles. If the atom is not a body particle, only its $x$, $y$, $z$ coordinates can be referenced, using the *integer* keywords 1,2,3. Note that this means that if you want to access more fields than this for body particles, then you cannot include non-body particles in the group.

For a body particle, the *integer* keywords refer to fields calculated by the body style for each sub-particle. The body style, as specified by the *atom_style body*, determines how many fields exist and what they are. See the *Howto_body* doc page for details of the different styles.

Here is an example of how to output body information using the *dump local* command with this compute. If fields 1, 2, and 3 for the body sub-particles are ($x$, $y$, $z$) coordinates, then the dump file will be formatted similar to the output of a *dump atom or custom* command.

```
compute 1 all body/local type 1 2 3
dump 1 all local 1000 tmp.dump index c_1[1] c_1[2] c_1[3] c_1[4]
```

### 3.8.4 Output info

This compute calculates a local vector or local array depending on the number of keywords. The length of the vector or number of rows in the array is the number of data as described above. If a single keyword is specified, a local vector is produced. If two or more keywords are specified, a local array is produced where the number of columns = the number of keywords. The vector or array can be accessed by any command that uses local values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The *units* for output values depend on the body style.

### 3.8.5 Restrictions

none

### 3.8.6 Related commands

*dump local*

### 3.8.7 Default

none

## 3.9 compute bond command

### 3.9.1 Syntax

```
compute ID group-ID bond
```

- ID, group-ID are documented in *compute* command
- bond = style name of this compute command

### 3.9.2 Examples

```
compute 1 all bond
```

### 3.9.3 Description

Define a computation that extracts the bond energy calculated by each of the bond sub-styles used in the *bond_style hybrid* command. These values are made accessible for output or further processing by other commands. The group specified for this command is ignored.

This compute is useful when using *bond_style hybrid* if you want to know the portion of the total energy contributed by one or more of the hybrid sub-styles.

### 3.9.4 Output info

This compute calculates a global vector of length $N$, where $N$ is the number of sub_styles defined by the *bond_style hybrid* command, which can be accessed by indices 1 through $N$. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The vector values are "extensive" and will be in energy *units*.

### 3.9.5 Restrictions

none

### 3.9.6 Related commands

*compute pe*, *compute pair*

### 3.9.7 Default

none

## 3.10 compute bond/local command

### 3.10.1 Syntax

```
compute ID group-ID bond/local value1 value2 ... keyword args ...
```

- ID, group-ID are documented in *compute* command
- bond/local = style name of this compute command
- one or more values may be appended
- value = *dist* or *dx* or *dy* or *dz* or *engpot* or *force* or *fx* or *fy* or *fz* or *engvib* or *engrot* or *engtrans* or *omega* or *velvib* or *v_name* or *bN*

```
dist = bond distance
engpot = bond potential energy
force = bond force
dx,dy,dz = components of pairwise distance
fx,fy,fz = components of bond force
engvib = bond kinetic energy of vibration
engrot = bond kinetic energy of rotation
engtrans = bond kinetic energy of translation
```

```
omega = magnitude of bond angular velocity
velvib = vibrational velocity along the bond length
v_name = equal-style variable with name (see below)
bN = bond style specific quantities for allowed N values
```

- zero or more keyword/args pairs may be appended
- keyword = *set*

```
set args = dist name
  dist = only currently allowed arg
  name = name of variable to set with distance (dist)
```

### 3.10.2 Examples

```
compute 1 all bond/local engpot
compute 1 all bond/local dist engpot force
compute 1 all bond/local dist fx fy fz b1 b2
compute 1 all bond/local dist v_distsq set dist d
```

### 3.10.3 Description

Define a computation that calculates properties of individual bond interactions. The number of datums generated, aggregated across all processors, equals the number of bonds in the system, modified by the group parameter as explained below.

All these properties are computed for the pair of atoms in a bond, whether the two atoms represent a simple diatomic molecule, or are part of some larger molecule.

Changed in version 12Jun2025: The sign of *dx*, *dy*, *dz* is no longer determined by the atom IDs of the bonded atoms but by their order in the bond list to be consistent with *fx*, *fy*, and *fz*.

The value *dist* is the current length of the bond. The values *dx*, *dy*, and *dz* are the $(x, y, z)$ components of the distance vector $\vec{x}_i - \vec{x}_j$ between the atoms in the bond. The order of the atoms is determined by the bond list and the respective atom-IDs can be output with *compute property/local*.

The value *engpot* is the potential energy for the bond, based on the current separation of the pair of atoms in the bond.

The value *force* is the magnitude of the force acting between the pair of atoms in the bond, which is positive for a repulsive force and negative for an attractive force.

The values *fx*, *fy*, and *fz* are the $(x, y, z)$ components of the force on the first atom $i$ in the bond due to the second atom $j$. Mathematically, they are obtained by multiplying the value of *force* from above with a unit vector created from the *dx*, *dy*, and *dz* components of the distance vector also described above. For bond styles that apply non-central forces, such as *bond_style bpm/rotational*, these values only include the $(x, y, z)$ components of the normal force component.

The remaining properties are all computed for motion of the two atoms relative to the center of mass (COM) velocity of the two atoms in the bond.

The value *engvib* is the vibrational kinetic energy of the two atoms in the bond, which is simply $\frac{1}{2}m_1v_1^2 + \frac{1}{2}m_2v_2^2$, where $v_1$ and $v_2$ are the magnitude of the velocity of the two atoms along the bond direction, after the COM velocity has been subtracted from each.

The value *engrot* is the rotational kinetic energy of the two atoms in the bond, which is simply $\frac{1}{2}m_1v_1^2 + \frac{1}{2}m_2v_2^2$, where $v_1$ and $v_2$ are the magnitude of the velocity of the two atoms perpendicular to the bond direction, after the COM velocity has been subtracted from each.

The value *engtrans* is the translational kinetic energy associated with the motion of the COM of the system itself, namely $\frac{1}{2}(m_1 + m_2)V_{\mathrm{cm}}^2$, where *Vcm* = magnitude of the velocity of the COM.

Note that these three kinetic energy terms are simply a partitioning of the summed kinetic energy of the two atoms themselves. That is, the total kinetic energy is $\frac{1}{2}m_1v_1^2 + \frac{1}{2}m_2v_2^2$ = engvib + engrot + engtrans, where $v_1$ and $v_2$ are the magnitude of the velocities of the two atoms, without any adjustment for the COM velocity.

The value *omega* is the magnitude of the angular velocity of the two atoms around their COM position.

The value *velvib* is the magnitude of the relative velocity of the two atoms in the bond towards each other. A negative value means the two atoms are moving toward each other; a positive value means they are moving apart.

The value *v_name* can be used together with the *set* keyword to compute a user-specified function of the bond distance. The *name* specified for the *v_name* value is the name of an *equal-style variable* which should evaluate a formula based on a variable which stores the bond distance. This other variable must be the *internal-style variable* specified by the *set* keyword. It is an internal-style variable, because this command resets its value directly. The internal-style variable does not need to be defined in the input script (though it can be); if it is not defined, then the *set* option creates an *internal-style variable* with the specified name.

As an example, these commands can be added to the bench/in.rhodo script to compute the length$^2$ of every bond in the system and output the statistics in various ways:

```
variable dsq equal v_d*v_d

compute 1 all property/local batom1 batom2 btype
compute 2 all bond/local engpot dist v_dsq set dist d
dump 1 all local 100 tmp.dump c_1[*] c_2[*]

compute 3 all reduce ave c_2[*] inputs local
thermo_style custom step temp press c_3[*]

fix 10 all ave/histo 10 10 100 0 6 20 c_2[3] mode vector file tmp.histo
```

The *dump local* command will output the energy, length, and length$^2$ for every bond in the system. The *thermo_style* command will print the average of those quantities via the *compute reduce* command with thermo output, and the *fix ave/histo* command will histogram the length$^2$ values and write them to a file.

A bond style may define additional bond quantities which can be accessed as *b1* to *bN*, where N is defined by the bond style. Most bond styles do not define any additional quantities, so N = 0. An example of ones that do are the *BPM bond styles* which store the reference state between two particles. See individual bond styles for details.

When using *bN* with bond style *hybrid*, the output will be the Nth quantity from the sub-style that computes the bonded interaction (based on bond type). If that sub-style does not define a *bN*, the output will be 0.0. The maximum allowed N is the maximum number of quantities provided by any sub-style.

---

The local data stored by this command is generated by looping over all the atoms owned on a processor and their bonds. A bond will only be included if both atoms in the bond are in the specified compute group. Any bonds that have been broken (see the *bond_style* command) by setting their bond type to 0 are not included. Bonds that have been turned off (see the *fix shake* or *delete_bonds* commands) by setting their bond type negative are written into the file, but their energy will be 0.0.

Note that as atoms migrate from processor to processor, there will be no consistent ordering of the entries within the local vector or array from one timestep to the next. The only consistency that is guaranteed is that the ordering on a particular timestep will be the same for local vectors or arrays generated by other compute commands. For example, bond output from the *compute property/local* command can be combined with data from this command and output by the *dump local* command in a consistent way.

Here is an example of how to do this:

```
compute 1 all property/local btype batom1 batom2
compute 2 all bond/local dist engpot
dump 1 all local 1000 tmp.dump index c_1[*] c_2[*]
```

### 3.10.4 Output info

This compute calculates a local vector or local array depending on the number of values. The length of the vector or number of rows in the array is the number of bonds. If a single value is specified, a local vector is produced. If two or more values are specified, a local array is produced where the number of columns = the number of values. The vector or array can be accessed by any command that uses local values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The output for *dist* will be in distance *units*. The output for *velvib* will be in velocity *units*. The output for *omega* will be in velocity/distance *units*. The output for *engtrans*, *engvib*, *engrot*, and *engpot* will be in energy *units*. The output for *force* will be in force *units*.

### 3.10.5 Restrictions

none

### 3.10.6 Related commands

*dump local*, *compute property/local*

### 3.10.7 Default

none

## 3.11 compute born/matrix command

### 3.11.1 Syntax

```
compute ID group-ID born/matrix keyword value ...
```

- ID, group-ID are documented in *compute* command
- born/matrix = style name of this compute command
- zero or more keywords or keyword/value pairs may be appended

```
keyword = numdiff or pair or bond or angle or dihedral or improper
  numdiff values = delta virial-ID
    delta = magnitude of strain (dimensionless)
    virial-ID = ID of pressure compute for virial (string)
    (numdiff cannot be used with any other keyword)
  pair = compute pair-wise contributions
  bond = compute bonding contributions
  angle = compute angle contributions
  dihedral = compute dihedral contributions
  improper = compute improper contributions
```

### 3.11.2 Examples

```
compute 1 all born/matrix
compute 1 all born/matrix bond angle
compute 1 all born/matrix numdiff 1.0e-4 myvirial
```

### 3.11.3 Description

Added in version 4May2022.

Define a compute that calculates $\frac{\partial^2 U}{\partial \varepsilon_i \partial \varepsilon_j}$, the second derivatives of the potential energy $U$ with respect to the strain tensor $\varepsilon$ elements. These values are related to:

$$C^B_{i,j} = \frac{1}{V} \frac{\partial^2 U}{\partial \varepsilon_i \partial \varepsilon_j}$$

also called the Born term of elastic constants in the stress-stress fluctuation formalism. This quantity can be used to compute the elastic constant tensor. Using the symmetric Voigt notation, the elastic constant tensor can be written as a 6x6 symmetric matrix:

$$C_{i,j} = \langle C^B_{i,j} \rangle + \frac{V}{k_B T} \left( \langle \sigma_i \sigma_j \rangle \ - \langle \sigma_i \rangle \langle \sigma_j \rangle \right) + \frac{N k_B T}{V} \left( \delta_{i,j} + (\delta_{1,i} + \delta_{2,i} + \delta_{3,i}) \ * (\delta_{1,j} + \delta_{2,j} + \delta_{3,j}) \right)$$

In the above expression, $\sigma$ stands for the virial stress tensor, $\delta$ is the Kronecker delta and the usual notation apply for the number of particle, the temperature and volume respectively $N$, $T$ and $V$. $k_B$ is the Boltzmann constant. For a more detailed explanation of the terms appearing in the above equation, see *(Lutsko)*.

The Born term is a symmetric 6x6 matrix, as is the matrix of second derivatives of potential energy w.r.t strain, whose 21 independent elements are output in this order:

$$\begin{bmatrix} C_1 & C_7 & C_8 & C_9 & C_{10} & C_{11} \\ C_7 & C_2 & C_{12} & C_{13} & C_{14} & C_{15} \\ \vdots & C_{12} & C_3 & C_{16} & C_{17} & C_{18} \\ \vdots & C_{13} & C_{16} & C_4 & C_{19} & C_{20} \\ \vdots & \vdots & \vdots & C_{19} & C_5 & C_{21} \\ \vdots & \vdots & \vdots & \vdots & C_{21} & C_6 \end{bmatrix}$$

in this matrix the indices of $C_k$ value are the corresponding element $k$ in the global vector output by this compute. Each term comes from the sum of the derivatives of every contribution to the potential energy in the system as explained in *(VanWorkum)*.

The output can be accessed using the usual LAMMPS routines:

```
compute 1 all born/matrix
compute 2 all pressure NULL virial
variable S1 equal -c_2[1]
variable S2 equal -c_2[2]
variable S3 equal -c_2[3]
variable S4 equal -c_2[6]
variable S5 equal -c_2[5]
variable S6 equal -c_2[4]
fix 1 all ave/time 1 1 1 v_S1 v_S2 v_S3 v_S4 v_S5 v_S6 c_1[*] file born.out
```

Note interchange of 4th and 6th stress values to satisfy Voigt notation. In this example, the file *born.out* will contain the information needed to compute the first and second terms of the elastic constant matrix in a post processing procedure. The third term depends only on $T$, which can be specified directly or estimated by sampling the atom velocities.

Several examples of this method are provided in the examples/ELASTIC_T/BORN_MATRIX directory described on the *Examples* doc page.

NOTE: In the above $C_{i,j}$ computation, the fluctuation term involving the virial stress tensor $\sigma$ is the covariance between each elements. In a solid the stress fluctuations can vary rapidly, while average fluctuations can be slow to converge. A detailed analysis of the convergence rate of all the terms in the elastic tensor is provided in the paper by Clavier et al. *(Clavier)*.

Two different computation methods for the Born matrix are implemented in this compute and are mutually exclusive.

The first one is a direct computation from the analytical formula from the different terms of the potential used for the simulations *(VanWorkum)*. However, the implementation of such derivations must be done for every potential form. This has not been done yet and can be very complicated for complex potentials. At the moment a warning message is displayed for every term that is not supporting the compute at the moment. This method is the default for now.

The second method uses finite differences of energy to numerically approximate the second derivatives *(Zhen)*. This is useful when using interaction styles for which the analytical second derivatives have not been implemented. In this cases, the compute applies linear strain fields of magnitude *delta* to all the atoms relative to a point at the center of the box. The strain fields are in six different directions, corresponding to the six Cartesian components of the stress tensor defined by LAMMPS. For each direction it applies the strain field in both the positive and negative senses, and the new stress virial tensor of the entire system is calculated after each. The difference in these two virials divided by two times *delta*, approximates the corresponding components of the second derivative, after applying a suitable unit conversion.

**Note**

It is important to choose a suitable value for delta, the magnitude of strains that are used to generate finite difference approximations to the exact virial stress. For typical systems, a value in the range of 1 part in 1e5 to 1e6 will be sufficient. However, the best value will depend on a multitude of factors including the stiffness of the interatomic potential, the thermodynamic state of the material being probed, and so on. The only way to be sure that you have made a good choice is to do a sensitivity study on a representative atomic configuration, sweeping over a wide range of values of delta. If delta is too small, the output values will vary erratically due to truncation effects. If delta is increased beyond a certain point, the output values will start to vary smoothly with delta, due to growing contributions from higher order derivatives. In between these two limits, the numerical virial values should be largely independent of delta.

The keyword requires the additional arguments *delta* and *virial-ID*. *delta* gives the size of the applied strains. *virial-ID* gives the ID string of the pressure compute that provides the virial stress tensor, requiring that it use the virial keyword e.g.

```
compute myvirial all pressure NULL virial
compute 1 all born/matrix numdiff 1.0e-4 myvirial
```

**Output info:**

This compute calculates a global vector with 21 values that are the second derivatives of the potential energy with respect to strain. The values are in energy units. The values are ordered as explained above. These values can be used by any command that uses global values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

The array values calculated by this compute are all “extensive”.

### 3.11.4 Restrictions

This compute is part of the EXTRA-COMPUTE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info. LAMMPS was built with that package. See the *Build package* page for more info.

The Born term can be decomposed as a product of two terms. The first one is a general term which depends on the configuration. The second one is specific to every interaction composing your force field (non-bonded, bonds, angle, ...). Currently not all LAMMPS interaction styles implement the *born_matrix* method giving first and second order derivatives and LAMMPS will exit with an error if this compute is used with such interactions unless the *numdiff* option is also used. The *numdiff* option cannot be used with any other keyword. In this situation, LAMMPS will also exit with an error.

### 3.11.5 Default

none

---

**(Van Workum)** K. Van Workum et al., J Chem Phys, 125, 144506 (2006).

**(Clavier)** G. Clavier, N. Desbiens, E. Bourasseau, V. Lachet, N. Brusselle-Dupend and B. Rousseau, Mol Sim, 43, 1413 (2017).

**(Zhen)** Y. Zhen, C. Chu, Comp Phys Comm, 183, 261-265 (2012).

**(Lutsko)** J. F. Lutsko, J Appl Phys, 65, 2991-2997 (1989).

## 3.12 compute centro/atom command

Accelerator Variants: *centro/atom/omp*

### 3.12.1 Syntax

```
compute ID group-ID centro/atom lattice keyword value ...
```

- ID, group-ID are documented in *compute* command
- centro/atom = style name of this compute command
- lattice = *fcc* or *bcc* or N = # of neighbors per atom to include
- zero or more keyword/value pairs may be appended
- keyword = *axes*

```
axes value = no or yes
  no = do not calculate 3 symmetry axes
  yes = calculate 3 symmetry axes
```

### 3.12.2 Examples

```
compute 1 all centro/atom fcc

compute 1 all centro/atom 8
```

### 3.12.3 Description

Define a computation that calculates the centro-symmetry parameter for each atom in the group, for either FCC or BCC lattices, depending on the choice of the *lattice* argument. In solid-state systems the centro-symmetry parameter is a useful measure of the local lattice disorder around an atom and can be used to characterize whether the atom is part of a perfect lattice, a local defect (e.g. a dislocation or stacking fault), or at a surface.

The value of the centro-symmetry parameter will be 0.0 for atoms not in the specified compute group.

This parameter is computed using the following formula from *(Kelchner)*

$$CS = \sum_{i=1}^{N/2} |\vec{R}_i + \vec{R}_{i+N/2}|^2$$

where the $N$ nearest neighbors of each atom are identified and $\vec{R}_i$ and $\vec{R}_{i+N/2}$ are vectors from the central atom to a particular pair of nearest neighbors. There are $N(N-1)/2$ possible neighbor pairs that can contribute to this formula. The quantity in the sum is computed for each, and the $N/2$ smallest are used. This will typically be for pairs of atoms in symmetrically opposite positions with respect to the central atom; hence the $i + N/2$ notation.

$N$ is an input parameter, which should be set to correspond to the number of nearest neighbors in the underlying lattice of atoms. If the keyword *fcc* or *bcc* is used, $N$ is set to 12 and 8 respectively. More generally, $N$ can be set to a positive, even integer.

For an atom on a lattice site, surrounded by atoms on a perfect lattice, the centro-symmetry parameter will be 0. It will be near 0 for small thermal perturbations of a perfect lattice. If a point defect exists, the symmetry is broken, and the parameter will be a larger positive value. An atom at a surface will have a large positive parameter. If the atom does not have $N$ neighbors (within the potential cutoff), then its centro-symmetry parameter is set to 0.0.

If the keyword *axes* has the setting *yes*, then this compute also estimates three symmetry axes for each atom's local neighborhood. The first two of these are the vectors joining the two pairs of neighbor atoms with smallest contributions to the centrosymmetry parameter, i.e. the two most symmetric pairs of atoms. The third vector is normal to the first two by the right-hand rule. All three vectors are normalized to unit length. For FCC crystals, the first two vectors will lie along a $\langle 110 \rangle$ direction, while the third vector will lie along either a $\langle 100 \rangle$ or $\langle 111 \rangle$ direction. For HCP crystals, the first two vectors will lie along $\langle 1000 \rangle$ directions, while the third vector will lie along $\langle 0001 \rangle$. This provides a simple way to measure local orientation in HCP structures. In general, the *axes* keyword can be used to estimate the orientation of symmetry axes in the neighborhood of any atom.

Only atoms within the cutoff of the pairwise neighbor list are considered as possible neighbors. Atoms not in the compute group are included in the $N$ neighbors used in this calculation.

The neighbor list needed to compute this quantity is constructed each time the calculation is performed (e.g., each time a snapshot of atoms is dumped). Thus it can be inefficient to compute/dump this quantity too frequently or to have multiple compute/dump commands, each with a *centro/atom* style.

### 3.12.4 Output info

By default, this compute calculates the centrosymmetry value for each atom as a per-atom vector, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

If the *axes* keyword setting is *yes*, then a per-atom array is calculated. The first column is the centrosymmetry parameter. The next three columns are the $x$, $y$, and $z$ components of the first symmetry axis, followed by the second, and third symmetry axes in columns 5–7 and 8–10.

The centrosymmetry values are unitless values $\geq 0.0$. Their magnitude depends on the lattice style due to the number of contributing neighbor pairs in the summation in the formula above. And it depends on the local defects surrounding the central atom, as described above. For the *axes yes* case, the vector components are also unitless, since they represent spatial directions.

Here are typical centro-symmetry values, from a nanoindentation simulation into gold (FCC). These were provided by Jon Zimmerman (Sandia):

```
Bulk lattice = 0
Dislocation core ~ 1.0 (0.5 to 1.25)
Stacking faults ~ 5.0 (4.0 to 6.0)
Free surface ~ 23.0
```

These values are **not** normalized by the square of the lattice parameter. If they were, normalized values would be:

```
Bulk lattice = 0
Dislocation core ~ 0.06 (0.03 to 0.075)
Stacking faults ~ 0.3 (0.24 to 0.36)
Free surface ~ 1.38
```

For BCC materials, the values for dislocation cores and free surfaces would be somewhat different, due to their being only 8 neighbors instead of 12.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 3.12.5 Restrictions

none

### 3.12.6 Related commands

*compute cna/atom*

### 3.12.7 Default

The default value for the optional keyword is axes = no.

---

**(Kelchner)** Kelchner, Plimpton, Hamilton, Phys Rev B, 58, 11085 (1998).

## 3.13 compute chunk/atom command

### 3.13.1 Syntax

```
compute ID group-ID chunk/atom style args keyword values ...
```

- ID, group-ID are documented in *compute* command

- chunk/atom = style name of this compute command

```
style = bin/1d or bin/2d or bin/3d or bin/sphere or bin/cylinder or type or␣
↪molecule or c_ID, c_ID[I], f_ID, f_ID[I], v_name
  bin/1d args = dim origin delta
    dim = x or y or z
    origin = lower or center or upper or coordinate value (distance units)
    delta = thickness of spatial bins in dim (distance units)
  bin/2d args = dim origin delta dim origin delta
    dim = x or y or z
    origin = lower or center or upper or coordinate value (distance units)
    delta = thickness of spatial bins in dim (distance units)
  bin/3d args = dim origin delta dim origin delta dim origin delta
    dim = x or y or z
    origin = lower or center or upper or coordinate value (distance units)
    delta = thickness of spatial bins in dim (distance units)
  bin/sphere args = xorig yorig zorig rmin rmax nsbin
    xorig,yorig,zorig = center point of sphere
    srmin,srmax = bin from sphere radius rmin to rmax
    nsbin = # of spherical shell bins between rmin and rmax
  bin/cylinder args = dim origin delta c1 c2 rmin rmax ncbin
    dim = x or y or z = axis of cylinder axis
    origin = lower or center or upper or coordinate value (distance units)
    delta = thickness of spatial bins in dim (distance units)
    c1,c2 = coords of cylinder axis in other 2 dimensions (distance units)
    crmin,crmax = bin from cylinder radius rmin to rmax (distance units)
    ncbin = # of concentric circle bins between rmin and rmax
  type args = none
  molecule args = none
  c_ID, c_ID[I], f_ID, f_ID[I], v_name args = none
    c_ID = per-atom vector calculated by a compute with ID
    c_ID[I] = Ith column of per-atom array calculated by a compute with ID
    f_ID = per-atom vector calculated by a fix with ID
    f_ID[I] = Ith column of per-atom array calculated by a fix with ID
    v_name = per-atom vector calculated by an atom-style variable with name
```

- zero or more keyword/values pairs may be appended
- keyword = *region* or *nchunk* or *limit* or *ids* or *compress* or *discard* or *bound* or *pbc* or *units*

```
region value = region-ID
  region-ID = ID of region atoms must be in to be part of a chunk
nchunk value = once or every
  once = only compute the number of chunks once
  every = re-compute the number of chunks whenever invoked
limit values = 0 or Nc max or Nc exact
  0 = no limit on the number of chunks
  Nc max = limit number of chunks to be <= Nc
  Nc exact = set number of chunks to exactly Nc
ids value = once or nfreq or every
  once = assign chunk IDs to atoms only once, they persist thereafter
  nfreq = assign chunk IDs to atoms only once every Nfreq steps (if invoked by fix␣
↪ave/chunk which sets Nfreq)
  every = assign chunk IDs to atoms whenever invoked
compress value = yes or no
  yes = compress chunk IDs to eliminate IDs with no atoms
```

```
    no = do not compress chunk IDs even if some IDs have no atoms
  discard value = yes or no or mixed
    yes = discard atoms with out-of-range chunk IDs by assigning a chunk ID = 0
    no = keep atoms with out-of-range chunk IDs by assigning a valid chunk ID
    mixed = keep or discard such atoms according to spatial binning rule
  bound values = x/y/z lo hi
    x/y/z = x or y or z to bound spatial bins in this dimension
    lo = lower or coordinate value (distance units)
    hi = upper or coordinate value (distance units)
  pbc value = no or yes
    yes = use periodic distance for bin/sphere and bin/cylinder styles
  units value = box or lattice or reduced
```

### 3.13.2 Examples

```
compute 1 all chunk/atom type
compute 1 all chunk/atom bin/1d z lower 0.02 units reduced
compute 1 all chunk/atom bin/2d z lower 1.0 y 0.0 2.5
compute 1 all chunk/atom molecule region sphere nchunk once ids once compress yes
compute 1 all chunk/atom bin/sphere 5 5 5 2.0 5.0 5 discard yes
compute 1 all chunk/atom bin/cylinder z lower 2 10 10 2.0 5.0 3 discard yes
compute 1 all chunk/atom c_cluster
```

### 3.13.3 Description

Define a computation that calculates an integer chunk ID from 1 to Nchunk for each atom in the group. Values of chunk IDs are determined by the *style* of chunk, which can be based on atom type or molecule ID or spatial binning or a per-atom property or value calculated by another *compute*, *fix*, or *atom-style variable*. Per-atom chunk IDs can be used by other computes with "chunk" in their style name, such as *compute com/chunk* or *compute msd/chunk*. Or they can be used by the *fix ave/chunk* command to sum and time average a variety of per-atom properties over the atoms in each chunk. Or they can simply be accessed by any command that uses per-atom values from a compute as input, as discussed on the *Howto output* doc page.

See the *Howto chunk* page for an overview of how this compute can be used with a variety of other commands to tabulate properties of a simulation. The page gives several examples of input script commands that can be used to calculate interesting properties.

Conceptually it is important to realize that this compute does two simple things. First, it sets the value of *Nchunk* = the number of chunks, which can be a constant value or change over time. Second, it assigns each atom to a chunk via a chunk ID. Chunk IDs range from 1 to *Nchunk* inclusive; some chunks may have no atoms assigned to them. Atoms that do not belong to any chunk are assigned a value of 0. Note that the two operations are not always performed together. For example, spatial bins can be setup once (which sets *Nchunk*), and atoms assigned to those bins many times thereafter (setting their chunk IDs).

All other commands in LAMMPS that use chunk IDs assume there are *Nchunk* number of chunks, and that every atom is assigned to one of those chunks, or not assigned to any chunk.

There are many options for specifying for how and when *Nchunk* is calculated, and how and when chunk IDs are assigned to atoms. The details depend on the chunk *style* and its *args*, as well as optional keyword settings. They can also depend on whether a *fix ave/chunk* command is using this compute, since that command requires *Nchunk* to remain static across windows of timesteps it specifies, while it accumulates per-chunk averages.

The details are described below.

---

The different chunk styles operate as follows. For each style, how it calculates *Nchunk* and assigns chunk IDs to atoms

is explained. Note that using the optional keywords can change both of those actions, as described further below where the keywords are discussed.

---

The *binning* styles perform a spatial binning of atoms, and assign an atom the chunk ID corresponding to the bin number it is in. *Nchunk* is set to the number of bins, which can change if the simulation box size changes. This also depends on the setting of the *units* keyword (e.g., for *reduced* units the number of chunks may not change even if the box size does).

The *bin/1d*, *bin/2d*, and *bin/3d* styles define bins as 1d layers (slabs), 2d pencils, or 3d boxes. The *dim*, *origin*, and *delta* settings are specified 1, 2, or 3 times. For 2d or 3d bins, there is no restriction on specifying dim = $x$ before dim = $y$ or $z$, or dim = $y$ before dim = $z$. Bins in a particular *dim* have a bin size in that dimension given by *delta*. In each dimension, bins are defined relative to a specified *origin*, which may be the lower/upper edge of the simulation box (in that dimension), or its center point, or a specified coordinate value. Starting at the origin, sufficient bins are created in both directions to completely span the simulation box or the bounds specified by the optional *bounds* keyword.

For orthogonal simulation boxes, the bins are layers, pencils, or boxes aligned with the xyz coordinate axes. For triclinic (non-orthogonal) simulation boxes, the bin faces are parallel to the tilted faces of the simulation box. See the *Howto triclinic* page for a discussion of the geometry of triclinic boxes in LAMMPS. As described there, a tilted simulation box has edge vectors $\vec{a}$, $\vec{b}$, and $\vec{c}$. In that nomenclature, bins in the $x$ dimension have faces with normals in the $\vec{b} \times \vec{c}$ direction, bins in $y$ have faces normal to the $\vec{a} \times \vec{c}$ direction, and bins in $z$ have faces normal to the $\vec{a} \times \vec{b}$ direction. Note that in order to define the size and position of these bins in an unambiguous fashion, the *units* option must be set to *reduced* when using a triclinic simulation box, as noted below.

The meaning of *origin* and *delta* for triclinic boxes is as follows. Consider a triclinic box with bins that are 1d layers or slabs in the x dimension. No matter how the box is tilted, an *origin* of 0.0 means start layers at the lower $\vec{b} \times \vec{c}$ plane of the simulation box and an *origin* of 1.0 means to start layers at the upper $\vec{b} \times \vec{c}$ face of the box. A *delta* value of 0.1 in *reduced* units means there will be 10 layers from 0.0 to 1.0, regardless of the current size or shape of the simulation box.

The *bin/sphere* style defines a set of spherical shell bins around the origin (*xorig*,*yorig*,*zorig*), using *nsbin* bins with radii equally spaced between *srmin* and *srmax*. This is effectively a 1d vector of bins. For example, if *srmin* = 1.0 and *srmax* = 10.0 and *nsbin* = 9, then the first bin spans $1.0 < r < 2.0$, and the last bin spans $9.0 < r < 10.0$. The geometry of the bins is the same whether the simulation box is orthogonal or triclinic (i.e., the spherical shells are not tilted or scaled differently in different dimensions to transform them into ellipsoidal shells).

The *bin/cylinder* style defines bins for a cylinder oriented along the axis *dim* with the axis coordinates in the other two radial dimensions at (*c1*,*c2*). For dim = $x$, $c_1/c_2 = y/z$; for dim = $y$, $c_1/c_2 = x/z$; for dim = $z$, $c_1/c_2 = x/y$. This is effectively a 2d array of bins. The first dimension is along the cylinder axis, the second dimension is radially outward from the cylinder axis. The bin size and positions along the cylinder axis are specified by the *origin* and *delta* values, the same as for the *bin/1d*, *bin/2d*, and *bin/3d* styles. There are *ncbin* concentric circle bins in the radial direction from the cylinder axis with radii equally spaced between *crmin* and *crmax*. For example, if *crmin* = 1.0 and *crmax* = 10.0 and *ncbin* = 9, then the first bin spans $1.0 < r < 2.0$ and the last bin spans $9.0 < r < 10.0$. The geometry of the bins in the radial dimensions is the same whether the simulation box is orthogonal or triclinic (i.e., the concentric circles are not tilted or scaled differently in the two different dimensions to transform them into ellipses).

The created bins (and hence the chunk IDs) are numbered consecutively from 1 to the number of bins = *Nchunk*. For *bin2d* and *bin3d*, the numbering varies fastest in the last dimension (which could be $x$, $y$, or $z$), slower in the second dimension, and slowest in the first dimension. For *bin/sphere*, the bin with smallest radius is chunk 1 and the bin with largest radius is chunk Nchunk = *ncbin*. For *bin/cylinder*, the numbering varies faster in the dimension along the cylinder axis and slower in the radial direction. In all cases, for a given dimension, the numbering increases with increasing value of the coordinate (Cartesian coordinate, sphere or cylinder radius, axial position).

Each time this compute is invoked, each atom is mapped to a bin based on its current position. Note that between reneighboring timesteps, atoms can move outside the current simulation box. If the box is periodic (in that dimension) the atom is remapping into the periodic box for purposes of binning. If the box in not periodic, the atom may have moved outside the bounds of all bins. If an atom is not inside any bin, the *discard* keyword is used to determine how a chunk ID is assigned to the atom.

---

The *type* style uses the atom type as the chunk ID. *Nchunk* is set to the number of atom types defined for the simulation (e.g., via the *create_box* or *read_data* commands).

---

The *molecule* style uses the molecule ID of each atom as its chunk ID. *Nchunk* is set to the largest chunk ID. Note that this excludes molecule IDs for atoms which are not in the specified group or optional region.

There is no requirement that all atoms in a particular molecule are assigned the same chunk ID (zero or non-zero), though you probably want that to be the case, if you wish to compute a per-molecule property. LAMMPS will issue a warning if that is not the case, but only the first time that *Nchunk* is calculated.

Note that atoms with a molecule ID = 0, which may be non-molecular solvent atoms, have an out-of-range chunk ID. These atoms are discarded (not assigned to any chunk) or assigned to *Nchunk*, depending on the value of the *discard* keyword.

---

The *compute/fix/variable* styles set the chunk ID of each atom based on a quantity calculated and stored by a compute, fix, or variable. In each case, it must be a per-atom quantity. In each case the referenced floating point values are converted to an integer chunk ID as follows. The floating point value is truncated (rounded down) to an integer value. If the integer value is $\leq 0$, then a chunk ID of 0 is assigned to the atom. If the integer value is $> 0$, it becomes the chunk ID to the atom. *Nchunk* is set to the largest chunk ID. Note that this excludes atoms which are not in the specified group or optional region.

If the style begins with "c_", a compute ID must follow which has been previously defined in the input script. If no bracketed integer is appended, the per-atom vector calculated by the compute is used. If a bracketed integer is appended, the Ith column of the per-atom array calculated by the compute is used. Users can also write code for their own compute styles and *add them to LAMMPS*.

If the style begins with "f_", a fix ID must follow which has been previously defined in the input script. If no bracketed integer is appended, the per-atom vector calculated by the fix is used. If a bracketed integer is appended, the Ith column of the per-atom array calculated by the fix is used. Note that some fixes only produce their values on certain timesteps, which must be compatible with the timestep on which this compute accesses the fix, else an error results. Users can also write code for their own fix styles and *add them to LAMMPS*.

If a value begins with "v_", a variable name for an *atom* or *atomfile* style *variable* must follow which has been previously defined in the input script. Variables of style *atom* can reference thermodynamic keywords and various per-atom attributes, or invoke other computes, fixes, or variables when they are evaluated, so this is a very general means of generating per-atom quantities to treat as a chunk ID.

---

Normally, *Nchunk* = the number of chunks, is re-calculated every time this fix is invoked, though the value may or may not change. As explained below, the *nchunk* keyword can be set to *once* which means *Nchunk* will never change.

If a *fix ave/chunk* command uses this compute, it can also turn off the re-calculation of *Nchunk* for one or more windows of timesteps. The extent of the windows, during which Nchunk is held constant, are determined by the *Nevery*, *Nrepeat*, *Nfreq* values and the *ave* keyword setting that are used by the *fix ave/chunk* command.

Specifically, if *ave* = *one*, then for each span of *Nfreq* timesteps, *Nchunk* is held constant between the first timestep when averaging is done (within the Nfreq-length window), and the last timestep when averaging is done (multiple of Nfreq). If *ave* = *running* or *window*, then *Nchunk* is held constant forever, starting on the first timestep when the *fix ave/chunk* command invokes this compute.

Note that multiple *fix ave/chunk* commands can use the same compute chunk/atom compute. However, the time windows they induce for holding *Nchunk* constant must be identical, else an error will be generated.

---

The various optional keywords operate as follows. Note that some of them function differently or are ignored by different chunk styles. Some of them also have different default values, depending on the chunk style, as listed below.

The *region* keyword applies to all chunk styles. If used, an atom must be in both the specified group and the specified geometric *region* to be assigned to a chunk.

---

The *nchunk* keyword applies to all chunk styles. It specifies how often *Nchunk* is recalculated, which in turn can affect the chunk IDs assigned to individual atoms.

If *nchunk* is set to *once*, then *Nchunk* is only calculated once, the first time this compute is invoked. If *nchunk* is set to *every*, then *Nchunk* is re-calculated every time the compute is invoked. Note that, as described above, the use of this compute by the *fix ave/chunk* command can override the *every* setting.

The default values for *nchunk* are listed below and depend on the chunk style and other system and keyword settings. They attempt to represent typical use cases for the various chunk styles. The *nchunk* value can always be set explicitly if desired.

---

The *limit* keyword can be used to limit the calculated value of *Nchunk* = the number of chunks. The limit is applied each time *Nchunk* is calculated, which also limits the chunk IDs assigned to any atom. The *limit* keyword is used by all chunk styles except the *binning* styles, which ignore it. This is because the number of bins can be tailored using the *bound* keyword (described below) which effectively limits the size of *Nchunk*.

If *limit* is set to *Nc* = 0, then no limit is imposed on *Nchunk*, though the *compress* keyword can still be used to reduce *Nchunk*, as described below.

If $Nc > 0$, then the effect of the *limit* keyword depends on whether the *compress* keyword is also used with a setting of *yes*, and whether the *compress* keyword is specified before the *limit* keyword or after.

In all cases, *Nchunk* is first calculated in the usual way for each chunk style, as described above.

First, here is what occurs if *compress yes* is not set. If *limit* is set to *Nc max*, then *Nchunk* is reset to the smaller of *Nchunk* and *Nc*. If *limit* is set to *Nc exact*, then *Nchunk* is reset to *Nc*, whether the original *Nchunk* was larger or smaller than *Nc*. If *Nchunk* shrank due to the *limit* setting, then atom chunk IDs $>$ *Nchunk* will be reset to 0 or *Nchunk*, depending on the setting of the *discard* keyword. If *Nchunk* grew, there will simply be some chunks with no atoms assigned to them.

If *compress yes* is set, and the *compress* keyword comes before the *limit* keyword, the compression operation is performed first, as described below, which resets *Nchunk*. The *limit* keyword is then applied to the new *Nchunk* value, exactly as described in the preceding paragraph. Note that in this case, all atoms will end up with chunk IDs $\leq$ *Nc*, but their original values (e.g., molecule ID or compute/fix/variable) may have been $>$ *Nc*, because of the compression operation.

If *compress yes* is set, and the *compress* keyword comes after the *limit* keyword, then the *limit* value of *Nc* is applied first to the uncompressed value of *Nchunk*, but only if $Nc < Nchunk$ (whether *Nc max* or *Nc exact* is used). This effectively means all atoms with chunk IDs $>$ *Nc* have their chunk IDs reset to 0 or *Nc*, depending on the setting of the *discard* keyword. The compression operation is then performed, which may shrink *Nchunk* further. If the new $Nchunk < Nc$ and *limit* = *Nc exact* is specified, then *Nchunk* is reset to *Nc*, which results in extra chunks with no atoms assigned to them. Note that in this case, all atoms will end up with chunk IDs $\leq$ *Nc*, and their original values (e.g., molecule ID or compute/fix/variable value) will also have been $\leq$ *Nc*.

---

The *ids* keyword applies to all chunk styles. If the setting is *once* then the chunk IDs assigned to atoms the first time this compute is invoked will be permanent, and never be re-computed.

If the setting is *nfreq* and if a *fix ave/chunk* command is using this compute, then in each of the *Nchunk* = constant time windows (discussed above), the chunk ID's assigned to atoms on the first step of the time window will persist until the end of the time window.

If the setting is *every*, which is the default, then chunk IDs are re-calculated on any timestep this compute is invoked.

**Note**

If you want the persistent chunk-IDs calculated by this compute to be continuous when running from a *restart file*, then you should use the same ID for this compute, as in the original run. This is so that the fix this compute creates to store per-atom quantities will also have the same ID, and thus be initialized correctly with chunk IDs from the restart file.

---

The *compress* keyword applies to all chunk styles and affects how *Nchunk* is calculated, which in turn affects the chunk IDs assigned to each atom. It is useful for converting a "sparse" set of chunk IDs (with many IDs that have no atoms assigned to them), into a "dense" set of IDs, where every chunk has one or more atoms assigned to it.

Two possible use cases are as follows. If a large simulation box is mostly empty space, then the *binning* style may produce many bins with no atoms. If *compress* is set to *yes*, only bins with atoms will be contribute to *Nchunk*. Likewise, the *molecule* or *compute/fix/variable* styles may produce large *Nchunk* values. For example, the *compute cluster/atom* command assigns every atom an atom ID for one of the atoms it is clustered with. For a million-atom system with 5 clusters, there would only be 5 unique chunk IDs, but the largest chunk ID might be 1 million, resulting in *Nchunk* = 1 million. If *compress* is set to *yes*, *Nchunk* will be reset to 5.

If *compress* is set to *no*, which is the default, no compression is done. If it is set to *yes*, all chunk IDs with no atoms are removed from the list of chunk IDs, and the list is sorted. The remaining chunk IDs are renumbered from 1 to *Nchunk* where *Nchunk* is the new length of the list. The chunk IDs assigned to each atom reflect the new renumbering from 1 to *Nchunk*.

The original chunk IDs (before renumbering) can be accessed by the *compute property/chunk* command and its *id* keyword, or by the *fix ave/chunk* command which outputs the original IDs as one of the columns in its global output array. For example, using the "compute cluster/atom" command discussed above, the original 5 unique chunk IDs might be atom IDs (27,4982,58374,857838,1000000). After compression, these will be renumbered to (1,2,3,4,5). The original values (27,...,1000000) can be output to a file by the *fix ave/chunk* command, or by using the *fix ave/time* command in conjunction with the *compute property/chunk* command.

**Note**

The compression operation requires global communication across all processors to share their chunk ID values. It can require large memory on every processor to store them, even after they are compressed, if there are a large number of unique chunk IDs with atoms assigned to them. It uses a STL map to find unique chunk IDs and store them in sorted order. Each time an atom is assigned a compressed chunk ID, it must access the STL map. All of this means that compression can be expensive, both in memory and CPU time. The use of the *limit* keyword in conjunction with the *compress* keyword can affect these costs, depending on which keyword is used first. So use this option with care.

---

The *discard* keyword applies to all chunk styles. It affects what chunk IDs are assigned to atoms that do not match one of the valid chunk IDs from 1 to *Nchunk*. Note that it does not apply to atoms that are not in the specified group or optionally specified region. Those atoms are always assigned a chunk ID = 0.

If the calculated chunk ID for an atom is not within the range 1 to *Nchunk* then it is a "discard" atom. Note that *Nchunk* may have been shrunk by the *limit* keyword. Or the *compress* keyword may have eliminated chunk IDs that were valid before the compression took place, and are now not in the compressed list. Also note that for the *molecule* chunk style, if new molecules are added to the system, their chunk IDs may exceed a previously calculated *Nchunk*. Likewise, evaluation of a compute/fix/variable on a later timestep may return chunk IDs that are invalid for the previously calculated *Nchunk*.

All the chunk styles except the *binning* styles, must use *discard* set to either *yes* or *no*. If *discard* is set to *yes*, which is the default, then every "discard" atom has its chunk ID set to 0. If *discard* is set to *no*, every "discard" atom has its chunk ID set to *Nchunk*. I.e. it becomes part of the last chunk.

The *binning* styles use the *discard* keyword to decide whether to discard atoms outside the spatial domain covered by bins, or to assign them to the bin they are nearest to.

For the *bin/1d*, *bin/2d*, *bin/3d* styles the details are as follows. If *discard* is set to *yes*, an out-of-domain atom will have its chunk ID set to 0. If *discard* is set to *no*, the atom will have its chunk ID set to the first or last bin in that dimension. If *discard* is set to *mixed*, which is the default, it will only have its chunk ID set to the first or last bin if bins extend to the simulation box boundary in that dimension. This is the case if the *bound* keyword settings are *lower* and *upper*, which is the default. If the *bound* keyword settings are numeric values, then the atom will have its chunk ID set to 0 if it is outside the bounds of any bin. Note that in this case, it is possible that the first or last bin extends beyond the numeric *bounds* settings, depending on the specified *origin*. If this is the case, the chunk ID of the atom is only set to 0 if it is outside the first or last bin, not if it is simply outside the numeric *bounds* setting.

For the *bin/sphere* style the details are as follows. If *discard* is set to *yes*, an out-of-domain atom will have its chunk ID set to 0. If *discard* is set to *no* or *mixed*, the atom will have its chunk ID set to the first or last bin, i.e. the innermost or outermost spherical shell. If the distance of the atom from the origin is less than *rmin*, it will be assigned to the first bin. If the distance of the atom from the origin is greater than *rmax*, it will be assigned to the last bin.

For the *bin/cylinder* style the details are as follows. If *discard* is set to *yes*, an out-of-domain atom will have its chunk ID set to 0. If *discard* is set to *no*, the atom will have its chunk ID set to the first or last bin in both the radial and axis dimensions. If *discard* is set to *mixed*, which is the default, the radial dimension is treated the same as for *discard* = no. But for the axis dimension, it will only have its chunk ID set to the first or last bin if bins extend to the simulation box boundary in the axis dimension. This is the case if the *bound* keyword settings are *lower* and *upper*, which is the default. If the *bound* keyword settings are numeric values, then the atom will have its chunk ID set to 0 if it is outside the bounds of any bin. Note that in this case, it is possible that the first or last bin extends beyond the numeric *bounds* settings, depending on the specified *origin*. If this is the case, the chunk ID of the atom is only set to 0 if it is outside the first or last bin, not if it is simply outside the numeric *bounds* setting.

If *discard* is set to *no* or *mixed*, the atom will have its chunk ID set to the first or last bin, i.e. the innermost or outermost spherical shell. If the distance of the atom from the origin is less than *rmin*, it will be assigned to the first bin. If the distance of the atom from the origin is greater than *rmax*, it will be assigned to the last bin.

---

The *bound* keyword only applies to the *bin/1d*, *bin/2d*, *bin/3d* styles and to the axis dimension of the *bin/cylinder* style; otherwise it is ignored. It can be used one or more times to limit the extent of bin coverage in a specified dimension, i.e. to only bin a portion of the box. If the *lo* setting is *lower* or the *hi* setting is *upper*, the bin extent in that direction extends to the box boundary. If a numeric value is used for *lo* and/or *hi*, then the bin extent in the *lo* or *hi* direction extends only to that value, which is assumed to be inside (or at least near) the simulation box boundaries, though LAMMPS does not check for this. Note that using the *bound* keyword typically reduces the total number of bins and thus the number of chunks *Nchunk*.

The *pbc* keyword only applies to the *bin/sphere* and *bin/cylinder* styles. If set to *yes*, the distance an atom is from the sphere origin or cylinder axis is calculated in a minimum image sense with respect to periodic dimensions, when determining which bin the atom is in. I.e. if x is a periodic dimension and the distance between the atom and the sphere center in the x dimension is greater than 0.5 * simulation box length in x, then a box length is subtracted to give a distance < 0.5 * simulation box length. This allosws the sphere or cylinder center to be near a box edge, and atoms on the other side of the periodic box will still be close to the center point/axis. Note that with a setting of *yes*, the outer sphere or cylinder radius must also be <= 0.5 * simulation box length in any periodic dimension except for the cylinder axis dimension, or an error is generated.

The *units* keyword only applies to the *binning* styles; otherwise it is ignored. For the *bin/1d*, *bin/2d*, *bin/3d* styles, it determines the meaning of the distance units used for the bin sizes *delta* and for *origin* and *bounds* values if they are coordinate values. For the *bin/sphere* style it determines the meaning of the distance units used for *xorig*,*yorig*,*zorig* and the radii *srmin* and *srmax*. For the *bin/cylinder* style it determines the meaning of the distance units used for *delta*,*c1*,*c2* and the radii *crmin* and *crmax*.

For orthogonal simulation boxes, any of the 3 options may be used. For non-orthogonal (triclinic) simulation boxes, only the *reduced* option may be used.

A *box* value selects standard distance units as defined by the *units* command (e.g., Å for units = *real* or *metal*). A *lattice* value means the distance units are in lattice spacings. The *lattice* command must have been previously used to define the lattice spacing. A *reduced* value means normalized unitless values between 0 and 1, which represent the lower and upper faces of the simulation box respectively. Thus an *origin* value of 0.5 means the center of the box in any dimension. A *delta* value of 0.1 means 10 bins span the box in that dimension.

Note that for the *bin/sphere* style, the radii *srmin* and *srmax* are scaled by the lattice spacing or reduced value of the *x* dimension.

Note that for the *bin/cylinder* style, the radii *crmin* and *crmax* are scaled by the lattice spacing or reduced value of the first dimension perpendicular to the cylinder axis (e.g., *y* for an *x*-axis cylinder, *x* for a *y*-axis cylinder, and *x* for a *z*-axis cylinder).

---

### 3.13.4 Output info

This compute calculates a per-atom vector (the chunk ID), which can be accessed by any command that uses per-atom values from a compute as input. It also calculates a global scalar (the number of chunks), which can be similarly accessed everywhere outside of a per-atom context. See the *Howto output* page for an overview of LAMMPS output options.

The per-atom vector values are unitless chunk IDs, ranging from 1 to *Nchunk* (inclusive) for atoms assigned to chunks, and 0 for atoms not belonging to a chunk. The scalar contains the value of *Nchunk*.

---

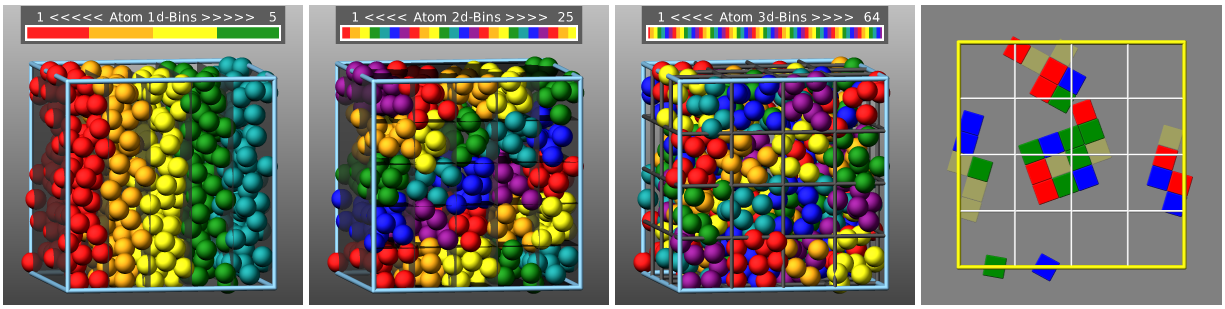
### 3.13.5 Dump image info

Compute *chunk/atom* can be used with the *compute* keyword of *dump image*. The compute will add graphics to the rendered image to visualize the bin boundaries. Currently, only the *bin/1d*, *bin/2d*, and *bin/3d* modes are supported. For *bin/1d* and *bin/2d* in three-dimensional systems, triangles are used to form rectangles representing the bin boundaries. For two-dimensional systems or when using *bin/3d*, cylinders are used to represent the bin boundaries as lines. For *bin/3d*, this results in a wireframe visualization of the 3D grid.

The color of the rectangles or lines is by default that of the first atom type when using color styles "type" or "element". With color style "const" the default value of "white" can be changed using *dump_modify ccolor*. The transparency is by default fully opaque and can be changed with *dump_modify ctrans*.

For 2d systems or 3d systems with mode *bin/3d*, the *fflag1* setting determines whether the cylinders representing the wall are capped with a sphere at the ends: 0 means no caps, 1 means the lower end is capped, 2 means the upper end is capped, and 3 means both ends are capped. The *fflag2* setting allows to adjust the radius of the rendered cylinder. It should be set to a value > 0 or the cylinder will not be visible since the diameter is set internally to zero due to lack of a suitable heuristic for deriving a meaningful diameter for all types of walls and unit settings.

For 3d systems with modes *bin/1d* and *bin/2d* the *cflag1* and *cflag2* settings are ignored.



### 3.13.6 Restrictions

Even if the *nchunk* keyword is set to *once*, the chunk IDs assigned to each atom are not stored in restart files. This means you cannot expect those assignments to persist in a restarted simulation. Instead you must re-specify this command and assign atoms to chunks when the restarted simulation begins.

### 3.13.7 Related commands

*fix ave/chunk*, *compute global/atom*

### 3.13.8 Default

The option defaults are as follows:

- region = none
- nchunk = every, if compress is yes, overriding other defaults listed here
- nchunk = once, for type style
- nchunk = once, for mol style if region is none
- nchunk = every, for mol style if region is set
- nchunk = once, for binning style if the simulation box size is static or units = reduced
- nchunk = every, for binning style if the simulation box size is dynamic and units is lattice or box
- nchunk = every, for compute/fix/variable style
- limit = 0
- ids = every
- compress = no
- discard = yes, for all styles except binning
- discard = mixed, for binning styles
- bound = lower and upper in all dimensions
- pbc = no
- units = lattice

## 3.14 compute chunk/spread/atom command

### 3.14.1 Syntax

```
compute ID group-ID chunk/spread/atom chunkID input1 input2 ...
```

- ID, group-ID are documented in *compute* command
- chunk/spread/atom = style name of this compute command
- chunkID = ID of *compute chunk/atom* command
- one or more inputs can be listed
- input = c_ID, c_ID[N], f_ID, f_ID[N]

```
c_ID = global vector calculated by a compute with ID
c_ID[I] = Ith column of global array calculated by a compute with ID, I can include␣
↪wildcard (see below)
f_ID = global vector calculated by a fix with ID
f_ID[I] = Ith column of global array calculated by a fix with ID, I can include␣
↪wildcard (see below)
```

### 3.14.2 Examples

```
compute 1 all chunk/spread/atom mychunk c_com[*] c_gyration
```

### 3.14.3 Description

Define a calculation that "spreads" one or more per-chunk values to each atom in the chunk. This can be useful in several scenarios:

- For creating a *dump file* where each atom lists info about the chunk it is in, e.g. for post-processing purposes.
- To access chunk value in *atom-style variables* that need info about the chunk each atom is in.
- To use the *fix ave/chunk* command to spatially average per-chunk values calculated by a per-chunk compute.

Examples are given below.

In LAMMPS, chunks are collections of atoms defined by a *compute chunk/atom* command, which assigns each atom to a single chunk (or no chunk). The ID for this command is specified as chunkID. For example, a single chunk could be the atoms in a molecule or atoms in a spatial bin. See the *compute chunk/atom* and *Howto chunk* doc pages for details of how chunks can be defined and examples of how they can be used to measure properties of a system.

For inputs that are computes, they must be a compute that calculates per-chunk values. These are computes whose style names end in "/chunk".

For inputs that are fixes, they should be a fix that calculates per-chunk values. For example, *fix ave/chunk* or *fix ave/time* (assuming it is time-averaging per-chunk data).

For each atom, this compute accesses its chunk ID from the specified *chunkID* compute, then accesses the per-chunk value in each input. Those values are copied to this compute to become the output for that atom.

The values generated by this compute will be 0.0 for atoms not in the specified compute group *group-ID*. They will also be 0.0 if the atom is not in a chunk, as assigned by the *chunkID* compute. They will also be 0.0 if the current chunk ID for the atom is out-of-bounds with respect to the number of chunks stored by a particular input compute or fix.

**Note**

LAMMPS does not check that a compute or fix which calculates per-chunk values uses the same definition of chunks as this compute. It's up to you to be consistent. Likewise, for a fix input, LAMMPS does not check that it is per-chunk data. It only checks that the fix produces a global vector or array.

---

Each listed input is operated on independently.

If a bracketed index I is used, it can be specified using a wildcard asterisk with the index to effectively specify multiple values. This takes the form "*" or "*n" or "n*" or "m*n". If N = the number of columns in the array, then an asterisk with no numeric values means all indices from 1 to N. A leading asterisk means all indices from 1 to n (inclusive). A trailing asterisk means all indices from n to N (inclusive). A middle asterisk means all indices from m to n (inclusive).

Using a wildcard is the same as if the individual columns of the array had been listed one by one. E.g. these 2 compute chunk/spread/atom commands are equivalent, since the *compute com/chunk* command creates a per-atom array with 3 columns:

```
compute com all com/chunk mychunk
compute 10 all chunk/spread/atom mychunk c_com[*]
compute 10 all chunk/spread/atom mychunk c_com[1] c_com[2] c_com[3]
```

---

Here is an example of writing a dump file the with the center-of-mass (COM) for the chunk each atom is in. The commands below can be added to the bench/in.chain script.

```
compute         cmol all chunk/atom molecule
compute         com all com/chunk cmol
compute         comchunk all chunk/spread/atom cmol c_com[*]
dump            1 all custom 50 tmp.dump id mol type x y z c_comchunk[*]
dump_modify     1 sort id
```

The same per-chunk data for each atom could be used to define per-atom forces for the *fix addforce* command. In this example the forces act to pull atoms of an extended polymer chain towards its COM in an attractive manner.

```
compute         prop all property/atom xu yu zu
variable        k equal 0.1
variable        fx atom v_k*(c_comchunk[1]-c_prop[1])
variable        fy atom v_k*(c_comchunk[2]-c_prop[2])
variable        fz atom v_k*(c_comchunk[3]-c_prop[3])
fix             3 all addforce v_fx v_fy v_fz
```

Note that *compute property/atom* is used to generate unwrapped coordinates for use in the per-atom force calculation, so that the effect of periodic boundaries is accounted for properly.

Over time this applied force could shrink each polymer chain's radius of gyration in a polymer mixture simulation. Here is output from the bench/in.chain script. Thermo output is shown for 1000 steps, where the last column is the average radius of gyration over all 320 chains in the 32000 atom system:

```
compute         gyr all gyration/chunk cmol
variable        ave equal ave(c_gyr)
thermo_style    custom step etotal press v_ave

       0    22.394765    4.6721833     5.128278
     100    22.445002    4.8166709    5.0348372
     200    22.500128    4.8790392    4.9364875
     300    22.534686    4.9183766    4.8590693
     400    22.557196    4.9492211    4.7937849
     500    22.571017    4.9161853    4.7412008
     600    22.573944    5.0229708    4.6931243
     700    22.581804    5.0541301    4.6440647
     800    22.584683    4.9691734    4.6000016
     900     22.59128    5.0247538    4.5611513
    1000    22.586832      4.94697    4.5238362
```

---

Here is an example for using one set of chunks, defined for molecules, to compute the dipole moment vector for each chunk. E.g. for water molecules. Then spreading those values to each atom in each chunk. Then defining a second set

of chunks based on spatial bins. And finally, using the *fix ave/chunk* command to calculate an average dipole moment vector per spatial bin.

```
compute       cmol all chunk/atom molecule
compute       dipole all dipole/chunk cmol
compute       spread all chunk/spread/atom cmol c_dipole[1] c_dipole[2] c_dipole[3]
compute       cspatial all chunk/atom bin/1d z lower 0.1 units reduced
fix           ave all ave/chunk 100 10 1000 cspatial c_spread[*]
```

Note that the *fix ave/chunk* command requires per-atom values as input. That is why the compute chunk/spread/atom command is used to assign per-chunk values to each atom in the chunk. If a molecule straddles bin boundaries, each of its atoms contributes in a weighted manner to the average dipole moment of the spatial bin it is in.

---

### 3.14.4 Output info

This compute calculates a per-atom vector or array, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The output is a per-atom vector if a single input value is specified, otherwise a per-atom array is output. The number of columns in the array is the number of inputs provided. The per-atom values for the vector or each column of the array will be in whatever *units* the corresponding input value is in.

The vector or array values are "intensive".

### 3.14.5 Restrictions

none

### 3.14.6 Related commands

*compute chunk/atom*, *fix ave/chunk*, *compute reduce/chunk*

### 3.14.7 Default

none

## 3.15 compute cluster/atom command

## 3.16 compute fragment/atom command

## 3.17 compute aggregate/atom command

### 3.17.1 Syntax

```
compute ID group-ID cluster/atom cutoff
compute ID group-ID fragment/atom keyword value ...
compute ID group-ID aggregate/atom cutoff
```

- ID, group-ID are documented in *compute* command
- *cluster/atom* or *fragment/atom* or *aggregate/atom* = style name of this compute command
- cutoff = distance within which to label atoms as part of same cluster (distance units)

- zero or more keyword/value pairs may be appended to *fragment/atom*
- keyword = *single*

```
single value = yes or no to treat single atoms (no bonds) as fragments
```

### 3.17.2 Examples

```
compute 1 all cluster/atom 3.5
compute 1 all fragment/atom
compute 1 all fragment/atom single no
compute 1 all aggregate/atom 3.5
```

### 3.17.3 Description

Define a computation that assigns each atom a cluster, fragment, or aggregate ID. Only atoms in the compute group are clustered and assigned cluster IDs. Atoms not in the compute group are assigned an ID = 0.

A cluster is defined as a set of atoms, each of which is within the cutoff distance from one or more other atoms in the cluster. If an atom has no neighbors within the cutoff distance, then it is a 1-atom cluster.

A fragment is similarly defined as a set of atoms, each of which has a bond to another atom in the fragment. Bonds can be defined initially via the *data file* or *create_bonds* commands, or dynamically by fixes which create or break bonds like *fix bond/react*, *fix bond/create*, *fix bond/swap*, or *fix bond/break*. The cluster ID or fragment ID of every atom in the cluster will be set to the smallest atom ID of any atom in the cluster or fragment, respectively.

For the *fragment/atom* style, the *single* keyword determines whether single atoms (not bonded to another atom) are treated as one-atom fragments or not, based on the *yes* or *no* setting. If the setting is *no* (the default), their fragment IDs are set to 0.

An aggregate is defined by combining the rules for clusters and fragments (i.e., a set of atoms, where each of them is within the cutoff distance from one or more atoms within a fragment that is part of the same cluster). This measure can be used to track molecular assemblies like micelles.

For computes *cluster/atom* and *aggregate/atom* a neighbor list needed to compute cluster IDs is constructed each time the compute is invoked. Thus it can be inefficient to compute/dump this quantity too frequently or to have multiple *cluster/atom* or *aggregate/atom* style computes.

**Note**

If you have a bonded system, then the settings of *special_bonds* command can remove pairwise interactions between atoms in the same bond, angle, or dihedral. This is the default setting for the *special_bonds* command, and means those pairwise interactions do not appear in the neighbor list. Because this fix uses the neighbor list, it also means those pairs will not be included when computing the clusters. This does not apply when using long-range coulomb (*coul/long*, *coul/msm*, *coul/wolf* or similar. One way to get around this would be to set special_bond scaling factors to very tiny numbers that are not exactly zero (e.g., $1.0 \times 10^{-50}$). Another workaround is to write a dump file and use the *rerun* command to compute the clusters for snapshots in the dump file. The rerun script can use a *special_bonds* command that includes all pairs in the neighbor list.

**Note**

For the compute fragment/atom style, each fragment is identified using the current bond topology. This will not account for bonds broken by the *bond_style quartic* command because it does not perform a full update of the bond topology data structures within LAMMPS.

### 3.17.4 Output info

This compute calculates a per-atom vector, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-atom vector values will be an ID $> 0$, as explained above.

### 3.17.5 Restrictions

none

### 3.17.6 Related commands

*compute coord/atom*

### 3.17.7 Default

The default for fragment/atom is single=no.

## 3.18 compute cna/atom command

Accelerator Variants: *cna/atom/omp*

### 3.18.1 Syntax

```
compute ID group-ID cna/atom cutoff
```

- ID, group-ID are documented in *compute* command
- cna/atom = style name of this compute command
- cutoff = cutoff distance for nearest neighbors (distance units)

### 3.18.2 Examples

```
compute 1 all cna/atom 3.08
```

### 3.18.3 Description

Define a computation that calculates the CNA (Common Neighbor Analysis) pattern for each atom in the group. In solid-state systems the CNA pattern is a useful measure of the local crystal structure around an atom. The CNA methodology is described in *(Faken)* and *(Tsuzuki)*.

Currently, there are five kinds of CNA patterns LAMMPS recognizes:

- fcc = 1
- hcp = 2
- bcc = 3
- icosahedral = 4
- unknown = 5

The value of the CNA pattern will be 0 for atoms not in the specified compute group. Note that normally a CNA calculation should only be performed on mono-component systems.

The CNA calculation can be sensitive to the specified cutoff value. You should ensure the appropriate nearest neighbors of an atom are found within the cutoff distance for the presumed crystal structure (e.g., 12 nearest neighbor for perfect FCC and HCP crystals, 14 nearest neighbors for perfect BCC crystals). These formulas can be used to obtain a good cutoff distance:

$$r_c^{\mathrm{fcc}} = \frac{1}{2}\left(\frac{\sqrt{2}}{2}+1\right)a \approx 0.8536a$$

$$r_c^{\mathrm{bcc}} = \frac{1}{2}(\sqrt{2}+1)a \approx 1.207a$$

$$r_c^{\mathrm{hcp}} = \frac{1}{2}\left(1+\sqrt{\frac{4+2x^2}{3}}\right)a$$

where $a$ is the lattice constant for the crystal structure concerned and in the HCP case, $x = (c/a)/1.633$, where 1.633 is the ideal $c/a$ for HCP crystals.

Also note that since the CNA calculation in LAMMPS uses the neighbors of an owned atom to find the nearest neighbors of a ghost atom, the following relation should also be satisfied:

$$r_c + r_s > 2 * \mathrm{cutoff}$$

where $r_c$ is the cutoff distance of the potential, $r_s$ is the skin distance as specified by the *neighbor* command, and cutoff is the argument used with the compute cna/atom command. LAMMPS will issue a warning if this is not the case.

The neighbor list needed to compute this quantity is constructed each time the calculation is performed (e.g. each time a snapshot of atoms is dumped). Thus it can be inefficient to compute/dump this quantity too frequently or to have multiple compute/dump commands, each with a *cna/atom* style.

### 3.18.4 Output info

This compute calculates a per-atom vector, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-atom vector values will be a number from 0 to 5, as explained above.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 3.18.5 Restrictions

none

### 3.18.6 Related commands

*compute centro/atom*

### 3.18.7 Default

none

---

**(Faken)** Faken, Jonsson, Comput Mater Sci, 2, 279 (1994).

**(Tsuzuki)** Tsuzuki, Branicio, Rino, Comput Phys Comm, 177, 518 (2007).

## 3.19 compute cnp/atom command

Accelerator Variants: *cnp/atom/omp*

### 3.19.1 Syntax

```
compute ID group-ID cnp/atom cutoff
```

- ID, group-ID are documented in *compute* command
- cnp/atom = style name of this compute command
- cutoff = cutoff distance for nearest neighbors (distance units)

### 3.19.2 Examples

```
compute 1 all cnp/atom 3.08
```

### 3.19.3 Description

Define a computation that calculates the Common Neighborhood Parameter (CNP) for each atom in the group. In solid-state systems the CNP is a useful measure of the local crystal structure around an atom and can be used to characterize whether the atom is part of a perfect lattice, a local defect (e.g., a dislocation or stacking fault), or at a surface.

The value of the CNP parameter will be 0.0 for atoms not in the specified compute group. Note that normally a CNP calculation should only be performed on single component systems.

This parameter is computed using the following formula from *(Tsuzuki)*

$$Q_i = \frac{1}{n_i} \sum_{j=1}^{n_i} \left\| \sum_{k=1}^{n_{ij}} \vec{R}_{ik} + \vec{R}_{jk} \right\|^2$$

where the index $j$ goes over the $n_i$ nearest neighbors of atom $i$, and the index $k$ goes over the $n_{ij}$ common nearest neighbors between atom $i$ and atom $j$. $\vec{R}_{ik}$ and $\vec{R}_{jk}$ are the vectors connecting atom $k$ to atoms $i$ and $j$. The quantity in the double sum is computed for each atom.

The CNP calculation is sensitive to the specified cutoff value. You should ensure that the appropriate nearest neighbors of an atom are found within the cutoff distance for the presumed crystal structure. E.g. 12 nearest neighbor for perfect FCC and HCP crystals, 14 nearest neighbors for perfect BCC crystals. These formulas can be used to obtain a good cutoff distance:

$$r_c^{\mathrm{fcc}} = \frac{1}{2}\left(\frac{\sqrt{2}}{2} + 1\right) a \approx 0.8536a$$

$$r_c^{\mathrm{bcc}} = \frac{1}{2}(\sqrt{2} + 1)a \approx 1.207a$$

$$r_c^{\mathrm{hcp}} = \frac{1}{2}\left(1 + \sqrt{\frac{4 + 2x^2}{3}}\right) a$$

where $a$ is the lattice constant for the crystal structure concerned and in the HCP case, $x = (c/a)/1.633$, where 1.633 is the ideal $c/a$ for HCP crystals.

Also note that since the CNP calculation in LAMMPS uses the neighbors of an owned atom to find the nearest neighbors of a ghost atom, the following relation should also be satisfied:

$$r_c + r_s > 2 * \text{cutoff}$$

where $r_c$ is the cutoff distance of the potential, $r_s$ is the skin distance as specified by the *neighbor* command, and cutoff is the argument used with the compute cnp/atom command. LAMMPS will issue a warning if this is not the case.

The neighbor list needed to compute this quantity is constructed each time the calculation is performed (e.g., each time a snapshot of atoms is dumped). Thus it can be inefficient to compute/dump this quantity too frequently or to have multiple compute/dump commands, each with a *cnp/atom* style.

### 3.19.4 Output info

This compute calculates a per-atom vector, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-atom vector values will be real positive numbers. Some typical CNP values:

```
FCC lattice = 0.0
BCC lattice = 0.0
HCP lattice = 4.4

FCC (111) surface = 13.0
FCC (100) surface = 26.5
FCC dislocation core = 11
```

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 3.19.5 Restrictions

This compute is part of the EXTRA-COMPUTE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.19.6 Related commands

*compute cna/atom compute centro/atom*

### 3.19.7 Default

none

---

**(Tsuzuki)** Tsuzuki, Branicio, Rino, Comput Phys Comm, 177, 518 (2007).

## 3.20 compute com command

### 3.20.1 Syntax

```
compute ID group-ID com
```

- ID, group-ID are documented in *compute* command
- com = style name of this compute command

### 3.20.2 Examples

```
compute 1 all com
```

### 3.20.3 Description

Define a computation that calculates the center-of-mass of the group of atoms, including all effects due to atoms passing through periodic boundaries.

A vector of three quantities is calculated by this compute, which are the $(x, y, z)$ coordinates of the center of mass.

> **Note**
>
> The coordinates of an atom contribute to the center-of-mass in "unwrapped" form, by using the image flags associated with each atom. See the *dump custom* command for a discussion of "unwrapped" coordinates. See the Atoms section of the *read_data* command for a discussion of image flags and how they are set for each atom. You can reset the image flags (e.g., to 0) before invoking this compute by using the *set image* command.

### 3.20.4 Output info

This compute calculates a global vector of length 3, which can be accessed by indices 1–3 by any command that uses global vector values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

The vector values are "intensive". The vector values will be in distance *units*.

### 3.20.5 Restrictions

none

### 3.20.6 Related commands

*compute com/chunk*

### 3.20.7 Default

none

## 3.21 compute com/chunk command

### 3.21.1 Syntax

```
compute ID group-ID com/chunk chunkID keyword args ...
```

- ID, group-ID are documented in *compute* command
- com/chunk = style name of this compute command
- chunkID = ID of *compute chunk/atom* command
- zero or more keyword/value pairs may be appended
- keyword = *wrap*

  *wrap* logical = wrap center of mass back to cell if argument is *yes*, *true*, *on* or *1*, otherwise do not (default)

### 3.21.2 Examples

```
compute 1 fluid com/chunk molchunk
compute 1 all com/chunk molchunk wrap on
```

### 3.21.3 Description

Define a computation that calculates the center-of-mass for multiple chunks of atoms.

In LAMMPS, chunks are collections of atoms defined by a *compute chunk/atom* command, which assigns each atom to a single chunk (or no chunk). The ID for this command is specified as chunkID. For example, a single chunk could be the atoms in a molecule or atoms in a spatial bin. See the *compute chunk/atom* and *Howto chunk* doc pages for details of how chunks can be defined and examples of how they can be used to measure properties of a system.

This compute calculates the $(x, y, z)$ coordinates of the center of mass for each chunk, which includes all effects due to atoms passing through periodic boundaries.

Added in version 11Feb2026.

If the *wrap* flag is used with an argument of *yes*, *true*, *on* or *1* the computed center of mass is wrapped back into the simulation cell. With an argument of *no*, *false*, *off* or 0 it is not (the default).

Note that only atoms in the specified group contribute to the calculation. The *compute chunk/atom* command defines its own group; atoms will have a chunk ID = 0 if they are not in that group, signifying they are not assigned to a chunk, and will thus also not contribute to this calculation. You can specify the "all" group for this command if you simply want to include atoms with non-zero chunk IDs.

**Note**

The coordinates of an atom contribute to the chunk's center-of-mass in "unwrapped" form, by using the image flags associated with each atom. See the *dump custom* command for a discussion of "unwrapped" coordinates. See the Atoms section of the *read_data* command for a discussion of image flags and how they are set for each atom. You can reset the image flags (e.g., to 0) before invoking this compute by using the *reset_atoms image* or *set image* commands.

The simplest way to output the results of the compute com/chunk calculation to a file is to use the *fix ave/time* command, for example:

```
compute cc1 all chunk/atom molecule
compute myChunk all com/chunk cc1
fix 1 all ave/time 100 1 100 c_myChunk[*] file tmp.out mode vector
```

### 3.21.4 Output info

This compute calculates a global array where the number of rows = the number of chunks *Nchunk* as calculated by the specified *compute chunk/atom* command. The number of columns is 3 for the $(x, y, z)$ center-of-mass coordinates of each chunk. These values can be accessed by any command that uses global array values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

The array values are "intensive". The array values will be in distance *units*.

### 3.21.5 Restrictions

none

### 3.21.6 Related commands

*compute com*

### 3.21.7 Default

wrap = off

## 3.22 compute composition/atom command

Accelerator Variants: *composition/atom/kk*

### 3.22.1 Syntax

```
compute ID group-ID composition/atom keyword values ...
```

- ID, group-ID are documented in *compute* command
- composition/atom = style name of this compute command
- one or more keyword/value pairs may be appended

  keyword = *cutoff*
    *cutoff* value = distance cutoff

### 3.22.2 Examples

```
compute 1 all composition/atom

compute 1 all composition/atom cutoff 9.0
comm_modify cutoff 9.0
```

### 3.22.3 Description

Added in version 21Nov2023.

Define a computation that calculates a local composition vector for each atom. For a central atom with $M$ neighbors within the neighbor cutoff sphere, composition is defined as the number of atoms of a given type (including the central atom) divided by ($M$ + 1). For a given central atom, the sum of all compositions equals one.

**Note**

This compute uses the number of atom types, not chemical species, assigned in *pair_coeff* command. If an interatomic potential has two species (i.e., Cu and Ni) assigned to four different atom types in *pair_coeff* (i.e., 'Cu Cu Ni Ni'), the compute will output four fractional values. In those cases, the user may desire an extra calculation step to consolidate per-type fractions into per-species fractions. This calculation can be conducted within LAMMPS using another compute such as *compute reduce*, an atom-style *variable command*, or as a post-processing step.

---

The optional keyword *cutoff* defines the distance cutoff used when searching for neighbors. The default value is the cutoff specified by the pair style. If no pair style is defined, then a cutoff must be defined using this keyword. If the specified cutoff is larger than that of the pair_style plus neighbor skin (or no pair style is defined), the *comm_modify cutoff* option must also be set to match that of the *cutoff* keyword.

The neighbor list needed to compute this quantity is constructed each time the calculation is performed (i.e. each time a snapshot of atoms is dumped). Thus it can be inefficient to compute/dump this quantity too frequently.

**Note**

If you have a bonded system, then the settings of *special_bonds* command can remove pairwise interactions between atoms in the same bond, angle, or dihedral. This is the default setting for the *special_bonds* command, and means those pairwise interactions do not appear in the neighbor list. Because this compute uses the neighbor list, it also means those pairs will not be included in the order parameter. This difficulty can be circumvented by writing a dump file, and using the *rerun* command to compute the order parameter for snapshots in the dump file. The rerun script can use a *special_bonds* command that includes all pairs in the neighbor list.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 3.22.4 Output info

This compute calculates a per-atom array with $1 + N$ columns, where $N$ is the number of atom types. The first column is a count of the number of atoms used to calculate composition (including the central atom), and each subsequent column indicates the fraction of that atom type within the cutoff sphere.

These values can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

### 3.22.5 Restrictions

This compute is part of the EXTRA-COMPUTE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This compute requires *neighbor styles 'bin' or 'nsq'*.

### 3.22.6 Related commands

*comm_modify*

### 3.22.7 Default

The option defaults are *cutoff* = pair style cutoff.

## 3.23 compute contact/atom command

### 3.23.1 Syntax

```
compute ID group-ID contact/atom group2-ID
```

- ID, group-ID are documented in *compute* command
- contact/atom = style name of this compute command
- group2-ID = optional argument to restrict which atoms to consider for contacts (see below)

### 3.23.2 Examples

```
compute 1 all contact/atom
compute 1 all contact/atom mygroup
```

### 3.23.3 Description

Define a computation that calculates the number of contacts for each atom in a group.

The contact number is defined for finite-size spherical particles as the number of neighbor atoms which overlap the central particle, meaning that their distance of separation is less than or equal to the sum of the radii of the two particles.

The value of the contact number will be 0.0 for atoms not in the specified compute group.

The optional *group2-ID* argument allows to specify from which group atoms contribute to the coordination number. Default setting is group 'all'.

### 3.23.4 Output info

This compute calculates a per-atom vector, whose values can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-atom vector values will be a number $\geq 0.0$, as explained above.

### 3.23.5 Restrictions

This compute is part of the GRANULAR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This compute requires that atoms store a radius as defined by the *atom_style sphere* command.

### 3.23.6 Related commands

*compute coord/atom*

### 3.23.7 Default

*group2-ID* = all

## 3.24 compute coord/atom command

Accelerator Variants: *coord/atom/kk*, *coord/atom/omp*

### 3.24.1 Syntax

```
compute ID group-ID coord/atom style args ...
```

- ID, group-ID are documented in *compute* command
- coord/atom = style name of this compute command
- style = *cutoff* or *orientorder*

```
cutoff args = cutoff [group group2-ID] typeN
  cutoff = distance within which to count coordination neighbors (distance units)
  group group2-ID = select group-ID to restrict which atoms to consider for␣
↪coordination number (optional)
  typeN = atom type for Nth coordination count (see asterisk form below)
orientorder args = orientorderID threshold
  orientorderID = ID of an orientorder/atom compute
  threshold = minimum value of the product of two "connected" atoms
```

### 3.24.2 Examples

```
compute 1 all coord/atom cutoff 2.0
compute 1 all coord/atom cutoff 6.0 1 2
compute 1 all coord/atom cutoff 6.0 2*4 5*8 *
compute 1 solute coord/atom cutoff 2.0 group solvent
compute 1 all coord/atom orientorder 2 0.5
```

### 3.24.3 Description

This compute performs calculations between neighboring atoms to determine a coordination value. The specific calculation and the meaning of the resulting value depend on the *cstyle* keyword used.

The *cutoff* cstyle calculates one or more traditional coordination numbers for each atom. A coordination number is defined as the number of neighbor atoms with specified atom type(s), and optionally within the specified group, that are within the specified cutoff distance from the central atom. The compute group selects only the central atoms; all neighboring atoms, unless selected by type, type range, or group option, are included in the coordination number tally.

The optional *group* keyword allows to specify from which group atoms contribute to the coordination number. Default setting is group 'all.'

The *typeN* keywords allow specification of which atom types contribute to each coordination number. One coordination number is computed for each of the *typeN* keywords listed. If no *typeN* keywords are listed, a single coordination number is calculated, which includes atoms of all types (same as the "*" format, see below).

The *typeN* keywords can be specified in one of two ways. An explicit numeric value can be used, as in the second example above. Or a wild-card asterisk can be used to specify a range of atom types. This takes the form "*" or "*n" or "m*" or "m*n". If *N* is the number of atom types, then an asterisk with no numeric values means all types from 1 to *N*. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from m to *N* (inclusive). A middle asterisk means all types from m to n (inclusive).

The *orientorder* cstyle calculates the number of "connected" neighbor atoms *j* around each central atom *i*. For this *cstyle*, connected is defined by the orientational order parameter calculated by the *compute orientorder/atom* command. This *cstyle* thus allows one to apply the ten Wolde's criterion to identify crystal-like atoms in a system, as discussed in *ten Wolde*.

The ID of the previously specified *compute orientorder/atom* command is specified as *orientorderID*. The compute must invoke its *components* option to calculate components of the *Ybar_lm* vector for each atoms, as described in its documentation. Note that orientorder/atom compute defines its own criteria for identifying neighboring atoms. If the scalar product (*Ybar_lm(i)*, *Ybar_lm(j)*), calculated by the orientorder/atom compute is larger than the specified *threshold*, then *i* and *j* are connected, and the coordination value of *i* is incremented by one.

For all *cstyle* settings, all coordination values will be 0.0 for atoms not in the specified compute group.

The neighbor list needed to compute this quantity is constructed each time the calculation is performed (i.e., each time a snapshot of atoms is dumped). Thus it can be inefficient to compute/dump this quantity too frequently.

**Note**

If you have a bonded system, then the settings of *special_bonds* command can remove pairwise interactions between atoms in the same bond, angle, or dihedral. This is the default setting for the *special_bonds* command, and means those pairwise interactions do not appear in the neighbor list. Because this fix uses the neighbor list, it also means those pairs will not be included in the coordination count. One way to get around this, is to write a dump file, and use the *rerun* command to compute the coordination for snapshots in the dump file. The rerun script can use a *special_bonds* command that includes all pairs in the neighbor list.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 3.24.4 Output info

For *cstyle* cutoff, this compute can calculate a per-atom vector or array. If single *type1* keyword is specified (or if none are specified), this compute calculates a per-atom vector. If multiple *typeN* keywords are specified, this compute calculates a per-atom array, with *N* columns.

For *cstyle* orientorder, this compute calculates a per-atom vector.

These values can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

The per-atom vector or array values will be a number $\geq 0.0$, as explained above.

### 3.24.5 Restrictions

none

### 3.24.6 Related commands

*compute cluster/atom compute orientorder/atom*

### 3.24.7 Default

group = all

---

**(tenWolde)** P. R. ten Wolde, M. J. Ruiz-Montero, D. Frenkel, J. Chem. Phys. 104, 9932 (1996).

## 3.25 compute count/type command

### 3.25.1 Syntax

```
compute ID group-ID count/type mode
```

- ID, group-ID are documented in *compute* command
- count/type = style name of this compute command
- mode = *atom* or *bond* or *angle* or *dihedral* or *improper*

### 3.25.2 Examples

```
compute 1 all count/type atom
compute 1 flowmols count/type bond
```

### 3.25.3 Description

Added in version 15Jun2023.

Define a computation that counts the current number of atoms for each atom type. Or the number of bonds (angles, dihedrals, impropers) for each bond (angle, dihedral, improper) type.

The former can be useful if atoms are added to or deleted from the system in random ways, e.g. via the *fix deposit*, *fix pour*, or *fix evaporate* commands. The latter can be useful in reactive simulations where molecular bonds are broken or created, as well as angles, dihedrals, impropers.

Note that for this command, bonds (angles, etc) are the topological kind enumerated in a data file, initially read by the *read_data* command or defined by the *molecule* command. They do not refer to implicit bonds defined on-the-fly by bond-order or reactive pair styles based on the current conformation of small clusters of atoms.

These commands can turn off topological bonds (angles, etc) by setting their bond (angle, etc) types to negative values. This command includes the turned-off bonds (angles, etc) in the count for each type:

- *fix shake*
- *delete_bonds*

These commands can create and/or break topological bonds (angles, etc). In the case of breaking, they remove the bond (angle, etc) from the system, so that they no longer exist (*bond_style quartic* and *BPM bond styles* are exceptions, see the discussion below). Thus they are not included in the counts for each type:

- *delete_bonds remove*
- *bond_style quartic*
- *fix bond/react*
- *fix bond/create*
- *fix bond/break*
- *BPM package* bond styles

---

If the *mode* setting is *atom* then the count of atoms for each atom type is tallied. Only atoms in the specified group are counted.

The atom count for each type can be normalized by the total number of atoms like so:

```
compute typevec all count/type atom # number of atoms of each type
variable normtypes vector c_typevec/atoms # divide by total number of atoms
variable ntypes equal extract_setting(ntypes) # number of atom types
thermo_style custom step v_normtypes[*${ntypes}] # vector variable needs upper limit
```

Similarly, bond counts can be normalized by the total number of bonds. The same goes for angles, dihedrals, and impropers (see below).

If the *mode* setting is *bond* then the count of bonds for each bond type is tallied. Only bonds with both atoms in the specified group are counted.

For *mode* = *bond*, broken bonds with a bond type of zero are also counted. The *bond_style quartic* and *BPM bond styles* break bonds by doing this. See the *Howto broken bonds* doc page for more details. Note that the group setting is ignored for broken bonds; all broken bonds in the system are counted.

If the *mode* setting is *angle* then the count of angles for each angle type is tallied. Only angles with all 3 atoms in the specified group are counted.

If the *mode* setting is *dihedral* then the count of dihedrals for each dihedral type is tallied. Only dihedrals with all 4 atoms in the specified group are counted.

If the *mode* setting is *improper* then the count of impropers for each improper type is tallied. Only impropers with all 4 atoms in the specified group are counted.

---

### 3.25.4 Output info

This compute calculates a global vector of counts. If the mode is *atom* or *bond* or *angle* or *dihedral* or *improper*, then the vector length is the number of atom types or bond types or angle types or dihedral types or improper types, respectively.

If the mode is *bond* this compute also calculates a global scalar which is the number of broken bonds with type = 0, as explained above.

These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The scalar and vector values calculated by this compute are both "intensive".

### 3.25.5 Restrictions

none

### 3.25.6 Related commands

none

### 3.25.7 Default

none

## 3.26 compute damage/atom command

### 3.26.1 Syntax

```
compute ID group-ID damage/atom
```

- ID, group-ID are documented in *compute* command
- damage/atom = style name of this compute command

### 3.26.2 Examples

```
compute 1 all damage/atom
```

### 3.26.3 Description

Define a computation that calculates the per-atom damage for each atom in a group. This is a quantity relevant for *Peridynamics models*. See this document for an overview of LAMMPS commands for Peridynamics modeling.

The "damage" of a Peridynamics particles is based on the bond breakage between the particle and its neighbors. If all the bonds are broken the particle is considered to be fully damaged.

See the *Peridynamics Howto* for a formal definition of "damage" and more details about Peridynamics as it is implemented in LAMMPS.

This command can be used with all the Peridynamic pair styles.

The damage value will be 0.0 for atoms not in the specified compute group.

### 3.26.4 Output info

This compute calculates a per-atom vector, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-atom vector values are unitless numbers (damage) $\geq 0.0$.

### 3.26.5 Restrictions

This compute is part of the PERI package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.26.6 Related commands

*compute dilatation/atom*, *compute plasticity/atom*

### 3.26.7 Default

none

## 3.27 compute dihedral command

### 3.27.1 Syntax

```
compute ID group-ID dihedral
```

- ID, group-ID are documented in *compute* command
- dihedral = style name of this compute command

### 3.27.2 Examples

```
compute 1 all dihedral
```

### 3.27.3 Description

Define a computation that extracts the dihedral energy calculated by each of the dihedral sub-styles used in the *dihedral_style hybrid* command. These values are made accessible for output or further processing by other commands. The group specified for this command is ignored.

This compute is useful when using *dihedral_style hybrid* if you want to know the portion of the total energy contributed by one or more of the hybrid sub-styles.

### 3.27.4 Output info

This compute calculates a global vector of length $N$, where $N$ is the number of sub_styles defined by the *dihedral_style hybrid* command, which can be accessed by the indices 1 through $N$. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The vector values are "extensive" and will be in energy *units*.

### 3.27.5 Restrictions

none

### 3.27.6 Related commands

*compute pe*, *compute pair*

### 3.27.7 Default

none

## 3.28 compute dihedral/local command

### 3.28.1 Syntax

```
compute ID group-ID dihedral/local value1 value2 ... keyword args ...
```

- ID, group-ID are documented in *compute* command
- dihedral/local = style name of this compute command
- one or more values may be appended
- value = *phi* or *v_name*

  ```
  phi = tabulate dihedral angles
  v_name = equal-style variable with name (see below)
  ```

- zero or more keyword/args pairs may be appended
- keyword = *set*

  ```
  set args = phi name
    phi = only currently allowed arg
    name = name of variable to set with phi
  ```

### 3.28.2 Examples

```
compute 1 all dihedral/local phi
compute 1 all dihedral/local phi v_cos set phi p
```

### 3.28.3 Description

Define a computation that calculates properties of individual dihedral interactions. The number of datums generated, aggregated across all processors, equals the number of dihedral angles in the system, modified by the group parameter as explained below.

The value *phi* ($\phi$) is the dihedral angle, as defined in the diagram on the *dihedral_style* doc page.

The value *v_name* can be used together with the *set* keyword to compute a user-specified function of the dihedral angle $\phi$. The *name* specified for the *v_name* value is the name of an *equal-style variable* which should evaluate a formula based on a variable which will store the angle $\phi$. This other variable must be an *internal-style variable* specified by the *set* keyword. It is an internal-style variable, because this command resets its value directly. The internal-style variable does not need to be defined in the input script (though it can be); if it is not defined, then the *set* option creates an *internal-style variable* with the specified name.

Note that the value of $\phi$ for each angle which stored in the internal variable is in radians, not degrees.

As an example, these commands can be added to the bench/in.rhodo script to compute the $\cos\phi$ and $\cos^2\phi$ of every dihedral angle in the system and output the statistics in various ways:

```
variable cos equal cos(v_p)
variable cossq equal cos(v_p)*cos(v_p)

compute 1 all property/local datom1 datom2 datom3 datom4 dtype
compute 2 all dihedral/local phi v_cos v_cossq set phi p
dump 1 all local 100 tmp.dump c_1[*] c_2[*]

compute 3 all reduce ave c_2[*] inputs local
thermo_style custom step temp press c_3[*]

fix 10 all ave/histo 10 10 100 -1 1 20 c_2[2] mode vector file tmp.histo
```

The *dump local* command will output the angle ($\phi$), $\cos(\phi)$, and $\cos^2(\phi)$ for every dihedral in the system. The *thermo_style* command will print the average of those quantities via the *compute reduce* command with thermo output. And the *fix ave/histo* command will histogram the cosine(angle) values and write them to a file.

---

The local data stored by this command is generated by looping over all the atoms owned on a processor and their dihedrals. A dihedral will only be included if all four atoms in the dihedral are in the specified compute group.

Note that as atoms migrate from processor to processor, there will be no consistent ordering of the entries within the local vector or array from one timestep to the next. The only consistency that is guaranteed is that the ordering on a particular timestep will be the same for local vectors or arrays generated by other compute commands. For example, dihedral output from the *compute property/local* command can be combined with data from this command and output by the *dump local* command in a consistent way.

Here is an example of how to do this:

```
compute 1 all property/local dtype datom1 datom2 datom3 datom4
compute 2 all dihedral/local phi
dump 1 all local 1000 tmp.dump index c_1[1] c_1[2] c_1[3] c_1[4] c_1[5] c_2[1]
```

### 3.28.4 Output info

This compute calculates a local vector or local array depending on the number of values. The length of the vector or number of rows in the array is the number of dihedrals. If a single value is specified, a local vector is produced. If two or more values are specified, a local array is produced where the number of columns is equal to the number of values. The vector or array can be accessed by any command that uses local values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The output for *phi* will be in degrees.

### 3.28.5 Restrictions

none

### 3.28.6 Related commands

*dump local*, *compute property/local*

### 3.28.7 Default

none

## 3.29 compute dilatation/atom command

### 3.29.1 Syntax

```
compute ID group-ID dilatation/atom
```

- ID, group-ID are documented in compute command
- dilatation/atom = style name of this compute command

### 3.29.2 Examples

```
compute 1 all dilatation/atom
```

### 3.29.3 Description

Define a computation that calculates the per-atom dilatation for each atom in a group. This is a quantity relevant for *Peridynamics models*. See this document for an overview of LAMMPS commands for Peridynamics modeling.

For small deformation, dilatation of is the measure of the volumetric strain.

The dilatation $\theta$ for each peridynamic particle $i$ is calculated as a sum over its neighbors with unbroken bonds, where the contribution of the $ij$ pair is a function of the change in bond length (versus the initial length in the reference state), the volume fraction of the particles and an influence function. See the *Peridynamics Howto* for a formal definition of dilatation.

This command can only be used with a subset of the Peridynamic *pair styles*: *peri/lps*, *peri/ves*, and *peri/eps*.

The dilatation value will be 0.0 for atoms not in the specified compute group.

### 3.29.4 Output info

This compute calculates a per-atom vector, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-atom vector values are unitless numbers ($\theta \geq 0.0$).

### 3.29.5 Restrictions

This compute is part of the PERI package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.29.6 Related commands

*compute damage/atom*, *compute plasticity/atom*

### 3.29.7 Default

none

# 3.30 compute dipole command

# 3.31 compute dipole/tip4p command

## 3.31.1 Syntax

```
compute ID group-ID style arg
```

- ID, group-ID are documented in *compute* command
- style = *dipole* or *dipole/tip4p*
- arg = *mass* or *geometry* = use COM or geometric center for charged chunk correction (optional)

## 3.31.2 Examples

```
compute 1 fluid dipole
compute dw water dipole geometry
compute dw water dipole/tip4p
```

## 3.31.3 Description

Define a computation that calculates the dipole vector and total dipole for a group of atoms.

These computes calculate the x,y,z coordinates of the dipole vector and the total dipole moment for the atoms in the compute group. This includes all effects due to atoms passing through periodic boundaries. For a group with a net charge the resulting dipole is made position independent by subtracting the position vector of the center of mass or geometric center times the net charge from the computed dipole vector. Both per-atom charges and per-atom dipole moments, if present, contribute to the computed dipole.

Added in version 28Mar2023.

Compute *dipole/tip4p* includes adjustments for the charge carrying point M in molecules with TIP4P water geometry. The corresponding parameters are extracted from the pair style.

**Note**

The coordinates of an atom contribute to the dipole in "unwrapped" form, by using the image flags associated with each atom. See the *dump custom* command for a discussion of "unwrapped" coordinates. See the Atoms section of the *read_data* command for a discussion of image flags and how they are set for each atom. You can reset the image flags (e.g., to 0) before invoking this compute by using the *set image* command.

## 3.31.4 Output info

These computes calculate a global scalar containing the magnitude of the computed dipole moment and a global vector of length 3 with the dipole vector. See the *Howto output* page for an overview of LAMMPS output options.

The computed values are "intensive". The array values will be in dipole units (i.e., charge *units* times distance *units*).

## 3.31.5 Restrictions

Compute style *dipole/tip4p* is part of the EXTRA-COMPUTE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Compute style *dipole/tip4p* can only be used with tip4p pair styles.

### 3.31.6 Related commands

*compute dipole/chunk*

### 3.31.7 Default

Using the center of mass is the default setting for the net charge correction.

## 3.32 compute dipole/chunk command

## 3.33 compute dipole/tip4p/chunk command

### 3.33.1 Syntax

```
compute ID group-ID style chunkID arg
```

- ID, group-ID are documented in *compute* command
- style = *dipole/chunk* or *dipole/tip4p/chunk*
- chunkID = ID of *compute chunk/atom* command
- arg = *mass* or *geometry* = use COM or geometric center for charged chunk correction (optional)

### 3.33.2 Examples

```
compute 1 fluid dipole/chunk molchunk
compute dw water dipole/chunk 1 geometry
```

### 3.33.3 Description

Define a computation that calculates the dipole vector and total dipole for multiple chunks of atoms.

In LAMMPS, chunks are collections of atoms defined by a *compute chunk/atom* command, which assigns each atom to a single chunk (or no chunk). The ID for this command is specified as chunkID. For example, a single chunk could be the atoms in a molecule or atoms in a spatial bin. See the *compute chunk/atom* and *Howto chunk* doc pages for details of how chunks can be defined and examples of how they can be used to measure properties of a system.

These computes calculate the $(x, y, z)$ coordinates of the dipole vector and the total dipole moment for each chunk, which includes all effects due to atoms passing through periodic boundaries. For chunks with a net charge the resulting dipole is made position independent by subtracting the position vector of the center of mass or geometric center times the net charge from the computed dipole vector. Both per-atom charges and per-atom dipole moments, if present, contribute to the computed dipole.

Added in version 28Mar2023.

Compute *dipole/tip4p/chunk* includes adjustments for the charge carrying point M in molecules with TIP4P water geometry. The corresponding parameters are extracted from the pair style.

Note that only atoms in the specified group contribute to the calculation. The *compute chunk/atom* command defines its own group; atoms will have a chunk ID = 0 if they are not in that group, signifying they are not assigned to a chunk, and will thus also not contribute to this calculation. You can specify the "all" group for this command if you simply want to include atoms with non-zero chunk IDs.

> **Note**
>
> The coordinates of an atom contribute to the chunk's dipole in "unwrapped" form, by using the image flags associated with each atom. See the *dump custom* command for a discussion of "unwrapped" coordinates. See the Atoms section of the *read_data* command for a discussion of image flags and how they are set for each atom. You can reset the image flags (e.g., to 0) before invoking this compute by using the *set image* command.

The simplest way to output the results of the compute com/chunk calculation to a file is to use the *fix ave/time* command, for example:

```
compute cc1 all chunk/atom molecule
compute myChunk all dipole/chunk cc1
fix 1 all ave/time 100 1 100 c_myChunk[*] file tmp.out mode vector
```

### 3.33.4 Output info

These computes calculate a global array where the number of rows = the number of chunks *Nchunk* as calculated by the specified *compute chunk/atom* command. The number of columns is 4 for the $(x, y, z)$ dipole vector components and the total dipole of each chunk. These values can be accessed by any command that uses global array values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The array values are "intensive". The array values will be in dipole units (i.e., charge *units* times distance *units*).

### 3.33.5 Restrictions

Compute style *dipole/tip4p/chunk* is part of the EXTRA-COMPUTE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Compute style *dipole/tip4p/chunk* can only be used with tip4p pair styles.

### 3.33.6 Related commands

*compute com/chunk*, *compute dipole*

### 3.33.7 Default

Using the center of mass is the default setting for the net charge correction.

## 3.34 compute displace/atom command

### 3.34.1 Syntax

```
compute ID group-ID displace/atom
```

- ID, group-ID are documented in *compute* command
- displace/atom = style name of this compute command
- zero or more keyword/arg pairs may be appended
- keyword = *refresh*

  *refresh* arg = name of per-atom variable

### 3.34.2 Examples

```
compute 1 all displace/atom
compute 1 all displace/atom refresh myVar
```

### 3.34.3 Description

Define a computation that calculates the current displacement of each atom in the group from its original (reference) coordinates, including all effects due to atoms passing through periodic boundaries.

A vector of four quantities per atom is calculated by this compute. The first three elements of the vector are the $(dx, dy, dz)$ displacements. The fourth component is the total displacement (i.e., $\sqrt{dx^2 + dy^2 + dz^2}$).

The displacement of an atom is from its original position at the time the compute command was issued. The value of the displacement will be 0.0 for atoms not in the specified compute group.

**Note**

Initial coordinates are stored in "unwrapped" form, by using the image flags associated with each atom. See the *dump custom* command for a discussion of "unwrapped" coordinates. See the Atoms section of the *read_data* command for a discussion of image flags and how they are set for each atom. You can reset the image flags (e.g., to 0) before invoking this compute by using the *set image* command.

**Note**

If you want the quantities calculated by this compute to be continuous when running from a *restart file*, then you should use the same ID for this compute, as in the original run. This is so that the fix this compute creates to store per-atom quantities will also have the same ID, and thus be initialized correctly with time = 0 atom coordinates from the restart file.

---

The *refresh* option can be used in conjunction with the "dump_modify refresh" command to generate incremental dump files.

The definition and motivation of an incremental dump file is as follows. Instead of outputting all atoms at each snapshot (with some associated values), you may only wish to output the subset of atoms with a value that has changed in some way compared to the value the last time that atom was output. In some scenarios this can result in a dramatically smaller dump file. If desired, by post-processing the sequence of snapshots, the values for all atoms at all timesteps can be inferred.

A concrete example using this compute, is a simulation of atom diffusion in a solid, represented as atoms on a lattice. Diffusive hops are rare. Imagine that when a hop occurs an atom moves more than a distance *Dhop*. For any snapshot we only want to output atoms that have hopped since the last snapshot. This can be accomplished with something like the following commands:

```
write_dump      all custom tmp.dump id type x y z    # see comment below

variable        Dhop equal 0.6
variable        check atom "c_dsp[4] > v_Dhop"
compute         dsp all displace/atom refresh check
dump            1 all custom 100 tmp.dump id type x y z
dump_modify     1 append yes thresh c_dsp[4] > ${Dhop} &
                refresh c_dsp delay 100
```

The *dump_modify thresh* command will only output atoms that have displaced more than 0.6 Å on each snapshot (assuming metal units). The dump_modify *refresh* option triggers a call to this compute at the end of every dump.

The *refresh* argument for this compute is the ID of an *atom-style variable* which calculates a Boolean value (0 or 1) based on the same criterion used by dump_modify thresh. This compute evaluates the atom-style variable. For each atom that returns 1 (true), the original (reference) coordinates of the atom (stored by this compute) are updated.

The effect of these commands is that a particular atom will only be output in the dump file on the snapshot after it makes a diffusive hop. It will not be output again until it makes another hop.

Note that in the first snapshot of a subsequent run, no atoms will be typically be output. That is because the initial displacement for all atoms is 0.0. If an initial dump snapshot is desired, containing the initial reference positions of all atoms, one way to do this is illustrated above. An initial write_dump command can be used before the first run. It will contain the positions of all the atoms, Options in the *dump_modify* command above will append new output to that same file and delay the output until a later timestep. The *delay* setting avoids a second time = 0 snapshot which would be empty.

---

### 3.34.4 Output info

This compute calculates a per-atom array with four columns, which can be accessed by indices 1–4 by any command that uses per-atom values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

The per-atom array values will be in distance *units*.

This compute supports the *refresh* option as explained above, for use in conjunction with *dump_modify refresh* to generate incremental dump files.

### 3.34.5 Restrictions

none

### 3.34.6 Related commands

*compute msd*, *dump custom*, *fix store/state*

### 3.34.7 Default

none

## 3.35 compute dpd command

### 3.35.1 Syntax

```
compute ID group-ID dpd
```

- ID, group-ID are documented in *compute* command
- dpd = style name of this compute command

### 3.35.2 Examples

```
compute 1 all dpd
```

### 3.35.3 Description

Define a computation that accumulates the total internal conductive energy ($U^{\text{cond}}$), the total internal mechanical energy ($U^{\text{mech}}$), the total chemical energy ($U^{\text{chem}}$) and the *harmonic* average of the internal temperature ($\theta_{\text{avg}}$) for the entire system of particles. See the *compute dpd/atom* command if you want per-particle internal energies and internal temperatures.

The system internal properties are computed according to the following relations:

$$U^{\text{cond}} = \sum_{i=1}^{N} u_i^{\text{cond}}$$

$$U^{\text{mech}} = \sum_{i=1}^{N} u_i^{\text{mech}}$$

$$U^{\text{chem}} = \sum_{i=1}^{N} u_i^{\text{chem}}$$

$$U = \sum_{i=1}^{N} (u_i^{\text{cond}} + u_i^{\text{mech}} + u_i^{\text{chem}})$$

$$\theta_{avg} = \left( \frac{1}{N} \sum_{i=1}^{N} \frac{1}{\theta_i} \right)^{-1}$$

where $N$ is the number of particles in the system.

---

### 3.35.4 Output info

This compute calculates a global vector of length 5 ($U^{\text{cond}}$, $U^{\text{mech}}$, $U^{\text{chem}}$, $\theta_{\text{avg}}$, $N$), which can be accessed by indices 1 through 5. See the *Howto output* page for an overview of LAMMPS output options.

The vector values will be in energy and temperature *units*.

### 3.35.5 Restrictions

This command is part of the DPD-REACT package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This command also requires use of the *atom_style dpd* command.

### 3.35.6 Related commands

*compute dpd/atom*, *thermo_style*

### 3.35.7 Default

none

---

**(Larentzos)** J.P. Larentzos, J.K. Brennan, J.D. Moore, and W.D. Mattson, "LAMMPS Implementation of Constant Energy Dissipative Particle Dynamics (DPD-E)", ARL-TR-6863, U.S. Army Research Laboratory, Aberdeen Proving Ground, MD (2014).

# 3.36 compute dpd/atom command

## 3.36.1 Syntax

```
compute ID group-ID dpd/atom
```

- ID, group-ID are documented in *compute* command
- dpd/atom = style name of this compute command

## 3.36.2 Examples

```
compute 1 all dpd/atom
```

## 3.36.3 Description

Define a computation that accesses the per-particle internal conductive energy ($u^{\text{cond}}$), internal mechanical energy ($u^{\text{mech}}$), internal chemical energy ($u^{\text{chem}}$) and internal temperatures ($\theta$) for each particle in a group. See the *compute dpd* command if you want the total internal conductive energy, the total internal mechanical energy, the total chemical energy and average internal temperature of the entire system or group of dpd particles.

## 3.36.4 Output info

This compute calculates a per-particle array with four columns ($u^{\text{cond}}$, $u^{\text{mech}}$, $u^{\text{chem}}$, $\theta$), which can be accessed by indices 1–4 by any command that uses per-particle values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-particle array values will be in energy ($u^{\text{cond}}$, $u^{\text{mech}}$, $u^{\text{chem}}$) and temperature ($\theta$) *units*.

## 3.36.5 Restrictions

This command is part of the DPD-REACT package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This command also requires use of the *atom_style dpd* command.

## 3.36.6 Related commands

*dump custom*, *compute dpd*

## 3.36.7 Default

none

---

**(Larentzos)** J.P. Larentzos, J.K. Brennan, J.D. Moore, and W.D. Mattson, "LAMMPS Implementation of Constant Energy Dissipative Particle Dynamics (DPD-E)", ARL-TR-6863, U.S. Army Research Laboratory, Aberdeen Proving Ground, MD (2014).

# 3.37 compute edpd/temp/atom command

## 3.37.1 Syntax

```
compute ID group-ID edpd/temp/atom
```

- ID, group-ID are documented in *compute* command
- edpd/temp/atom = style name of this compute command

### 3.37.2 Examples

```
compute 1 all edpd/temp/atom
```

### 3.37.3 Description

Define a computation that calculates the per-atom temperature for each eDPD particle in a group.

The temperature is a local temperature derived from the internal energy of each eDPD particle based on the local equilibrium hypothesis. For more details please see *(Espanol1997)* and *(Li2014)*.

### 3.37.4 Output info

This compute calculates a per-atom vector, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-atom vector values will be in temperature *units*.

### 3.37.5 Restrictions

This compute is part of the DPD-MESO package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.37.6 Related commands

*pair_style edpd*

### 3.37.7 Default

none

---

**(Espanol1997)** Espanol, Europhys Lett, 40(6): 631-636 (1997). DOI: 10.1209/epl/i1997-00515-8

**(Li2014)** Li, Tang, Lei, Caswell, Karniadakis, J Comput Phys, 265: 113-127 (2014). DOI: 10.1016/j.jcp.2014.02.003.

## 3.38 compute efield/atom command

### 3.38.1 Syntax

```
compute ID group-ID efield/atom keyword val
```

- ID, group-ID are documented in *compute* command
- efield/atom = style name of this compute command
- zero or more keyword/value pairs may be appended
- keyword = *pair* or *kspace*

  ```
  pair args = yes or no
  kspace args = yes or no
  ```

### 3.38.2 Examples

```
compute 1 all efield/atom
compute 1 all efield/atom pair yes kspace no
```

Used in input scripts:

```
examples/PACKAGES/dielectric/in.confined
examples/PACKAGES/dielectric/in.nopbc
```

### 3.38.3 Description

Define a computation that calculates the electric field at each atom in a group. The compute should only enabled with pair and kspace styles that are provided by the DIELECTRIC package because only these styles compute the per-atom electric field at every time step.

The electric field is a 3-component vector. The value of the electric field components will be 0.0 for atoms not in the specified compute group.

---

The keyword/value option pairs are used in the following ways.

For the *pair* and *kspace* keywords, the real-space and reciprocal-space contributions to the electric field can be turned off and on.

### 3.38.4 Output info

This compute calculates a per-atom vector, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-atom vector values will be in electric field *units*.

### 3.38.5 Restrictions

This compute is part of the DIELECTRIC package. It is only enabled if LAMMPS was built with that package.

### 3.38.6 Related commands

*dump custom*

### 3.38.7 Default

The option defaults are pair = yes and kspace = yes.

## 3.39 compute efield/wolf/atom command

### 3.39.1 Syntax

```
compute ID group-ID efield/wolf/atom alpha keyword val
```

- ID, group-ID are documented in *compute* command
- efield/atom/wolf = style name of this compute command
- alpha = damping parameter (inverse distance units)
- zero or more keyword/value pairs may be appended

- keyword = *limit* or *cutoff*

```
limit group2-ID = limit computing the electric field contributions to a group␣
↪(default: all)
cutoff value = set cutoff for computing contributions to this value (default:␣
↪maximum cutoff of pair style)
```

### 3.39.2 Examples

```
compute 1 all efield/wolf/atom 0.2
compute 1 mols efield/wolf/atom 0.25 limit water cutoff 10.0
```

### 3.39.3 Description

Added in version 8Feb2023.

Define a computation that approximates the electric field at each atom in a group.

$$\vec{E}_i = \frac{\vec{F}coul_i}{q_i} = \sum_{j \neq i} \frac{q_j}{r_{ij}^2} \qquad r < r_c$$

The electric field at the position of the atom $i$ is the coulomb force on a unit charge at that point, which is equivalent to dividing the Coulomb force by the charge of the individual atom.

In this compute the electric field is approximated as the derivative of the potential energy using the Wolf summation method, described in *Wolf*, given by:

$$E_i = \frac{1}{2} \sum_{j \neq i} \frac{q_i q_j \mathrm{erfc}(\alpha r_{ij})}{r_{ij}} + \frac{1}{2} \sum_{j \neq i} \frac{q_i q_j \mathrm{erf}(\alpha r_{ij})}{r_{ij}} \qquad r < r_c$$

where $\alpha$ is the damping parameter, and *erf()* and *erfc()* are error-function and complementary error-function terms. This potential is essentially a short-range, spherically-truncated, charge-neutralized, force-shifted, pairwise *1/r* summation. With a manipulation of adding and subtracting a self term (for i = j) to the first and second term on the right-hand-side, respectively, and a small enough $\alpha$ damping parameter, the second term shrinks and the potential becomes a rapidly-converging real-space summation. With a long enough cutoff and small enough $\alpha$ parameter, the electric field calculated by the Wolf summation method approaches that computed using the Ewald sum.

The value of the electric field components will be 0.0 for atoms not in the specified compute group.

When the *limit* keyword is used, only contributions from atoms in the selected group will be considered, otherwise contributions from all atoms within the cutoff are included.

When the *cutoff* keyword is used, the cutoff used for the electric field approximation can be set explicitly. By default it is the largest cutoff of any pair style force computation.

**Computational Efficiency**

This compute will loop over a full neighbor list just like a pair style does when computing forces, thus it can be quite time-consuming and slow down a calculation significantly when its data is used in every time step. The *compute efield/atom* command of the DIELECTRIC package is more efficient in comparison, since the electric field data is collected and stored as part of the force computation at next to no extra computational cost.

### 3.39.4 Output info

This compute calculates a per-atom vector, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The vector contains 3 values per atom which are the x-, y-, and z-direction electric field components in force units.

### 3.39.5 Restrictions

This compute is part of the EXTRA-COMPUTE package. It is only enabled if LAMMPS was built with that package.

This compute requires *neighbor styles 'bin' or 'nsq'*.

### 3.39.6 Related commands

*pair_style coul/wolf*, *compute efield/atom*

### 3.39.7 Default

The option defaults are *limit* = all and *cutoff* = largest cutoff for pair styles.

---

**(Wolf)** D. Wolf, P. Keblinski, S. R. Phillpot, J. Eggebrecht, J Chem Phys, 110, 8254 (1999).

## 3.40 compute entropy/atom command

Accelerator Variants: *entropy/atom/omp*

### 3.40.1 Syntax

```
compute ID group-ID entropy/atom sigma cutoff keyword value ...
```

- ID, group-ID are documented in *compute* command
- entropy/atom = style name of this compute command
- sigma = width of Gaussians used in the $g(r)$ smoothing
- cutoff = cutoff for the $g(r)$ calculation
- one or more keyword/value pairs may be appended

```
keyword = avg or local
  avg args = neigh cutoff2
    neigh value = yes or no = whether to average the pair entropy over neighbors
    cutoff2 = cutoff for the averaging over neighbors
  local arg = yes or no = use the local density around each atom to normalize the g(r)
```

### 3.40.2 Examples

```
compute 1 all entropy/atom 0.25 5.
compute 1 all entropy/atom 0.25 5. avg yes 5.
compute 1 all entropy/atom 0.125 7.3 avg yes 5.1 local yes
```

### 3.40.3 Description

Define a computation that calculates the pair entropy fingerprint for each atom in the group. The fingerprint is useful to distinguish between ordered and disordered environments, for instance liquid and solid-like environments, or glassy and crystalline-like environments. Some applications could be the identification of grain boundaries, a melt-solid interface, or a solid cluster emerging from the melt. The advantage of this parameter over others is that no a priori information about the solid structure is required.

This parameter for atom i is computed using the following formula from *(Piaggi)* and *(Nettleton)* ,

$$s_S^i = -2\pi\rho k_B \int_0^{r_m} [g(r)\ln g(r) - g(r) + 1]\, r^2 dr$$

where $r$ is a distance, $g(r)$ is the radial distribution function of atom $i$, and $\rho$ is the density of the system. The $g(r)$ computed for each atom $i$ can be noisy and therefore it is smoothed using

$$g_m^i(r) = \frac{1}{4\pi\rho r^2} \sum_j \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(r-r_{ij})^2/(2\sigma^2)}$$

where the sum over $j$ goes through the neighbors of atom $i$ and $\sigma$ is a parameter to control the smoothing.

The input parameters are *sigma* the smoothing parameter $\sigma$, and the *cutoff* for the calculation of $g(r)$.

If the keyword *avg* has the setting *yes*, then this compute also averages the parameter over the neighbors of atom $i$ according to

$$\left\langle s_S^i \right\rangle = \frac{\sum_j s_S^j + s_S^i}{N+1},$$

where the sum over $j$ goes over the neighbors of atom $i$ and $N$ is the number of neighbors. This procedure provides a sharper distinction between order and disorder environments. In this case the input parameter *cutoff2* is the cutoff for the averaging over the neighbors and must also be specified.

If the *avg yes* option is used, the effective cutoff of the neighbor list should be *cutoff* +*cutoff2* and therefore it might be necessary to increase the skin of the neighbor list with:

```
neighbor <skin distance> bin
```

See *neighbor* for details.

If the *local yes* option is used, the $g(r)$ is normalized by the local density around each atom, that is to say the density around each atom is the number of neighbors within the neighbor list cutoff divided by the corresponding volume. This option can be useful when dealing with inhomogeneous systems such as those that have surfaces.

Here are typical input parameters for fcc aluminum (lattice constant 4.05 Å),

```
compute 1 all entropy/atom 0.25 5.7 avg yes 3.7
```

and for bcc sodium (lattice constant 4.23 Å),

```
compute 1 all entropy/atom 0.25 7.3 avg yes 5.1
```

### 3.40.4 Output info

By default, this compute calculates the pair entropy value for each atom as a per-atom vector, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The pair entropy values have units of the Boltzmann constant. They are always negative, and lower values (lower entropy) correspond to more ordered environments.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 3.40.5 Restrictions

This compute is part of the EXTRA-COMPUTE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.40.6 Related commands

*compute cna/atom compute centro/atom*

### 3.40.7 Default

The default values for the optional keywords are avg = no and local = no.

---

**(Piaggi)** Piaggi and Parrinello, J Chem Phys, 147, 114112 (2017).

**(Nettleton)** Nettleton and Green, J Chem Phys, 29, 6 (1958).

## 3.41 compute erotate/asphere command

Accelerator Variants: *erotate/asphere/kk*

### 3.41.1 Syntax

```
compute ID group-ID erotate/asphere
```

- ID, group-ID are documented in *compute* command
- erotate/asphere = style name of this compute command

### 3.41.2 Examples

```
compute 1 all erotate/asphere
```

### 3.41.3 Description

Define a computation that calculates the rotational kinetic energy of a group of aspherical particles. The aspherical particles can be ellipsoids, or line segments, or triangles. See the *atom_style* and *read_data* commands for descriptions of these options.

For all 3 types of particles, the rotational kinetic energy is computed as $\frac{1}{2} I \omega^2$, where $I$ is the inertia tensor for the aspherical particle and $\omega$ is its angular velocity, which is computed from its angular momentum if needed.

**Note**

For *2d models*, ellipsoidal particles are treated as ellipsoids, not ellipses, meaning their moments of inertia will be the same as in 3d.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 3.41.4 Output info

This compute calculates a global scalar (the KE). This value can be used by any command that uses a global scalar value from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "extensive". The scalar value will be in energy *units*.

### 3.41.5 Restrictions

This compute requires that ellipsoidal particles atoms store a shape and quaternion orientation and angular momentum as defined by the *atom_style ellipsoid* command.

This compute requires that line segment particles atoms store a length and orientation and angular velocity as defined by the *atom_style line* command.

This compute requires that triangular particles atoms store a size and shape and quaternion orientation and angular momentum as defined by the *atom_style tri* command.

All particles in the group must be of finite size. They cannot be point particles.

### 3.41.6 Related commands

none

*compute erotate/sphere*

### 3.41.7 Default

none

## 3.42 compute erotate/rigid command

### 3.42.1 Syntax

```
compute ID group-ID erotate/rigid fix-ID
```

- ID, group-ID are documented in *compute* command
- erotate/rigid = style name of this compute command
- fix-ID = ID of rigid body fix

### 3.42.2 Examples

```
compute 1 all erotate/rigid myRigid
```

### 3.42.3 Description

Define a computation that calculates the rotational kinetic energy of a collection of rigid bodies, as defined by one of the *fix rigid* command variants.

The rotational energy of each rigid body is computed as $\frac{1}{2} I \omega_{\text{body}}^2$, where $I$ is the inertia tensor for the rigid body and $\omega_{\text{body}}$ is its angular velocity vector. Both $I$ and $\omega_{\text{body}}$ are in the frame of reference of the rigid body (i.e., $I$ is diagonal).

The *fix-ID* should be the ID of one of the *fix rigid* commands which defines the rigid bodies. The group specified in the compute command is ignored. The rotational energy of all the rigid bodies defined by the fix rigid command in included in the calculation.

### 3.42.4 Output info

This compute calculates a global scalar (the summed rotational energy of all the rigid bodies). This value can be used by any command that uses a global scalar value from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "extensive". The scalar value will be in energy *units*.

### 3.42.5 Restrictions

This compute is part of the RIGID package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.42.6 Related commands

*compute ke/rigid*

### 3.42.7 Default

none

## 3.43 compute erotate/sphere command

Accelerator Variants: *erotate/sphere/kk*

### 3.43.1 Syntax

```
compute ID group-ID erotate/sphere
```

- ID, group-ID are documented in *compute* command
- erotate/sphere = style name of this compute command

### 3.43.2 Examples

```
compute 1 all erotate/sphere
```

### 3.43.3 Description

Define a computation that calculates the rotational kinetic energy of a group of spherical particles.

The rotational energy is computed as $\frac{1}{2} I \omega^2$, where $I$ is the moment of inertia for a sphere and $\omega$ is the particle's angular velocity.

**Note**

For *2d models*, particles are treated as spheres, not disks, meaning their moment of inertia will be the same as in 3d.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 3.43.4 Output info

This compute calculates a global scalar (the KE). This value can be used by any command that uses a global scalar value from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "extensive". The scalar value will be in energy *units*.

### 3.43.5 Restrictions

This compute requires that atoms store a radius and angular velocity (omega) as defined by the *atom_style sphere* command.

All particles in the group must be finite-size spheres or point particles. They cannot be aspherical. Point particles will not contribute to the rotational energy.

### 3.43.6 Related commands

*compute erotate/asphere*

### 3.43.7 Default

none

# 3.44 compute erotate/sphere/atom command

Accelerator Variants: *erotate/sphere/atom/omp*

## 3.44.1 Syntax

```
compute ID group-ID erotate/sphere/atom
```

- ID, group-ID are documented in *compute* command
- erotate/sphere/atom = style name of this compute command

## 3.44.2 Examples

```
compute 1 all erotate/sphere/atom
```

## 3.44.3 Description

Define a computation that calculates the rotational kinetic energy for each particle in a group.

The rotational energy is computed as $\frac{1}{2} I \omega^2$, where $I$ is the moment of inertia for a sphere and $\omega$ is the particle's angular velocity.

**Note**

For *2d models*, particles are treated as spheres, not disks, meaning their moment of inertia will be the same as in 3d.

The value of the rotational kinetic energy will be 0.0 for atoms not in the specified compute group or for point particles with a radius of 0.0.

## 3.44.4 Output info

This compute calculates a per-atom vector, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-atom vector values will be in energy *units*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 3.44.5 Restrictions

none

### 3.44.6 Related commands

*dump custom*

### 3.44.7 Default

none

## 3.45 compute event/displace command

### 3.45.1 Syntax

```
compute ID group-ID event/displace threshold
```

- ID, group-ID are documented in *compute* command
- event/displace = style name of this compute command
- threshold = minimum distance any particle must move to trigger an event (distance units)

### 3.45.2 Examples

```
compute 1 all event/displace 0.5
```

### 3.45.3 Description

Define a computation that flags an "event" if any particle in the group has moved a distance greater than the specified threshold distance when compared to a previously stored reference state (i.e., the previous event). This compute is typically used in conjunction with the *prd* and *tad* commands, to detect if a transition to a new minimum energy basin has occurred.

This value calculated by the compute is equal to 0 if no particle has moved far enough, and equal to 1 if one or more particles have moved further than the threshold distance.

**Note**

If the system is undergoing significant center-of-mass motion, due to thermal motion, an external force, or an initial net momentum, then this compute will not be able to distinguish that motion from local atom displacements and may generate "false positives".

### 3.45.4 Output info

This compute calculates a global scalar (the flag). This value can be used by any command that uses a global scalar value from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "intensive". The scalar value will be a 0 or 1 as explained above.

### 3.45.5 Restrictions

This command can only be used if LAMMPS was built with the REPLICA package. See the *Build package* doc page for more info.

### 3.45.6 Related commands

*prd*, *tad*

### 3.45.7 Default

none

## 3.46 compute fabric command

### 3.46.1 Syntax

```
compute ID group-ID fabric cutoff attribute ... keyword values ...
```

- ID, group-ID are documented in *compute* command
- fabric = style name of this compute command
- cutoff = *type* or *radius*

  *type* = cutoffs determined based on atom types
  *radius* = cutoffs determined based on atom diameters (atom style sphere)

- one or more attributes may be appended
- attribute = *contact* or *branch* or *force/normal* or *force/tangential*

  *contact* = contact tensor
  *branch* = branch tensor
  *force/normal* = normal force tensor
  *force/tangential* = tangential force tensor

- zero or more keyword/value pairs may be appended
- keyword = *type/include*

  *type/include* value = arg1 arg2
    arg = separate lists of types (see below)

### 3.46.2 Examples

```
compute 1 all fabric type contact force/normal type/include 1,2 3*4
compute 1 all fabric radius force/normal force/tangential
```

### 3.46.3 Description

Define a compute that calculates various fabric tensors for pairwise interaction *(Ouadfel)*. Fabric tensors are commonly used to quantify the anisotropy or orientation of granular contacts but can also be used to characterize the direction of pairwise interactions in general systems. The *type* and *radius* settings are used to select whether interactions cutoffs are determined by atom types or by the sum of atomic radii (atom style sphere), respectively. Calling this compute is roughly the cost of a pair style invocation as it involves a loop over the neighbor list. If the normal or tangential force tensors are requested, it will be more expensive than a pair style invocation as it will also recalculate all pair forces.

Four fabric tensors are available: the contact, branch, normal force, or tangential force tensor. The contact tensor is calculated as

$$C_{ab} = \frac{15}{2}(\phi_{ab} - \frac{1}{3}\mathrm{Tr}(\phi)\delta_{ab})$$

where $a$ and $b$ are the $x$, $y$, $z$ directions, $\delta_{ab}$ is the Kronecker delta function, and the tensor $\phi$ is defined as

$$\phi_{ab} = \sum_{n=1}^{N_p} \frac{r_a r_b}{r^2}$$

where $n$ loops over the $N_p$ pair interactions in the simulation, $r_a$ is the $a$ component of the radial vector between the two pairwise interacting particles, and $r$ is the magnitude of the radial vector.

The branch tensor is calculated as

$$B_{ab} = \frac{15}{2\,\mathrm{Tr}(D)}(D_{ab} - \frac{1}{3}\mathrm{Tr}(D)\delta_{ab})$$

where the tensor $D$ is defined as

$$D_{ab} = \sum_{n=1}^{N_p} \frac{1}{N_c(r^2 + C_{cd}r_c r_d)} \frac{r_a r_b}{r}$$

where $N_c$ is the total number of contacts in the system and the subscripts $c$ and $d$ indices are summed according to Einstein notation.

The normal force fabric tensor is calculated as

$$F^n_{ab} = \frac{15}{2\,\mathrm{Tr}(N)}(N_{ab} - \frac{1}{3}\mathrm{Tr}(N)\delta_{ab})$$

where the tensor $N$ is defined as

$$N_{ab} = \sum_{n=1}^{N_p} \frac{1}{N_c(r^2 + C_{cd}r_c r_d)} \frac{r_a r_b}{r^2} f_n$$

and $f_n$ is the magnitude of the normal, central-body force between the two atoms.

Finally, the tangential force fabric tensor is only defined for pair styles that apply tangential forces to particles, namely granular pair styles. It is calculated as

$$F^t_{ab} = \frac{5}{\mathrm{Tr}(N)}(T_{ab} - \frac{1}{3}\mathrm{Tr}(T)\delta_{ab})$$

where the tensor $T$ is defined as

$$T_{ab} = \sum_{n=1}^{N_p} \frac{1}{N_c(r^2 + C_{cd}r_c r_d)} \frac{r_a r_b}{r^2} f_t$$

and $f_t$ is the magnitude of the tangential force between the two atoms.

The *type/include* keyword filters interactions based on the types of the two atoms. Interactions between two atoms are only included in calculations if the atom types are in the two lists. Each list consists of a series of type ranges separated by commas. The range can be specified as a single numeric value, or a wildcard asterisk can be used to specify a range of values. This takes the form "*" or "*n" or "m*" or "m*n". For example, if $M$ is the number of atom types, then an asterisk with no numeric values means all types from 1 to $M$. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from m to $M$ (inclusive). A middle asterisk means all types from m to n (inclusive). Multiple *type/include* keywords may be added.

### 3.46.4 Output info

This compute calculates a global vector of doubles and a global scalar. The vector stores the unique components of the first requested tensor in the order *xx*, *yy*, *zz*, *xy*, *xz*, *yz* followed by the same components for all subsequent tensors. The length of the vector is therefore six times the number of requested tensors. The scalar output is the number of pairwise interactions included in the calculation of the fabric tensor.

### 3.46.5 Restrictions

This fix is part of the GRANULAR package. It is only enabled if LAMMPS was built with that package. See the *Build package* doc page for more info.

Currently, compute *fabric* does not support pair styles with many-body interactions. It also does not support models with long-range Coulombic or dispersion forces, i.e. the kspace_style command in LAMMPS. It also does not support the following fixes which add rigid-body constraints: *fix shake*, *fix rattle*, *fix rigid*, *fix rigid/small*. It does not support granular pair styles that extend beyond the contact of atomic radii (e.g., JKR and DMT).

### 3.46.6 Related commands

none

### 3.46.7 Default

none

---

**(Ouadfel)** Ouadfel and Rothenburg "Stress-force-fabric relationship for assemblies of ellipsoids", Mechanics of Materials (2001). (link to paper)

## 3.47 compute fep command

### 3.47.1 Syntax

```
compute ID group-ID fep temp attribute args ... keyword value ...
```

- ID, group-ID are documented in the *compute* command
- fep = name of this compute command
- temp = external temperature (as specified for constant-temperature run)
- one or more attributes with args may be appended
- attribute = *pair* or *atom*

```
pair args = pstyle pparam I J v_delta
  pstyle = pair style name (e.g., lj/cut)
  pparam = parameter to perturb
  I,J = type pair(s) to set parameter for
  v_delta = variable with perturbation to apply (in the units of the parameter)
atom args = aparam I v_delta
  aparam = charge = parameter to perturb
  I = type to set parameter for
  v_delta = variable with perturbation to apply (in the units of the parameter)
```

- zero or more keyword/value pairs may be appended
- keyword = *tail* or *volume*

```
tail value = no or yes
  no = ignore tail correction to pair energies (usually small in fep)
  yes = include tail correction to pair energies
volume value = no or yes
  no = ignore volume changes (e.g., in NVE or NVT trajectories)
  yes = include volume changes (e.g., in NPT trajectories)
```

### 3.47.2 Examples

```
compute 1 all fep 298 pair lj/cut epsilon 1 * v_delta pair lj/cut sigma 1 * v_delta␣
↪volume yes
compute 1 all fep 300 atom charge 2 v_delta
```

Example input scripts available: examples/PACKAGES/fep

### 3.47.3 Description

Apply a perturbation to parameters of the interaction potential and recalculate the pair potential energy without changing the atomic coordinates from those of the reference, unperturbed system. This compute can be used to calculate free energy differences using several methods, such as free-energy perturbation (FEP), finite-difference thermodynamic integration (FDTI) or Bennet's acceptance ratio method (BAR).

The potential energy of the system is decomposed in three terms: a background term corresponding to interaction sites whose parameters remain constant, a reference term $U_0$ corresponding to the initial interactions of the atoms that will undergo perturbation, and a term $U_1$ corresponding to the final interactions of these atoms:

$$U(\lambda) = U_{\mathrm{bg}} + U_1(\lambda) + U_0(\lambda)$$

A coupling parameter $\lambda$ varying from 0 to 1 connects the reference and perturbed systems:

$$\begin{aligned} \lambda = 0 \quad &\Rightarrow \quad U = U_{\mathrm{bg}} + U_0 \\ \lambda = 1 \quad &\Rightarrow \quad U = U_{\mathrm{bg}} + U_1 \end{aligned}$$

It is possible but not necessary that the coupling parameter (or a function thereof) appears as a multiplication factor of the potential energy. Therefore, this compute can apply perturbations to interaction parameters that are not directly proportional to the potential energy (e.g., $\sigma$ in Lennard-Jones potentials).

This command can be combined with *fix adapt* to perform multistage free-energy perturbation calculations along stepwise alchemical transformations during a simulation run:

$$\Delta_0^1 A = \sum_{i=0}^{n-1} \Delta_{\lambda_i}^{\lambda_{i+1}} A = -k_B T \sum_{i=0}^{n-1} \ln \left\langle \exp\left( -\frac{U(\lambda_{i+1}) - U(\lambda_i)}{k_B T} \right) \right\rangle_{\lambda_i}$$

This compute is suitable for the finite-difference thermodynamic integration (FDTI) method *(Mezei)*, which is based on an evaluation of the numerical derivative of the free energy by a perturbation method using a very small $\delta$:

$$\Delta_0^1 A = \int_{\lambda=0}^{\lambda=1} \left( \frac{\partial A(\lambda)}{\partial \lambda} \right)_\lambda \mathrm{d}\lambda \approx \sum_{i=0}^{n-1} w_i \frac{A(\lambda_i + \delta) - A(\lambda_i)}{\delta}$$

where $w_i$ are weights of a numerical quadrature. The *fix adapt* command can be used to define the stages of $\lambda$ at which the derivative is calculated and averaged.

The compute fep calculates the exponential Boltzmann term and also the potential energy difference $U_1 - U_0$. By choosing a very small perturbation $\delta$ the thermodynamic integration method can be implemented using a numerical evaluation of the derivative of the potential energy with respect to $\lambda$:

$$\Delta_0^1 A = \int_{\lambda=0}^{\lambda=1} \left\langle \frac{\partial U(\lambda)}{\partial \lambda} \right\rangle_\lambda \mathrm{d}\lambda \approx \sum_{i=0}^{n-1} w_i \left\langle \frac{U(\lambda_i + \delta) - U(\lambda_i)}{\delta} \right\rangle_{\lambda_i}$$

Another technique to calculate free energy differences is the acceptance ratio method *(Bennet)*, which can be implemented by calculating the potential energy differences with $\delta = 1.0$ on both the forward and reverse routes:

$$\left\langle \frac{1}{1+\exp\left[\left(U_1 - U_0 - \Delta_0^1 A\right)/k_B T\right]} \right\rangle_0 = \left\langle \frac{1}{1+\exp\left[\left(U_0 - U_1 + \Delta_0^1 A\right)/k_B T\right]} \right\rangle_1$$

The value of the free energy difference is determined by numerical root finding to establish the equality.

Concerning the choice of how the atomic parameters are perturbed in order to setup an alchemical transformation route, several strategies are available, such as single-topology or double-topology strategies *(Pearlman)*. The latter does not require modification of bond lengths, angles or other internal coordinates.

NOTES: This compute command does not take kinetic energy into account, therefore the masses of the particles should not be modified between the reference and perturbed states, or along the alchemical transformation route. This compute command does not change bond lengths or other internal coordinates *(Boresch, Karplus)*.

---

The *pair* attribute enables various parameters of potentials defined by the *pair_style* and *pair_coeff* commands to be changed, if the pair style supports it.

The *pstyle* argument is the name of the pair style. For example, *pstyle* could be specified as "lj/cut". The *pparam* argument is the name of the parameter to change. This is a list of pair styles and parameters that can be used with this compute. See the doc pages for individual pair styles and their energy formulas for the meaning of these parameters:

| | | |
|---|---|---|
| *born* | a,b,c | type pairs |
| *buck, buck/coul/cut, buck/coul/long, buck/coul/msm* | a,c | type pairs |
| *buck/mdf* | a,c | type pairs |
| *coul/cut* | scale | type pairs |
| *coul/cut/soft* | lambda | type pairs |
| *coul/long, coul/msm* | scale | type pairs |
| *coul/long/soft* | scale, lambda | type pairs |
| *eam* | scale | type pairs |
| *gauss* | a | type pairs |
| *lennard/mdf* | a,b | type pairs |
| *lj/class2* | epsilon,sigma | type pairs |
| *lj/class2/coul/cut, lj/class2/coul/long* | epsilon,sigma | type pairs |
| *lj/cut* | epsilon,sigma | type pairs |
| *lj/cut/soft* | epsilon,sigma,lambda | type pairs |
| *lj/cut/coul/cut, lj/cut/coul/long, lj/cut/coul/msm* | epsilon,sigma | type pairs |
| *lj/cut/coul/cut/soft, lj/cut/coul/long/soft* | epsilon,sigma,lambda | type pairs |
| *lj/cut/tip4p/cut, lj/cut/tip4p/long* | epsilon,sigma | type pairs |
| *lj/cut/tip4p/long/soft* | epsilon,sigma,lambda | type pairs |
| *lj/expand* | epsilon,sigma,delta | type pairs |
| *lj/mdf* | epsilon,sigma | type pairs |
| *lj/sf/dipole/sf* | epsilon,sigma,scale | type pairs |
| *mie/cut* | epsilon,sigma,gamR,gamA | type pairs |
| *morse, morse/smooth/linear* | d0,r0,alpha | type pairs |
| *morse/soft* | d0,r0,alpha,lambda | type pairs |
| *nm/cut* | e0,r0,nn,mm | type pairs |
| *nm/cut/coul/cut, nm/cut/coul/long* | e0,r0,nn,mm | type pairs |
| *ufm* | epsilon,sigma,scale | type pairs |
| *soft* | a | type pairs |

Note that it is easy to add new potentials and their parameters to this list. All it typically takes is adding an extract() method to the pair_*.cpp file associated with the potential.

Similar to the *pair_coeff* command, I and J can be specified in one of two ways. Explicit numeric values can be used for each, as in the first example above. I $\leq$ J is required. LAMMPS sets the coefficients for the symmetric J,I interaction to the same values. A wild-card asterisk can be used in place of or in conjunction with the I,J arguments to set the coefficients for multiple pairs of atom types. This takes the form "*" or "*n" or "m*" or "m*n". If *N* is the number of atom types, then an asterisk with no numeric values means all types from 1 to *N*. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from m to N (inclusive). A middle asterisk means all types from m to n (inclusive). Note that only type pairs with I $\leq$ J are considered; if asterisks imply type pairs where J $<$ I, they are ignored.

If *pair_style hybrid or hybrid/overlay* is being used, then the *pstyle* will be a sub-style name. You must specify I,J arguments that correspond to type pair values defined (via the *pair_coeff* command) for that sub-style.

The *v_name* argument for keyword *pair* is the name of an *equal-style variable* which will be evaluated each time this compute is invoked. It should be specified as v_name, where name is the variable name.

---

The *atom* attribute enables atom properties to be changed. The *aparam* argument is the name of the parameter to change. This is the current list of atom parameters that can be used with this compute:

- charge = charge on particle

The *v_name* argument for keyword *pair* is the name of an *equal-style variable* which will be evaluated each time this compute is invoked. It should be specified as v_name, where name is the variable name.

---

The *tail* keyword controls the calculation of the tail correction to "van der Waals" pair energies beyond the cutoff, if this has been activated via the *pair_modify* command. If the perturbation is small, the tail contribution to the energy difference between the reference and perturbed systems should be negligible.

If the keyword *volume* = *yes*, then the Boltzmann term is multiplied by the volume so that correct ensemble averaging can be performed over trajectories during which the volume fluctuates or changes *(Allen and Tildesley)*:

$$\Delta_0^1 A = -k_B T \sum_{i=0}^{n-1} \ln \frac{\left\langle V \exp\left(-\frac{U(\lambda_{i+1})-U(\lambda_i)}{k_B T}\right)\right\rangle_{\lambda_i}}{\langle V \rangle_{\lambda_i}}$$

---

## 3.47.4 Output info

This compute calculates a global vector of length 3 which contains the energy difference ( $U_1 - U_0$ ) as c_ID[1], the Boltzmann factor $\exp(-(U_1 - U_0)/k_B T)$, or $V \exp(-(U_1 - U_0)/k_B T)$, as c_ID[2] and the volume of the simulation box $V$ as c_ID[3]. $U_1$ is the pair potential energy obtained with the perturbed parameters and $U_0$ is the pair potential energy obtained with the unperturbed parameters. The energies include kspace terms if these are used in the simulation.

These output results can be used by any command that uses a global scalar or vector from a compute as input. See the *Howto output* page for an overview of LAMMPS output options. For example, the computed values can be averaged using *fix ave/time*.

The values calculated by this compute are "extensive".

## 3.47.5 Restrictions

This compute is distributed as the FEP package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.47.6 Related commands

*fix adapt/fep*, *fix ave/time*, *pair_style . . . /soft*

### 3.47.7 Default

The option defaults are *tail* = *no*, *volume* = *no*.

---

**(Pearlman)** Pearlman, J Chem Phys, 98, 1487 (1994)

**(Mezei)** Mezei, J Chem Phys, 86, 7084 (1987)

**(Bennet)** Bennet, J Comput Phys, 22, 245 (1976)

**(BoreschKarplus)** Boresch and Karplus, J Phys Chem A, 103, 103 (1999)

**(AllenTildesley)** Allen and Tildesley, Computer Simulation of Liquids, Oxford University Press (1987)

## 3.48 compute fep/ta command

### 3.48.1 Syntax

```
compute ID group-ID fep/ta temp plane scale_factor keyword value ...
```

- ID, group-ID are documented in the *compute* command
- fep/ta = name of this compute command
- temp = external temperature (as specified for constant-temperature run)
- plane = *xy* or *xz* or *yz*
- scale_factor = multiplicative factor for change in plane area
- zero or more keyword/value pairs may be appended
- keyword = *tail*

```
tail value = no or yes
  no = ignore tail correction to pair energies (usually small in fep)
  yes = include tail correction to pair energies
```

### 3.48.2 Examples

```
compute 1 all fep/ta 298 xy 1.0005
```

### 3.48.3 Description

Added in version 4May2022.

Define a computation that calculates the change in the free energy due to a test-area (TA) perturbation *(Gloor)*. The test-area approach can be used to determine the interfacial tension of the system in a single simulation:

$$\gamma = \lim_{\Delta\mathcal{A}\to 0}\left(\frac{\Delta A_{0\to 1}}{\Delta\mathcal{A}}\right)_{N,V,T} = -\frac{k_B T}{\Delta\mathcal{A}}\ln\left\langle \exp\left(\frac{-(U_1-U_0)}{k_B T}\right)\right\rangle_0$$

During the perturbation, both axes of *plane* are scaled by multiplying $\sqrt{\text{scale_factor}}$, while the other axis divided by scale_factor such that the overall volume of the system is maintained.

The *tail* keyword controls the calculation of the tail correction to "van der Waals" pair energies beyond the cutoff, if this has been activated via the *pair_modify* command. If the perturbation is small, the tail contribution to the energy difference between the reference and perturbed systems should be negligible.

---

### 3.48.4 Output info

This compute calculates a global vector of length 3 which contains the energy difference ($U_1 - U_0$) as c_ID[1], the Boltzmann factor $\exp\left(-(U_1-U_0)/k_BT\right)$, as c_ID[2] and the change in the *plane* area $\Delta\mathcal{A}$ as c_ID[3]. $U_1$ is the potential energy of the perturbed state and $U_0$ is the potential energy of the reference state. The energies include kspace terms if these are used in the simulation.

These output results can be used by any command that uses a global scalar or vector from a compute as input. See the *Howto output* page for an overview of LAMMPS output options. For example, the computed values can be averaged using *fix ave/time*.

### 3.48.5 Restrictions

Constraints, like fix shake, may lead to incorrect values for energy difference.

This compute is distributed as the FEP package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.48.6 Related commands

*compute fep*

### 3.48.7 Default

The option defaults are *tail* = *no*.

---

**(Gloor)** Gloor, J Chem Phys, 123, 134703 (2005)

## 3.49 compute gaussian/grid/local command

Accelerator Variants: *gaussian/grid/local/kk*

### 3.49.1 Syntax

```
compute ID group-ID gaussian/grid/local grid nx ny nz rcutfac  R_1 R_2 ... sigma_1 sigma_
↪2
```

- ID, group-ID are documented in *compute* command
- gaussian/grid/local = style name of this compute command
- *grid* values = nx, ny, nz, number of grid points in x, y, and z directions (positive integer)
- *rcutfac* = scale factor applied to all cutoff radii (positive real)
- *R_1, R_2,...* = list of cutoff radii, one for each type (distance units)
- *sigma_1, sigma_2,...* = Gaussian widths, one for each type (distance units)

### 3.49.2 Examples

```
compute mygrid all gaussian/grid/local grid 40 40 40 4.0 0.5 0.5 0.4 0.4
```

### 3.49.3 Description

Added in version 4Feb2025.

Define a computation that calculates a Gaussian representation of the ionic structure. This representation is used for the efficient evaluation of quantities related to the structure factor in a grid-based workflow, such as the ML-DFT workflow MALA *(Ellis)*, for which it was originally implemented. Usage of the workflow is described in a separate publication *(Fiedler)*.

For each LAMMPS type, a separate sum of Gaussians is calculated, using a separate Gaussian broadening per type. The computation is always performed on the numerical grid, no atom-based version of this compute exists. The Gaussian representation can only be executed in a local fashion, thus the output array only contains rows for grid points that are local to the processor subdomain. The layout of the grid is the same as for the see *sna/grid/local* command.

Namely, the array contains one row for each of the local grid points, looping over the global index *ix* fastest, then *iy*, and *iz* slowest. Each row of the array contains the global indexes *ix*, *iy*, and *iz* first, followed by the *x*, *y*, and *z* coordinates of the grid point, followed by the values of the Gaussians (one floating point number per type per grid point).

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 3.49.4 Output info

Compute *gaussian/grid/local* evaluates a local array. The array contains one row for each of the local grid points, looping over the global index *ix* fastest, then *iy*, and *iz* slowest. The array contains math $ntypes + 6$ columns, where *ntypes* is the number of LAMMPS types. The first three columns are the global indexes *ix*, *iy*, and *iz*, followed by the *x*, *y*, and *z* coordinates of the grid point, followed by the *ntypes* columns containing the values of the Gaussians for each type.

### 3.49.5 Restrictions

These computes are part of the ML-SNAP package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.49.6 Related commands

*compute sna/grid/local*

---

**(Ellis)** Ellis, Fiedler, Popoola, Modine, Stephens, Thompson, Cangi, Rajamanickam, Phys. Rev. B, 104, 035120, (2021)

**(Fiedler)** Fiedler, Modine, Schmerler, Vogel, Popoola, Thompson, Rajamanickam, and Cangi, npj Comp. Mater., 9, 115 (2023)

# 3.50 compute global/atom command

## 3.50.1 Syntax

```
compute ID group-ID style index input1 input2 ...
```

- ID, group-ID are documented in *compute* command
- global/atom = style name of this compute command
- index = c_ID, c_ID[N], f_ID, f_ID[N], v_name

```
c_ID = per-atom vector calculated by a compute with ID
c_ID[I] = Ith column of per-atom array calculated by a compute with ID
f_ID = per-atom vector calculated by a fix with ID
f_ID[I] = Ith column of per-atom array calculated by a fix with ID
v_name = per-atom vector calculated by an atom-style variable with name
```

- one or more inputs can be listed
- input = c_ID, c_ID[N], f_ID, f_ID[N], v_name

```
c_ID = global vector calculated by a compute with ID
c_ID[I] = Ith column of global array calculated by a compute with ID, I can include␣
↪wildcard (see below)
f_ID = global vector calculated by a fix with ID
f_ID[I] = Ith column of global array calculated by a fix with ID, I can include␣
↪wildcard (see below)
v_name = global vector calculated by a vector-style variable with name
```

## 3.50.2 Examples

```
compute 1 all global/atom c_chunk c_com[1] c_com[2] c_com[3]
compute 1 all global/atom c_chunk c_com[*]
```

## 3.50.3 Description

Define a calculation that assigns global values to each atom from vectors or arrays of global values. The specified *index* parameter is used to determine which global value is assigned to each atom.

The *index* parameter must reference a per-atom vector or array from a *compute* or *fix* or the evaluation of an atom-style *variable*. Each *input* value must reference a global vector or array from a *compute* or *fix* or the evaluation of an vector-style *variable*. Details are given below.

The *index* value for an atom is used as an index $I$ (from 1 to $N$, where $N$ is the number of atoms) into the vector associated with each of the input values. The $I$th value from the input vector becomes one output value for that atom. If the atom is not in the specified group, or the index $I < 1$ or $I > M$, where $M$ is the actual length of the input vector, then an output value of 0.0 is assigned to the atom.

An example of how this command is useful, is in the context of "chunks" which are static or dynamic subsets of atoms. The *compute chunk/atom* command assigns unique chunk IDs to each atom. Its output can be used as the *index* parameter for this command. Various other computes with "chunk" in their style name, such as *compute com/chunk* or *compute msd/chunk*, calculate properties for each chunk. The output of these commands are global vectors or arrays, with one or more values per chunk, and can be used as input values for this command. This command will then assign the global chunk value to each atom in the chunk, producing a per-atom vector or per-atom array as output. The per-atom values can then be output to a dump file or used by any command that uses per-atom values from a compute as input, as discussed on the *Howto output* doc page.

As a concrete example, these commands will calculate the displacement of each atom from the center-of-mass of the molecule it is in, and dump those values to a dump file. In this case, each molecule is a chunk.

```
compute cc1 all chunk/atom molecule
compute myChunk all com/chunk cc1
compute prop all property/atom xu yu zu
compute glob all global/atom c_cc1 c_myChunk[*]
variable dx atom c_prop[1]-c_glob[1]
variable dy atom c_prop[2]-c_glob[2]
variable dz atom c_prop[3]-c_glob[3]
variable dist atom sqrt(v_dx*v_dx+v_dy*v_dy+v_dz*v_dz)
dump 1 all custom 100 tmp.dump id xu yu zu c_glob[1] c_glob[2] c_glob[3] &
     v_dx v_dy v_dz v_dist
dump_modify 1 sort id
```

You can add these commands to the bench/in.chain script to see how they work.

---

Note that for input values from a compute or fix, the bracketed index I can be specified using a wildcard asterisk with the index to effectively specify multiple values. This takes the form "*" or "*n" or "m*" or "m*n". If *N* is the size of the vector (for *mode* = scalar) or the number of columns in the array (for *mode* = vector), then an asterisk with no numeric values means all indices from 1 to *N*. A leading asterisk means all indices from 1 to n (inclusive). A trailing asterisk means all indices from m to *N* (inclusive). A middle asterisk means all indices from m to n (inclusive).

Using a wildcard is the same as if the individual columns of the array had been listed one by one. For example, the following two compute global/atom commands are equivalent, since the *compute com/chunk* command creates a global array with three columns:

```
compute cc1 all chunk/atom molecule
compute com all com/chunk cc1
compute 1 all global/atom c_cc1 c_com[1] c_com[2] c_com[3]
compute 1 all global/atom c_cc1 c_com[*]
```

---

This section explains the *index* parameter. Note that it must reference per-atom values, as contrasted with the *input* values, which must reference global values.

Note that all of these options generate floating point values. When they are used as an index into the specified input vectors, they simple rounded down to convert the value to integer indices. The final values should range from 1 to *N* (inclusive), since they are used to access values from *N*-length vectors.

If *index* begins with "c_", a compute ID must follow which has been previously defined in the input script. The compute must generate per-atom quantities. See the individual *compute* doc page for details. If no bracketed integer is appended, the per-atom vector calculated by the compute is used. If a bracketed integer is appended, the Ith column of the per-atom array calculated by the compute is used. Users can also write code for their own compute styles and *add them to LAMMPS*. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

If *index* begins with "f_", a fix ID must follow which has been previously defined in the input script. The Fix must generate per-atom quantities. See the individual *fix* page for details. Note that some fixes only produce their values on certain timesteps, which must be compatible with when compute global/atom references the values, else an error results. If no bracketed integer is appended, the per-atom vector calculated by the fix is used. If a bracketed integer is appended, the Ith column of the per-atom array calculated by the fix is used. Users can also write code for their own fix style and *add them to LAMMPS*. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

If *index* begins with "v_", a variable name must follow which has been previously defined in the input script. It must be an *atom-style variable*. Atom-style variables can reference thermodynamic keywords and various per-atom attributes, or invoke other computes, fixes, or variables when they are evaluated, so this is a very general means of generating per-atom quantities to use as *index*.

---

This section explains the kinds of *input* values that can be used. Note that inputs reference global values, as contrasted with the *index* parameter which must reference per-atom values.

If a value begins with "c_", a compute ID must follow which has been previously defined in the input script. The compute must generate a global vector or array. See the individual *compute* doc page for details. If no bracketed integer is appended, the vector calculated by the compute is used. If a bracketed integer is appended, the Ith column of the array calculated by the compute is used. Users can also write code for their own compute styles and *add them to LAMMPS*. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

If a value begins with "f_", a fix ID must follow which has been previously defined in the input script. The fix must generate a global vector or array. See the individual *fix* doc page for details. Note that some fixes only produce their values on certain timesteps, which must be compatible with when compute global/atom references the values, else an error results. If no bracketed integer is appended, the vector calculated by the fix is used. If a bracketed integer is appended, the Ith column of the array calculated by the fix is used. Users can also write code for their own fix style and *add them to LAMMPS*. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

If a value begins with "v_", a variable name must follow which has been previously defined in the input script. It must be a *vector-style variable*. Vector-style variables can reference thermodynamic keywords and various other attributes of atoms, or invoke other computes, fixes, or variables when they are evaluated, so this is a very general means of generating a vector of global quantities which the *index* parameter will reference for assignment of global values to atoms.

---

### 3.50.4 Output info

If a single input is specified this compute produces a per-atom vector. If multiple inputs are specified, this compute produces a per-atom array values, where the number of columns is equal to the number of inputs specified. These values can be used by any command that uses per-atom vector or array values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-atom vector or array values will be in whatever units the corresponding input values are in.

### 3.50.5 Restrictions

none

### 3.50.6 Related commands

*compute*, *fix*, *variable*, *compute chunk/atom*, *compute reduce*

### 3.50.7 Default

none

## 3.51 compute group/group command

### 3.51.1 Syntax

```
compute ID group-ID group/group group2-ID keyword value ...
```

- ID, group-ID are documented in *compute* command
- group/group = style name of this compute command
- group2-ID = group ID of second (or same) group
- zero or more keyword/value pairs may be appended
- keyword = *pair* or *kspace* or *boundary* or *molecule*

```
pair value = yes or no
kspace value = yes or no
boundary value = yes or no
molecule value = off or inter or intra
```

### 3.51.2 Examples

```
compute 1 lower group/group upper
compute 1 lower group/group upper kspace yes
compute mine fluid group/group wall
```

### 3.51.3 Description

Define a computation that calculates the total energy and force interaction between two groups of atoms: the compute group and the specified group2. The two groups can be the same.

If the *pair* keyword is set to *yes*, which is the default, then the the interaction energy will include a pair component which is defined as the pairwise energy between all pairs of atoms where one atom in the pair is in the first group and the other is in the second group. Likewise, the interaction force calculated by this compute will include the force on the compute group atoms due to pairwise interactions with atoms in the specified group2.

**Note**

The energies computed by the *pair* keyword do not include tail corrections, even if they are enabled via the *pair_modify* command.

If the *molecule* keyword is set to *inter* or *intra* than an additional check is made based on the molecule IDs of the two atoms in each pair before including their pairwise interaction energy and force. For the *inter* setting, the two atoms must be in different molecules. For the *intra* setting, the two atoms must be in the same molecule.

If the *kspace* keyword is set to *yes*, which is not the default, and if a *kspace_style* is defined, then the interaction energy will include a Kspace component which is the long-range Coulombic energy between all the atoms in the first group and all the atoms in the second group. Likewise, the interaction force calculated by this compute will include the force on the compute group atoms due to long-range Coulombic interactions with atoms in the specified group2.

Normally the long-range Coulombic energy converges only when the net charge of the unit cell is zero. However, one can assume the net charge of the system is neutralized by a uniform background plasma, and a correction to the system energy can be applied to reduce artifacts. For more information see *(Bogusz)*. If the *boundary* keyword is set to *yes*, which is the default, and *kspace* contributions are included, then this energy correction term will be added to the total group-group energy. This correction term does not affect the force calculation and will be zero if one or both of the groups are charge neutral. This energy correction term is the same as that included in the regular Ewald and PPPM routines.

**Note**

The *molecule* setting only affects the group/group contributions calculated by the *pair* keyword. It does not affect the group/group contributions calculated by the *kspace* keyword.

This compute does not calculate any bond or angle or dihedral or improper interactions between atoms in the two groups.

---

The pairwise contributions to the group-group interactions are calculated by looping over a neighbor list. The Kspace contribution to the group-group interactions require essentially the same amount of work (FFTs, Ewald summation) as computing long-range forces for the entire system. Thus it can be costly to invoke this compute too frequently.

**Note**

If you have a bonded system, then the settings of *special_bonds* command can remove pairwise interactions between atoms in the same bond, angle, or dihedral. This is the default setting for the *special_bonds* command, and means those pairwise interactions do not appear in the neighbor list. Because this compute uses a neighbor list, it also means those pairs will not be included in the group/group interaction. This does not apply when using long-range Coulomb interactions (*coul/long*, *coul/msm*, *coul/wolf* or similar). One way to get around this would be to set *special_bond* scaling factors to very tiny numbers that are not exactly zero (e.g., $1.0 \times 10^{-50}$). Another workaround would be to write a dump file and use the *rerun* command to compute the group/group interactions for snapshots in the dump file. The rerun script can use a *special_bonds* command that includes all pairs in the neighbor list.

If you desire a breakdown of the interactions into a pairwise and Kspace component, simply invoke the compute twice with the appropriate yes/no settings for the *pair* and *kspace* keywords. This is no more costly than using a single compute with both keywords set to *yes*. The individual contributions can be summed in a *variable* if desired.

This document describes how the long-range group-group calculations are performed.

---

### 3.51.4 Output info

This compute calculates a global scalar (the energy) and a global vector of length 3 (force), which can be accessed by indices 1–3. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

Both the scalar and vector values calculated by this compute are “extensive”. The scalar value will be in energy *units*. The vector values will be in force *units*.

### 3.51.5 Restrictions

Not all pair styles can be evaluated in a pairwise mode as required by this compute. For example, three-body and other many-body potentials, such as *Tersoff* and *Stillinger-Weber* cannot be used. *EAM* potentials will re-use previously

computed embedding term contributions, so the computed pairwise forces and energies are based on the whole system and not valid if particles have been moved since.

Not all *Kspace styles* support the calculation of group/group interactions. The regular *ewald* and *pppm* styles do.

### 3.51.6 Related commands

none

### 3.51.7 Default

The option defaults are pair = yes, kspace = no, boundary = yes, molecule = off.

---

Bogusz et al, J Chem Phys, 108, 7070 (1998)

## 3.52 compute gyration command

### 3.52.1 Syntax

```
compute ID group-ID gyration
```

- ID, group-ID are documented in *compute* command
- gyration = style name of this compute command

### 3.52.2 Examples

```
compute 1 molecule gyration
```

### 3.52.3 Description

Define a computation that calculates the radius of gyration $R_g$ of the group of atoms, including all effects due to atoms passing through periodic boundaries.

$R_g$ is a measure of the size of the group of atoms, and is computed as the square root of the $R_g^2$ value in this formula

$$R_g^2 = \frac{1}{M} \sum_i m_i (r_i - r_{\rm cm})^2$$

where $M$ is the total mass of the group, $r_{\rm cm}$ is the center-of-mass position of the group, and the sum is over all atoms in the group.

A $R_g^2$ tensor, stored as a 6-element vector, is also calculated by this compute. The formula for the components of the tensor is the same as the above formula, except that $(r_i - r_{\rm cm})^2$ is replaced by $(r_{i,x} - r_{{\rm cm},x}) \cdot (r_{i,y} - r_{{\rm cm},y})$ for the $xy$ component, and so on. The six components of the vector are ordered $xx$, $yy$, $zz$, $xy$, $xz$, $yz$. Note that unlike the scalar $R_g$, each of the six values of the tensor is effectively a "squared" value, since the cross-terms may be negative and taking a square root would be invalid.

**Note**

The coordinates of an atom contribute to $R_g$ in "unwrapped" form, by using the image flags associated with each atom. See the *dump custom* command for a discussion of "unwrapped" coordinates. See the Atoms section of the *read_data* command for a discussion of image flags and how they are set for each atom. You can reset the image flags (e.g., to 0) before invoking this compute by using the *set image* command.

### 3.52.4 Output info

This compute calculates a global scalar ($R_g$) and a global vector of length 6 ($R_g^2$ tensor), which can be accessed by indices 1–6. These values can be used by any command that uses a global scalar value or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The scalar and vector values calculated by this compute are "intensive". The scalar and vector values will be in distance and distance$^2$ *units*, respectively.

### 3.52.5 Restrictions

none

### 3.52.6 Related commands

*compute gyration/chunk*, *compute gyration/shape*

### 3.52.7 Default

none

## 3.53 compute gyration/chunk command

### 3.53.1 Syntax

```
compute ID group-ID gyration/chunk chunkID keyword value ...
```

- ID, group-ID are documented in *compute* command
- gyration/chunk = style name of this compute command
- chunkID = ID of *compute chunk/atom* command
- zero or more keyword/value pairs may be appended
- keyword = *tensor*

  *tensor* value = none

### 3.53.2 Examples

```
compute 1 molecule gyration/chunk molchunk
compute 2 molecule gyration/chunk molchunk tensor
```

### 3.53.3 Description

Define a computation that calculates the radius of gyration $R_g$ for multiple chunks of atoms.

In LAMMPS, chunks are collections of atoms defined by a *compute chunk/atom* command, which assigns each atom to a single chunk (or no chunk). The ID for this command is specified as chunkID. For example, a single chunk could be the atoms in a molecule or atoms in a spatial bin. See the *compute chunk/atom* and *Howto chunk* doc pages for details of how chunks can be defined and examples of how they can be used to measure properties of a system.

This compute calculates the radius of gyration $R_g$ for each chunk, which includes all effects due to atoms passing through periodic boundaries.

$R_g$ is a measure of the size of a chunk, and is computed by the formula

$$R_g^2 = \frac{1}{M} \sum_i m_i (r_i - r_{\rm cm})^2$$

where $M$ is the total mass of the chunk, $r_{\rm cm}$ is the center-of-mass position of the chunk, and the sum is over all atoms in the chunk.

Note that only atoms in the specified group contribute to the calculation. The *compute chunk/atom* command defines its own group; atoms will have a chunk ID = 0 if they are not in that group, signifying they are not assigned to a chunk, and will thus also not contribute to this calculation. You can specify the "all" group for this command if you simply want to include atoms with non-zero chunk IDs.

If the *tensor* keyword is specified, then the scalar $R_g$ value is not calculated, but an $R_g$ tensor is instead calculated for each chunk. The formula for the components of the tensor is the same as the above formula, except that $(r_i - r_{\rm cm})^2$ is replaced by $(r_{i,x} - r_{{\rm cm},x}) \cdot (r_{i,y} - r_{{\rm cm},y})$ for the $xy$ component, and so on. The six components of the tensor are ordered $xx$, $yy$, $zz$, $xy$, $xz$, $yz$.

**Note**

The coordinates of an atom contribute to $R_g$ in "unwrapped" form, by using the image flags associated with each atom. See the *dump custom* command for a discussion of "unwrapped" coordinates. See the Atoms section of the *read_data* command for a discussion of image flags and how they are set for each atom. You can reset the image flags (e.g., to 0) before invoking this compute by using the *set image* command.

The simplest way to output the results of the compute gyration/chunk calculation to a file is to use the *fix ave/time* command, for example:

```
compute cc1 all chunk/atom molecule
compute myChunk all gyration/chunk cc1
fix 1 all ave/time 100 1 100 c_myChunk file tmp.out mode vector
```

### 3.53.4 Output info

This compute calculates a global vector if the *tensor* keyword is not specified and a global array if it is. The length of the vector or number of rows in the array = the number of chunks *Nchunk* as calculated by the specified *compute chunk/atom* command. If the *tensor* keyword is specified, the global array has six columns. The vector or array can be accessed by any command that uses global values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

All the vector or array values calculated by this compute are "intensive". The vector or array values will be in distance *units*, since they are the square root of values represented by the formula above.

### 3.53.5 Restrictions

none

### 3.53.6 Related commands

none

*compute gyration*

### 3.53.7 Default

none

## 3.54 compute gyration/shape command

### 3.54.1 Syntax

```
compute ID group-ID gyration/shape compute-ID
```

- ID, group-ID are documented in *compute* command
- gyration/shape = style name of this compute command
- compute-ID = ID of *compute gyration* command

### 3.54.2 Examples

```
compute 1 molecule gyration/shape pe
```

### 3.54.3 Description

Define a computation that calculates the eigenvalues of the gyration tensor of a group of atoms and three shape parameters. The computation includes all effects due to atoms passing through periodic boundaries.

The three computed shape parameters are the asphericity, $b$, the acylindricity, $c$, and the relative shape anisotropy, $k$, viz.,

$$b = l_z - \frac{1}{2}(l_y + l_x)$$
$$c = l_y - l_x$$
$$k = \frac{3}{2}\frac{l_x^2 + l_y^2 + l_z^2}{(l_x + l_y + l_z)^2} - \frac{1}{2}$$

where $l_x \leq l_y \leq l_z$ are the three eigenvalues of the gyration tensor. A general description of these parameters is provided in *(Mattice)* while an application to polymer systems can be found in *(Theodorou)*. The asphericity is always non-negative and zero only when the three principal moments are equal. This zero condition is met when the distribution of particles is spherically symmetric (hence the name asphericity) but also whenever the particle distribution is symmetric with respect to the three coordinate axes (e.g., when the particles are distributed uniformly on a cube, tetrahedron or other Platonic solid). The acylindricity is always non-negative and zero only when the two principal moments are equal. This zero condition is met when the distribution of particles is cylindrically symmetric (hence the name, acylindricity), but also whenever the particle distribution is symmetric with respect to the two coordinate axes (e.g., when the particles are distributed uniformly on a regular prism). The relative shape anisotropy is bounded between zero (if all points are spherically symmetric) and one (if all points lie on a line).

**Note**

The coordinates of an atom contribute to the gyration tensor in "unwrapped" form, by using the image flags associated with each atom. See the *dump custom* command for a discussion of "unwrapped" coordinates. See the Atoms section of the *read_data* command for a discussion of image flags and how they are set for each atom. You can reset the image flags (e.g., to 0) before invoking this compute by using the *set image* command.

### 3.54.4 Output info

This compute calculates a global vector of length 6, which can be accessed by indices 1–6. The first three values are the eigenvalues of the gyration tensor followed by the asphericity, the acylindricity and the relative shape anisotropy. The computed values can be used by any command that uses global vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The vector values calculated by this compute are "intensive". The first five vector values will be in distance2 *units* while the sixth one is dimensionless.

### 3.54.5 Restrictions

This compute is part of the EXTRA-COMPUTE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.54.6 Related commands

*compute gyration*

### 3.54.7 Default

none

---

**(Mattice)** Mattice, Suter, Conformational Theory of Large Molecules, Wiley, New York, 1994.

**(Theodorou)** Theodorou, Suter, Macromolecules, 18, 1206 (1985).

## 3.55 compute gyration/shape/chunk command

### 3.55.1 Syntax

```
compute ID group-ID gyration/shape/chunk compute-ID
```

- ID, group-ID are documented in *compute* command
- gyration/shape/chunk = style name of this compute command
- compute-ID = ID of *compute gyration/chunk* command

### 3.55.2 Examples

```
compute 1 molecule gyration/shape/chunk pe
```

### 3.55.3 Description

Define a computation that calculates the eigenvalues of the gyration tensor and three shape parameters of multiple chunks of atoms. The computation includes all effects due to atoms passing through periodic boundaries.

The three computed shape parameters are the asphericity, $b$, the acylindricity, $c$, and the relative shape anisotropy, $k$, viz.,

$$b = l_z - \frac{1}{2}(l_y + l_x)$$
$$c = l_y - l_x$$
$$k = \frac{3}{2}\frac{l_x^2 + l_y^2 + l_z^2}{(l_x + l_y + l_z)^2} - \frac{1}{2}$$

where $l_x \leq l_y \leq l_z$ are the three eigenvalues of the gyration tensor. A general description of these parameters is provided in *(Mattice)* while an application to polymer systems can be found in *(Theodorou)*. The asphericity is always non-negative and zero only when the three principal moments are equal. This zero condition is met when the distribution of particles is spherically symmetric (hence the name asphericity) but also whenever the particle distribution is symmetric with respect to the three coordinate axes (e.g., when the particles are distributed uniformly on a cube, tetrahedron, or other Platonic solid). The acylindricity is always non-negative and zero only when the two principal moments are equal. This zero condition is met when the distribution of particles is cylindrically symmetric (hence the name, acylindricity), but also whenever the particle distribution is symmetric with respect to the two coordinate axes (e.g., when the particles are distributed uniformly on a regular prism). The relative shape anisotropy is bounded between 0 (if all points are spherically symmetric) and 1 (if all points lie on a line).

The tensor keyword must be specified in the compute gyration/chunk command.

**Note**

The coordinates of an atom contribute to the gyration tensor in "unwrapped" form, by using the image flags associated with each atom. See the *dump custom* command for a discussion of "unwrapped" coordinates. See the Atoms section of the *read_data* command for a discussion of image flags and how they are set for each atom. You can reset the image flags (e.g., to 0) before invoking this compute by using the *set image* command.

### 3.55.4 Output info

This compute calculates a global array with six columns, which can be accessed by indices 1–6. The first three columns are the eigenvalues of the gyration tensor followed by the asphericity, the acylindricity and the relative shape anisotropy. The computed values can be used by any command that uses global array values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The array calculated by this compute is "intensive". The first five columns will be in distance$^2$ *units* while the sixth one is dimensionless.

### 3.55.5 Restrictions

This compute is part of the EXTRA-COMPUTE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.55.6 Related commands

*compute gyration/chunk compute gyration/shape*

### 3.55.7 Default

none

---

**(Mattice)** Mattice, Suter, Conformational Theory of Large Molecules, Wiley, New York, 1994.

**(Theodorou)** Theodorou, Suter, Macromolecules, 18, 1206 (1985).

## 3.56 compute hbond/local command

### 3.56.1 Syntax

```
compute ID group-ID hbond/local rcut acut dgroup-ID agroup-ID hgroup-ID value1 value2 ...
```

- ID, group-ID are documented in *compute* command
- hbond/local = style name of this compute command
- rcut = distance cutoff between hydrogen bond donor and acceptor (distance units)
- acut = angle cutoff for the hydrogen - donor - acceptor angle (degrees)
- dgroup-ID = group-ID of the hydrogen bond donor atoms
- agroup-ID = group-ID of the hydrogen bond acceptor atoms
- hgroup-ID = group-ID of the hydrogen bond hydrogen atoms
- zero or more values may be appended to select which additional properties to compute and provide
- value = *dist* or *angle* or *hdist* or *ehb*

```
dist = distance between hydrogen bond donor and acceptor atom (distance units)
angle = hydrogen - donor - acceptor angle (degrees)
hdist = distance between hydrogen bond hydrogen and acceptor atom (distance␣
↪units)
ehb = hydrogen bond strength (energy units)
```

- zero or more keyword/value pair may be appended
- keyword = *ecut*

```
ecut value = minimum hydrogen bond strength cutoff (energy units)
```

### 3.56.2 Examples

```
compute hb all hbond/local 3.2 30.0 dgroup agroup hgroup
compute hb all hbond/local 3.2 30.0 oxygen oxygen hydrogen dist hdist angle ehb ecut 1.5
```

### 3.56.3 Description

Added in version 11Feb2026.

Define a computation that determines the number of hydrogen bonds and computes some related properties according to the provided parameters. To be counted as a hydrogen bond the following conditions have to be met

- the donor atom has to be in the group *dgroup-ID*
- the acceptor atom has to be in the group *agroup-ID*
- the hydrogen atom has to be in the group *hgroup-ID*
- all three atoms have to be in the compute group
- the hydrogen atom has to be connected to the donor with a bond
- the donor - acceptor distance has to be less than *rcut*
- the hydrogen - donor - acceptor angle has to be less than *acut*

The following values can be computed and output.

- The *dist* value is the current distance between the hydrogen bond donor and acceptor atom in distance units
- The *angle* value is the current hydrogen-donor-acceptor angle in degrees
- The *hdist* value is the current distance between the hydrogen atom and hydrogen bond acceptor atom in distance units

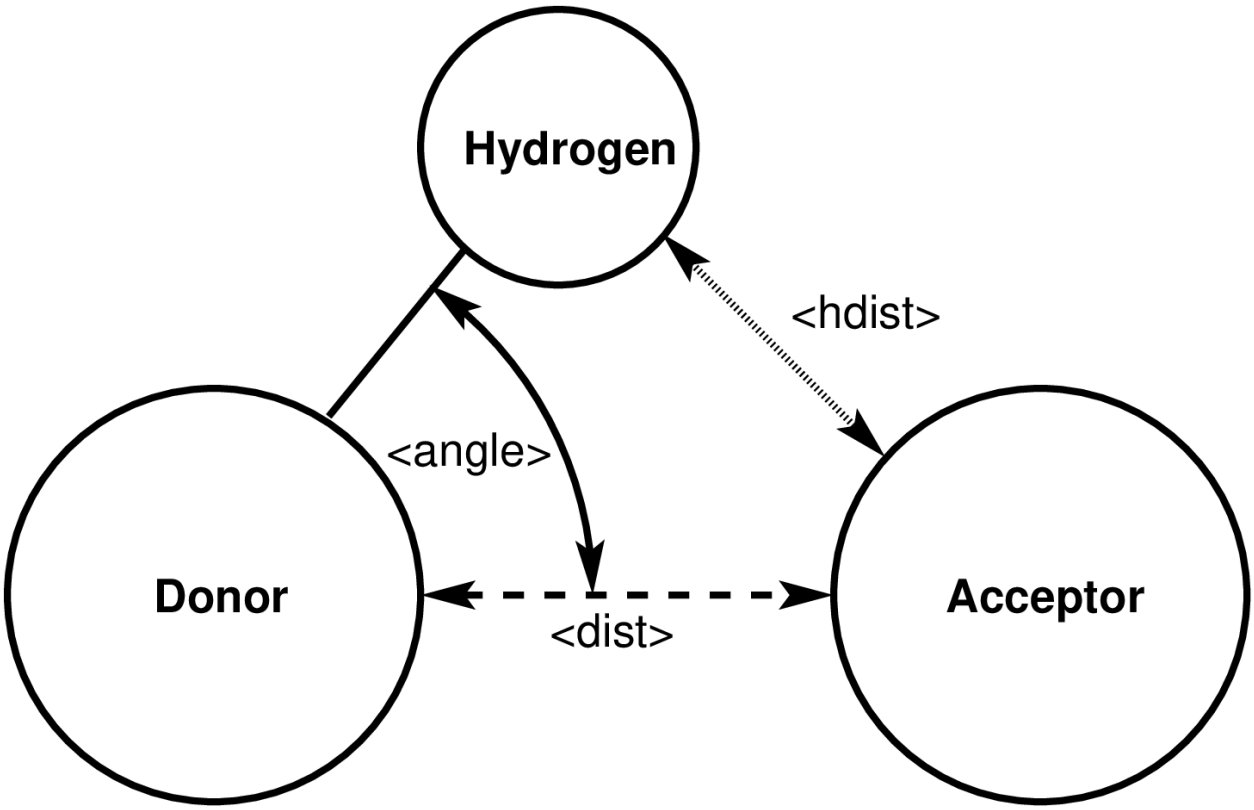



Fig. 1: Diagram of the hydrogen bond definition for compute hbond/local

- The *ehb* value is the hydrogen bond strength computed as the sum of the pairwise potential energy between a) the donor atom and the acceptor atom, and b) the hydrogen atom and the acceptor atom. This is a positive value for an attractive interaction.

If the *ecut* keyword is used, an additional energy cutoff is applied. The computed hydrogen bond strength must be larger than the *ecut* value or else the potential hydrogen bond is not counted as such. The energy cutoff is otherwise not applied.

**Restrictions for computing *ehb* and applying *ecut***

Computing the hydrogen bond strength and applying an energy cutoff for hydrogen bonds requires that the *pair_style* in use is capable of computing pair-wise energies. This is usually available for *lj/cut/coul/cut* or similar but not for most many-body and machine learning force fields.

If a *kspace solver* is used, this energy *only* contains the real-space contributions. But since the distances between the atoms are small, the missing long-range contribution should be small, too.

### 3.56.4 Output info

This compute calculates a global scalar (the number of detected hydrogen bonds summed over all MPI processes) and a local array containing in its columns in this order: the atom-ID of the hydrogen bond hydrogen atom, the atom-ID of the hydrogen bond donor atom, the atom-ID of the hydrogen bond acceptor atom, followed by the properties in the order they were selected in the compute command line. To avoid double counting, hydrogen bonds are only counted and their information stored on the MPI process where the hydrogen bond donor atom is a *local* atom; hydrogen and acceptor atoms may be ghost atoms. The number of columns is thus three plus the number of selected value to compute and store. The array can be accessed by any command that uses local data.

As an example, the commands shown below can be added to the `examples/rdf-adf/in.spce` example input file to compute and output the hydrogen bond information of the compute for a bulk water system in multiple ways.

```
group           ogroup type 1
group           hgroup type 2
variable        nmol equal count(ogroup)

# water oxygen atoms are both hydrogen bond donor and acceptor
compute         hb       all      hbond/local 3.3 30.0 ogroup ogroup hgroup dist hdist␣
↪angle ehb

# output all hydrogen bonds. each line contains: hydrogen-ID donor-ID acceptor-ID r_DA r_
↪HA theta_HDA e_hb
dump            hb       all      local 100 hbonds.dump c_hb[*]
dump_modify     hb label "HBONDS" colname 1 Hydrogen colname 2 "   Donor " colname 3␣
↪Acceptor &
                  colname 4 " r_DA  " colname 5 "  r_HA  " colname 6 "  theta " colname␣
↪7 "  e_hb"  &
                  format line "%20.0f %8.0f %8.0f %8.3f %8.3f %8.3f %8.3f"

# for the number of hydrogen bonds per molecule we must multiply by 2
# since water oxygens are in equal parts hydrogen bond donor and acceptor
variable        nhb_mol equal 2.0*c_hb/${nmol}
# get average values for distances, angles and strength for the current step
compute         avg      all      reduce ave c_hb[4] c_hb[5] c_hb[6] c_hb[7] inputs local
# get running average of those values
fix             ave      all      ave/time 100 1 100 v_nhb_mol c_avg[1] c_avg[2] c_
```

(continues on next page)

(continued from previous page)

```
↪avg[3] c_avg[4] ave running

# get histogram of hydrogen bond distance
fix       dhist all ave/histo 100 100 10000 2.3 3.3 30 c_hb[4] file hbond_histo_dist.dat␣
↪mode vector kind local
# get histogram of hydrogen bond angle
fix       ahist all ave/histo 100 100 10000 0.0 30.0 30 c_hb[6] file hbond_histo_angle.
↪dat mode vector kind local
# get histogram of hydrogen bond strength
fix       ahist all ave/histo 100 100 10000 -11.0 1.0 30 c_hb[7] file hbond_histo_eng.
↪dat mode vector kind local

# output computed global data as thermo output, first for current step, then running␣
↪averages
thermo_style    custom step temp press v_nhb_mol c_avg[*] f_ave[*]
thermo_modify   colname 4 "n_HB/mol" colname 5 "r_DA " colname 6 "r_HA " colname 7
↪"theta_HDA" colname 8 "e_HB " &
                colname 9 "<n_HB/mol>" colname 10 "<r_DA> " colname 11 "<r_HA> " colname␣
↪12 "<theta_HDA>" colname 13 "<e_HB> "
thermo          100
```

The *dump local* command will output the three atom-IDs for hydrogen bond donor, acceptor, and hydrogen atom, then donor-acceptor distance, hydrogen-donor-acceptor angle, hydrogen-acceptor distance, and hydrogen bond strength for the hydrogen bond. The *custom thermo output* includes the number of hydrogen bonds per molecule and the distances and angles averaged over the system and then over time.

See the *Visualize LAMMPS snapshots* page for examples of visualizing the computed hydrogen bonds with *dump image*.

---

The local data stored by this command is generated by three nested loops: the outer loop is over all atoms that are in the compute group and in the donor atom group, the middle loop is over all 1-2 neighbors of potential donor atoms that also are included in the compute group and the hydrogen atom group, the inner loop is over all non-bonded neighbors of the potential donor atoms that also match the compute group and the acceptor atom group. For any atom triple that matches all conditions, the donor-acceptor distance is computed and the hydrogen-donor-acceptor angle and if both are smaller than the corresponding cutoff values from the command line, the hydrogen bond is counted and its information stored. If requested, additional properties are computed and stored in the local array. Both atoms of each non-bonded pair are tried for being a hydrogen bond donor and acceptor.

Note that as atoms migrate from processor to processor, there will be no consistent ordering of the entries within the local array from one timestep to the next.

The output for *dist* and *hdist* will be in distance *units*. The output for *angle* will be in degrees. The output for *ehb* will be in energy *units*.

---

### 3.56.5 Dump image info

Compute *hbond/local* can be used with the *compute* keyword of *dump image*. The compute will add arrows based on the detected hydrogen bonds in the compute group to *dump image* so that they are included in the rendered image.

The color of the arrows is by default that of the hydrogen bond donor atom when using color styles "type" or "element". With color style "const" the default value of "white" can be changed using *dump_modify ccolor*. The transparency is by default fully opaque and can be changed with *dump_modify ctrans*.

The *cflag1* setting allows to adjust the length of the arrow. This allows for example to shrink the arrows so that the tip would otherwise be (partially) obscured by the sphere representing the hydrogen bond acceptor atom. Thus it is recommended to use a negative value of at least the atom diameter.

The *cflag2* setting allows you to adjust the radius of the rendered arrows. Since the radius of the arrows is not known by the compute and thus set to 0, it is recommended to set this flag to a value > 0.

---

### 3.56.6 Restrictions

This compute is part of the EXTRA-COMPUTE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This compute requires that the hydrogen atom of a hydrogen bond is bound to the donor atom with an explicit bond. It cannot be used with pair styles like *reaxff* where bonds are implicit.

To compute the hydrogen bond strength, the *pair style* must support computation of pair-wise forces and energies, which is generally not available for many-body and machine learning potentials.

### 3.56.7 Related commands

*dump local*, *dump image*, *compute bond/local*, *fix graphics/arrows*

### 3.56.8 Default

*ecut* = off, default outputs are the atom-IDs (in this order) for hydrogen bond hydrogen atom, donor atom, and acceptor atom.

## 3.57 compute heat/flux command

### 3.57.1 Syntax

```
compute ID group-ID heat/flux ke-ID pe-ID stress-ID
```

- ID, group-ID are documented in *compute* command
- heat/flux = style name of this compute command
- ke-ID = ID of a compute that calculates per-atom kinetic energy
- pe-ID = ID of a compute that calculates per-atom potential energy
- stress-ID = ID of a compute that calculates per-atom stress

### 3.57.2 Examples

```
compute myFlux all heat/flux myKE myPE myStress
```

### 3.57.3 Description

Define a computation that calculates the heat flux vector based on contributions from atoms in the specified group. This can be used by itself to measure the heat flux through a set of atoms (e.g., a region between two thermostatted reservoirs held at different temperatures), or to calculate a thermal conductivity using the equilibrium Green-Kubo formalism.

For other non-equilibrium ways to compute a thermal conductivity, see the *Howto kappa* doc page. These include use of the *fix thermal/conductivity* command for the Muller-Plathe method. Or the *fix heat* command which can add or subtract heat from groups of atoms.

The compute takes three arguments which are IDs of other *computes*. One calculates per-atom kinetic energy (*ke-ID*), one calculates per-atom potential energy (*pe-ID*), and the third calculates per-atom stress (*stress-ID*).

**Note**

These other computes should provide values for all the atoms in the group this compute specifies. That means the other computes could use the same group as this compute, or they can just use group "all" (or any group whose atoms are superset of the atoms in this compute's group). LAMMPS does not check for this.

In case of two-body interactions, the heat flux **J** is defined as

$$\begin{aligned}\mathbf{J} &= \frac{1}{V}\left[\sum_i e_i \mathbf{v}_i - \sum_i \mathbf{S}_i \mathbf{v}_i\right] \\ &= \frac{1}{V}\left[\sum_i e_i \mathbf{v}_i + \sum_{i<j} \left(\mathbf{F}_{ij} \cdot \mathbf{v}_j\right) \mathbf{r}_{ij}\right] \\ &= \frac{1}{V}\left[\sum_i e_i \mathbf{v}_i + \frac{1}{2}\sum_{i<j} \left(\mathbf{F}_{ij} \cdot \left(\mathbf{v}_i + \mathbf{v}_j\right)\right) \mathbf{r}_{ij}\right]\end{aligned}$$

$e_i$ in the first term of the equation is the per-atom energy (potential and kinetic). This is calculated by the computes *ke-ID* and *pe-ID*. $\mathbf{S}_i$ in the second term is the per-atom stress tensor calculated by the compute *stress-ID*. See *compute stress/atom* and *compute centroid/stress/atom* for possible definitions of atomic stress $\mathbf{S}_i$ in the case of bonded and many-body interactions. The tensor multiplies $\mathbf{v}_i$ by a $3 \times 3$ matrix to yield a vector. Note that as discussed below, the $1/V$ scaling factor in the equation for **J** is **not** included in the calculation performed by these computes; you need to add it for a volume appropriate to the atoms included in the calculation.

**Note**

The *compute pe/atom* and *compute stress/atom* commands have options for which terms to include in their calculation (pair, bond, etc). The heat flux calculation will thus include exactly the same terms. Normally you should use *compute stress/atom virial* or *compute centroid/stress/atom virial* so as not to include a kinetic energy term in the heat flux.

**Warning**

The compute *heat/flux* has been reported to produce unphysical values for angle, dihedral, improper and constraint force contributions when used with *compute stress/atom*, as discussed in *(Surblys2019)*, *(Boone)* and *(Surblys2021)*. You are strongly advised to use *compute centroid/stress/atom*, which has been implemented specifically for such cases.

**Warning**

Due to an implementation detail, the $y$ and $z$ components of heat flux from *fix rigid* contribution when computed via *compute stress/atom* are highly unphysical and should not be used.

The Green–Kubo formulas relate the ensemble average of the auto-correlation of the heat flux **J** to the thermal conductivity $\kappa$:

$$\kappa = \frac{V}{k_B T^2}\int_0^\infty \langle J_x(0) J_x(t)\rangle \, \mathrm{d}t = \frac{V}{3k_B T^2}\int_0^\infty \langle \mathbf{J}(0) \cdot \mathbf{J}(t)\rangle \, \mathrm{d}t$$

The heat flux can be output every so many timesteps (e.g., via the *thermo_style custom* command). Then as a post-processing operation, an auto-correlation can be performed, its integral estimated, and the Green–Kubo formula above evaluated.

The *fix ave/correlate* command can calculate the auto-correlation. The trap() function in the *variable* command can calculate the integral.

An example LAMMPS input script for solid argon is appended below. The result should be an average conductivity $\approx 0.29$ W/m · K.

## 3.57.4 Output info

This compute calculates a global vector of length 6. The first three components are the $x$, $y$, and $z$ components of the full heat flux vector (i.e., $J_x$, $J_y$, and $J_z$). The next three components are the $x$, $y$, and $z$ components of just the convective portion of the flux (i.e., the first term in the equation for **J**). Each component can be accessed by indices 1–6. These values can be used by any command that uses global vector values from a compute as input. See the *Howto output* documentation for an overview of LAMMPS output options.

The vector values calculated by this compute are "extensive", meaning they scale with the number of atoms in the simulation. They can be divided by the appropriate volume to get a flux, which would then be an "intensive" value, meaning independent of the number of atoms in the simulation. Note that if the compute group is "all", then the appropriate volume to divide by is the simulation box volume. However, if a group with a subset of atoms is used, it should be the volume containing those atoms.

The vector values will be in energy*velocity *units*. Once divided by a volume the units will be that of flux, namely energy/area/time *units*

## 3.57.5 Restrictions

none

## 3.57.6 Related commands

*fix thermal/conductivity*, *fix ave/correlate*, *variable*

## 3.57.7 Default

none

### Example Input File

```
# Sample LAMMPS input script for thermal conductivity of solid Ar

units       real
variable    T equal 70
variable    V equal vol
variable    dt equal 4.0
variable    p equal 200     # correlation length
variable    s equal 10      # sample interval
variable    d equal $p*$s   # dump interval
```

(continues on next page)

(continued from previous page)

```
# convert from LAMMPS real units to SI

variable    kB equal 1.3806504e-23    # [J/K] Boltzmann
variable    kCal2J equal 4186.0/6.02214e23
variable    A2m equal 1.0e-10
variable    fs2s equal 1.0e-15
variable    convert equal ${kCal2J}*${kCal2J}/${fs2s}/${A2m}

# setup problem

dimension    3
boundary     p p p
lattice      fcc 5.376 orient x 1 0 0 orient y 0 1 0 orient z 0 0 1
region       box block 0 4 0 4 0 4
create_box   1 box
create_atoms 1 box
mass         1 39.948
pair_style   lj/cut 13.0
pair_coeff   * * 0.2381 3.405
timestep     ${dt}
thermo       $d

# equilibration and thermalization

velocity     all create $T 102486 mom yes rot yes dist gaussian
fix          NVT all nvt temp $T $T 10 drag 0.2
run          8000

# thermal conductivity calculation, switch to NVE if desired

#unfix       NVT
#fix         NVE all nve

reset_timestep 0
compute      myKE all ke/atom
compute      myPE all pe/atom
compute      myStress all stress/atom NULL virial
compute      flux all heat/flux myKE myPE myStress
variable     Jx equal c_flux[1]/vol
variable     Jy equal c_flux[2]/vol
variable     Jz equal c_flux[3]/vol
fix          JJ all ave/correlate $s $p $d &
             c_flux[1] c_flux[2] c_flux[3] type auto file J0Jt.dat ave running
variable     scale equal ${convert}/${kB}/$T/$T/$V*$s*${dt}
variable     k11 equal trap(f_JJ[3])*${scale}
variable     k22 equal trap(f_JJ[4])*${scale}
variable     k33 equal trap(f_JJ[5])*${scale}
thermo_style custom step temp v_Jx v_Jy v_Jz v_k11 v_k22 v_k33
run          100000
variable     k equal (v_k11+v_k22+v_k33)/3.0
variable     ndens equal count(all)/vol
```

(continues on next page)

(continued from previous page)

```
print       "average conductivity: $k[W/mK] @ $T K, ${ndens} /A\^3"
```

**(Surblys2019)** Surblys, Matsubara, Kikugawa, Ohara, Phys Rev E, 99, 051301(R) (2019).

**(Boone)** Boone, Babaei, Wilmer, J Chem Theory Comput, 15, 5579–5587 (2019).

**(Surblys2021)** Surblys, Matsubara, Kikugawa, Ohara, J Appl Phys 130, 215104 (2021).

# 3.58 compute hexorder/atom command

Accelerator Variants: *hexorder/atom/omp*

## 3.58.1 Syntax

```
compute ID group-ID hexorder/atom keyword values ...
```

- ID, group-ID are documented in *compute* command
- hexorder/atom = style name of this compute command
- one or more keyword/value pairs may be appended

```
keyword = degree or nnn or cutoff
  cutoff value = distance cutoff
  nnn value = number of nearest neighbors
  degree value = degree n of order parameter
```

## 3.58.2 Examples

```
compute 1 all hexorder/atom
compute 1 all hexorder/atom degree 4 nnn 4 cutoff 1.2
```

## 3.58.3 Description

Define a computation that calculates $q_n$ the bond-orientational order parameter for each atom in a group. The hexatic ($n$ = 6) order parameter was introduced by *Nelson and Halperin* as a way to detect hexagonal symmetry in two-dimensional systems. For each atom, $q_n$ is a complex number (stored as two real numbers) defined as follows:

$$q_n = \frac{1}{nnn} \sum_{j=1}^{nnn} e^{ni\theta(\mathbf{r}_{ij})}$$

where the sum is over the *nnn* nearest neighbors of the central atom. The angle $\theta$ is formed by the bond vector $r_{ij}$ and the $x$ axis. $\theta$ is calculated only using the $x$ and $y$ components, whereas the distance from the central atom is calculated using all three $x$, $y$, and $z$ components of the bond vector. Neighbor atoms not in the group are included in the order parameter of atoms in the group.

The optional keyword *cutoff* defines the distance cutoff used when searching for neighbors. The default value, also the maximum allowable value, is the cutoff specified by the pair style.

The optional keyword *nnn* defines the number of nearest neighbors used to calculate $q_n$. The default value is 6. If the value is NULL, then all neighbors up to the distance cutoff are used.

The optional keyword *degree* sets the degree $n$ of the order parameter. The default value is 6. For a perfect hexagonal lattice with *nnn* = 6, $q_6 = e^{6i\phi}$ for all atoms, where the constant $0 < \phi < \frac{\pi}{3}$ depends only on the orientation of the

lattice relative to the $x$ axis. In an isotropic liquid, local neighborhoods may still exhibit weak hexagonal symmetry, but because the orientational correlation decays quickly with distance, the value of $\phi$ will be different for different atoms, and so when $q_6$ is averaged over all the atoms in the system, $|\langle q_6 \rangle| << 1$.

The value of $q_n$ is set to zero for atoms not in the specified compute group, as well as for atoms that have less than *nnn* neighbors within the distance cutoff.

The neighbor list needed to compute this quantity is constructed each time the calculation is performed (i.e. each time a snapshot of atoms is dumped). Thus it can be inefficient to compute/dump this quantity too frequently.

**Note**

If you have a bonded system, then the settings of *special_bonds* command can remove pairwise interactions between atoms in the same bond, angle, or dihedral. This is the default setting for the *special_bonds* command, and means those pairwise interactions do not appear in the neighbor list. Because this fix uses the neighbor list, it also means those pairs will not be included in the order parameter. This difficulty can be circumvented by writing a dump file, and using the *rerun* command to compute the order parameter for snapshots in the dump file. The rerun script can use a *special_bonds* command that includes all pairs in the neighbor list.

### 3.58.4 Output info

This compute calculates a per-atom array with 2 columns, giving the real and imaginary parts $q_n$, a complex number restricted to the unit disk of the complex plane (i.e., $\Re(q_n)^2 + \Im(q_n)^2 \leq 1$).

These values can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 3.58.5 Restrictions

This compute is part of the EXTRA-COMPUTE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.58.6 Related commands

*compute orientorder/atom*, *compute coord/atom*, *compute centro/atom*

### 3.58.7 Default

The option defaults are *cutoff* = pair style cutoff, *nnn* = 6, *degree* = 6

---

**(Nelson)** Nelson, Halperin, Phys Rev B, 19, 2457 (1979).

# 3.59 compute hma command

## 3.59.1 Syntax

```
compute ID group-ID hma temp-ID keyword ...
```

- ID, group-ID are documented in *compute* command
- hma = style name of this compute command
- temp-ID = ID of fix that specifies the set temperature during canonical simulation
- one or more keywords or keyword/argument pairs must be appended
- keyword = *anharmonic* or *u* or *p* or *cv*

```
anharmonic = compute will return anharmonic property values
u = compute will return potential energy
p value = Pharm = compute will return pressure
   Pharm = difference between the harmonic pressure and lattice pressure
           as described below
cv = compute will return the heat capacity
```

## 3.59.2 Examples

```
compute 2 all hma 1 u
compute 2 all hma 1 anharmonic u p 0.9
compute 2 all hma 1 u cv
```

## 3.59.3 Description

Define a computation that calculates the properties of a solid (potential energy, pressure or heat capacity), using the harmonically-mapped averaging (HMA) method. This command yields much higher precision than the equivalent compute commands (*compute pe*, *compute pressure*, etc.) commands during a canonical simulation of an atomic crystal. Specifically, near melting HMA can yield averages of a given precision an order of magnitude faster than conventional methods, and this only improves as the temperatures is lowered. This is particularly important for evaluating the free energy by thermodynamic integration, where the low-temperature contributions are the greatest source of statistical uncertainty. Moreover, HMA has other advantages, including smaller potential-truncation effects, finite-size effects, smaller timestep inaccuracy, faster equilibration and shorter decorrelation time.

HMA should not be used if atoms are expected to diffuse. It is also restricted to simulations in the NVT ensemble. While this compute may be used with any potential in LAMMPS, it will provide inaccurate results for potentials that do not go to 0 at the truncation distance; *pair_style lj/smooth/linear* and Ewald summation should work fine, while *pair_style lj/cut* will perform poorly unless the potential is shifted (via *pair_modify* shift) or the cutoff is large. Furthermore, computation of the heat capacity with this compute is restricted to those that implement the *single_hessian* method in Pair. Implementing *single_hessian* in additional pair styles is simple. Please contact Andrew Schultz (ajs42 at buffalo.edu) and David Kofke (kofke at buffalo.edu) if your desired pair style does not have this method. This is the list of pair styles that currently implement *single_hessian*:

- *pair_style lj/smooth/linear*

In this method, the analytically known harmonic behavior of a crystal is removed from the traditional ensemble averages, which leads to an accurate and precise measurement of the anharmonic contributions without contamination by noise produced by the already-known harmonic behavior. A detailed description of this method can be found in (*Moustafa*). The potential energy is computed by the formula:

$$\langle U \rangle_{\mathrm{HMA}} = \frac{d}{2}(N-1)k_B T + \left\langle U + \frac{1}{2}\vec{F}\cdot\Delta\vec{r} \right\rangle$$

where $N$ is the number of atoms in the system, $k_B$ is Boltzmann's constant, $T$ is the temperature, $d$ is the dimensionality of the system (2 or 3 for 2d/3d), $\vec{F} \cdot \Delta\vec{r}$ is the sum of dot products of the atomic force vectors and displacement (from lattice sites) vectors, and $U$ is the sum of pair, bond, angle, dihedral, improper, kspace (long-range), and fix energies.

The pressure is computed by the formula:

$$\langle P \rangle_{HMA} = \Delta\hat{P} + \left\langle P_{\text{vir}} + \frac{\beta\Delta\hat{P} - \rho}{d(N-1)}\vec{F} \cdot \Delta\vec{r} \right\rangle$$

where $\rho$ is the number density of the system, $\Delta\hat{P}$ is the difference between the harmonic and lattice pressure, $P_{\text{vir}}$ is the virial pressure computed as the sum of pair, bond, angle, dihedral, improper, kspace (long-range), and fix contributions to the force on each atom, and $k_B = 1/k_BT$. Although the method will work for any value of $\Delta\hat{P}$ specified (use pressure *units*), the precision of the resultant pressure is sensitive to $\Delta\hat{P}$; the precision tends to be best when $\Delta\hat{P}$ is the actual the difference between the lattice pressure and harmonic pressure.

$$\langle C_V \rangle_{\text{HMA}} = \frac{d}{2}(N-1)k_B + \frac{1}{k_BT^2}\left(\left\langle U_{\text{HMA}}^2 \right\rangle - \langle U_{\text{HMA}} \rangle^2\right) + \frac{1}{4T}\left\langle \vec{F} \cdot \Delta\vec{r} + \Delta r \cdot \Phi \cdot \Delta\vec{r} \right\rangle$$

where $\Phi$ is the Hessian matrix. The compute hma command computes the full expression for $C_V$ except for the $\langle U_{\text{HMA}} \rangle^2$ in the variance term, which can be obtained by passing the *u* keyword; you must add this extra contribution to the $C_V$ value reported by this compute. The variance term can cause significant round-off error when computing $C_V$. To address this, the *anharmonic* keyword can be passed and/or the output format can be specified with more digits.

```
thermo_modify format float '%22.15e'
```

The *anharmonic* keyword will instruct the compute to return anharmonic properties rather than the full properties, which include lattice, harmonic and anharmonic contributions. When using this keyword, the compute must be first active (it must be included via a *thermo_style custom* command) while the atoms are still at their lattice sites (before equilibration).

The temp-ID specified with compute hma command should be same as the fix-ID of the Nose–Hoover (*fix nvt*) or Berendsen (*fix temp/berendsen*) thermostat used for the simulation. While using this command, the Langevin thermostat (*fix langevin*) should be avoided as its extra forces interfere with the HMA implementation.

**Note**

Compute hma command should be used right after the energy minimization, when the atoms are at their lattice sites. The simulation should not be started before this command has been used in the input script.

The following example illustrates the placement of this command in the input script:

```
min_style cg
minimize 1e-35 1e-15 50000 500000
compute 1 all hma thermostatid u
fix thermostatid all nvt temp 600.0 600.0 100.0
```

**Note**

Compute hma should be used when the atoms of the solid do not diffuse. Diffusion will reduce the precision in the potential energy computation.

> **Note**
>
> The *fix_modify energy yes* command must also be specified if a fix is to contribute potential energy to this command.

An example input script that uses this compute is included in examples/PACKAGES/hma/ along with corresponding LAMMPS output showing that the HMA properties fluctuate less than the corresponding conventional properties.

### 3.59.4 Output info

This compute calculates a global vector that includes the n properties requested as arguments to the command (the potential energy, pressure and/or heat capacity). The elements of the vector can be accessed by indices 1–n by any command that uses global vector values as input. See the *Howto output* page for an overview of LAMMPS output options.

The vector values calculated by this compute are "extensive". The scalar value will be in energy *units*.

### 3.59.5 Restrictions

This compute is part of the EXTRA-COMPUTE package. It is enabled only if LAMMPS was built with that package. See the *Build package* page for more info.

Usage restricted to canonical (NVT) ensemble simulation only.

### 3.59.6 Related commands

*compute pe*, *compute pressure*

*dynamical matrix* provides a finite difference formulation of the Hessian provided by Pair's single_hessian, which is used by this compute.

### 3.59.7 Default

none

---

**(Moustafa)** Sabry G. Moustafa, Andrew J. Schultz, and David A. Kofke, *Very fast averaging of thermal properties of crystals by molecular simulation*, Phys. Rev. E [92], 043303 (2015)

## 3.60 compute improper command

### 3.60.1 Syntax

```
compute ID group-ID improper
```

- ID, group-ID are documented in *compute* command
- improper = style name of this compute command

### 3.60.2 Examples

```
compute 1 all improper
```

### 3.60.3 Description

Define a computation that extracts the improper energy calculated by each of the improper sub-styles used in the *improper_style hybrid* command. These values are made accessible for output or further processing by other commands. The group specified for this command is ignored.

This compute is useful when using *improper_style hybrid* if you want to know the portion of the total energy contributed by one or more of the hybrid sub-styles.

### 3.60.4 Output info

This compute calculates a global vector of length $N$, where $N$ is the number of sub_styles defined by the *improper_style hybrid* command. These styles can be accessed by the indices 1 through $N$. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The vector values are "extensive" and will be in energy *units*.

### 3.60.5 Restrictions

none

### 3.60.6 Related commands

*compute pe*, *compute pair*

### 3.60.7 Default

none

## 3.61 compute improper/local command

### 3.61.1 Syntax

```
compute ID group-ID improper/local value1 value2 ...
```

- ID, group-ID are documented in *compute* command
- improper/local = style name of this compute command
- one or more values may be appended
- value = *chi*

  *chi* = tabulate improper angles

### 3.61.2 Examples

```
compute 1 all improper/local chi
```

### 3.61.3 Description

Define a computation that calculates properties of individual improper interactions. The number of datums generated, aggregated across all processors, equals the number of impropers in the system, modified by the group parameter as explained below.

The value *chi* is the improper angle, as defined in the doc pages for the individual improper styles listed on *improper_style* doc page.

The local data stored by this command is generated by looping over all the atoms owned on a processor and their impropers. An improper will only be included if all four atoms in the improper are in the specified compute group.

Note that as atoms migrate from processor to processor, there will be no consistent ordering of the entries within the local vector or array from one timestep to the next. The only consistency that is guaranteed is that the ordering on a particular timestep will be the same for local vectors or arrays generated by other compute commands. For example, improper output from the *compute property/local* command can be combined with data from this command and output by the *dump local* command in a consistent way.

Here is an example of how to do this:

```
compute 1 all property/local itype iatom1 iatom2 iatom3 iatom4
compute 2 all improper/local chi
dump 1 all local 1000 tmp.dump index c_1[1] c_1[2] c_1[3] c_1[4] c_1[5] c_2[1]
```

### 3.61.4 Output info

This compute calculates a local vector or local array depending on the number of keywords. The length of the vector or number of rows in the array is the number of impropers. If a single keyword is specified, a local vector is produced. If two or more keywords are specified, a local array is produced where the number of columns = the number of keywords. The vector or array can be accessed by any command that uses local values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The output for *chi* will be in degrees.

### 3.61.5 Restrictions

none

### 3.61.6 Related commands

*dump local*, *compute property/local*

### 3.61.7 Default

none

## 3.62 compute inertia/chunk command

### 3.62.1 Syntax

```
compute ID group-ID inertia/chunk chunkID
```

- ID, group-ID are documented in *compute* command
- inertia/chunk = style name of this compute command
- chunkID = ID of *compute chunk/atom* command

### 3.62.2 Examples

```
compute 1 fluid inertia/chunk molchunk
```

### 3.62.3 Description

Define a computation that calculates the inertia tensor for multiple chunks of atoms.

In LAMMPS, chunks are collections of atoms defined by a *compute chunk/atom* command, which assigns each atom to a single chunk (or no chunk). The ID for this command is specified as chunkID. For example, a single chunk could be the atoms in a molecule or atoms in a spatial bin. See the *compute chunk/atom* and *Howto chunk* doc pages for details of how chunks can be defined and examples of how they can be used to measure properties of a system.

This compute calculates the six components of the symmetric inertia tensor for each chunk, ordered $I_{xx}, I_{yy}, I_{zz}, I_{xy}, I_{yz}, I_{xz}$. The calculation includes all effects due to atoms passing through periodic boundaries.

Note that only atoms in the specified group contribute to the calculation. The *compute chunk/atom* command defines its own group; atoms will have a chunk ID = 0 if they are not in that group, signifying they are not assigned to a chunk, and will thus also not contribute to this calculation. You can specify the "all" group for this command if you simply want to include atoms with non-zero chunk IDs.

**Note**

The coordinates of an atom contribute to the chunk's inertia tensor in "unwrapped" form, by using the image flags associated with each atom. See the *dump custom* command for a discussion of "unwrapped" coordinates. See the Atoms section of the *read_data* command for a discussion of image flags and how they are set for each atom. You can reset the image flags (e.g., to 0) before invoking this compute by using the *set image* command.

The simplest way to output the results of the compute inertia/chunk calculation to a file is to use the *fix ave/time* command, for example:

```
compute cc1 all chunk/atom molecule
compute myChunk all inertia/chunk cc1
fix 1 all ave/time 100 1 100 c_myChunk[*] file tmp.out mode vector
```

### 3.62.4 Output info

This compute calculates a global array where the number of rows = the number of chunks *Nchunk* as calculated by the specified *compute chunk/atom* command. The number of columns is 6, one for each of the 6 components of the inertia tensor for each chunk, ordered as listed above. These values can be accessed by any command that uses global array values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The array values are "intensive". The array values will be in mass*distance$^2$ *units*.

### 3.62.5 Restrictions

none

### 3.62.6 Related commands

*variable inertia() function*

### 3.62.7 Default

none

## 3.63 compute ke command

### 3.63.1 Syntax

```
compute ID group-ID ke
```

- ID, group-ID are documented in *compute* command
- ke = style name of this compute command

### 3.63.2 Examples

```
compute 1 all ke
```

### 3.63.3 Description

Define a computation that calculates the translational kinetic energy of a group of particles.

The kinetic energy of each particle is computed as $\frac{1}{2}mv^2$, where $m$ and $v$ are the mass and velocity of the particle, respectively.

There is a subtle difference between the quantity calculated by this compute and the kinetic energy calculated by the *ke* or *etotal* keyword used in thermodynamic output, as specified by the *thermo_style* command. For this compute, kinetic energy is "translational" kinetic energy, calculated by the simple formula above. For thermodynamic output, the *ke* keyword infers kinetic energy from the temperature of the system with $\frac{1}{2}k_BT$ of energy for each degree of freedom. For the default temperature computation via the *compute temp* command, these are the same. However, different computes that calculate temperature can subtract out different non-thermal components of velocity and/or include different degrees of freedom (translational, rotational, etc.).

### 3.63.4 Output info

This compute calculates a global scalar (the summed KE). This value can be used by any command that uses a global scalar value from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "extensive". The scalar value will be in energy *units*.

### 3.63.5 Restrictions

none

### 3.63.6 Related commands

*compute erotate/sphere*

### 3.63.7 Default

none

# 3.64 compute ke/atom command

Accelerator Variants: *ke/atom/omp*

## 3.64.1 Syntax

```
compute ID group-ID ke/atom
```

- ID, group-ID are documented in *compute* command
- ke/atom = style name of this compute command

## 3.64.2 Examples

```
compute 1 all ke/atom
```

## 3.64.3 Description

Define a computation that calculates the per-atom translational kinetic energy for each atom in a group.

The kinetic energy is simply $\frac{1}{2}mv^2$, where $m$ is the mass and $v$ is the velocity of each atom.

The value of the kinetic energy will be 0.0 for atoms not in the specified compute group.

## 3.64.4 Output info

This compute calculates a per-atom vector, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-atom vector values will be in energy *units*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

## 3.64.5 Restrictions

none

## 3.64.6 Related commands

*dump custom*

### 3.64.7 Default

none

## 3.65 compute ke/atom/eff command

### 3.65.1 Syntax

```
compute ID group-ID ke/atom/eff
```

- ID, group-ID are documented in *compute* command
- ke/atom/eff = style name of this compute command

### 3.65.2 Examples

```
compute 1 all ke/atom/eff
```

### 3.65.3 Description

Define a computation that calculates the per-atom translational (nuclei and electrons) and radial kinetic energy (electron only) in a group. The particles are assumed to be nuclei and electrons modeled with the *electronic force field*.

The kinetic energy for each nucleus is computed as $\frac{1}{2}mv^2$, where $m$ corresponds to the corresponding nuclear mass, and the kinetic energy for each electron is computed as $\frac{1}{2}(m_e v^2 + \frac{3}{4}m_e s^2)$, where $m_e$ and $v$ correspond to the mass and translational velocity of each electron, and $s$ to its radial velocity, respectively.

There is a subtle difference between the quantity calculated by this compute and the kinetic energy calculated by the *ke* or *etotal* keyword used in thermodynamic output, as specified by the *thermo_style* command. For this compute, kinetic energy is "translational" plus electronic "radial" kinetic energy, calculated by the simple formula above. For thermodynamic output, the *ke* keyword infers kinetic energy from the temperature of the system with $\frac{1}{2}k_B T$ of energy for each (nuclear-only) degree of freedom in eFF.

**Note**

The temperature in eFF should be monitored via the *compute temp/eff* command, which can be printed with thermodynamic output by using the *thermo_modify* command, as shown in the following example:

```
compute         effTemp all temp/eff
thermo_style    custom step etotal pe ke temp press
thermo_modify   temp effTemp
```

The value of the kinetic energy will be 0.0 for atoms (nuclei or electrons) not in the specified compute group.

### 3.65.4 Output info

This compute calculates a scalar quantity for each atom, which can be accessed by any command that uses per-atom computes as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-atom vector values will be in energy *units*.

### 3.65.5 Restrictions

This compute is part of the EFF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.65.6 Related commands

*dump custom*

### 3.65.7 Default

none

## 3.66 compute ke/eff command

### 3.66.1 Syntax

```
compute ID group-ID ke/eff
```

- ID, group-ID are documented in *compute* command
- ke/eff = style name of this compute command

### 3.66.2 Examples

```
compute 1 all ke/eff
```

### 3.66.3 Description

Define a computation that calculates the kinetic energy of motion of a group of eFF particles (nuclei and electrons), as modeled with the *electronic force field*.

The kinetic energy for each nucleus is computed as $\frac{1}{2}mv^2$ and the kinetic energy for each electron is computed as $\frac{1}{2}(m_e v^2 + \frac{3}{4}m_e s^2)$, where $m$ corresponds to the nuclear mass, $m_e$ to the electron mass, $v$ to the translational velocity of each particle, and $s$ to the radial velocity of the electron, respectively.

There is a subtle difference between the quantity calculated by this compute and the kinetic energy calculated by the *ke* or *etotal* keyword used in thermodynamic output, as specified by the *thermo_style* command. For this compute, kinetic energy is "translational" and "radial" (only for electrons) kinetic energy, calculated by the simple formula above. For thermodynamic output, the *ke* keyword infers kinetic energy from the temperature of the system with $\frac{1}{2}k_B T$ of energy for each degree of freedom. For the eFF temperature computation via the *compute temp_eff* command, these are the same. But different computes that calculate temperature can subtract out different non-thermal components of velocity and/or include other degrees of freedom.

**Warning**

The temperature in eFF models should be monitored via the *compute temp/eff* command, which can be printed with thermodynamic output by using the *thermo_modify* command, as shown in the following example:

```
compute          effTemp all temp/eff
thermo_style     custom step etotal pe ke temp press
thermo_modify    temp effTemp
```

See *compute temp/eff*.

### 3.66.4 Output info

This compute calculates a global scalar (the KE). This value can be used by any command that uses a global scalar value from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "extensive". The scalar value will be in energy *units*.

### 3.66.5 Restrictions

This compute is part of the EFF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.66.6 Related commands

none

### 3.66.7 Default

none

## 3.67 compute ke/rigid command

### 3.67.1 Syntax

```
compute ID group-ID ke/rigid fix-ID
```

- ID, group-ID are documented in *compute* command
- ke = style name of this compute command
- fix-ID = ID of rigid body fix

### 3.67.2 Examples

```
compute 1 all ke/rigid myRigid
```

### 3.67.3 Description

Define a computation that calculates the translational kinetic energy of a collection of rigid bodies, as defined by one of the *fix rigid* command variants.

The kinetic energy of each rigid body is computed as $\frac{1}{2}MV_{\mathrm{cm}}^2$, where $M$ is the total mass of the rigid body, and $V_{\mathrm{cm}}$ is its center-of-mass velocity.

The *fix-ID* should be the ID of one of the *fix rigid* commands which defines the rigid bodies. The group specified in the compute command is ignored. The kinetic energy of all the rigid bodies defined by the fix rigid command in included in the calculation.

### 3.67.4 Output info

This compute calculates a global scalar (the summed KE of all the rigid bodies). This value can be used by any command that uses a global scalar value from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "extensive". The scalar value will be in energy *units*.

### 3.67.5 Restrictions

This compute is part of the RIGID package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.67.6 Related commands

*compute erotate/rigid*

### 3.67.7 Default

none

## 3.68 compute mliap command

### 3.68.1 Syntax

```
compute ID group-ID mliap ... keyword values ...
```

- ID, group-ID are documented in *compute* command
- mliap = style name of this compute command
- two or more keyword/value pairs must be appended
- keyword = *model* or *descriptor* or *gradgradflag*

```
model values = style
  style = linear or quadratic or mliappy
descriptor values = style filename
  style = sna or ace
  filename = name of file containing descriptor definitions
gradgradflag value = 0/1
  toggle gradgrad method for force gradient
```

### 3.68.2 Examples

```
compute mliap model linear descriptor sna Ta06A.mliap.descriptor
compute mliap model linear descriptor ace H_N_O_ccs.yace gradgradflag 1
```

### 3.68.3 Description

Compute style *mliap* provides a general interface to the gradient of machine-learning interatomic potentials with respect to model parameters. It is used primarily for calculating the gradient of energy, force, and stress components with respect to model parameters, which is useful when training *mliap pair_style* models to match target data. It provides separate definitions of the interatomic potential functional form (*model*) and the geometric quantities that characterize the atomic positions (*descriptor*). By defining *model* and *descriptor* separately, it is possible to use many different models with a given descriptor, or many different descriptors with a given model. Currently, the compute supports *linear* and *quadratic* SNAP descriptor computes used in *pair_style snap*, *linear* SO3 descriptor computes, and *linear* ACE descriptor computes used in *pair_style pace*, and it is straightforward to add new descriptor styles.

The compute *mliap* command must be followed by two keywords *model* and *descriptor* in either order.

The *model* keyword is followed by the model style (*linear*, *quadratic* or *mliappy*). The *mliappy* model is only available if LAMMPS is built with the *mliappy* Python module. There are *specific installation instructions* for that module. For the *mliap* compute, specifying a *linear* model will compute the specified descriptors and gradients with respect to linear model parameters whereas *quadratic* will do the same, but for the quadratic products of descriptors.

The *descriptor* keyword is followed by a descriptor style, and additional arguments. The compute currently supports three descriptor styles: *sna*, *so3*, and *ace*, but it is is straightforward to add additional descriptor styles. The SNAP descriptor style *sna* is the same as that used by *pair_style snap*, including the linear, quadratic, and chem variants. A single additional argument specifies the descriptor filename containing the parameters and setting used by the SNAP descriptor. The descriptor filename usually ends in the *.mliap.descriptor* extension. The format of this file is identical to the descriptor file in the *pair_style mliap*, and is described in detail there.

The ACE descriptor style *ace* is the same as *pair_style pace*. A single additional argument specifies the *ace* descriptor filename that contains parameters and settings for the ACE descriptors. This file format differs from the SNAP or SO3 descriptor files, and has a *.yace* or *.ace* extension. However, as with other mliap descriptor styles, this file is identical to the ace descriptor file in *pair_style mliap*, where it is described in further detail.

**Note**

The number of LAMMPS atom types (and the value of *nelems* in the model) must match the value of *nelems* in the descriptor file.

Compute *mliap* calculates a global array containing gradient information. The number of columns in the array is *nelems* $\times$ *nparams* + 1. The first row of the array contain the derivative of potential energy with respect to. to each parameter and each element. The last six rows of the array contain the corresponding derivatives of the virial stress tensor, listed in Voigt notation: *pxx*, *pyy*, *pzz*, *pyz*, *pxz*, and *pxy*. In between the energy and stress rows are the $3N$ rows containing the derivatives of the force components. See section below on output for a detailed description of how rows and columns are ordered.

The element in the last column of each row contains the potential energy, force, or stress, according to the row. These quantities correspond to the user-specified reference potential that must be subtracted from the target data when training a model. The potential energy calculation uses the built in compute *thermo_pe*. The stress calculation uses a compute called *mliap_press* that is automatically created behind the scenes, according to the following command:

```
compute mliap_press all pressure NULL virial
```

See section below on output for a detailed explanation of the data layout in the global array.

The optional keyword *gradgradflag* controls how the force gradient is calculated. A value of 1 requires that the model provide the matrix of double gradients of energy with respect to both parameters and descriptors. For the linear and quadratic models this matrix is sparse and so is easily calculated and stored. For other models, this matrix may be prohibitively expensive to calculate and store. A value of 0 requires that the descriptor provide the derivative of the descriptors with respect to the position of every neighbor atom. This is not optimal for linear and quadratic models, but may be a better choice for more complex models.

Atoms not in the group do not contribute to this compute. Neighbor atoms not in the group do not contribute to this compute. The neighbor list needed to compute this quantity is constructed each time the calculation is performed (i.e., each time a snapshot of atoms is dumped). Thus it can be inefficient to compute/dump this quantity too frequently.

**Note**

If the user-specified reference potentials includes bonded and non-bonded pairwise interactions, then the settings of *special_bonds* command can remove pairwise interactions between atoms in the same bond, angle, or dihedral. This is the default setting for the *special_bonds* command, and means those pairwise interactions do not appear in the neighbor list. Because this fix uses the neighbor list, it also means those pairs will not be included in the calculation. The *rerun* command is not an option here, since the reference potential is required for the last column of the global array. A work-around is to prevent pairwise interactions from being removed by explicitly adding a *tiny* positive value for every pairwise interaction that would otherwise be set to zero in the *special_bonds* command.

### 3.68.4 Output info

Compute *mliap* evaluates a global array. The columns are arranged into *nelems* blocks, listed in order of element *I*. Each block contains one column for each of the *nparams* model parameters. A final column contains the corresponding energy, force component on an atom, or virial stress component. The rows of the array appear in the following order:

- 1 row: Derivatives of potential energy with respect to each parameter of each element.
- 3*N* rows: Derivatives of force components; the *x*, *y*, and *z* components of the force on atom *i* appear in consecutive rows. The atoms are sorted based on atom ID.
- 6 rows: Derivatives of the virial stress tensor with respect to each parameter of each element. The ordering of the rows follows Voigt notation: *pxx*, *pyy*, *pzz*, *pyz*, *pxz*, *pxy*.

These values can be accessed by any command that uses a global array from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options. To see how this command can be used within a Python workflow to train machine-learning interatomic potentials, see the examples in FitSNAP.

### 3.68.5 Restrictions

This compute is part of the ML-IAP package. It is only enabled if LAMMPS was built with that package. In addition, building LAMMPS with the ML-IAP package requires building LAMMPS with the ML-SNAP package. The *mliappy* model also requires building LAMMPS with the PYTHON package. The *ace* descriptor also requires building LAMMPS with the ML-PACE package. See the *Build package* page for more info. Note that *kk* (KOKKOS) accelerated variants of SNAP and ACE descriptors are not compatible with *mliap descriptor*.

### 3.68.6 Related commands

*pair_style mliap*

### 3.68.7 Default

The keyword defaults are gradgradflag = 1

## 3.69 compute momentum command

### 3.69.1 Syntax

```
compute ID group-ID momentum
```

- ID, group-ID are documented in *compute* command
- momentum = style name of this compute command

### 3.69.2 Examples

```
compute 1 all momentum
```

### 3.69.3 Description

Define a computation that calculates the translational momentum $p$ of a group of particles. It is computed as the sum $\vec{p} = \sum_i m_i \cdot \vec{v}_i$ over all particles in the compute group, where $m$ and $v$ are the mass and velocity vector of the particle, respectively.

### 3.69.4 Output info

This compute calculates a global vector (the summed momentum) of length 3. This value can be used by any command that uses a global vector value from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The vector value calculated by this compute is "extensive". The vector value will be in mass*velocity *units*.

### 3.69.5 Restrictions

This compute is part of the EXTRA-COMPUTE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.69.6 Related commands

### 3.69.7 Default

none

## 3.70 compute msd command

### 3.70.1 Syntax

```
compute ID group-ID msd keyword values ...
```

- ID, group-ID are documented in *compute* command
- msd = style name of this compute command
- zero or more keyword/value pairs may be appended
- keyword = *com* or *average*

  *com* value = *yes* or *no*
  *average* value = *yes* or *no*

### 3.70.2 Examples

```
compute 1 all msd
compute 1 upper msd com yes average yes
```

### 3.70.3 Description

Define a computation that calculates the mean-squared displacement (MSD) of the group of atoms, including all effects due to atoms passing through periodic boundaries. For computation of the non-Gaussian parameter of mean-squared displacement, see the *compute msd/nongauss* command.

A vector of four quantities is calculated by this compute. The first three elements of the vector are the squared $dx$, $dy$, and $dz$ displacements, summed and averaged over atoms in the group. The fourth element is the total squared displacement (i.e., $dx^2 + dy^2 + dz^2$), summed and averaged over atoms in the group.

The slope of the mean-squared displacement (MSD) versus time is proportional to the diffusion coefficient of the diffusing atoms.

The displacement of an atom is from its reference position. This is normally the original position at the time the compute command was issued, unless the *average* keyword is set to *yes*.

If the *com* option is set to *yes* then the effect of any drift in the center-of-mass of the group of atoms is subtracted out before the displacement of each atom is calculated.

If the *average* option is set to *yes* then the reference position of an atom is based on the average position of that atom, corrected for center-of-mass motion if requested. The average position is a running average over all previous calls to the compute, including the current call. So on the first call it is current position, on the second call it is the arithmetic average of the current position and the position on the first call, and so on. Note that when using this option, the precise value of the mean square displacement will depend on the number of times the compute is called. So, for example, changing the frequency of thermo output may change the computed displacement. Also, the precise values will be changed if a single simulation is broken up into two parts, using either multiple run commands or a restart file. It only makes sense to use this option if the atoms are not diffusing, so that their average positions relative to the center of mass of the system are stationary. The most common case is crystalline solids undergoing thermal motion.

**Note**

Initial coordinates are stored in "unwrapped" form, by using the image flags associated with each atom. See the *dump custom* command for a discussion of "unwrapped" coordinates. See the Atoms section of the *read_data* command for a discussion of image flags and how they are set for each atom. You can reset the image flags (e.g. to 0) before invoking this compute by using the *set image* command.

**Note**

If you want the quantities calculated by this compute to be continuous when running from a *restart file*, then you should use the same ID for this compute, as in the original run. This is so that the fix this compute creates to store per-atom quantities will also have the same ID, and thus be initialized correctly with atom reference positions from the restart file. When *average* is set to yes, then the atom reference positions are restored correctly, but not the number of samples used obtain them. As a result, the reference positions from the restart file are combined with subsequent positions as if they were from a single sample, instead of many, which will change the values of msd somewhat.

### 3.70.4 Output info

This compute calculates a global vector of length 4, which can be accessed by indices 1–4 by any command that uses global vector values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

The vector values are "intensive". The vector values will be in distance$^2$ *units*.

### 3.70.5 Restrictions

Compute *msd* cannot be used with a dynamic group and the number of atoms in the compute group must not be changed by some fixes like, for example, *fix deposit* or *fix evaporate*.

### 3.70.6 Related commands

*compute msd/nongauss*, *compute displace_atom*, *fix store/state*, *compute msd/chunk*

### 3.70.7 Default

The option default are com = no, average = no.

# 3.71 compute msd/chunk command

## 3.71.1 Syntax

```
compute ID group-ID msd/chunk chunkID
```

- ID, group-ID are documented in *compute* command
- msd/chunk = style name of this compute command
- chunkID = ID of *compute chunk/atom* command

## 3.71.2 Examples

```
compute 1 all msd/chunk molchunk
```

## 3.71.3 Description

Define a computation that calculates the mean-squared displacement (MSD) for multiple chunks of atoms.

In LAMMPS, chunks are collections of atoms defined by a *compute chunk/atom* command, which assigns each atom to a single chunk (or no chunk). The ID for this command is specified as chunkID. For example, a single chunk could be the atoms in a molecule or atoms in a spatial bin. See the *compute chunk/atom* and *Howto chunk* doc pages for details of how chunks can be defined and examples of how they can be used to measure properties of a system.

Four quantities are calculated by this compute for each chunk. The first 3 quantities are the squared $dx$, $dy$, and $dz$ displacements of the center-of-mass. The fourth component is the total squared displacement (i.e., $dx^2 + dy^2 + dz^2$) of the center-of-mass. These calculations include all effects due to atoms passing through periodic boundaries.

Note that only atoms in the specified group contribute to the calculation. The *compute chunk/atom* command defines its own group; atoms will have a chunk ID = 0 if they are not in that group, signifying they are not assigned to a chunk, and will thus also not contribute to this calculation. You can specify the "all" group for this command if you simply want to include atoms with non-zero chunk IDs.

The slope of the mean-squared displacement (MSD) versus time is proportional to the diffusion coefficient of the diffusing chunks.

The displacement of the center-of-mass of the chunk is from its original center-of-mass position, calculated on the timestep this compute command was first invoked.

**Note**

The number of chunks *Nchunk* calculated by the *compute chunk/atom* command must remain constant each time this compute is invoked, so that the displacement for each chunk from its original position can be computed consistently. If *Nchunk* does not remain constant, an error will be generated. If needed, you can enforce a constant *Nchunk* by using the *nchunk once* or *ids once* options when specifying the *compute chunk/atom* command.

**Note**

This compute stores the original position (of the center-of-mass) of each chunk. When a displacement is calculated on a later timestep, it is assumed that the same atoms are assigned to the same chunk ID. However LAMMPS has no simple way to ensure this is the case, though you can use the *ids once* option when specifying the *compute chunk/atom* command. Note that if this is not the case, the MSD calculation does not have a sensible meaning.

**Note**

The initial coordinates of the atoms in each chunk are stored in "unwrapped" form, by using the image flags associated with each atom. See the *dump custom* command for a discussion of "unwrapped" coordinates. See the Atoms section of the *read_data* command for a discussion of image flags and how they are set for each atom. You can reset the image flags (e.g., to 0) before invoking this compute by using the *set image* command.

**Note**

If you want the quantities calculated by this compute to be continuous when running from a *restart file*, then you should use the same ID for this compute, as in the original run. This is so that the fix this compute creates to store per-chunk quantities will also have the same ID, and thus be initialized correctly with chunk reference positions from the restart file.

The simplest way to output the results of the compute msd/chunk calculation to a file is to use the *fix ave/time* command, for example:

```
compute cc1 all chunk/atom molecule
compute myChunk all msd/chunk cc1
fix 1 all ave/time 100 1 100 c_myChunk[*] file tmp.out mode vector
```

### 3.71.4 Output info

This compute calculates a global array where the number of rows = the number of chunks *Nchunk* as calculated by the specified *compute chunk/atom* command. The number of columns = 4 for *dx*, *dy*, *dz*, and the total displacement. These values can be accessed by any command that uses global array values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The array values are "intensive". The array values will be in distance$^2$ *units*.

### 3.71.5 Restrictions

none

### 3.71.6 Related commands

*compute msd*, *compute vacf/chunk*

### 3.71.7 Default

none

## 3.72 compute msd/nongauss command

### 3.72.1 Syntax

```
compute ID group-ID msd/nongauss keyword values ...
```

- ID, group-ID are documented in *compute* command
- msd/nongauss = style name of this compute command
- zero or more keyword/value pairs may be appended

- keyword = *com*

  *com* value = *yes* or *no*

### 3.72.2 Examples

```
compute 1 all msd/nongauss
compute 1 upper msd/nongauss com yes
```

### 3.72.3 Description

Define a computation that calculates the mean-squared displacement (MSD) and non-Gaussian parameter (NGP) of the group of atoms, including all effects due to atoms passing through periodic boundaries.

A vector of three quantities is calculated by this compute. The first element of the vector is the total squared displacement, $dr^2 = dx^2 + dy^2 + dz^2$, of the atoms, and the second is the fourth power of these displacements, $dr^4 = (dx^2 + dy^2 + dz^2)^2$, summed and averaged over atoms in the group. The third component is the non-Gaussian diffusion parameter NGP,

$$\mathrm{NGP}(t) = \frac{3\left\langle (r(t) - r(0))^4 \right\rangle}{5\left\langle (r(t) - r(0))^2 \right\rangle^2} - 1.$$

The NGP is a commonly used quantity in studies of dynamical heterogeneity. Its minimum theoretical value ($-0.4$) occurs when all atoms have the same displacement magnitude. NGP = 0 for Brownian diffusion, while NGP $> 0$ when some mobile atoms move faster than others.

If the *com* option is set to *yes* then the effect of any drift in the center-of-mass of the group of atoms is subtracted out before the displacement of each atom is calculated.

See the *compute msd* page for further important NOTEs, which also apply to this compute.

### 3.72.4 Output info

This compute calculates a global vector of length 3, which can be accessed by indices 1–3 by any command that uses global vector values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

The vector values are "intensive". The first vector value will be in distance$^2$ *units*, the second is in distance$^4$ units, and the third is dimensionless.

### 3.72.5 Restrictions

Compute *msd/nongauss* cannot be used with a dynamic group.

This compute is part of the EXTRA-COMPUTE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.72.6 Related commands

*compute msd*

### 3.72.7 Default

The option default is com = no.

# 3.73 compute nbond/atom command

## 3.73.1 Syntax

```
compute ID group-ID nbond/atom keyword value
```

- ID, group-ID are documented in *compute* command
- nbond/atom = style name of this compute command
- zero or more keyword/value pairs may be appended
- keyword = *bond/type*

  *bond/type* value = *btype*
    *btype* = bond type included in count

## 3.73.2 Examples

```
compute 1 all nbond/atom
compute 1 all nbond/atom bond/type 2
```

## 3.73.3 Description

Added in version 4May2022.

Define a computation that computes the number of bonds each atom is part of. Bonds which are broken are not counted in the tally. See the *Howto broken bonds* page for more information. The number of bonds will be zero for atoms not in the specified compute group. This compute does not depend on Newton bond settings.

If the keyword *bond/type* is specified, only bonds of *btype* are counted.

## 3.73.4 Output info

This compute calculates a per-atom vector, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

## 3.73.5 Restrictions

This compute is part of the BPM package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

## 3.73.6 Related commands

## 3.73.7 Default

none

# 3.74 compute omega/chunk command

## 3.74.1 Syntax

```
compute ID group-ID omega/chunk chunkID
```

- ID, group-ID are documented in *compute* command
- omega/chunk = style name of this compute command

- chunkID = ID of *compute chunk/atom* command

### 3.74.2 Examples

```
compute 1 fluid omega/chunk molchunk
```

### 3.74.3 Description

Define a computation that calculates the angular velocity (omega) of multiple chunks of atoms.

In LAMMPS, chunks are collections of atoms defined by a *compute chunk/atom* command, which assigns each atom to a single chunk (or no chunk). The ID for this command is specified as chunkID. For example, a single chunk could be the atoms in a molecule or atoms in a spatial bin. See the *compute chunk/atom* and *Howto chunk* doc pages for details of how chunks can be defined and examples of how they can be used to measure properties of a system.

This compute calculates the three components of the angular velocity vector for each chunk via the formula $\vec{L} = \mathrm{I} \cdot \vec{\omega}$, where $\vec{L}$ is the angular momentum vector of the chunk, I is its moment of inertia tensor, and $\omega$ is the angular velocity of the chunk. The calculation includes all effects due to atoms passing through periodic boundaries.

Note that only atoms in the specified group contribute to the calculation. The *compute chunk/atom* command defines its own group; atoms will have a chunk ID = 0 if they are not in that group, signifying they are not assigned to a chunk, and will thus also not contribute to this calculation. You can specify the "all" group for this command if you simply want to include atoms with non-zero chunk IDs.

**Note**

The coordinates of an atom contribute to the chunk's angular velocity in "unwrapped" form, by using the image flags associated with each atom. See the *dump custom* command for a discussion of "unwrapped" coordinates. See the Atoms section of the *read_data* command for a discussion of image flags and how they are set for each atom. You can reset the image flags (e.g., to 0) before invoking this compute by using the *set image* command.

The simplest way to output the results of the compute omega/chunk calculation to a file is to use the *fix ave/time* command, for example:

```
compute cc1 all chunk/atom molecule
compute myChunk all omega/chunk cc1
fix 1 all ave/time 100 1 100 c_myChunk[*] file tmp.out mode vector
```

### 3.74.4 Output info

This compute calculates a global array where the number of rows is the number of chunks *Nchunk* as calculated by the specified *compute chunk/atom* command. The number of columns is 3 for the three ($x$, $y$, $z$) components of the angular velocity for each chunk. These values can be accessed by any command that uses global array values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The array values are "intensive". The array values will be in velocity/distance *units*.

### 3.74.5 Restrictions

none

### 3.74.6 Related commands

*variable omega() function*

### 3.74.7 Default

none

## 3.75 compute orientorder/atom command

Accelerator Variants: *orientorder/atom/kk*, *orientorder/atom/omp*

### 3.75.1 Syntax

```
compute ID group-ID orientorder/atom keyword values ...
```

- ID, group-ID are documented in *compute* command
- orientorder/atom = style name of this compute command
- one or more keyword/value pairs may be appended

```
keyword = cutoff or nnn or degrees or wl or wl/hat or components or chunksize
  cutoff value = distance cutoff
  nnn value = number of nearest neighbors
  degrees values = nlvalues, l1, l2,...
  wl value = yes or no
  wl/hat value = yes or no
  components value = ldegree
  chunksize value = number of atoms in each pass
```

### 3.75.2 Examples

```
compute 1 all orientorder/atom
compute 1 all orientorder/atom degrees 5 4 6 8 10 12 nnn NULL cutoff 1.5
compute 1 all orientorder/atom wl/hat yes
compute 1 all orientorder/atom components 6
```

### 3.75.3 Description

Define a computation that calculates a set of bond-orientational order parameters $Q_\ell$ for each atom in a group. These order parameters were introduced by *Steinhardt et al.* as a way to characterize the local orientational order in atomic structures. For each atom, $Q_\ell$ is a real number defined as follows:

$$\bar{Y}_{\ell m} = \frac{1}{nnn} \sum_{j=1}^{nnn} Y_{\ell m}\left(\theta(\mathbf{r}_{ij}), \phi(\mathbf{r}_{ij})\right)$$

$$Q_\ell = \sqrt{\frac{4\pi}{2\ell+1} \sum_{m=-\ell}^{m=\ell} \bar{Y}_{\ell m} \bar{Y}^*_{\ell m}}$$

The first equation defines the local order parameters as averages of the spherical harmonics $Y_{\ell m}$ for each neighbor. These are complex number components of the 3D analog of the 2D order parameter $q_n$, which is implemented as LAMMPS compute *hexorder/atom*. The summation is over the *nnn* nearest neighbors of the central atom. The angles $\theta$ and $\phi$ are the standard spherical polar angles defining the direction of the bond vector $r_{ij}$. The phase and sign of $Y_{\ell m}$ follow the

standard conventions, so that $\mathrm{sign}(Y_{\ell\ell}(0,0)) = (-1)^{\ell}$. The second equation defines $Q_\ell$, which is a rotationally invariant non-negative amplitude obtained by summing over all the components of degree $\ell$.

The optional keyword *cutoff* defines the distance cutoff used when searching for neighbors. The default value, also the maximum allowable value, is the cutoff specified by the pair style.

The optional keyword *nnn* defines the number of nearest neighbors used to calculate $Q_\ell$. The default value is 12. If the value is NULL, then all neighbors up to the specified distance cutoff are used.

The optional keyword *degrees* defines the list of order parameters to be computed. The first argument *nlvalues* is the number of order parameters. This is followed by that number of non-negative integers giving the degree of each order parameter. Because $Q_2$ and all odd-degree order parameters are zero for atoms in cubic crystals (see *Steinhardt*), the default order parameters are $Q_4$, $Q_6$, $Q_8$, $Q_{10}$, and $Q_{12}$. For the FCC crystal with *nnn* =12,

$$Q_4 = \sqrt{\frac{7}{192}} \approx 0.19094$$

The numerical values of all order parameters up to $Q_{12}$ for a range of commonly encountered high-symmetry structures are given in Table I of *Mickel et al.*, and these can be reproduced with this compute.

The optional keyword *wl* will output the third-order invariants $W_\ell$ (see Eq. 1.4 in *Steinhardt*) for the same degrees as for the $Q_\ell$ parameters. For the FCC crystal with *nnn* = 12,

$$W_4 = -\sqrt{\frac{14}{143}}\left(\frac{49}{4096}\right)\pi^{-3/2} \approx -0.0006722136$$

The optional keyword *wl/hat* will output the normalized third-order invariants $\hat{W}_\ell$ (see Eq. 2.2 in *Steinhardt*) for the same degrees as for the $Q_\ell$ parameters. For the FCC crystal with *nnn* =12,

$$\hat{W}_4 = -\frac{7}{3}\sqrt{\frac{2}{429}} \approx -0.159317$$

The numerical values of $\hat{W}_\ell$ for a range of commonly encountered high-symmetry structures are given in Table I of *Steinhardt*, and these can be reproduced with this keyword.

The optional keyword *components* will output the components of the *normalized* complex vector $\hat{Y}_{\ell m} = \bar{Y}_{\ell m}/|\bar{Y}_{\ell m}|$ of degree *ldegree*, which must be included in the list of order parameters to be computed. This option can be used in conjunction with *compute coord_atom* to calculate the ten Wolde's criterion to identify crystal-like particles, as discussed in *ten Wolde*.

The optional keyword *chunksize* is only applicable when using the the KOKKOS package and is ignored otherwise. This keyword controls the number of atoms in each pass used to compute the bond-orientational order parameters and is used to avoid running out of memory. For example if there are 32768 atoms in the simulation and the *chunksize* is set to 16384, the parameter calculation will be broken up into two passes.

The value of $Q_\ell$ is set to zero for atoms not in the specified compute group, as well as for atoms that have less than *nnn* neighbors within the distance cutoff, unless *nnn* is NULL.

The neighbor list needed to compute this quantity is constructed each time the calculation is performed (i.e., each time a snapshot of atoms is dumped). Thus it can be inefficient to compute/dump this quantity too frequently.

**Note**

If you have a bonded system, then the settings of *special_bonds* command can remove pairwise interactions between atoms in the same bond, angle, or dihedral. This is the default setting for the *special_bonds* command, and means those pairwise interactions do not appear in the neighbor list. Because this fix uses the neighbor list, it also means those pairs will not be included in the order parameter. This difficulty can be circumvented by writing a dump file, and using the *rerun* command to compute the order parameter for snapshots in the dump file. The rerun script can use a *special_bonds* command that includes all pairs in the neighbor list.

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 3.75.4 Output info

This compute calculates a per-atom array with *nlvalues* columns, giving the $Q_\ell$ values for each atom, which are real numbers in the range $0 \leq Q_\ell \leq 1$.

If the keyword *wl* is set to yes, then the $W_\ell$ values for each atom will be added to the output array, which are real numbers.

If the keyword *wl/hat* is set to yes, then the $\hat{W}_\ell$ values for each atom will be added to the output array, which are real numbers.

If the keyword *components* is set, then the real and imaginary parts of each component of *normalized* $\hat{Y}_{\ell m}$ will be added to the output array in the following order: $\Re(\hat{Y}_{-m})$, $\Im(\hat{Y}_{-m})$, $\Re(\hat{Y}_{-m+1})$, $\Im(\hat{Y}_{-m+1})$, ..., $\Re(\hat{Y}_{m})$, $\Im(\hat{Y}_{m})$.

In summary, the per-atom array will contain *nlvalues* columns, followed by an additional *nlvalues* columns if *wl* is set to yes, followed by an additional *nlvalues* columns if *wl/hat* is set to yes, followed by an additional 2*(2* *ldegree*+1) columns if the *components* keyword is set.

These values can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

### 3.75.5 Restrictions

none

### 3.75.6 Related commands

*compute coord/atom*, *compute centro/atom*, *compute hexorder/atom*

### 3.75.7 Default

The option defaults are *cutoff* = pair style cutoff, *nnn* = 12, *degrees* = 5 4 6 8 10 12 (i.e., $Q_4$, $Q_6$, $Q_8$, $Q_{10}$, and $Q_{12}$), *wl* = no, *wl/hat* = no, *components* off, and *chunksize* = 16384

**(Steinhardt)** P. Steinhardt, D. Nelson, and M. Ronchetti, Phys. Rev. B 28, 784 (1983).

**(Mickel)** W. Mickel, S. C. Kapfer, G. E. Schroeder-Turkand, K. Mecke, J. Chem. Phys. 138, 044501 (2013).

**(tenWolde)** P. R. ten Wolde, M. J. Ruiz-Montero, D. Frenkel, J. Chem. Phys. 104, 9932 (1996).

# 3.76 compute pace command

## 3.76.1 Syntax

```
compute ID group-ID pace ace_potential_filename ... keyword values ...
```

- ID, group-ID are documented in *compute* command
- pace = style name of this compute command
- ace_potential_filename = file name (in the .yace or .ace format from *pace pair_style*) including ACE hyper-parameters, bonds, and generalized coupling coefficients
- keyword = *bikflag* or *dgradflag*

```
bikflag value = 0 or 1
   0 = descriptors are summed over atoms of each type
   1 = descriptors are listed separately for each atom
dgradflag value = 0 or 1
   0 = descriptor gradients are summed over atoms of each type
   1 = descriptor gradients are listed separately for each atom pair
```

## 3.76.2 Examples

```
compute pace all pace coupling_coefficients.yace
compute pace all pace coupling_coefficients.yace 0 1
compute pace all pace coupling_coefficients.yace 1 1
```

## 3.76.3 Description

Added in version 7Feb2024.

This compute calculates a set of quantities related to the atomic cluster expansion (ACE) descriptors of the atoms in a group. ACE descriptors are highly general atomic descriptors, encoding the radial and angular distribution of neighbor atoms, up to arbitrary bond order (rank). The detailed mathematical definition is given in the paper by *(Drautz)*. These descriptors are used in the *pace pair_style*. Quantities obtained from *compute pace* are related to those used in *pace pair_style* to evaluate atomic energies, forces, and stresses for linear ACE models.

For example, the energy for a linear ACE model is calculated as: $E = \sum_i^{N_atoms} \sum_\nu c_\nu B_{i,\nu}$. The ACE descriptors for atom $i$ $B_{i,\nu}$, and $c_\nu$ are linear model parameters. The detailed definition and indexing convention for ACE descriptors is given in *(Drautz)*. In short, body order $N$, angular character, radial character, and chemical elements in the $N$-*body* descriptor are encoded by $\nu$. In the *pace pair_style*, the linear model parameters and the ACE descriptors are combined for efficient evaluation of energies and forces. The details and benefits of this efficient implementation are given in *(Lysogorskiy)*, but the combined descriptors and linear model parameters for the purposes of *compute pace* may be expressed in terms of the ACE descriptors mentioned above.

$c_\nu B_{i,\nu} = \sum_{\nu' \in \nu} \left[c_\nu C(\nu')\right] A_{i,\nu'}$

where the bracketed terms on the right-hand side are the combined functions with linear model parameters typically provided in the *<name>.yace* potential file for *pace pair_style*. When these bracketed terms are multiplied by the products of the atomic base from *(Drautz)*, $A_{i,\nu'}$, the ACE descriptors are recovered but they are also scaled by linear model parameters. The generalized coupling coefficients, written in short-hand here as $C(\nu')$, are the generalized Clebsch-Gordan or generalized Wigner symbols. It may be desirable to reverse the combination of these descriptors and the linear model parameters so that the ACE descriptors themselves may be used. The ACE descriptors and their gradients are often used when training ACE models, performing custom data analysis, generalizing ACE model forms, and other tasks that involve direct computation of descriptors. The key utility of *compute pace* is that it can compute the ACE descriptors and gradients so that these tasks can be performed during a LAMMPS simulation or so that LAMMPS

can be used as a driver for tasks like ACE model parameterization. To see how this command can be used within a Python workflow to train ACE potentials, see the examples in FitSNAP. Examples on using outputs from this compute to construct general ACE potential forms are demonstrated in *(Goff)*. The various keywords and inputs to *compute pace* determine what ACE descriptors and related quantities are returned in a compute array.

The coefficient file, *<name>.yace*, ultimately defines the number of ACE descriptors to be computed, their maximum body-order, the degree of angular character they have, the degree of radial character they have, the chemical character (which element-element interactions are encoded by descriptors), and other hyper-parameters defined in *(Drautz)*. These may be modeled after the potential files in *pace pair_style*, and have the same format. Details on how to generate the coefficient files to train ACE models may be found in FitSNAP.

The keyword *bikflag* determines whether or not to list the descriptors of each atom separately, or sum them together and list in a single row. If *bikflag* is set to *0* then a single descriptor row is used, which contains the per-atom ACE descriptors $B_{i,v}$ summed over all atoms $i$ to produce $B_v$. If *bikflag* is set to *1* this is replaced by a separate per-atom ACE descriptor row for each atom. In this case, the entries in the final column for these rows are set to zero.

The keyword *dgradflag* determines whether to sum atom gradients or list them separately. If *dgradflag* is set to 0, the ACE descriptor gradients w.r.t. atom $j$ are summed over all atoms $i$' of, which may be useful when training linear ACE models on atomic forces. If *dgradflag* is set to 1, gradients are listed separately for each pair of atoms. Each row corresponds to a single term $\frac{\partial B_{i,v}}{\partial r_j^a}$ where $r_j^a$ is the *a-th* position coordinate of the atom with global index $j$. This also changes the number of columns to be equal to the number of ACE descriptors, with 3 additional columns representing the indices $i$, $j$, and $a$, as explained more in the Output info section below. The option *dgradflag=1* requires that *bikflag=1*.

**Note**

It is noted here that in contrast to *pace pair_style*, the *.yace* file for *compute pace* typically should not contain linear parameters for an ACE potential. If $c_v$ are included, the value of the descriptor will not be returned in the *compute* array, but instead, the energy contribution from that descriptor will be returned. Do not do this unless it is the desired behavior. *In short, you should not plug in a '.yace' for a pace potential into this compute to evaluate descriptors.*

**Note**

*Generalized Clebsch-Gordan or Generalized Wigner symbols (with appropriate factors) must be used to evaluate ACE descriptors with this compute.* There are multiple ways to define the generalized coupling coefficients. Because of this, this compute will not revert your potential file to a coupling coefficient file. Instead this compute allows the user to supply coupling coefficients that follow any convention.

**Note**

Using *dgradflag* = 1 produces a global array with $N + 3N^2 + 1$ rows which becomes expensive for systems with more than 1000 atoms.

**Note**

If you have a bonded system, then the settings of *special_bonds* command can remove pairwise interactions between atoms in the same bond, angle, or dihedral. This is the default setting for the *special_bonds* command, and means those pairwise interactions do not appear in the neighbor list. Because this fix uses the neighbor list, it also means those pairs will not be included in the calculation. One way to get around this, is to write a dump file, and use

the *rerun* command to compute the ACE descriptors for snapshots in the dump file. The rerun script can use a *special_bonds* command that includes all pairs in the neighbor list.

---

### 3.76.4 Output info

Compute *pace* evaluates a global array. The columns are arranged into *ntypes* blocks, listed in order of atom type *I*. Each block contains one column for each ACE descriptor, the same as for compute *sna/atom*in *compute snap*. A final column contains the corresponding energy, force component on an atom, or virial stress component. The rows of the array appear in the following order:

- 1 row: *pace* average descriptor values for all atoms of type *I*
- 3*$n$ force rows: quantities, with derivatives w.r.t. x, y, and z coordinate of atom $i$ appearing in consecutive rows. The atoms are sorted based on atom ID and run up to the total number of atoms, $n$.
- 6 rows: *virial* quantities summed for all atoms of type *I*

For example, if # $B_{i,v}$ =30 and ntypes=1, the number of columns in the The number of columns in the global array generated by *pace* are 31, and 931, respectively, while the number of rows is 1+3*$n$+6, where $n$ is the total number of atoms.

If the *bik* keyword is set to 1, the structure of the pace array is expanded. The first $N$ rows of the pace array correspond to # $B_{i,v}$ instead of a single row summed over atoms $i$. In this case, the entries in the final column for these rows are set to zero. Also, each row contains only non-zero entries for the columns corresponding to the type of that atom. This is not true in the case of *dgradflag* keyword = 1 (see below).

If the *dgradflag* keyword is set to 1, this changes the structure of the global array completely. Here the per-atom quantities are replaced with rows corresponding to descriptor gradient components on single atoms:

$$\frac{\partial B_{i,v}}{\partial r_j^a}$$

where $r_j^a$ is the *a-th* position coordinate of the atom with global index $j$. The rows are organized in chunks, where each chunk corresponds to an atom with global index $j$. The rows in an atom $j$ chunk correspond to atoms with global index $i$. The total number of rows for these descriptor gradients is therefore $3N^2$. The number of columns is equal to the number of ACE descriptors, plus 3 additional left-most columns representing the global atom indices $i$, $j$, and Cartesian direction $a$ (0, 1, 2, for x, y, z). The first 3 columns of the first $N$ rows belong to the reference potential force components. The remaining K columns contain the $B_{i,v}$ per-atom descriptors corresponding to the non-zero entries obtained when *bikflag* = 1. The first column of the last row, after the first $N + 3N^2$ rows, contains the reference potential energy. The virial components are not used with this option. The total number of rows is therefore $N + 3N^2 + 1$ and the number of columns is $K + 3$.

These values can be accessed by any command that uses global values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

### 3.76.5 Restrictions

These computes are part of the ML-PACE package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.76.6 Related commands

*pair_style pace pair_style snap compute snap*

### 3.76.7 Default

The optional keyword defaults are *bikflag* = 0, *dgradflag* = 0

---

**(Drautz)** Drautz, Phys Rev B, 99, 014104 (2019).

**(Lysogorskiy)** Lysogorskiy, van der Oord, Bochkarev, Menon, Rinaldi, Hammerschmidt, Mrovec, Thompson, Csanyi, Ortner, Drautz, npj Comp Mat, 7, 97 (2021).

**(Goff)** Goff, Zhang, Negre, Rohskopf, Niklasson, Journal of Chemical Theory and Computation 19, no. 13 (2023).

## 3.77 compute pair command

### 3.77.1 Syntax

```
compute ID group-ID pair pstyle [nstyle] [evalue]
```

- ID, group-ID are documented in *compute* command
- pair = style name of this compute command
- pstyle = style name of a pair style that calculates additional values
- nsub = *n*-instance of a sub-style, if a pair style is used multiple times in a hybrid style
- *evalue* = *epair* or *evdwl* or *ecoul* or blank (optional)

### 3.77.2 Examples

```
compute 1 all pair gauss
compute 1 all pair lj/cut/coul/cut ecoul
compute 1 all pair tersoff 2 epair
compute 1 all pair reaxff
```

### 3.77.3 Description

Define a computation that extracts additional values calculated by a pair style, and makes them accessible for output or further processing by other commands.

> **Note**
>
> The group specified for this command is **ignored**.

The specified *pstyle* must be a pair style used in your simulation either by itself or as a sub-style in a *pair_style hybrid or hybrid/overlay* command. If the sub-style is used more than once, an additional number *nsub* has to be specified in order to choose which instance of the sub-style will be used by the compute. Not specifying the number in this case will cause the compute to fail.

The *evalue* setting is optional. All pair styles tally a potential energy *epair* which may be broken into two parts: *evdwl* and *ecoul* such that *epair* = *evdwl* + *ecoul*. If the pair style calculates Coulombic interactions, their energy will be tallied in *ecoul*. Everything else (whether it is a Lennard-Jones style van der Waals interaction or not) is tallied in *evdwl*. If *evalue* is blank or specified as *epair*, then *epair* is stored as a global scalar by this compute. This is useful when using *pair_style hybrid* if you want to know the portion of the total energy contributed by one sub-style. If *evalue* is specified as *evdwl* or *ecoul*, then just that portion of the energy is stored as a global scalar.

> **Note**
>
> The energy returned by the *evdwl* keyword does not include tail corrections, even if they are enabled via the *pair_modify* command.

Some pair styles tally additional quantities, e.g. a breakdown of potential energy into 14 components is tallied by the *pair_style reaxff* command. These values (1 or more) are stored as a global vector by this compute. See the page for *individual pair styles* for info on these values.

### 3.77.4 Output info

This compute calculates a global scalar which is *epair* or *evdwl* or *ecoul*. If the pair style supports it, it also calculates a global vector of length $\geq 1$, as determined by the pair style. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

The scalar and vector values calculated by this compute are "extensive".

The scalar value will be in energy *units*. The vector values will typically also be in energy *units*, but see the page for the pair style for details.

### 3.77.5 Restrictions

none

### 3.77.6 Related commands

*compute pe*, *compute bond*, *fix pair*

### 3.77.7 Default

The keyword defaults are *evalue* = *epair*, nsub = 0.

## 3.78 compute pair/local command

### 3.78.1 Syntax

```
compute ID group-ID pair/local value1 value2 ... keyword args ...
```

- ID, group-ID are documented in *compute* command
- pair/local = style name of this compute command
- one or more values may be appended
- value = *dist* or *dx* or *dy* or *dz* or *eng* or *force* or *fx* or *fy* or *fz* or *p1* or *p2* or ...

  ```
  dist = pairwise distance
  dx,dy,dz = components of pairwise distance
  eng = pairwise energy
  force = pairwise force
  fx,fy,fz = components of pairwise force
  p1, p2, ... = pair style specific quantities for allowed N values
  ```

- zero or more keyword/arg pairs may be appended
- keyword = *cutoff*

*cutoff* arg = *type* or *radius*

### 3.78.2 Examples

```
compute 1 all pair/local eng
compute 1 all pair/local dist eng force
compute 1 all pair/local dist eng fx fy fz
compute 1 all pair/local dist fx fy fz p1 p2 p3
```

### 3.78.3 Description

Define a computation that calculates properties of individual pairwise interactions. The number of datums generated, aggregated across all processors, equals the number of pairwise interactions in the system.

The local data stored by this command is generated by looping over the pairwise neighbor list. Info about an individual pairwise interaction will only be included if both atoms in the pair are in the specified compute group, and if the current pairwise distance is less than the force cutoff distance for that interaction, as defined by the *pair_style* and *pair_coeff* commands.

Changed in version 12Jun2025: The sign of *dx*, *dy*, *dz* is no longer determined by the value of their atom-IDs but by their order in the neighbor list to be consistent with *fx*, *fy*, and *fz*.

The value *dist* is the distance between the pair of atoms. The values *dx*, *dy*, and *dz* are the $(x, y, z)$ components of the distance vector $\vec{x}_i - \vec{x}_j$ between the pair of atoms. The order of the atoms is determined by the neighbor list and the respective atom-IDs can be output with *compute property/local*.

The value *eng* is the interaction energy for the pair of atoms.

The value *force* is the force acting between the pair of atoms, which is positive for a repulsive force and negative for an attractive force.

The values *fx*, *fy*, and *fz* are the $(x, y, z)$ components of the force vector on the first atom $i$ of a pair in the neighbor list due to the second atom $j$. Mathematically, they are obtained by multiplying the value of *force* from above with a unit vector created from the *dx*, *dy*, and *dz* components of the distance vector also described above. For pair styles that apply non-central forces, such as *granular pair styles*, these values only include the $(x, y, z)$ components of the normal force component.

A pair style may define additional pairwise quantities which can be accessed as *p1* to *pN*, where *N* is defined by the pair style. Most pair styles do not define any additional quantities, so $N = 0$. An example of ones that do are the *granular pair styles* which calculate the tangential force between two particles and return its components and magnitude acting on atom $I$ for $N \in \{1, 2, 3, 4\}$. See individual pair styles for details.

When using *pN* with pair style *hybrid*, the output will be the Nth quantity from the sub-style that computes the pairwise interaction (based on atom types). If that sub-style does not define a *pN*, the output will be 0.0. The maximum allowed *N* is the maximum number of quantities provided by any sub-style.

When using *pN* with pair style *hybrid/overlay* the quantities from all sub-styles that provide them are concatenated together into one long list. For example, if there are 3 sub-styles and 2 of them have additional output (with 3 and 4 quantities, respectively), then 7 values (*p1* up to *p7*) are defined. The values *p1* to *p3* refer to quantities defined by the first of the two sub-styles. Values *p4* to *p7* refer to quantities from the second of the two sub-styles. If the referenced *pN* is not computed for the specific pairwise interaction (based on atom types), then the output will be 0.0.

The value *dist*, *dx*, *dy* and *dz* will be in distance *units*. The value *eng* will be in energy *units*. The values *force*, *fx*, *fy*, and *fz* will be in force *units*. The values *pN* will be in whatever units the pair style defines.

The optional *cutoff* keyword determines how the force cutoff distance for an interaction is determined. For the default setting of *type*, the pairwise cutoff defined by the *pair_style* command for the types of the two atoms is used. For the *radius* setting, the sum of the radii of the two particles is used as a cutoff. For example, this is appropriate for granular particles which only interact when they are overlapping, as computed by *granular pair styles*. Note that if a granular

model defines atom types such that all particles of a specific type are monodisperse (same diameter), then the two settings are effectively identical.

Note that as atoms migrate from processor to processor, there will be no consistent ordering of the entries within the local vector or array from one timestep to the next. The only consistency that is guaranteed is that the ordering on a particular timestep will be the same for local vectors or arrays generated by other compute commands. For example, pair output from the *compute property/local* command can be combined with data from this command and output by the *dump local* command in a consistent way.

Here is an example of how to do this:

```
compute 1 all property/local patom1 patom2
compute 2 all pair/local dist eng force
dump 1 all local 1000 tmp.dump index c_1[1] c_1[2] c_2[1] c_2[2] c_2[3]
```

**Note**

For pairs, if two atoms I,J are involved in 1–2, 1–3, and 1–4 interactions within the molecular topology, their pairwise interaction may be turned off, and thus they may not appear in the neighbor list, and will not be part of the local data created by this command. More specifically, this will be true of I,J pairs with a weighting factor of 0.0; pairs with a non-zero weighting factor are included. The weighting factors for 1–2, 1–3, and 1–4 pairwise interactions are set by the *special_bonds* command. An exception is if long-range Coulombics are being computed via the *kspace_style* command, then atom pairs with weighting factors of zero are still included in the neighbor list, so that a portion of the long-range interaction contribution can be computed in the pair style. Hence in that case, those atom pairs will be part of the local data created by this command.

### 3.78.4 Output info

This compute calculates a local vector or local array depending on the number of keywords. The length of the vector or number of rows in the array is the number of pairs. If a single keyword is specified, a local vector is produced. If two or more keywords are specified, a local array is produced where the number of columns = the number of keywords. The vector or array can be accessed by any command that uses local values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The output for *dist* will be in distance *units*. The output for *eng* will be in energy *units*. The output for *force*, *fx*, *fy*, and *fz* will be in force *units*. The output for *pN* will be in whatever units the pair style defines.

### 3.78.5 Restrictions

none

### 3.78.6 Related commands

*dump local*, *compute property/local*

### 3.78.7 Default

The keyword default is cutoff = type.

# 3.79 compute pe command

## 3.79.1 Syntax

```
compute ID group-ID pe keyword ...
```

- ID, group-ID are documented in *compute* command
- pe = style name of this compute command
- zero or more keywords may be appended
- keyword = *pair* or *bond* or *angle* or *dihedral* or *improper* or *kspace* or *fix*

## 3.79.2 Examples

```
compute 1 all pe
compute molPE all pe bond angle dihedral improper
```

## 3.79.3 Description

Define a computation that calculates the potential energy of the entire system of atoms. The specified group must be "all". See the *compute pe/atom* command if you want per-atom energies. These per-atom values could be summed for a group of atoms via the *compute reduce* command.

The energy is calculated by the various pair, bond, etc. potentials defined for the simulation. If no extra keywords are listed, then the potential energy is the sum of pair, bond, angle, dihedral, improper, *k*-space (long-range), and fix energy (i.e., it is as though all the keywords were listed). If any extra keywords are listed, then only those components are summed to compute the potential energy.

The *k*-space contribution requires 1 extra FFT each timestep the energy is calculated, if using the PPPM solver via the *kspace_style pppm* command. Thus it can increase the cost of the PPPM calculation if it is needed on a large fraction of the simulation timesteps.

Various fixes can contribute to the total potential energy of the system if the *fix* contribution is included. See the doc pages for *individual fixes* for details of which ones compute a potential energy.

**Note**

The *fix_modify energy yes* command must also be specified if a fix is to contribute potential energy to this command.

A compute of this style with the ID of "thermo_pe" is created when LAMMPS starts up, as if this command were in the input script:

```
compute thermo_pe all pe
```

See the "thermo_style" command for more details.

---

## 3.79.4 Output info

This compute calculates a global scalar (the potential energy). This value can be used by any command that uses a global scalar value from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "extensive". The scalar value will be in energy *units*.

### 3.79.5 Restrictions

none

### 3.79.6 Related commands

*compute pe/atom*

### 3.79.7 Default

none

## 3.80 compute pe/atom command

### 3.80.1 Syntax

```
compute ID group-ID pe/atom keyword ...
```

- ID, group-ID are documented in *compute* command
- pe/atom = style name of this compute command
- zero or more keywords may be appended
- keyword = *pair* or *bond* or *angle* or *dihedral* or *improper* or *kspace* or *fix*

### 3.80.2 Examples

```
compute 1 all pe/atom
compute 1 all pe/atom pair
compute 1 all pe/atom pair bond
```

### 3.80.3 Description

Define a computation that computes the per-atom potential energy for each atom in a group. See the *compute pe* command if you want the potential energy of the entire system.

The per-atom energy is calculated by the various pair, bond, etc potentials defined for the simulation. If no extra keywords are listed, then the potential energy is the sum of pair, bond, angle, dihedral, improper, *k*-space (long-range), and fix energy (i.e., it is as though all the keywords were listed). If any extra keywords are listed, then only those components are summed to compute the potential energy.

Note that the energy of each atom is due to its interaction with all other atoms in the simulation, not just with other atoms in the group.

For an energy contribution produced by a small set of atoms (e.g., 4 atoms in a dihedral or 3 atoms in a Tersoff 3-body interaction), that energy is assigned in equal portions to each atom in the set (e.g., 1/4 of the dihedral energy to each of the four atoms).

The *dihedral_style charmm* style calculates pairwise interactions between 1–4 atoms. The energy contribution of these terms is included in the pair energy, not the dihedral energy.

The KSpace contribution is calculated using the method in *(Heyes)* for the Ewald method and a related method for PPPM, as specified by the *kspace_style pppm* command. For PPPM, the calculation requires 1 extra FFT each timestep that per-atom energy is calculated. This document describes how the long-range per-atom energy calculation is performed.

Various fixes can contribute to the per-atom potential energy of the system if the *fix* contribution is included. See the doc pages for *individual fixes* for details of which ones compute a per-atom potential energy.

> **Note**
>
> The *fix_modify energy yes* command must also be specified if a fix is to contribute per-atom potential energy to this command.

As an example of per-atom potential energy compared to total potential energy, these lines in an input script should yield the same result in the last 2 columns of thermo output:

```
compute         peratom all pe/atom
compute         pe all reduce sum c_peratom
thermo_style    custom step temp etotal press pe c_pe
```

> **Note**
>
> The per-atom energy does not include any Lennard-Jones tail corrections to the energy added by the *pair_modify tail yes* command, since those are contributions to the global system energy.

### 3.80.4 Output info

This compute calculates a per-atom vector, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-atom vector values will be in energy *units*.

### 3.80.5 Restrictions

### 3.80.6 Related commands

*compute pe*, *compute stress/atom*

### 3.80.7 Default

none

---

**(Heyes)** Heyes, Phys Rev B 49, 755 (1994),

## 3.81 compute plasticity/atom command

### 3.81.1 Syntax

```
compute ID group-ID plasticity/atom
```

- ID, group-ID are documented in compute command
- plasticity/atom = style name of this compute command

### 3.81.2 Examples

```
compute 1 all plasticity/atom
```

### 3.81.3 Description

Define a computation that calculates the per-atom plasticity for each atom in a group. This is a quantity relevant for *Peridynamics models*. See this document for an overview of LAMMPS commands for Peridynamics modeling.

The plasticity for a Peridynamic particle is the so-called consistency parameter ($\lambda$). For elastic deformation, $\lambda = 0$, otherwise $\lambda > 0$ for plastic deformation. For details, see *(Mitchell)* and the PDF doc included in the LAMMPS distribution in doc/PDF/PDLammps_EPS.pdf.

This command can be invoked for one of the Peridynamic *pair styles*: peri/eps.

The plasticity value will be 0.0 for atoms not in the specified compute group.

### 3.81.4 Output info

This compute calculates a per-atom vector, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-atom vector values are unitless numbers $\lambda \geq 0.0$.

### 3.81.5 Restrictions

This compute is part of the PERI package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.81.6 Related commands

*compute damage/atom*, *compute dilatation/atom*

### 3.81.7 Default

none

---

**(Mitchell)** Mitchell, "A non-local, ordinary-state-based viscoelasticity model for peridynamics", Sandia National Lab Report, 8064:1-28 (2011).

## 3.82 compute pod/atom command

## 3.83 compute podd/atom command

## 3.84 compute pod/local command

## 3.85 compute pod/global command

### 3.85.1 Syntax

```
compute ID group-ID pod/atom param.pod coefficients.pod
compute ID group-ID podd/atom param.pod coefficients.pod
compute ID group-ID pod/local param.pod coefficients.pod
compute ID group-ID pod/global param.pod coefficients.pod
```

- ID, group-ID are documented in *compute* command
- pod/atom = style name of this compute command
- param.pod = the parameter file specifies parameters of the POD descriptors
- coefficients.pod = the coefficient file specifies coefficients of the POD potential

### 3.85.2 Examples

```
compute d all pod/atom Ta_param.pod
compute dd all podd/atom Ta_param.pod
compute ldd all pod/local Ta_param.pod
compute gdd all podd/global Ta_param.pod
compute d all pod/atom Ta_param.pod Ta_coefficients.pod
compute dd all podd/atom Ta_param.pod Ta_coefficients.pod
compute ldd all pod/local Ta_param.pod Ta_coefficients.pod
compute gdd all podd/global Ta_param.pod Ta_coefficients.pod
```

### 3.85.3 Description

Added in version 27June2024.

Define a computation that calculates a set of quantities related to the POD descriptors of the atoms in a group. These computes are used primarily for calculating the dependence of energy and force components on the linear coefficients in the *pod pair_style*, which is useful when training a POD potential to match target data. POD descriptors of an atom are characterized by the radial and angular distribution of neighbor atoms. The detailed mathematical definition is given in the papers by *(Nguyen and Rohskopf)*, *(Nguyen2023)*, *(Nguyen2024)*, and *(Nguyen and Sema)*.

Compute *pod/atom* calculates the per-atom POD descriptors.

Compute *podd/atom* calculates derivatives of the per-atom POD descriptors with respect to atom positions.

Compute *pod/local* calculates the per-atom POD descriptors and their derivatives with respect to atom positions.

Compute *pod/global* calculates the global POD descriptors and their derivatives with respect to atom positions.

Examples how to use Compute POD commands are found in the directory `examples/PACKAGES/pod`.

**Warning**

All of these compute styles produce *very* large per-atom output arrays that scale with the total number of atoms in the system. This will result in *very* large memory consumption for systems with a large number of atoms.

---

### 3.85.4 Output info

Compute *pod/atom* produces an 2D array of size $N \times M$, where $N$ is the number of atoms and $M$ is the number of descriptors. Each column corresponds to a particular POD descriptor.

Compute *podd/atom* produces an 2D array of size $N \times (M * 3N)$. Each column corresponds to a particular derivative of a POD descriptor.

Compute *pod/local* produces an 2D array of size $(1 + 3N) \times (M * N)$. The first row contains the per-atom descriptors, and the last 3N rows contain the derivatives of the per-atom descriptors with respect to atom positions.

Compute *pod/global* produces an 2D array of size $(1 + 3N) \times (M)$. The first row contains the global descriptors, and the last 3N rows contain the derivatives of the global descriptors with respect to atom positions.

### 3.85.5 Restrictions

These computes are part of the ML-POD package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.85.6 Related commands

*fitpod*, *pair_style pod*

### 3.85.7 Default

none

---

**(Nguyen and Rohskopf)** Nguyen and Rohskopf, Journal of Computational Physics, 480, 112030, (2023).

**(Nguyen2023)** Nguyen, Physical Review B, 107(14), 144103, (2023).

**(Nguyen2024)** Nguyen, Journal of Computational Physics, 113102, (2024).

**(Nguyen and Sema)** Nguyen and Sema, https://arxiv.org/abs/2405.00306, (2024).

## 3.86 compute pressure command

### 3.86.1 Syntax

```
compute ID group-ID pressure temp-ID keyword ...
```

- ID, group-ID are documented in *compute* command
- pressure = style name of this compute command
- temp-ID = ID of compute that calculates temperature, can be NULL if not needed
- zero or more keywords may be appended
- keyword = *ke* or *pair* or *bond* or *angle* or *dihedral* or *improper* or *kspace* or *fix* or *virial* or *pair/hybrid*

### 3.86.2 Examples

```
compute 1 all pressure thermo_temp
compute 1 all pressure NULL pair bond
compute 1 all pressure NULL pair/hybrid lj/cut
```

### 3.86.3 Description

Define a computation that calculates the pressure of the entire system of atoms. The specified group must be "all". See the *compute stress/atom* command if you want per-atom pressure (stress). These per-atom values could be summed for a group of atoms via the *compute reduce* command.

The pressure is computed by the formula

$$P = \frac{Nk_BT}{V} + \frac{1}{Vd}\sum_{i=1}^{N'} \vec{r}_i \cdot \vec{f}_i$$

where $N$ is the number of atoms in the system (see discussion of DOF below), $k_B$ is the Boltzmann constant, $T$ is the temperature, $d$ is the dimensionality of the system (2 for 2d, 3 for 3d), and $V$ is the system volume (or area in 2d). The second term is the virial, equal to $-dU/dV$, computed for all pairwise as well as 2-body, 3-body, 4-body, many-body,

and long-range interactions, where $\vec{r}_i$ and $\vec{f}_i$ are the position and force vector of atom $i$, and the dot indicates the dot product (scalar product). This is computed in parallel for each subdomain and then summed over all parallel processes. Thus $N'$ necessarily includes atoms from neighboring subdomains (so-called ghost atoms) and the position and force vectors of ghost atoms are thus included in the summation. Only when running in serial and without periodic boundary conditions is $N' = N$ the number of atoms in the system. *Fixes* that impose constraints (e.g., the *fix shake* command) may also contribute to the virial term.

A symmetric pressure tensor, stored as a 6-element vector, is also calculated by this compute. The six components of the vector are ordered $xx$, $yy$, $zz$, $xy$, $xz$, $yz$. The equation for the $(I, J)$ components (where $I$ and $J$ are $x$, $y$, or $z$) is similar to the above formula, except that the first term uses components related to the kinetic energy tensor and the second term uses components of the virial tensor:

$$P_{IJ} = \frac{1}{V} \sum_{k=1}^{N} m_k v_{k_I} v_{k_J} + \frac{1}{V} \sum_{k=1}^{N'} r_{k_I} f_{k_J}.$$

If no extra keywords are listed, the entire equations above are calculated. This includes a kinetic energy (temperature) term and the virial as the sum of pair, bond, angle, dihedral, improper, kspace (long-range), and fix contributions to the force on each atom. If any extra keywords are listed, then only those components are summed to compute temperature or ke and/or the virial. The *virial* keyword means include all terms except the kinetic energy *ke*.

The *pair/hybrid* keyword means to only include contribution from a sub-style in a *hybrid* or *hybrid/overlay* pair style.

Details of how LAMMPS computes the virial efficiently for the entire system, including for many-body potentials and accounting for the effects of periodic boundary conditions are discussed in *(Thompson)*.

The temperature and kinetic energy tensor are not calculated by this compute, but rather by the temperature compute specified with the command. See the doc pages for individual compute temp variants for an explanation of how they calculate temperature and a symmetric tensor (6-element vector) whose components are twice that of the traditional KE tensor. That tensor is what appears in the pressure tensor formula above.

If the kinetic energy is not included in the pressure, than the temperature compute is not used and can be specified as NULL. Normally the temperature compute used by compute pressure should calculate the temperature of all atoms for consistency with the virial term, but any compute style that calculates temperature can be used (e.g., one that excludes frozen atoms or other degrees of freedom).

Note that if desired the specified temperature compute can be one that subtracts off a bias to calculate a temperature using only the thermal velocity of the atoms (e.g., by subtracting a background streaming velocity). See the doc pages for individual *compute commands* to determine which ones include a bias.

Also note that the $N$ in the first formula above is really degrees-of-freedom divided by $d$ = dimensionality, where the DOF value is calculated by the temperature compute. See the various *compute temperature* styles for details.

A compute of this style with the ID of thermo_press is created when LAMMPS starts up, as if this command were in the input script:

```
compute thermo_press all pressure thermo_temp
```

where thermo_temp is the ID of a similarly defined compute of style "temp". See the *thermo_style* command for more details.

---

### 3.86.4 Output info

This compute calculates a global scalar (the pressure) and a global vector of length 6 (pressure tensor), which can be accessed by indices 1–6. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The ordering of values in the symmetric pressure tensor is as follows: $p_{xx}$, $p_{yy}$, $p_{zz}$, $p_{xy}$, $p_{xz}$, $p_{yz}$.

The scalar and vector values calculated by this compute are "intensive". The scalar and vector values will be in pressure *units*.

### 3.86.5 Restrictions

none

### 3.86.6 Related commands

*compute temp*, *compute stress/atom*, *thermo_style*, *fix numdiff/virial*,

### 3.86.7 Default

By default the compute includes contributions from the keywords: `ke pair bond angle dihedral improper kspace fix`

---

**(Thompson)** Thompson, Plimpton, Mattson, J Chem Phys, 131, 154107 (2009).

## 3.87 compute pressure/alchemy command

### 3.87.1 Syntax

```
compute ID group-ID pressure/alchemy fix-ID
```

- ID, group-ID are documented in *compute* command
- pressure/alchemy = style name of this compute command
- fix-ID = ID of *fix alchemy* command

### 3.87.2 Examples

```
fix trans all alchemy
compute mixed all pressure/alchemy trans
thermo_modify press mixed
```

### 3.87.3 Description

Added in version 28Mar2023.

Define a compute style that makes the "mixed" system pressure available for a system that uses the *fix alchemy* command to transform one topology to another. This can be used in combination with either *thermo_modify press* or *fix_modify press* to output and access a pressure consistent with the simulated combined two topology system.

The actual pressure is determined with *compute pressure* commands that are internally used by *fix alchemy* for each topology individually and then combined. This command just extracts the information from the fix.

The `examples/PACKAGES/alchemy` folder contains an example input for this command.

---

### 3.87.4 Output info

This compute calculates a global scalar (the pressure) and a global vector of length 6 (the pressure tensor), which can be accessed by indices 1–6. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The ordering of values in the symmetric pressure tensor is as follows: $p_{xx}$, $p_{yy}$, $p_{zz}$, $p_{xy}$, $p_{xz}$, $p_{yz}$.

The scalar and vector values calculated by this compute are "intensive". The scalar and vector values will be in pressure *units*.

### 3.87.5 Restrictions

This compute is part of the REPLICA package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.87.6 Related commands

*fix alchemy*, *compute pressure*, *thermo_modify*, *fix_modify*

### 3.87.7 Default

none

## 3.88 compute pressure/uef command

### 3.88.1 Syntax

```
compute ID group-ID pressure/uef temp-ID keyword ...
```

- ID, group-ID are documented in *compute* command
- pressure/uef = style name of this compute command
- temp-ID = ID of compute that calculates temperature, can be NULL if not needed
- zero or more keywords may be appended
- keyword = *ke* or *pair* or *bond* or *angle* or *dihedral* or *improper* or *kspace* or *fix* or *virial*

### 3.88.2 Examples

```
compute 1 all pressure/uef my_temp_uef
compute 2 all pressure/uef my_temp_uef virial
```

### 3.88.3 Description

This command is used to compute the pressure tensor in the reference frame of the applied flow field when *fix nvt/uef* or *fix npt/uef* is used. It is not necessary to use this command to compute the scalar value of the pressure. A *compute pressure* may be used for that purpose.

The keywords and output information are documented in *compute_pressure*.

### 3.88.4 Restrictions

This fix is part of the UEF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This command can only be used when *fix nvt/uef* or *fix npt/uef* is active.

The kinetic contribution to the pressure tensor will be accurate only when the compute specified by *temp-ID* is a *compute temp/uef*.

### 3.88.5 Related commands

*compute pressure*, *fix nvt/uef*, *compute temp/uef*

### 3.88.6 Default

none

## 3.89 compute property/atom command

### 3.89.1 Syntax

```
compute ID group-ID property/atom input1 input2 ...
```

- ID, group-ID are documented in *compute* command
- property/atom = style name of this compute command
- input = one or more atom attributes

```
possible attributes = id, mol, proc, type, mass,
                      x, y, z, xs, ys, zs, xu, yu, zu, ix, iy, iz,
                      vx, vy, vz, fx, fy, fz,
                      q, mux, muy, muz, mu,
                      spx, spy, spz, sp, fmx, fmy, fmz,
                      nbonds,
                      radius, diameter, omegax, omegay, omegaz,
                      temperature, heatflow,
                      angmomx, angmomy, angmomz,
                      shapex, shapey, shapez,
                      block1, block2,
                      inertiax, inertiay, inertiaz,
                      quatw, quati, quatj, quatk, tqx, tqy, tqz,
                      end1x, end1y, end1z, end2x, end2y, end2z,
                      corner1x, corner1y, corner1z,
                      corner2x, corner2y, corner2z,
                      corner3x, corner3y, corner3z,
                      i_name, d_name, i2_name[I], d2_name[I],
      history[I][J],
                      vfrac, s0, espin, eradius, ervel, erforce,
                      rho, drho, e, de, cv, buckling,
                      apip_lambda, apip_lambda_input, apip_e_fast,
                      apip_e_precise, apip_la_inp, apip_la_avg,
                      apip_la_norm
```

*id* = atom ID

```
mol = molecule ID
proc = ID of processor that owns atom
type = atom type
mass = atom mass
x,y,z = unscaled atom coordinates
xs,ys,zs = scaled atom coordinates
xu,yu,zu = unwrapped atom coordinates
ix,iy,iz = box image that the atom is in
vx,vy,vz = atom velocities
fx,fy,fz = forces on atoms
q = atom charge
mux,muy,muz = orientation of dipole moment of atom
mu = magnitude of dipole moment of atom
spx, spy, spz = direction of the atomic magnetic spin
sp = magintude of atomic magnetic spin moment
fmx, fmy, fmz = magnetic force
nbonds = number of bonds assigned to an atom
radius,diameter = radius,diameter of spherical particle
omegax,omegay,omegaz = angular velocity of spherical particle
temperature = internal temperature of spherical particle
heatflow = internal heat flow of spherical particle
angmomx,angmomy,angmomz = angular momentum of aspherical particle
shapex,shapey,shapez = 3 diameters of aspherical particle
block1,block2 = 2 blockiness exponents of aspherical (superellipsoid) particle
inertiax,inertiay,inertiaz = 3 principal moments of inertia of aspherical␣
↪(superellipsoid) particle
quatw,quati,quatj,quatk = quaternion components for aspherical or body particles
tqx,tqy,tqz = torque on finite-size particles
end12x, end12y, end12z = end points of line segment
corner123x, corner123y, corner123z = corner points of triangle
i_name = custom integer vector with name
d_name = custom floating point vector with name
i2_name[I] = Ith column of custom integer array with name
d2_name[I] = Ith column of custom floating-point array with name
history[I][J] = Ith most recent history frame (1 to Nrepeat) for Jth attribute (1␣
↪to Nattribute)

APIP package per-atom properties:
apip_lambda = switching parameter
apip_lambda_input = input used to calculate the switching parameter
apip_e_fast,apip_e_precise = potential energies mixed by the adaptive-precision␣
↪potential
apip_la_inp = input used for the local averaging of the descriptor
apip_la_norm = locally averaged radial weighting function used for normalization
apip_la_avg = locally averaged descriptor used to calculate the switching parameter
```

```
PERI package per-atom properties:
vfrac = volume fraction
s0 = critical bond stretch (bond-breaking threshold) of a particle
```

```
EFF package per-atom properties:
espin = electron spin
eradius = electron radius
ervel = electron radial velocity
```

(continues on next page)

(continued from previous page)

```
erforce = electron radial force
```

```
SPH package per-atom properties:
rho = density of SPH particles
drho = change in density
e = energy
de = change in thermal energy
cv = heat capacity
```

```
APIP package per-atom properties:
apip_lambda = switching parameter
apip_lambda_input = input used to calculate the switching parameter
apip_e_fast,apip_e_precise = potential energies mixed by the adaptive-precision␣
↪potential
```

- zero or more keyword/arg pairs may be appended
- keyword = *history*

```
history arg = fixID
  fixID = ID of a fix store/state command with history enabled
```

### 3.89.2 Examples

```
compute 1 all property/atom xs vx fx mux
compute 2 all property/atom type
compute 1 all property/atom ix iy iz
compute 3 all property/atom sp spx spy spz
compute 1 all property/atom i_myFlag d_Sxyz[1] d_Sxyz[3]
```

### 3.89.3 Description

Define a computation that simply stores atom attributes for each atom in the group. This is useful so that the values can be used by other *output commands* that take computes as inputs. See for example, the *compute reduce*, *fix ave/atom*, *fix ave/histo*, *fix ave/chunk*, and *atom-style variable* commands.

The list of possible attributes is essentially the same as that used by the *dump custom* command, which describes their meaning, with some additional quantities that are only defined for certain *atom styles*. The goal of this augmented list gives an input script access to any per-atom quantity stored by LAMMPS.

The values are stored in a per-atom vector or array as discussed below. Zeroes are stored for atoms not in the specified group or for quantities that are not defined for a particular particle in the group (e.g., *shapex* if the particle is not an ellipsoid).

Attributes *i_name*, *d_name*, *i2_name*, *d2_name* refer to custom per-atom integer and floating-point vectors or arrays that have been added via the *fix property/atom* command. When that command is used specific names are given to each attribute which are the "name" portion of these attributes. For arrays *i2_name* and *d2_name*, the column of the array must also be included following the name in brackets (e.g., d2_xyz[2] or i2_mySpin[3]).

Attribute *history* refers to per-atom history values stored by the *fix store/state* command using its history keyword. If this attribute is used, the optional *history* keyword must also be used to specify the *fixID* of the fix store/state command.

The associated *fix store/state* command specifies *Nattribute* per-atom attributes, and its history keyword specifies *Nrepeat* and *Nfreq*. *Nfreq* determines which timesteps the history can be accessed, which must be compatible with each timestep this compute is invoked. *Nrepeat* sets the count of how many history values are stored for each attribute. The first *I* index of attribute *history[I][J]* must be a value from 1 to *Nrepeat*, where 1 is the most recent history and *Nrepeat*

is the oldest history. If this compute is invoked before all *Nrepeat* history values have been stored (e.g. early in a run), then zeroes will be stored by this compute for not-yet-available history. The second *J* index of attribute *history[I][J]* must be a value from 1 to *Nattribute* for the per-atom attribute to access. See the *fix store/state* doc page for more details.

The additional per-atom attributes only accessible via this command, and not directly via the *dump custom* command, are as follows.

*Nbonds* is available for all molecular atom styles and refers to the number of explicit bonds assigned to an atom. Note that if the *newton bond* command is set to *on*, which is the default, then every bond in the system is assigned to only one of the two atoms in the bond. Thus a bond between atoms *I* and *J* may be tallied for either atom *I* or atom *J*. If *newton bond off* is set, it will be tallied with both atom *I* and atom *J*.

The quantities *shapex*, *shapey*, and *shapez* are defined for ellipsoidal particles and define the 3d shape of each particle.

Added in version 30Mar2026.

The quantities *block1*, and *block2*, are defined for superellipsoidal particles and define the blockiness of each superellipsoid particle. See the *set* command for an explanation of the blockiness.

Added in version 30Mar2026.

The quantities *inertiax*, *inertiay*, and *inertiaz* are defined for superellipsoidal particles and define the 3 principal moments of inertia of each particle. These are with respect to the particle's center of mass and in a reference system aligned with the particle's principal axes.

The quantities *quatw*, *quati*, *quatj*, and *quatk* are defined for ellipsoidal particles and body particles and store the 4-vector quaternion representing the orientation of each particle. See the *set* command for an explanation of the quaternion vector.

*End1x*, *end1y*, *end1z*, *end2x*, *end2y*, *end2z*, are defined for line segment particles and define the end points of each line segment.

*Corner1x*, *corner1y*, *corner1z*, *corner2x*, *corner2y*, *corner2z*, *corner3x*, *corner3y*, *corner3z*, are defined for triangular particles and define the corner points of each triangle.

The accessible quantities from the *APIP package* are explained in the doc pages of this package in detail. In short: *apip_lambda* is the switching parameter $\lambda \in [0,1]$. The switching parameter can be calculated from *apip_lambda_input* and mixes the energies of a fast (*apip_e_fast*) and a precise (*apip_e_precise*) potential into an adaptive-precision energy.

Changed in version 4Jul2026.

Alternatively, the switching parameter can be calculated from a locally averaged descriptor (*apip_la_avg*) to obtain a conservative potential. The descriptor is calculated from an atomic property (*apip_la_inp*) and normalized with a locally averaged weighting function (*apip_la_norm*).

**Note**

The energy according to the fast and the precise potential are only computed for the subset of atoms, for which it is required, i.e., for an atom $i$ with $\lambda_i = 1$ one does not need $E_i^{\text{precise}}$ and with $\lambda_i = 0$ one does not need $E_i^{\text{fast}}$.

In addition, the various per-atom quantities listed above for specific packages are only accessible by this command.

Changed in version 15Sep2022: The *espin* property was previously called *spin*.

The accessible quantities from the *APIP package* are explained in detail in the doc pages for this package's commands. In short: *apip_lambda* is the switching parameter $\lambda \in [0,1]$, that is calculated from *apip_lambda_input* and that mixes the energies of a fast (*apip_e_fast*) and a precise (*apip_e_precise*) potential into an adaptive-precision energy.

**Note**

The energy according to the fast and the precise potential are only computed for the subset of atoms for which it is required. I.e., for an atom $i$ with $\lambda_i = 1$, $E_i^{\text{precise}}$ is not needed. And with $\lambda_i = 0$, $E_i^{\text{fast}}$. is not needed.

### 3.89.4 Output info

This compute calculates a per-atom vector or per-atom array depending on the number of input values. If a single input is specified, a per-atom vector is produced. If two or more inputs are specified, a per-atom array is produced where the number of columns = the number of inputs. The vector or array can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The vector or array values will be in whatever *units* the corresponding attribute is in (e.g., velocity units for *vx*, charge units for *q*).

For the spin quantities, *sp* is in the units of the Bohr magneton; *spx*, *spy*, and *spz* are unitless quantities; and *fmx*, *fmy*, and *fmz* are given in rad/THz.

### 3.89.5 Restrictions

none

### 3.89.6 Related commands

*dump custom*, *compute reduce*, *fix ave/atom*, *fix ave/chunk*, *fix property/atom*, *fix store/state*

### 3.89.7 Default

none

## 3.90 compute property/chunk command

### 3.90.1 Syntax

```
compute ID group-ID property/chunk chunkID input1 input2 ...
```

- ID, group-ID are documented in *compute* command
- property/chunk = style name of this compute command
- chunkID = ID of *compute chunk/atom* command that defines the chunks
- input1,etc = one or more attributes

```
attributes = count, id, coord1, coord2, coord3
  count = # of atoms in chunk
  id = original chunk IDs before compression by compute chunk/atom
  coord123 = coordinates for spatial bins calculated by compute chunk/atom
```

### 3.90.2 Examples

```
compute 1 all property/chunk bin2d id count
compute 1 all property/chunk myChunks id coord1
```

### 3.90.3 Description

Define a computation that stores the specified attributes of chunks of atoms.

In LAMMPS, chunks are collections of atoms defined by a *compute chunk/atom* command, which assigns each atom to a single chunk (or no chunk). The ID for this command is specified as chunkID. For example, a single chunk could be the atoms in a molecule or atoms in a spatial bin. See the *compute chunk/atom* and *Howto chunk* doc pages for details of how chunks can be defined and examples of how they can be used to measure properties of a system.

This compute calculates and stores the specified attributes of chunks as global data so they can be accessed by other *output commands* and used in conjunction with other commands that generate per-chunk data, such as *compute com/chunk* or *compute msd/chunk*.

Note that only atoms in the specified group contribute to the calculation of the *count* attribute. The *compute chunk/atom* command defines its own group; atoms will have a chunk ID = 0 if they are not in that group, signifying they are not assigned to a chunk, and will thus also not contribute to this calculation. You can specify the "all" group for this command if you simply want to include atoms with non-zero chunk IDs.

The *count* attribute is the number of atoms in the chunk.

The *id* attribute stores the original chunk ID for each chunk. It can only be used if the *compress* keyword was set to *yes* for the *compute chunk/atom* command referenced by chunkID. This means that the original chunk IDs (e.g., molecule IDs) will have been compressed to remove chunk IDs with no atoms assigned to them. Thus a compressed chunk ID of 3 may correspond to an original chunk ID (molecule ID in this case) of 415. The *id* attribute will then be 415 for the third chunk.

The *coordN* attributes can only be used if a *binning* style was used

in the *compute chunk/atom* command referenced by chunkID. For *bin/1d*, *bin/2d*, and *bin/3d* styles the attribute is the center point of the bin in the corresponding dimension. Style *bin/1d* only defines a *coord1* attribute. Style *bin/2d* adds a *coord2* attribute. Style *bin/3d* adds a *coord3* attribute.

Note that if the value of the *units* keyword used in the *compute chunk/atom command* is *box* or *lattice*, the *coordN* attributes will be in distance *units*. If the value of the *units* keyword is *reduced*, the *coordN* attributes will be in unitless reduced units (0-1).

The simplest way to output the results of the compute property/chunk calculation to a file is to use the *fix ave/time* command, for example:

```
compute cc1 all chunk/atom molecule
compute myChunk1 all property/chunk cc1 count
compute myChunk2 all com/chunk cc1
fix 1 all ave/time 100 1 100 c_myChunk1 c_myChunk2[*] file tmp.out mode vector
```

### 3.90.4 Output info

This compute calculates a global vector or global array depending on the number of input values. The length of the vector or number of rows in the array is the number of chunks.

This compute calculates a global vector or global array where the number of rows = the number of chunks *Nchunk* as calculated by the specified *compute chunk/atom* command. If a single input is specified, a global vector is produced. If two or more inputs are specified, a global array is produced where the number of columns = the number of inputs. The vector or array can be accessed by any command that uses global values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The vector or array values are "intensive". The values will be unitless or in the units discussed above.

### 3.90.5 Restrictions

none

### 3.90.6 Related commands

*fix ave/chunk*

### 3.90.7 Default

none

## 3.91 compute property/grid command

### 3.91.1 Syntax

```
compute ID group-ID property/grid Nx Ny Nz input1 input2 ...
```

- ID, group-ID are documented in *compute* command
- property/grid = style name of this compute command
- Nx, Ny, Nz = grid size in each dimension
- input1,etc = one or more attributes

```
attributes = id, ix, iy, iz, x, y, z, xs, ys, zs, xc, yc, zc, xsc, ysc, zsc
  id = ID of grid cell, x fastest, y next, z slowest
  proc = processor ID (0 to Nprocs-1) which owns the grid cell
  ix,iy,iz = grid indices in each dimension (1 to N inclusive)
  x,y,z = coords of lower left corner of grid cell
  xs,ys,zs = scaled coords of lower left corner of grid cell (0.0 to 1.0)
  xc,yc,zc = coords of center point of grid cell
  xsc,ysc,zsc = scaled coords of center point of grid cell (0.0 to 1.0)
```

### 3.91.2 Examples

```
compute 1 all property/grid 10 10 20 id ix iy iz
compute 1 all property/grid 100 100 1 id xc yc zc
```

### 3.91.3 Description

Define a computation that stores the specified attributes of a distributed grid. In LAMMPS, distributed grids are regular 2d or 3d grids which overlay a 2d or 3d simulation domain. Each processor owns the grid cells whose center points lie within its subdomain. See the *Howto grid* doc page for details of how distributed grids can be defined by various commands and referenced.

This compute stores the specified attributes of grids as per-grid data so they can be accessed by other *output commands* such as *dump grid*.

*Nx*, *Ny*, and *Nz* define the size of the grid. For a 2d simulation *Nz* must be 1. When this compute is used by *dump grid*, to output per-grid values from other computes of fixes, the grid size specified for this command must be consistent with the grid sizes used by the other commands.

The *id* attribute is the grid ID for each grid cell. For a global grid of size Nx by Ny by Nz (in 3d simulations) the grid IDs range from 1 to Nx*Ny*Nz. They are ordered with the X index of the 3d grid varying fastest, then Y, then Z slowest. For 2d grids (in 2d simulations), the grid IDs range from 1 to Nx*Ny, with X varying fastest and Y slowest.

Added in version 21Nov2023.

The *proc* attribute is the ID of the processor which owns the grid cell. Processor IDs range from 0 to Nprocs - 1, where Nprocs is the number of processors the simulation is running on. Each grid cell is owned by a single processor.

The *ix*, *iy*, *iz* attributes are the indices of a grid cell in each dimension. They range from 1 to Nx inclusive in the X dimension, and similar for Y and Z.

The *x*, *y*, *z* attributes are the coordinates of the lower left corner point of each grid cell.

The *xs*, *ys*, *zs* attributes are also coordinates of the lower left corner point of each grid cell, except in scaled coordinates, where the lower-left corner of the entire simulation box is (0,0,0) and the upper right corner is (1,1,1).

The *xc*, *yc*, *zc* attributes are the coordinates of the center point of each grid cell.

The *xsc*, *ysc*, *zsc* attributes are also coordinates of the center point each grid cell, except in scaled coordinates, where the lower-left corner of the entire simulation box is (0,0,0) and the upper right corner is (1,1,1).

For *triclinic simulation boxes*, the grid point coordinates for (x,y,z) and (xc,yc,zc) will reflect the triclinic geometry. For (xs,yz,zs) and (xsc,ysc,zsc), the coordinates are the same for orthogonal versus triclinic boxes.

### 3.91.4 Output info

This compute calculates a per-grid vector or array depending on the number of input values. The length of the vector or number of array rows (distributed across all processors) is Nx * Ny * Nz. For access by other commands, the name of the single grid produced by this command is "grid". The name of its per-grid data is "data".

The (x,y,z) and (xc,yc,zc) coordinates are in distance *units*.

### 3.91.5 Restrictions

For 2d simulations, the attributes which refer to the Z dimension cannot be used.

### 3.91.6 Related commands

*dump grid*

### 3.91.7 Default

none

## 3.92 compute property/local command

### 3.92.1 Syntax

```
compute ID group-ID property/local attribute1 attribute2 ... keyword args ...
```

- ID, group-ID are documented in *compute* command
- property/local = style name of this compute command
- one or more attributes of the same type (neighbor, pair, bond, angle, dihedral, or improper) may be appended

```
possible attributes = natom1, natom2, ntype1, ntype2,
                      patom1, patom2, ptype1, ptype2,
                      batom1, batom2, btype,
                      aatom1, aatom2, aatom3, atype,
                      datom1, datom2, datom3, datom4, dtype,
                      iatom1, iatom2, iatom3, iatom4, itype
```

  - Neighbor attributes

```
natom1, natom2 = store IDs of two atoms in each pair (within neighbor cutoff)
ntype1, ntype2 = store types of two atoms in each pair (within neighbor cutoff)
```

  - Pair attributes

```
patom1, patom2 = store IDs of two atoms in each pair (within force cutoff)
ptype1, ptype2 = store types of two atoms in each pair (within force cutoff)
```

  - Bond attributes

```
batom1, batom2 = store IDs of two atoms in each bond
btype = store bond type of each bond
```

  - Angle attributes

```
aatom1, aatom2, aatom3 = store IDs of three atoms in each angle
atype = store angle type of each angle
```

  - Dihedral attributes

```
datom1, datom2, datom3, datom4 = store IDs of 4 atoms in each dihedral
dtype = store dihedral type of each dihedral
```

  - Improper attributes

```
iatom1, iatom2, iatom3, iatom4 = store IDs of 4 atoms in each improper
itype = store improper type of each improper
```

- zero or more keyword/arg pairs may be appended
- keyword = *cutoff*

  *cutoff* arg = *type* or *radius*

### 3.92.2 Examples

```
compute 1 all property/local btype batom1 batom2
compute 1 all property/local atype aatom2
```

### 3.92.3 Description

Define a computation that stores the specified attributes as local data so it can be accessed by other *output commands*. If the input attributes refer to bond information, then the number of datums generated, aggregated across all processors, equals the number of bonds in the system. Ditto for pairs, angles, etc.

If multiple attributes are specified then they must all generate the same amount of information, so that the resulting local array has the same number of rows for each column. This means that only bond attributes can be specified together, or angle attributes, etc. Bond and angle attributes cannot be mixed in the same compute property/local command.

If the inputs are pair attributes, the local data is generated by looping over the pairwise neighbor list. Info about an individual pairwise interaction will only be included if both atoms in the pair are in the specified compute group. For *natom1* and *natom2*, all atom pairs in the neighbor list are considered (out to the neighbor cutoff = force cutoff + *neighbor skin*). For *patom1* and *patom2*, the distance between the atoms must be less than the force cutoff distance for that pair to be included, as defined by the *pair_style* and *pair_coeff* commands.

The optional *cutoff* keyword determines how the force cutoff distance for an interaction is determined for the *patom1* and *patom2* attributes. For the default setting of *type*, the pairwise cutoff defined by the *pair_style* command for the types of the two atoms is used. For the *radius* setting, the sum of the radii of the two particles is used as a cutoff. For example, this is appropriate for granular particles which only interact when they are overlapping, as computed by *granular pair styles*. Note that if a granular model defines atom types such that all particles of a specific type are monodisperse (same diameter), then the two settings are effectively identical.

If the inputs are bond, angle, etc attributes, the local data is generated by looping over all the atoms owned on a processor and extracting bond, angle, etc info. For bonds, info about an individual bond will only be included if both atoms in the bond are in the specified compute group. Likewise for angles, dihedrals, etc.

For bonds and angles, a bonds/angles that have been broken by setting their bond/angle type to 0 will not be included. Bonds/angles that have been turned off (see the *fix shake* or *delete_bonds* commands) by setting their bond/angle type negative are written into the file. This is consistent with the *compute bond/local* and *compute angle/local* commands

Note that as atoms migrate from processor to processor, there will be no consistent ordering of the entries within the local vector or array from one timestep to the next. The only consistency that is guaranteed is that the ordering on a particular timestep will be the same for local vectors or arrays generated by other compute commands. For example, output from the *compute bond/local* command can be combined with bond atom indices from this command and output by the *dump local* command in a consistent way.

The *natom1* and *natom2* or *patom1* and *patom2* attributes refer to the atom IDs of the two atoms in each pairwise interaction computed by the *pair_style* command. The *ntype1* and *ntype2* or *ptype1* and *ptype2* attributes refer to the atom types of the two atoms in each pairwise interaction.

**Note**

For pairs, if two atoms $I, J$ are involved in 1–2, 1–3, 1–4 interactions within the molecular topology, their pairwise interaction may be turned off, and thus they may not appear in the neighbor list, and will not be part of the local data created by this command. More specifically, this may be true of $I, J$ pairs with a weighting factor of 0.0; pairs with a non-zero weighting factor are included. The weighting factors for 1–2, 1–3, and 1–4 pairwise interactions are set by the *special_bonds* command.

The *batom1* and *batom2* attributes refer to the atom IDs of the 2 atoms in each *bond*. The *btype* attribute refers to the type of the bond, from 1 to Nbtypes = # of bond types. The number of bond types is defined in the data file read by the *read_data* command.

The attributes that start with "a", "d", and "i" refer to similar values for *angles*, *dihedrals*, and *impropers*.

### 3.92.4 Output info

This compute calculates a local vector or local array depending on the number of input values. The length of the vector or number of rows in the array is the number of bonds, angles, etc. If a single input is specified, a local vector is produced. If two or more inputs are specified, a local array is produced where the number of columns = the number of inputs. The vector or array can be accessed by any command that uses local values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The vector or array values will be integers that correspond to the specified attribute.

### 3.92.5 Restrictions

none

### 3.92.6 Related commands

*dump local*, *compute reduce*

### 3.92.7 Default

The keyword default is cutoff = type.

## 3.93 compute ptm/atom command

### 3.93.1 Syntax

```
compute ID group-ID ptm/atom structures threshold group2-ID
```

- ID, group-ID are documented in *compute* command
- ptm/atom = style name of this compute command
- structures = *default* or *all* or any hyphen-separated combination of *fcc*, *hcp*, *bcc*, *ico*, *sc*, *dcub*, *dhex*, or *graphene* = structure types to search for
- threshold = lattice distortion threshold (RMSD)
- group2-ID determines which group is used for neighbor selection (optional, default "all")

### 3.93.2 Examples

```
compute 1 all ptm/atom default 0.1 all
compute 1 all ptm/atom fcc-hcp-dcub-dhex 0.15 all
compute 1 all ptm/atom all 0
```

### 3.93.3 Description

Define a computation that determines the local lattice structure around an atom using the PTM (Polyhedral Template Matching) method. The PTM method is described in *(Larsen)*.

Currently, there are seven lattice structures PTM recognizes:

- fcc = 1
- hcp = 2
- bcc = 3
- ico (icosahedral) = 4
- sc (simple cubic) = 5
- dcub (diamond cubic) = 6
- dhex (diamond hexagonal) = 7
- graphene = 8

The value of the PTM structure will be 0 for unknown types and −1 for atoms not in the specified compute group. The choice of structures to search for can be specified using the "structures" argument, which is a hyphen-separated list of structure keywords. Two convenient pre-set options are provided:

- default: fcc-hcp-bcc-ico
- all: fcc-hcp-bcc-ico-sc-dcub-dhex-graphene

The 'default' setting detects the same structures as the Common Neighbor Analysis method. The 'all' setting searches for all structure types. A performance penalty is incurred for the diamond and graphene structures, so it is not recommended to use this option if it is known that the simulation does not contain these structures.

PTM identifies structures using two steps. First, a graph isomorphism test is used to identify potential structure matches. Next, the deviation is computed between the local structure (in the simulation) and a template of the ideal lattice structure. The deviation is calculated as:

$$\mathrm{RMSD}(\mathbf{u},\mathbf{v}) = \min_{s,\mathbf{Q}} \sqrt{\frac{1}{N}\sum_{i=1}^{N} \left\| s[\vec{u}_i - \bar{\mathbf{u}}] - \mathbf{Q}\cdot\vec{v}_i \right\|^2}$$

Here, $\vec{u}$ and $\vec{v}$ contain the coordinates of the local and ideal structures respectively, $s$ is a scale factor, and $\mathbf{Q}$ is a rotation. The best match is identified by the lowest RMSD value, using the optimal scaling, rotation, and correspondence between the points.

The *threshold* keyword sets an upper limit on the maximum permitted deviation before a local structure is identified as disordered. Typical values are in the range 0.1–0.15, but larger values may be desirable at higher temperatures. A value of 0 is equivalent to infinity and can be used if no threshold is desired.

The neighbor list needed to compute this quantity is constructed each time the calculation is performed (e.g., each time a snapshot of atoms is dumped). Thus it can be inefficient to compute/dump this quantity too frequently or to have multiple compute/dump commands, each with a *ptm/atom* style. By default the compute processes **all** neighbors unless the optional *group2-ID* argument is given, then only members of that group are considered as neighbors.

### 3.93.4 Output info

This compute calculates a per-atom array, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

Results are stored in the per-atom array in the following order:

- type
- rmsd
- interatomic distance
- qw
- qx
- qy
- qz

The type is a number from −1 to 8. The rmsd is a positive real number. The interatomic distance is computed from the scale factor in the RMSD equation. The $(qw, qx, qy, qz)$ parameters represent the orientation of the local structure in quaternion form. The reference coordinates for each template (from which the orientation is determined) can be found in the *ptm_constants.h* file in the PTM source directory. For atoms that are not within the compute group-ID, all values are set to zero.

### 3.93.5 Restrictions

This fix is part of the PTM package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.93.6 Related commands

*compute centro/atom compute cna/atom*

### 3.93.7 Default

none

---

**(Larsen)** Larsen, Schmidt, Schiotz, Modelling Simul Mater Sci Eng, 24, 055007 (2016).

## 3.94 compute rattlers/atom command

### 3.94.1 Syntax

```
compute ID group-ID rattlers/atom cutoff zmin ntries
```

- ID, group-ID are documented in *compute* command
- rattlers/atom = style name of this compute command
- cutoff = *type* or *radius*

  ```
  type = cutoffs determined based on atom types
  radius = cutoffs determined based on atom diameters (atom style sphere)
  ```

- zmin = minimum coordination for a non-rattler atom
- ntries = maximum number of iterations to remove rattlers

### 3.94.2 Examples

```
compute 1 all rattlers/atom type 4 10
```

### 3.94.3 Description

Added in version 7Feb2024.

Define a compute that identifies rattlers in a system. Rattlers are often identified in granular or glassy packings as under-coordinated atoms that do not have the required number of contacts to constrain their translational degrees of freedom. Such atoms are not considered rigid and can often freely rattle around in the system. This compute identifies rattlers which can be helpful for excluding them from analysis or providing extra damping forces to accelerate relaxation processes.

Rattlers are identified using an interactive approach. The coordination number of all atoms is first calculated. The *type* and *radius* settings are used to select whether interaction cutoffs are determined by atom types or by the sum of atomic radii (atom style sphere), respectively. Rattlers are then identified as atoms with a coordination number less than *zmin* and are removed from consideration. Atomic coordination numbers are then recalculated, excluding previously identified rattlers, to identify a new set of rattlers. This process is iterated up to a maximum of *ntries* or until no new rattlers are identified and the remaining atoms form a stable network of contacts.

In dense homogeneous systems where the average atom coordination number is expected to be larger than *zmin*, this process usually only takes a few iterations and a value of *ntries* around ten may be sufficient. In systems with significant

heterogeneity or average coordination numbers less than *zmin*, an appropriate value of *ntries* depends heavily on the specific system. For instance, a linear chain of N rattler atoms with a *zmin* of 2 would take N/2 iterations to identify that all the atoms are rattlers.

### 3.94.4 Output info

This compute calculates a per-atom vector and a global scalar. The vector designates which atoms are rattlers, indicated by a value 1. Non-rattlers have a value of 0. The global scalar returns the total number of rattlers in the system. See the *Howto output* page for an overview of LAMMPS output options.

### 3.94.5 Restrictions

This compute is part of the EXTRA-COMPUTE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The *radius* cutoff option requires that atoms store a radius as defined by the *atom_style sphere* or similar commands.

### 3.94.6 Related commands

*compute coord/atom compute contact/atom*

### 3.94.7 Default

none

## 3.95 compute rdf command

Accelerator Variants: *rdf/omp*

### 3.95.1 Syntax

```
compute ID group-ID rdf Nbin itype1 jtype1 itype2 jtype2 ... keyword/value ...
```

- ID, group-ID are documented in *compute* command
- rdf = style name of this compute command
- Nbin = number of RDF bins
- itypeN = central atom type for Nth RDF histogram (integer, type label, or asterisk form)
- jtypeN = distribution atom type for Nth RDF histogram (integer, type label, or asterisk form)
- zero or more keyword/value pairs may be appended
- keyword = *cutoff*

```
cutoff value = Rcut
  Rcut = cutoff distance for RDF computation (distance units)
```

### 3.95.2 Examples

```
compute 1 all rdf 100
compute 1 all rdf 100 1 1
compute 1 all rdf 100 * 3 cutoff 5.0
compute 1 fluid rdf 500 1 1 1 2 2 1 2 2
compute 1 fluid rdf 500 1*3 2 5 *10 cutoff 3.5
```

### 3.95.3 Description

Define a computation that calculates the radial distribution function (RDF), also called $g(r)$, and the coordination number for a group of particles. Both are calculated in histogram form by binning pairwise distances into *Nbin* bins from 0.0 to the maximum force cutoff defined by the *pair_style* command or the cutoff distance *Rcut* specified via the *cutoff* keyword. The bins are of uniform size in radial distance. Thus a single bin encompasses a thin shell of distances in 3d and a thin ring of distances in 2d.

**Note**

If you have a bonded system, then the settings of *special_bonds* command can remove pairwise interactions between atoms in the same bond, angle, or dihedral. This is the default setting for the *special_bonds* command, and means those pairwise interactions do not appear in the neighbor list. Because this fix uses a neighbor list, it also means those pairs will not be included in the RDF. This does not apply when using long-range coulomb interactions (*coul/long*, *coul/msm*, *coul/wolf* or similar. One way to get around this would be to set special_bond scaling factors to very tiny numbers that are not exactly zero (e.g., $1.0 \times 10^{-50}$). Another workaround is to write a dump file, and use the *rerun* command to compute the RDF for snapshots in the dump file. The rerun script can use a *special_bonds* command that includes all pairs in the neighbor list.

By default the RDF is computed out to the maximum force cutoff defined by the *pair_style* command. If the *cutoff* keyword is used, then the RDF is computed accurately out to the $Rcut > 0.0$ distance specified.

**Note**

Normally, you should only use the *cutoff* keyword if no pair style is defined (e.g., the *rerun* command is being used to post-process a dump file of snapshots) or if you really want the RDF for distances beyond the pair_style force cutoff and cannot easily post-process a dump file to calculate it. This is because using the *cutoff* keyword incurs extra computation and possibly communication, which may slow down your simulation. If you specify $Rcut \leq$ force cutoff, you will force an additional neighbor list to be built at every timestep this command is invoked (or every reneighboring timestep, whichever is less frequent), which is inefficient. LAMMPS will warn you if this is the case. If you specify a *Rcut* > force cutoff, you must ensure ghost atom information out to *Rcut* + *skin* is communicated, via the *comm_modify cutoff* command, else the RDF computation cannot be performed, and LAMMPS will give an error message. The *skin* value is what is specified with the *neighbor* command. In this case, you are forcing a large neighbor list to be built just for the RDF computation, and extra communication to be performed every timestep.

The *itypeN* and *jtypeN* arguments are optional. These arguments must come in pairs. If no pairs are listed, then a single histogram is computed for $g(r)$ between all atom types. If one or more pairs are listed, then a separate histogram is generated for each *itype*,*jtype* pair.

The *itypeN* and *jtypeN* settings can be specified in one of three ways. One or both of the types in the I,J pair can be a *type label*. Or an explicit numeric value can be used, as in the fourth example above. Or a wild-card asterisk can be used to specify a range of atom types. This takes the form "*" or "*n" or "m*" or "m*n". If $N$ is the number of atom types, then an asterisk with no numeric values means all types from 1 to $N$. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from m to $N$ (inclusive). A middle asterisk means all types from m to n (inclusive).

If both *itypeN* and *jtypeN* are single values, as in the fourth example above, this means that a $g(r)$ is computed where atoms of type *itypeN* are the central atom, and atoms of type *jtypeN* are the distribution atom. If either *itypeN* and *jtypeN* represent a range of values via the wild-card asterisk, as in the fifth example above, this means that a $g(r)$ is computed where atoms of any of the range of types represented by *itypeN* are the central atom, and atoms of any of the range of types represented by *jtypeN* are the distribution atom.

Pairwise distances are generated by looping over a pairwise neighbor list, just as they would be in a *pair_style* computation. The distance between two atoms $I$ and $J$ is included in a specific histogram if the following criteria are

met:

- atoms *I* and *J* are both in the specified compute group
- the distance between atoms *I* and *J* is less than the maximum force cutoff
- the type of the *I* atom matches *itypeN* (one or a range of types)
- the type of the *J* atom matches *jtypeN* (one or a range of types)

It is OK if a particular pairwise distance is included in more than one individual histogram, due to the way the *itypeN* and *jtypeN* arguments are specified.

The $g(r)$ value for a bin is calculated from the histogram count by scaling it by the idealized number of how many counts there would be if atoms of type *jtypeN* were uniformly distributed. Thus it involves the count of *itypeN* atoms, the count of *jtypeN* atoms, the volume of the entire simulation box, and the volume of the bin's thin shell in 3d (or the area of the bin's thin ring in 2d).

A coordination number coord($r$) is also calculated, which is the number of atoms of type *jtypeN* within the current bin or closer, averaged over atoms of type *itypeN*. This is calculated as the area- or volume-weighted sum of $g(r)$ values over all bins up to and including the current bin, multiplied by the global average volume density of atoms of type *jtypeN*.

The simplest way to output the results of the compute rdf calculation to a file is to use the *fix ave/time* command, for example:

```
compute myRDF all rdf 50
fix 1 all ave/time 100 1 100 c_myRDF[*] file tmp.rdf mode vector
```

### 3.95.4 Output info

This compute calculates a global array in which the number of rows is *Nbins* and the number of columns is $1 + 2N_{\text{pairs}}$, where $N_{\text{pairs}}$ is the number of $I, J$ pairings specified. The first column has the bin coordinate (center of the bin), and each successive set of two columns has the $g(r)$ and coord($r$) values for a specific set of *itypeN* versus *jtypeN* interactions, as described above. These values can be used by any command that uses a global values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The array values calculated by this compute are all "intensive".

The first column of array values will be in distance *units*. The $g(r)$ columns of array values are normalized numbers $\geq 0.0$. The coordination number columns of array values are also numbers $\geq 0.0$.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 3.95.5 Restrictions

By default, the RDF is not computed for distances longer than the largest force cutoff, since the neighbor list creation will only contain pairs up to that distance (plus neighbor list skin). This distance can be increased using the *cutoff* keyword but this keyword is only valid with *neighbor styles 'bin' and 'nsq'*.

If you want an RDF for larger distances, you can also use the *rerun* command to post-process a dump file, use *pair style zero* and set the force cutoff to be longer in the rerun script. Note that in the rerun context, the force cutoff is arbitrary and with pair style zero you are not computing any forces, and you are not running dynamics you are not changing the model that generated the trajectory.

The definition of $g(r)$ used by LAMMPS is only appropriate for characterizing atoms that are uniformly distributed throughout the simulation cell. In such cases, the coordination number is still correct and meaningful. As an example, if a large simulation cell contains only one atom of type *itypeN* and one of *jtypeN*, then $g(r)$ will register an arbitrarily large spike at whatever distance they happen to be at, and zero everywhere else. The function coord($r$) will show a step change from zero to one at the location of the spike in $g(r)$.

**Note**

compute rdf can handle dynamic groups and systems where atoms are added or removed, but this causes that certain normalization parameters need to be re-computed in every step and include collective communication operations. This will reduce performance and limit parallel efficiency and scaling. For systems, where only the type of atoms changes (e.g., when using *fix atom/swap*), you need to explicitly request the dynamic normalization updates via *compute_modify dynamic/dof yes*

### 3.95.6 Related commands

*fix ave/time*, *compute_modify*, *compute adf*

### 3.95.7 Default

The keyword defaults are cutoff = 0.0 (use the pairwise force cutoff).

## 3.96 compute reaxff/atom command

Accelerator Variants: *reaxff/atom/kk*

### 3.96.1 Syntax

```
compute ID group-ID reaxff/atom attribute args ... keyword value ...
```

- ID, group-ID are documented in *compute* command
- reaxff/atom = name of this compute command
- attribute = *pair*

```
pair args = nsub
  nsub = n-instance of a sub-style, if a pair style is used multiple times in a␣
↪hybrid style
```

- keyword = *bonds*

```
bonds value = no or yes
  no = ignore list of local bonds
  yes = include list of local bonds
```

### 3.96.2 Examples

```
compute 1 all reaxff/atom bonds yes
```

### 3.96.3 Description

Added in version 7Feb2024.

Define a computation that extracts bond information computed by the ReaxFF potential specified by *pair_style reaxff*.

By default, it produces per-atom data that includes the following columns:

- abo = atom bond order (sum of all bonds)
- nlp = number of lone pairs
- nb = number of bonds

Bonds will only be included if its atoms are in the group.

In addition, if `bonds` is set to `yes`, the compute will also produce a local array of all bonds on the current processor whose atoms are in the group. The columns of each entry of this local array are:

- id_i = atom i id of bond
- id_j = atom j id of bond
- bo = bond order of bond

### 3.96.4 Output info

This compute calculates a per-atom array and local array depending on the number of keywords. The number of rows in the local array is the number of bonds as described above. Both per-atom and local array have 3 columns.

The arrays can be accessed by any command that uses local and per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 3.96.5 Restrictions

The compute reaxff/atom command requires that the *pair_style reaxff* is invoked. This fix is part of the REAXFF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.96.6 Related commands

*pair_style reaxff*

### 3.96.7 Default

The option defaults are *bonds* = *no*.

## 3.97 compute reduce command

## 3.98 compute reduce/region command

### 3.98.1 Syntax

```
compute ID group-ID style arg mode input1 input2 ... keyword args ...
```

- ID, group-ID are documented in *compute* command
- style = *reduce* or *reduce/region*

  ```
  reduce arg = none
  reduce/region arg = region-ID
    region-ID = ID of region to use for choosing atoms
  ```

- mode = *sum* or *min* or *minabs* or *max* or *maxabs* or *ave* or *sumsq* or *avesq* or *sumabs* or *aveabs*
- one or more inputs can be listed
- input = *x* or *y* or *z* or *vx* or *vy* or *vz* or *fx* or *fy* or *fz* or c_ID or c_ID[N] or f_ID or f_ID[N] or v_name

  ```
  x,y,z,vx,vy,vz,fx,fy,fz = atom attribute (position, velocity, force component)
  c_ID = per-atom or local vector calculated by a compute with ID
  c_ID[I] = Ith column of per-atom or local array calculated by a compute with ID, I␣
  ↪can include wildcard (see below)
  f_ID = per-atom or local vector calculated by a fix with ID
  f_ID[I] = Ith column of per-atom or local array calculated by a fix with ID, I can␣
  ↪include wildcard (see below)
  v_name = per-atom vector calculated by an atom-style variable with name
  ```

- zero or more keyword/args pairs may be appended
- keyword = *replace* or *inputs*

  ```
  replace args = vec1 vec2
    vec1 = reduced value from this input vector will be replaced
    vec2 = replace it with vec1[N] where N is index of max/min value from vec2
  inputs arg = peratom or local
    peratom = all inputs are per-atom quantities (default)
    local = all input are local quantities
  ```

### 3.98.2 Examples

```
compute 1 all reduce sum c_force
compute 1 all reduce/region subbox sum c_force
compute 2 all reduce min c_press[2] f_ave v_myKE
compute 2 all reduce min c_press[*] f_ave v_myKE inputs peratom
```

(continues on next page)

(continued from previous page)

```
compute 3 fluid reduce max c_index[1] c_index[2] c_dist replace 1 3 replace 2 3
compute 4 all reduce max c_bond inputs local
```

### 3.98.3 Description

Define a calculation that "reduces" one or more vector inputs into scalar values, one per listed input. For the compute reduce command, the inputs can be either per-atom or local quantities and must all be of the same kind (per-atom or local); see discussion of the optional *inputs* keyword below. The compute reduce/region command can only be used with per-atom inputs.

Atom attributes are per-atom quantities, *computes* and *fixes* can generate either per-atom or local quantities, and *atom-style variables* generate per-atom quantities. See the *variable* command and its special functions which can perform the same reduction operations as the compute reduce command on global vectors.

The reduction operation is specified by the *mode* setting. The *sum* option adds the values in the vector into a global total. The *min* or *max* options find the minimum or maximum value across all vector values. The *minabs* or *maxabs* options find the minimum or maximum value across all absolute vector values. The *ave* setting adds the vector values into a global total, then divides by the number of values in the vector. The *sumsq* option sums the square of the values in the vector into a global total. The *avesq* setting does the same as *sumsq*, then divides the sum of squares by the number of values. The last two options can be useful for calculating the variance of some quantity (e.g., variance = $avesq - ave^2$). The *sumabs* option sums the absolute values in the vector into a global total. The *aveabs* setting does the same as *sumabs*, then divides the sum of absolute values by the number of values.

Each listed input is operated on independently. For per-atom inputs, the group specified with this command means only atoms within the group contribute to the result. Likewise for per-atom inputs, if the compute reduce/region command is used, the atoms must also currently be within the region. Note that an input that produces per-atom quantities may define its own group which affects the quantities it returns. For example, if a compute is used as an input which generates a per-atom vector, it will generate values of 0.0 for atoms that are not in the group specified for that compute.

Each listed input can be an atom attribute (position, velocity, force component) or can be the result of a *compute* or *fix* or the evaluation of an atom-style *variable*.

Note that for values from a compute or fix, the bracketed index *I* can be specified using a wildcard asterisk with the index to effectively specify multiple values. This takes the form "*" or "*n" or "m*" or "m*n". If *N* is the size of the vector (for *mode* = scalar) or the number of columns in the array (for *mode* = vector), then an asterisk with no numeric values means all indices from 1 to *N*. A leading asterisk means all indices from 1 to n (inclusive). A trailing asterisk means all indices from m to *N* (inclusive). A middle asterisk means all indices from m to n (inclusive).

Using a wildcard is the same as if the individual columns of the array had been listed one by one. For example, the following two compute reduce commands are equivalent, since the *compute stress/atom* command creates a per-atom array with six columns:

```
compute myPress all stress/atom NULL
compute 2 all reduce min c_myPress[*]
compute 2 all reduce min c_myPress[1] c_myPress[2] c_myPress[3] &
                         c_myPress[4] c_myPress[5] c_myPress[6]
```

---

The atom attribute values (*x*, *y*, *z*, *vx*, *vy*, *vz*, *fx*, *fy*, and *fz*) are self-explanatory. Note that other atom attributes can be used as inputs to this fix by using the *compute property/atom* command and then specifying an input value from that compute.

If a value begins with "c_", a compute ID must follow which has been previously defined in the input script. Valid computes can generate per-atom or local quantities. See the individual *compute* page for details. If no bracketed integer is appended, the vector calculated by the compute is used. If a bracketed integer is appended, the Ith column of the array

calculated by the compute is used. Users can also write code for their own compute styles and *add them to LAMMPS*. See the discussion above for how *I* can be specified with a wildcard asterisk to effectively specify multiple values.

If a value begins with "f_", a fix ID must follow which has been previously defined in the input script. Valid fixes can generate per-atom or local quantities. See the individual *fix* page for details. Note that some fixes only produce their values on certain timesteps, which must be compatible with when compute reduce references the values, else an error results. If no bracketed integer is appended, the vector calculated by the fix is used. If a bracketed integer is appended, the Ith column of the array calculated by the fix is used. Users can also write code for their own fix style and *add them to LAMMPS*. See the discussion above for how *I* can be specified with a wildcard asterisk to effectively specify multiple values.

If a value begins with "v_", a variable name must follow which has been previously defined in the input script. It must be an *atom-style variable*. Atom-style variables can reference thermodynamic keywords and various per-atom attributes, or invoke other computes, fixes, or variables when they are evaluated, so this is a very general means of generating per-atom quantities to reduce.

---

If the *replace* keyword is used, two indices *vec1* and *vec2* are specified, where each index ranges from 1 to the number of input values. The replace keyword can only be used if the *mode* is *min* or *max*. It works as follows. A min/max is computed as usual on the *vec2* input vector. The index *N* of that value within *vec2* is also stored. Then, instead of performing a min/max on the *vec1* input vector, the stored index is used to select the *N*th element of the *vec1* vector.

Thus, for example, if you wish to use this compute to find the bond with maximum stretch, you can do it as follows:

```
compute batoms all property/local batom1 batom2
compute blength all bond/local dist
compute 3 all reduce max c_batoms[1] c_batoms[2] c_blength replace 1 3 replace 2 3␣
↪inputs local
thermo_style custom step temp c_3[1] c_3[2] c_3[3]
```

The first two input values in the compute reduce command are vectors with the IDs of the two atoms in each bond, using the *compute property/local* command. The last input value is bond distance, using the *compute bond/local* command. Instead of taking the max of the two atom ID vectors, which does not yield useful information in this context, the *replace* keywords will extract the atom IDs for the two atoms in the bond of maximum stretch. These atom IDs and the bond stretch will be printed with thermodynamic output.

Added in version 21Nov2023.

The *inputs* keyword allows selection of whether all the inputs are per-atom or local quantities. As noted above, all the inputs must be the same kind (per-atom or local). Per-atom is the default setting. If a compute or fix is specified as an input, it must produce per-atom or local data to match this setting. If it produces both, like for example the *compute voronoi/atom* command, then this keyword selects between them. If a compute *only* produces local data, like for example the *compute bond/local command*, the setting "inputs local" is *required*.

---

If a single input is specified this compute produces a global scalar value. If multiple inputs are specified, this compute produces a global vector of values, the length of which is equal to the number of inputs specified.

As discussed below, for the *sum*, *sumabs*, and *sumsq* modes, the value(s) produced by this compute are all "extensive", meaning their value scales linearly with the number of atoms involved. If normalized values are desired, this compute can be accessed by the *thermo_style custom* command with *thermo_modify norm yes* set as an option. Or it can be accessed by a *variable* that divides by the appropriate atom count.

---

### 3.98.4 Output info

This compute calculates a global scalar if a single input value is specified or a global vector of length *N*, where *N* is the number of inputs, and which can be accessed by indices 1 to *N*. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

Added in version 10Dec2025.

This compute supports automatically generated thermo column names when using *thermo_modify colname auto*. The thermo column names are "c_", followed by the compute ID, followed by a colon, followed by the reduction operation (*mode*), followed by the compute being operated on in parentheses. E.g., for the first in-text example above, the first printed thermo column name would be "c_2:min(c_myPress[1])", rather than the default "c_2[1]". If the *replace* keyword is used, *vec1* of the *replace* keyword is listed after the colon, followed by '<-', followed by the reduction operation, followed by *vec2* of the *replace* keyword in parentheses. E.g., for the second in-text example above, the first printed thermo column name would be "c_3:c_batoms[1]<-max(c_blength)" rather than the default "c_3[1]".

All the scalar or vector values calculated by this compute are "intensive", except when the *sum*, *sumabs*, or *sumsq* modes are used on per-atom or local vectors, in which case the calculated values are "extensive".

The scalar or vector values will be in whatever *units* the quantities being reduced are in.

### 3.98.5 Restrictions

As noted above, the compute reduce/region command can only be used with per-atom inputs.

### 3.98.6 Related commands

*compute*, *fix*, *variable*

### 3.98.7 Default

The default value for the *inputs* keyword is peratom.

## 3.99 compute reduce/chunk command

### 3.99.1 Syntax

```
compute ID group-ID reduce/chunk chunkID mode input1 input2 ...
```

- ID, group-ID are documented in *compute* command
- reduce/chunk = style name of this compute command
- chunkID = ID of *compute chunk/atom* command
- mode = *sum* or *min* or *max*
- one or more inputs can be listed
- input = c_ID, c_ID[N], f_ID, f_ID[N], v_ID

```
c_ID = per-atom vector calculated by a compute with ID
c_ID[I] = Ith column of per-atom array calculated by a compute with ID, I can␣
↪include wildcard (see below)
f_ID = per-atom vector calculated by a fix with ID
f_ID[I] = Ith column of per-atom array calculated by a fix with ID, I can include␣
```

(continues on next page)

(continued from previous page)

```
↪wildcard (see below)
v_name = per-atom vector calculated by an atom-style variable with name
```

### 3.99.2 Examples

```
compute 1 all reduce/chunk mychunk min c_cluster
```

### 3.99.3 Description

Define a calculation that reduces one or more per-atom vectors into per-chunk values. This can be useful for diagnostic output. Or when used in conjunction with the *compute chunk/spread/atom* command it can be used to create per-atom values that induce a new set of chunks with a second *compute chunk/atom* command. An example is given below.

In LAMMPS, chunks are collections of atoms defined by a *compute chunk/atom* command, which assigns each atom to a single chunk (or no chunk). The ID for this command is specified as chunkID. For example, a single chunk could be the atoms in a molecule or atoms in a spatial bin. See the *compute chunk/atom* and *Howto chunk* doc pages for details of how chunks can be defined and examples of how they can be used to measure properties of a system.

For each atom, this compute accesses its chunk ID from the specified *chunkID* compute. The per-atom value from an input contributes to a per-chunk value corresponding the chunk ID.

The reduction operation is specified by the *mode* setting and is performed over all the per-atom values from the atoms in each chunk. The *sum* option adds the per-atom values to a per-chunk total. The *min* or *max* options find the minimum or maximum value of the per-atom values for each chunk.

Note that only atoms in the specified group contribute to the reduction operation. If the *chunkID* compute returns a 0 for the chunk ID of an atom (i.e., the atom is not in a chunk defined by the *compute chunk/atom* command), that atom will also not contribute to the reduction operation. An input that is a compute or fix may define its own group which affects the quantities it returns. For example, a compute will return a zero value for atoms that are not in the group specified for that compute.

Each listed input is operated on independently. Each input can be the result of a *compute* or *fix* or the evaluation of an atom-style *variable*.

Note that for values from a compute or fix, the bracketed index I can be specified using a wildcard asterisk with the index to effectively specify multiple values. This takes the form "*" or "*n" or "m*" or "m*n". If *N* is the size of the vector (for *mode* = scalar) or the number of columns in the array (for *mode* = vector), then an asterisk with no numeric values means all indices from 1 to *N*. A leading asterisk means all indices from 1 to n (inclusive). A trailing asterisk means all indices from n to *N* (inclusive). A middle asterisk means all indices from m to n (inclusive).

Using a wildcard is the same as if the individual columns of the array had been listed one by one. For example, the following two compute reduce/chunk commands are equivalent, since the *compute property/chunk* command creates a per-atom array with 3 columns:

```
compute prop all property/atom vx vy vz
compute 10 all reduce/chunk mychunk max c_prop[*]
compute 10 all reduce/chunk mychunk max c_prop[1] c_prop[2] c_prop[3]
```

---

Here is an example of using this compute, in conjunction with the compute chunk/spread/atom command to identify self-assembled micelles. The commands below can be added to the examples/in.micelle script.

Imagine a collection of polymer chains or small molecules with hydrophobic end groups. All the hydrophobic (HP) atoms are assigned to a group called "phobic".

These commands will assign a unique cluster ID to all HP atoms within a specified distance of each other. A cluster will contain all HP atoms in a single molecule, but also the HP atoms in nearby molecules (e.g., molecules that have clumped to form a micelle due to the attraction induced by the hydrophobicity). The output of the chunk/reduce command will be a cluster ID per chunk (molecule). Molecules with the same cluster ID are in the same micelle.

```
group phobic type 4     # specific to in.micelle model
compute cluster phobic cluster/atom 2.0
compute cmol all chunk/atom molecule
compute reduce phobic reduce/chunk cmol min c_cluster
```

This per-chunk info could be output in at least two ways:

```
fix 10 all ave/time 1000 1 1000 c_reduce file tmp.phobic mode vector

compute spread all chunk/spread/atom cmol c_reduce
dump 1 all custom 1000 tmp.dump id type mol x y z c_cluster c_spread
dump_modify 1 sort id
```

In the first case, each snapshot in the tmp.phobic file will contain one line per molecule. Molecules with the same value are in the same micelle. In the second case each dump snapshot contains all atoms, each with a final field with the cluster ID of the micelle that the HP atoms of that atom's molecule belong to.

The result from compute chunk/spread/atom can be used to define a new set of chunks, where all the atoms in all the molecules in the same micelle are assigned to the same chunk (i.e., one chunk per micelle).

```
compute micelle all chunk/atom c_spread compress yes
```

Further analysis on a per-micelle basis can now be performed using any of the per-chunk computes listed on the *Howto chunk* doc page (e.g., count the number of atoms in each micelle, calculate its center or mass, shape/moments of inertia, and radius of gyration).

```
compute prop all property/chunk micelle count
fix 20 all ave/time 1000 1 1000 c_prop file tmp.micelle mode vector
```

Each snapshot in the tmp.micelle file will have one line per micelle with its count of atoms, plus a first line for a chunk with all the solvent atoms. By the time 50000 steps have elapsed, there are a handful of large micelles.

---

### 3.99.4 Output info

This compute calculates a global vector if a single input value is specified, otherwise a global array is output. The number of columns in the array is the number of inputs provided. The length of the vector or the number of vector elements or array rows = the number of chunks *Nchunk* as calculated by the specified *compute chunk/atom* command. The vector or array can be accessed by any command that uses global values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-atom values for the vector or each column of the array will be in whatever *units* the corresponding input value is in. The vector or array values are "intensive".

### 3.99.5 Restrictions

none

### 3.99.6 Related commands

*compute chunk/atom*, *compute reduce*, *compute chunk/spread/atom*

### 3.99.7 Default

none

## 3.100 compute rheo/property/atom command

### 3.100.1 Syntax

```
compute ID group-ID rheo/property/atom input1 input2 ...
```

- ID, group-ID are documented in *compute* command
- rheo/property/atom = style name of this compute command
- input = one or more atom attributes

```
possible attributes = phase, surface, surface/r,
                      surface/divr, surface/n/a, coordination,
                      shift/v/a, energy, temperature, heatflow,
                      conductivity, cv, viscosity, pressure, rho,
                      grad/v/ab, stress/v/ab, stress/t/ab, nbond/shell
```

*phase* = atom phase state
*surface* = atom surface status
*surface/r* = atom distance from the surface
*surface/divr* = divergence of position at atom position
*surface/n/a* = a-component of surface normal vector
*coordination* = coordination number
*shift/v/a* = a-component of atom shifting velocity
*energy* = atom energy
*temperature* = atom temperature
*heatflow* = atom heat flow
*conductivity* = atom conductivity
*cv* = atom specific heat
*viscosity* = atom viscosity
*pressure* = atom pressure
*rho* = atom density
*grad/v/ab* = ab-component of atom velocity gradient tensor
*stress/v/ab* = ab-component of atom viscous stress tensor
*stress/t/ab* = ab-component of atom total stress tensor (pressure and viscous)
*nbond/shell* = number of oxide bonds

### 3.100.2 Examples

```
compute 1 all rheo/property/atom phase surface/r surface/n/* pressure
compute 2 all rheo/property/atom shift/v/x grad/v/xx stress/v/*
```

### 3.100.3 Description

Added in version 29Aug2024.

Define a computation that stores atom attributes specific to the RHEO package for each atom in the group. This is useful so that the values can be used by other *output commands* that take computes as inputs. See for example, the *compute reduce*, *fix ave/atom*, *fix ave/histo*, *fix ave/chunk*, and *atom-style variable* commands.

For vector attributes, e.g. *shift/v/*$\alpha$, one must specify $\alpha$ as the *x*, *y*, or *z* component, e.g. *shift/v/x*. Alternatively, a wild card * will include all components, *x* and *y* in 2D or *x*, *y*, and *z* in 3D.

For tensor attributes, e.g. *grad/v/*$\alpha\beta$, one must specify both $\alpha$ and $\beta$ as *x*, *y*, or *z*, e.g. *grad/v/xy*. Alternatively, a wild card * will include all components. In 2D, this includes *xx*, *xy*, *yx*, and *yy*. In 3D, this includes *xx*, *xy*, *xz*, *yx*, *yy*, *yz*, *zx*, *zy*, and *zz*.

Many properties require their respective fixes, listed below in related commands, be defined. For instance, the *viscosity* attribute is the viscosity of a particle calculated by *fix rheo/viscosity*. The meaning of less obvious properties is described below.

The *phase* property indicates whether the particle is in a fluid state, a value of 0, or a solid state, a value of 1.

The *surface* property indicates the surface designation produced by the *surface/detection* option of *fix rheo*. Bulk particles have a value of 0, surface particles have a value of 1, and splash particles have a value of 2. The *surface/r* property is the distance from the surface, up to the kernel cutoff length. Surface particles have a value of 0. The *surface/n/*$\alpha$ properties are the components of the surface normal vector.

The *shift/v/*$\alpha$ properties are the components of the shifting velocity produced by the *shift* option of *fix rheo*.

The *nbond/shell* property is the number of shell bonds that have been activated from *bond style rheo/shell*.

The values are stored in a per-atom vector or array as discussed below. Zeroes are stored for atoms not in the specified group or for quantities that are not defined for a particular particle in the group

### 3.100.4 Output info

This compute calculates a per-atom vector or per-atom array depending on the number of input values. Generally, if a single input is specified, a per-atom vector is produced. If two or more inputs are specified, a per-atom array is produced where the number of columns = the number of inputs. However, if a wild card * is used for a vector or tensor, then the number of inputs is considered to be incremented by the dimension or the dimension squared, respectively. The vector or array can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The vector or array values will be in whatever *units* the corresponding attribute is in (e.g., density units for *rho*).

### 3.100.5 Restrictions

This compute style is part of the RHEO package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.100.6 Related commands

*dump custom*, *compute reduce*, *fix ave/atom*, *fix ave/chunk*, *fix rheo/viscosity*, *fix rheo/pressure*, *fix rheo/thermal*, *fix rheo/oxdiation*, *fix rheo*

### 3.100.7 Default

none

# 3.101 compute rigid/local command

## 3.101.1 Syntax

```
compute ID group-ID rigid/local rigidID input1 input2 ...
```

- ID, group-ID are documented in *compute* command
- rigid/local = style name of this compute command
- rigidID = ID of fix rigid/small command or one of its variants
- input = one or more rigid body attributes

```
possible attributes = id, mol, mass,
                      x, y, z, xu, yu, zu, ix, iy, iz
                      vx, vy, vz, fx, fy, fz,
                      omegax, omegay, omegaz,
                      angmomx, angmomy, angmomz,
                      quatw, quati, quatj, quatk,
                      tqx, tqy, tqz,
                      inertiax, inertiay, inertiaz
```

```
id = atom ID of atom within body which owns body properties
mol = molecule ID used to define body in fix rigid/small command
mass = total mass of body
x,y,z = center of mass coords of body
xu,yu,zu = unwrapped center of mass coords of body
ix,iy,iz = box image that the center of mass is in
vx,vy,vz = center of mass velocities
fx,fy,fz = force of center of mass
omegax,omegay,omegaz = angular velocity of body
angmomx,angmomy,angmomz = angular momentum of body
quatw,quati,quatj,quatk = quaternion components for body
tqx,tqy,tqz = torque on body
inertiax,inertiay,inertiaz = diagonalized moments of inertia of body
```

## 3.101.2 Examples

```
compute 1 all rigid/local myRigid mol x y z
```

## 3.101.3 Description

Define a computation that simply stores rigid body attributes for rigid bodies defined by the *fix rigid/small* command or one of its NVE, NVT, NPT, NPH variants. The data is stored as local data so it can be accessed by other *output commands* that process local data, such as the *compute reduce* or *dump local* commands.

Note that this command only works with the *fix rigid/small* command or its variants, not the fix rigid command and its variants. The ID of the *fix rigid/small* command used to define rigid bodies must be specified as *rigidID*. The *fix rigid* command is typically used to define a handful of (potentially very large) rigid bodies. It outputs similar per-body information as this command directly from the fix as global data; see the *fix rigid* page for details

The local data stored by this command is generated by looping over all the atoms owned on a processor. If the atom is not in the specified *group-ID* or is not part of a rigid body it is skipped. If it is not the atom within a body that is assigned to store the body information it is skipped (only one atom per body is so assigned). If it is the assigned atom,

then the info for that body is output. This means that information for *N* bodies is generated. *N* may be less than the number of bodies defined by the fix rigid command, if the atoms in some bodies are not in the *group-ID*.

**Note**

Which atom in a body owns the body info is determined internal to LAMMPS; it's the one nearest the geometric center of the body. Typically you should avoid this complication, by defining the group associated with this fix to include/exclude entire bodies.

Note that as atoms and bodies migrate from processor to processor, there will be no consistent ordering of the entries within the local vector or array from one timestep to the next.

Here is an example of how to use this compute to dump rigid body info to a file:

```
compute 1 all rigid/local myRigid mol x y z fx fy fz
dump 1 all local 1000 tmp.dump index c_1[1] c_1[2] c_1[3] c_1[4] c_1[5] c_1[6] c_1[7]
```

---

This section explains the rigid body attributes that can be specified.

The *id* attribute is the atom-ID of the atom which owns the rigid body, which is assigned by the *fix rigid/small* command.

The *mol* attribute is the molecule ID of the rigid body. It should be the molecule ID which all of the atoms in the body belong to, since that is how the *fix rigid/small* command defines its rigid bodies.

The *mass* attribute is the total mass of the rigid body.

There are two options for outputting the coordinates of the center of mass (COM) of the body. The *x*, *y*, *z* attributes write the COM "unscaled", in the appropriate distance *units* (Å, $\sigma$, etc). Use *xu*, *yu*, *zu* if you want the COM "unwrapped" by the image flags for each body. Unwrapped means that if the body COM has passed through a periodic boundary one or more times, the value is generated what the COM coordinate would be if it had not been wrapped back into the periodic box.

The image flags for the body can be generated directly using the *ix*, *iy*, *iz* attributes. For periodic dimensions, they specify which image of the simulation box the COM is considered to be in. An image of 0 means it is inside the box as defined. A value of 2 means add 2 box lengths to get the true value. A value of $-1$ means subtract 1 box length to get the true value. LAMMPS updates these flags as the rigid body COMs cross periodic boundaries during the simulation.

The *vx*, *vy*, *vz*, *fx*, *fy*, *fz* attributes are components of the COM velocity and force on the COM of the body.

The *omegax*, *omegay*, and *omegaz* attributes are the angular velocity components of the body in the system frame around its COM.

The *angmomx*, *angmomy*, and *angmomz* attributes are the angular momentum components of the body in the system frame around its COM.

The *quatw*, *quati*, *quatj*, and *quatk* attributes are the components of the 4-vector quaternion representing the orientation of the rigid body. See the *set* command for an explanation of the quaternion vector.

The *tqx*, *tqy*, *tqz* attributes are components of the torque acting on the body around its COM.

The *inertiax*, *inertiay*, *inertiaz* attributes are components of diagonalized inertia tensor for the body (i.e., the three moments of inertia for the body around its principal axes), as computed internally by LAMMPS.

---

### 3.101.4 Output info

This compute calculates a local vector or local array depending on the number of keywords. The length of the vector or number of rows in the array is the number of rigid bodies. If a single keyword is specified, a local vector is produced. If two or more keywords are specified, a local array is produced where the number of columns = the number of keywords. The vector or array can be accessed by any command that uses local values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The vector or array values will be in whatever *units* the corresponding attribute is in:

- id,mol = unitless
- mass = mass units
- x,y,z and xy,yu,zu = distance units
- vx,vy,vz = velocity units
- fx,fy,fz = force units
- omegax,omegay,omegaz = radians/time units
- angmomx,angmomy,angmomz = mass*distance$^2$/time units
- quatw,quati,quatj,quatk = unitless
- tqx,tqy,tqz = torque units
- inertiax,inertiay,inertiaz = mass*distance$^2$ units

### 3.101.5 Restrictions

This compute is part of the RIGID package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.101.6 Related commands

*dump local*, *compute reduce*

### 3.101.7 Default

none

## 3.102 compute saed command

### 3.102.1 Syntax

```
compute ID group-ID saed lambda type1 type2 ... typeN keyword value ...
```

- ID, group-ID are documented in *compute* command
- saed = style name of this compute command
- lambda = wavelength of incident radiation (length units)
- type1 type2 ... typeN = chemical symbol of each atom type (see valid options below)
- zero or more keyword/value pairs may be appended
- keyword = *Kmax* or *Zone* or *dR_Ewald* or *c* or *manual* or *echo*

```
Kmax value = Maximum distance explored from reciprocal space origin
               (inverse length units)
Zone values = z1 z2 z3
  z1,z2,z3 = Zone axis of incident radiation. If z1=z2=z3=0 all
             reciprocal space will be meshed up to Kmax
dR_Ewald value = Thickness of Ewald sphere slice intercepting
                 reciprocal space (inverse length units)
c values = c1 c2 c3
  c1,c2,c3 = parameters to adjust the spacing of the reciprocal
             lattice nodes in the h, k, and l directions respectively
manual = flag to use manual spacing of reciprocal lattice points
          based on the values of the c parameters
echo = flag to provide extra output for debugging purposes
```

### 3.102.2 Examples

```
compute 1 all saed 0.0251 Al O Kmax 1.70 Zone 0 0 1 dR_Ewald 0.01 c 0.5 0.5 0.5
compute 2 all saed 0.0251 Ni Kmax 1.70 Zone 0 0 0 c 0.05 0.05 0.05 manual echo

fix 1 all saed/vtk 1 1 1 c_1 file Al2O3_001.saed
fix 2 all saed/vtk 1 1 1 c_2 file Ni_000.saed
```

### 3.102.3 Description

Define a computation that calculates electron diffraction intensity as described in *(Coleman)* on a mesh of reciprocal lattice nodes defined by the entire simulation domain (or manually) using simulated radiation of wavelength lambda.

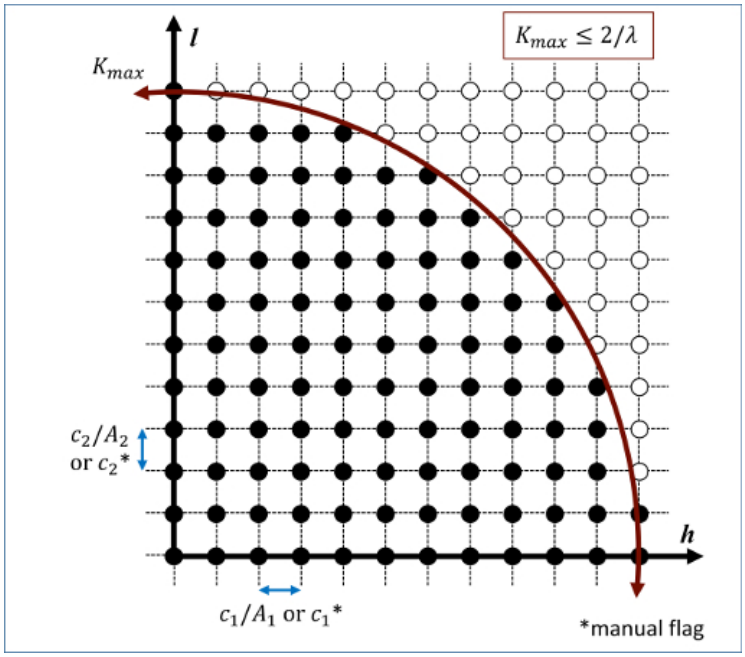
The electron diffraction intensity I at each reciprocal lattice point is computed from the structure factor F using the equations:

$$I = \frac{F^* F}{N}$$

$$F(\mathbf{k}) = \sum_{j=1}^{N} f_j(\theta) exp(2\pi i \mathbf{k} \cdot \mathbf{r}_j)$$

Here, K is the location of the reciprocal lattice node, $r_j$ is the position of each atom, $f_j$ are atomic scattering factors.

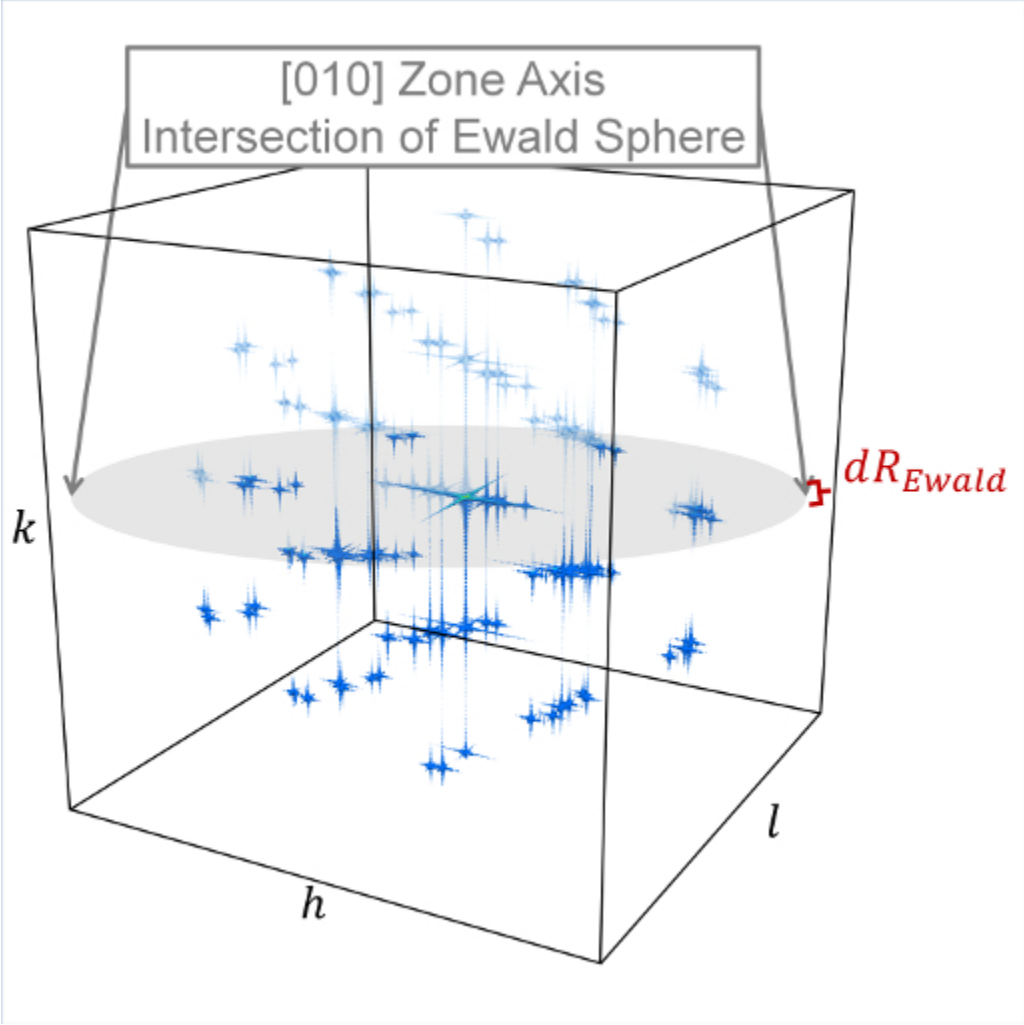
Diffraction intensities are calculated on a three-dimensional mesh of reciprocal lattice nodes. The mesh spacing is defined either (a) by the entire simulation domain or (b) manually using selected values as shown in the 2D diagram below.




For a mesh defined by the simulation domain, a rectilinear grid is constructed with spacing *c**inv(A) along each reciprocal lattice axis. Where A are the vectors corresponding to the edges of the simulation cell. If one or two directions has non-periodic boundary conditions, then the spacing in these directions is defined from the average of the (inversed) box lengths with periodic boundary conditions. Meshes defined by the simulation domain must contain at least one periodic boundary.

If the *manual* flag is included, the mesh of reciprocal lattice nodes will defined using the *c* values for the spacing along each reciprocal lattice axis. Note that manual mapping of the reciprocal space mesh is good for comparing diffraction results from multiple simulations; however it can reduce the likelihood that Bragg reflections will be satisfied unless small spacing parameters ($< 0.05$ Å$^{-}1$) are implemented. Meshes with manual spacing do not require a periodic boundary.

The limits of the reciprocal lattice mesh are determined by the use of the *Kmax*, *Zone*, and *dR_Ewald* parameters. The rectilinear mesh created about the origin of reciprocal space is terminated at the boundary of a sphere of radius *Kmax* centered at the origin. If *Zone* parameters *z1* = *z2* = *z3* = 0 are used, diffraction intensities are computed throughout the entire spherical volume - note this can greatly increase the cost of computation. Otherwise, *Zone* parameters will denote the $z1 = h$, $z2 = k$, and $z3 = \ell$ (in a global sense) zone axis of an intersecting Ewald sphere. Diffraction intensities will only be computed at the intersection of the reciprocal lattice mesh and a *dR_Ewald* thick surface of the Ewald sphere. See the example 3D intensity data and the intersection of a [010] zone axis in the below image.




The atomic scattering factors, fj, accounts for the reduction in diffraction intensity due to Compton scattering. Compute saed uses analytical approximations of the atomic scattering factors that vary for each atom type (type1 type2 ... typeN) and angle of diffraction. The analytic approximation is computed using the formula *(Brown)*:

$$f_j\left(\frac{sin(\theta)}{\lambda}\right) = \sum_i^5 a_i exp\left(-b_i \frac{sin^2(\theta)}{\lambda^2}\right)$$

Coefficients parameterized by *(Fox)* are assigned for each atom type designating the chemical symbol and charge of each atom type. Valid chemical symbols for compute saed are:

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | He | Li | Be | B | C | N | O | F | Ne | Na | Mg | Al | Si | P | S | Cl | Ar | K | Ca |
| Sc | Ti | V | Cr | Mn | Fe | Co | Ni | Cu | Zn | Ga | Ge | As | Se | Br | Kr | Rb | Sr | Y | Zr |
| Nb | Mo | Tc | Ru | Rh | Pd | Ag | Cd | In | Sn | Sb | Te | I | Xe | Cs | Ba | La | Ce | Pr | Nd |
| Pm | Sm | Eu | Gd | Tb | Dy | Ho | Er | Tm | Yb | Lu | Hf | Ta | W | Re | Os | Ir | Pt | Au | Hg |
| Tl | Pb | Bi | Po | At | Rn | Fr | Ra | Ac | Th | Pa | U | Np | Pu | Am | Cm | Bk | Cf | | |

If the *echo* keyword is specified, compute saed will provide extra reporting information to the screen.

### 3.102.4 Output info

This compute calculates a global vector. The length of the vector is the number of reciprocal lattice nodes that are explored by the mesh. The entries of the global vector are the computed diffraction intensities as described above.

The vector can be accessed by any command that uses global values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

All array values calculated by this compute are "intensive".

### 3.102.5 Restrictions

This compute is part of the DIFFRACTION package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The compute_saed command does not work for triclinic cells.

### 3.102.6 Related commands

*fix saed_vtk*, *compute xrd*

### 3.102.7 Default

The option defaults are Kmax = 1.70, Zone 1 0 0, c 1 1 1, dR_Ewald = 0.01.

---

**(Coleman)** Coleman, Spearot, Capolungo, MSMSE, 21, 055020 (2013).

**(Brown)** Brown et al. International Tables for Crystallography Volume C: Mathematical and Chemical Tables, 554-95 (2004).

**(Fox)** Fox, O'Keefe, Tabbernor, Acta Crystallogr. A, 45, 786-93 (1989).

## 3.103 compute slcsa/atom command

### 3.103.1 Syntax

```
compute ID group-ID slcsa/atom twojmax nclasses db_mean_descriptor_file lda_file lr_
↪decision_file lr_bias_file maha_file value
```

- ID, group-ID are documented in *compute* command
- slcsa/atom = style name of this compute command
- twojmax = band limit for bispectrum components (non-negative integer)
- nclasses = number of crystal structures used in the database for the classifier SL-CSA
- db_mean_descriptor_file = file name of file containing the database mean descriptor
- lda_file = file name of file containing the linear discriminant analysis matrix for dimension reduction
- lr_decision_file = file name of file containing the scaling matrix for logistic regression classification
- lr_bias_file = file name of file containing the bias vector for logistic regression classification
- maha_file = file name of file containing for each crystal structure: the Mahalanobis distance threshold for sanity check purposes, the average reduced descriptor and the inverse of the corresponding covariance matrix
- c_ID[1] = compute ID and output data column of previously defined *compute sna/atom* command

### 3.103.2 Examples

```
compute b1 all sna/atom 9.0 0.99363 8 0.5 1.0 rmin0 0.0 nnn 24 wmode 1 delta 0.3
compute b2 all slcsa/atom 8 4 mean_descriptors.dat lda_scalings.dat lr_decision.dat lr_
↪bias.dat maha_thresholds.dat c_b1[1]
```

### 3.103.3 Description

Added in version 7Feb2024.

Define a computation that performs the Supervised Learning Crystal Structure Analysis (SL-CSA) from *(Lafourcade)* for each atom in the group. The SL-CSA tool takes as an input a per-atom descriptor (bispectrum) that is computed through the *compute sna/atom* command and then proceeds to a dimension reduction step followed by a logistic regression in order to assign a probable crystal structure to each atom in the group. The SL-CSA tool is pre-trained on a database containing $C$ distinct crystal structures from which a crystal structure classifier is derived and a tutorial to build such a tool is available at SL-CSA.

The first step of the SL-CSA tool consists in performing a dimension reduction of the per-atom descriptor $\mathbf{B}^i \in \mathbb{R}^D$ through the Linear Discriminant Analysis (LDA) method, leading to a new projected descriptor $\mathbf{x}^i = \mathrm{P}_{\mathrm{LDA}}(\mathbf{B}^i) : \mathbb{R}^D \rightarrow \mathbb{R}^{d=C-1}$:

$$\mathbf{x}^i = \mathbf{C}_{\mathrm{LDA}}^T \cdot (\mathbf{B}^i - \mu_{\mathrm{db}}^{\mathbf{B}})$$

where $\mathbf{C}_{\mathrm{LDA}}^T \in \mathbb{R}^{D\times d}$ is the reduction coefficients matrix of the LDA model read in file *lda_file*, $\mathbf{B}^i \in \mathbb{R}^D$ is the bispectrum of atom $i$ and $\mu_{\mathrm{db}}^{\mathbf{B}} \in \mathbb{R}^D$ is the average descriptor of the entire database. The latter is computed from the average descriptors of each crystal structure read from the file *mean_descriptors_file*.

The new projected descriptor with dimension $d = C - 1$ allows for a good separation of different crystal structures fingerprints in the latent space.

Once the dimension reduction step is performed by means of LDA, the new descriptor $\mathbf{x}^i \in \mathbb{R}^{d=C-1}$ is taken as an input for performing a multinomial logistic regression (LR) which provides a score vector $\mathbf{s}^i = \mathrm{P}_{\mathrm{LR}}(\mathbf{x}^i) : \mathbb{R}^d \rightarrow \mathbb{R}^C$ defined as:

$$\mathbf{s}^i = \mathbf{b}_{\mathrm{LR}} + \mathbf{D}_{\mathrm{LR}} \cdot \mathbf{x}^{iT}$$

with $\mathbf{b}_{\mathrm{LR}} \in \mathbb{R}^C$ and $\mathbf{D}_{\mathrm{LR}} \in \mathbb{R}^{C\times d}$ the bias vector and decision matrix of the LR model after training both read in files *lr_fil1* and *lr_file2* respectively.

Finally, a probability vector $\mathbf{p}^i = \mathrm{P}_{\mathrm{LR}}(\mathbf{x}^i) : \mathbb{R}^d \rightarrow \mathbb{R}^C$ is defined as:

$$\mathbf{p}^i = \frac{\exp(\mathbf{s}^i)}{\sum_j \exp(s_j^i)}$$

from which the crystal structure assigned to each atom with descriptor $\mathbf{B}^i$ and projected descriptor $\mathbf{x}^i$ is computed as the *argmax* of the probability vector $\mathbf{p}^i$. Since the logistic regression step systematically attributes a crystal structure to each atom, a sanity check is needed to avoid misclassification. To this end, a per-atom Mahalanobis distance to each crystal structure *CS* present in the database is computed:

$$d_{\mathrm{Mahalanobis}}^{i\rightarrow\mathrm{CS}} = \sqrt{(\mathbf{x}^i - \mu_{\mathrm{CS}}^{\mathbf{x}})^{\mathrm{T}} \cdot \blacksquare_{\mathrm{CS}}^{-1} \cdot (\mathbf{x}^i - \mu_{\mathrm{CS}}^{\mathbf{x}})}$$

where $\mu_{\mathrm{CS}}^{\mathbf{x}} \in \mathbb{R}^d$ is the average projected descriptor of crystal structure *CS* in the database and where $\blacksquare_{\mathrm{CS}} \in \mathbb{R}^{d\times d}$ is the corresponding covariance matrix. Finally, if the Mahalanobis distance to crystal structure *CS* for atom $i$ is greater than the pre-determined threshold, no crystal structure is assigned to atom $i$. The Mahalanobis distance thresholds are read in file *maha_file* while the covariance matrices are read in file *covmat_file*.

The SL-CSA framework provides an automatic computation of the different matrices and thresholds required for a proper classification and writes down all the required files for calling the *compute slcsa/atom* command.

The *compute slcsa/atom* command requires that the *compute sna/atom* command is called before as it takes the resulting per-atom bispectrum as an input. In addition, it is crucial that the value *twojmax* is set to the same value of the value *twojmax* used in the *compute sna/atom* command, as well as that the value *nclasses* is set to the number of crystal structures used in the database to train the SL-CSA tool.

### 3.103.4 Output info

By default, this compute computes the Mahalanobis distances to the different crystal structures present in the database in addition to assigning a crystal structure for each atom as a per-atom vector, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

### 3.103.5 Restrictions

This compute is part of the EXTRA-COMPUTE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.103.6 Related commands

*compute sna/atom*

### 3.103.7 Default

none

---

**(Lafourcade)** Lafourcade, Maillet, Denoual, Duval, Allera, Goryaeva, and Marinica, Comp. Mat. Science, 230, 112534 (2023)

## 3.104 compute slice command

### 3.104.1 Syntax

```
compute ID group-ID slice Nstart Nstop Nskip input1 input2 ...
```

- ID, group-ID are documented in *compute* command
- slice = style name of this compute command
- Nstart = starting index within input vector(s)
- Nstop = stopping index within input vector(s)
- Nskip = extract every Nskip elements from input vector(s)
- input = c_ID, c_ID[N], f_ID, f_ID[N]

```
c_ID = global vector calculated by a compute with ID
c_ID[I] = Ith column of global array calculated by a compute with ID
f_ID = global vector calculated by a fix with ID
f_ID[I] = Ith column of global array calculated by a fix with ID
v_name = vector calculated by an vector-style variable with name
```

### 3.104.2 Examples

```
compute 1 all slice 1 100 10 c_msdmol[4]
compute 1 all slice 301 400 1 c_msdmol[4] v_myVec
```

### 3.104.3 Description

Define a calculation that "slices" one or more vector inputs into smaller vectors, one per listed input. The inputs can be global quantities; they cannot be per-atom or local quantities. *Computes* and *fixes* and vector-style *variables* can generate such global quantities. The group specified with this command is ignored.

The values extracted from the input vector(s) are determined by the *Nstart*, *Nstop*, and *Nskip* parameters. The elements of an input vector of length N are indexed from 1 to N. Starting at element *Nstart*, every Mth element is extracted, where M = *Nskip*, until element *Nstop* is reached. The extracted quantities are stored as a vector, which is typically shorter than the input vector.

Each listed input is operated on independently to produce one output vector. Each listed input must be a global vector or column of a global array calculated by another *compute* or *fix*.

If an input value begins with "c_", a compute ID must follow which has been previously defined in the input script and which generates a global vector or array. See the individual *compute* doc page for details. If no bracketed integer is appended, the vector calculated by the compute is used. If a bracketed integer is appended, the Ith column of the array calculated by the compute is used. Users can also write code for their own compute styles and *add them to LAMMPS*.

If a value begins with "f_", a fix ID must follow which has been previously defined in the input script and which generates a global vector or array. See the individual *fix* page for details. Note that some fixes only produce their values on certain timesteps, which must be compatible with when compute slice references the values, else an error results. If no bracketed integer is appended, the vector calculated by the fix is used. If a bracketed integer is appended, the Ith column of the array calculated by the fix is used. Users can also write code for their own fix style and *add them to LAMMPS*.

If an input value begins with "v_", a variable name must follow which has been previously defined in the input script. Only vector-style variables can be referenced. See the *variable* command for details. Note that variables of style *vector* define a formula which can reference individual atom properties or thermodynamic keywords, or they can invoke other computes, fixes, or variables when they are evaluated, so this is a very general means of specifying quantities to slice.

If a single input is specified this compute produces a global vector, even if the length of the vector is 1. If multiple inputs are specified, then a global array of values is produced, with the number of columns equal to the number of inputs specified.

---

### 3.104.4 Output info

This compute calculates a global vector if a single input value is specified or a global array with N columns where N is the number of inputs. The length of the vector or the number of rows in the array is equal to the number of values extracted from each input vector. These values can be used by any command that uses global vector or array values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The vector or array values calculated by this compute are simply copies of values generated by computes or fixes or variables that are input vectors to this compute. If there is a single input vector of intensive and/or extensive values, then each value in the vector of values calculated by this compute will be "intensive" or "extensive", depending on the corresponding input value. If there are multiple input vectors, and all the values in them are intensive, then the array values calculated by this compute are "intensive". If there are multiple input vectors, and any value in them is extensive, then the array values calculated by this compute are "extensive". Values produced by a variable are treated as intensive.

The vector or array values will be in whatever *units* the input quantities are in.

### 3.104.5 Restrictions

none

### 3.104.6 Related commands

*compute*, *fix*, *compute reduce*

### 3.104.7 Default

none

## 3.105 compute smd/contact/radius command

### 3.105.1 Syntax

```
compute ID group-ID smd/contact/radius
```

- ID, group-ID are documented in *compute* command
- smd/contact/radius = style name of this compute command

### 3.105.2 Examples

```
compute 1 all smd/contact/radius
```

### 3.105.3 Description

Define a computation which outputs the contact radius, i.e., the radius used to prevent particles from penetrating each other. The contact radius is used only to prevent particles belonging to different physical bodies from penetrating each other. It is used by the contact pair styles, e.g., smd/hertz and smd/tri_surface.

See this PDF guide to using Smooth Mach Dynamics in LAMMPS.

The value of the contact radius will be 0.0 for particles not in the specified compute group.

### 3.105.4 Output info

This compute calculates a per-particle vector, which can be accessed by any command that uses per-particle values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-particle vector values will be in distance *units*.

### 3.105.5 Restrictions

This compute is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.105.6 Related commands

*dump custom* smd/hertz smd/tri_surface

### 3.105.7 Default

none

## 3.106 compute smd/damage command

### 3.106.1 Syntax

```
compute ID group-ID smd/damage
```

- ID, group-ID are documented in *compute* command
- smd/damage = style name of this compute command

### 3.106.2 Examples

```
compute 1 all smd/damage
```

### 3.106.3 Description

Define a computation that calculates the damage status of SPH particles according to the damage model which is defined via the SMD SPH pair styles, e.g., the maximum plastic strain failure criterion.

See this PDF guide to use Smooth Mach Dynamics in LAMMPS.

**Output Info:**

This compute calculates a per-particle vector, which can be accessed by any command that uses per-particle values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-particle values are dimensionless an in the range of zero to one.

### 3.106.4 Restrictions

This compute is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the "Build

### 3.106.5 Related commands

*smd/plastic_strain*, *smd/tlsph_stress*

### 3.106.6 Default

none

## 3.107 compute smd/hourglass/error command

### 3.107.1 Syntax

```
compute ID group-ID smd/hourglass/error
```

- ID, group-ID are documented in *compute* command
- smd/hourglass/error = style name of this compute command

### 3.107.2 Examples

```
compute 1 all smd/hourglass/error
```

### 3.107.3 Description

Define a computation which outputs the error of the approximated relative separation with respect to the actual relative separation of the particles i and j. Ideally, if the deformation gradient is exact, and there exists a unique mapping between all particles' positions within the neighborhood of the central node and the deformation gradient, the approximated relative separation will coincide with the actual relative separation of the particles i and j in the deformed configuration. This compute is only really useful for debugging the hourglass control mechanism which is part of the Total-Lagrangian SPH pair style.

See this PDF guide to use Smooth Mach Dynamics in LAMMPS.

**Output Info:**

This compute calculates a per-particle vector, which can be accessed by any command that uses per-particle values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-particle vector values will are dimensionless. See *units*.

### 3.107.4 Restrictions

This compute is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This quantity will be computed only for particles which interact with tlsph pair style.

### 3.107.5 Related commands

*smd/tlsph_defgrad*

### 3.107.6 Default

## 3.108 compute smd/internal/energy command

### 3.108.1 Syntax

```
compute ID group-ID smd/internal/energy
```

- ID, group-ID are documented in *compute* command
- smd/smd/internal/energy = style name of this compute command

### 3.108.2 Examples

```
compute 1 all smd/internal/energy
```

### 3.108.3 Description

Define a computation which outputs the per-particle enthalpy, i.e., the sum of potential energy and heat.

See this PDF guide to use Smooth Mach Dynamics in LAMMPS.

### 3.108.4 Output Info

This compute calculates a per-particle vector, which can be accessed by any command that uses per-particle values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-particle vector values will be given in *units* of energy.

### 3.108.5 Restrictions

This compute is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info. This compute can only be used for particles which interact via the updated Lagrangian or total Lagrangian SPH pair styles.

### 3.108.6 Related commands

none

### 3.108.7 Default

## 3.109 compute smd/plastic/strain command

### 3.109.1 Syntax

```
compute ID group-ID smd/plastic/strain
```

- ID, group-ID are documented in *compute* command
- smd/plastic/strain = style name of this compute command

### 3.109.2 Examples

```
compute 1 all smd/plastic/strain
```

### 3.109.3 Description

Define a computation that outputs the equivalent plastic strain per particle. This command is only meaningful if a material model with plasticity is defined.

See this PDF guide to use Smooth Mach Dynamics in LAMMPS.

**Output Info:**

This compute calculates a per-particle vector, which can be accessed by any command that uses per-particle values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-particle values will be given dimensionless. See *units*.

### 3.109.4 Restrictions

This compute is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info. This compute can only be used for particles which interact via the updated Lagrangian or total Lagrangian SPH pair styles.

### 3.109.5 Related commands

*smd/plastic/strain/rate*, *smd/tlsph/strain/rate*, *smd/tlsph/strain*

### 3.109.6 Default

none

## 3.110 compute smd/plastic/strain/rate command

### 3.110.1 Syntax

```
compute ID group-ID smd/plastic/strain/rate
```

- ID, group-ID are documented in *compute* command
- smd/plastic/strain/rate = style name of this compute command

### 3.110.2 Examples

```
compute 1 all smd/plastic/strain/rate
```

### 3.110.3 Description

Define a computation that outputs the time rate of the equivalent plastic strain. This command is only meaningful if a material model with plasticity is defined.

See this PDF guide to use Smooth Mach Dynamics in LAMMPS.

**Output Info:**

This compute calculates a per-particle vector, which can be accessed by any command that uses per-particle values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-particle values will be given in *units* of one over time.

### 3.110.4 Restrictions

This compute is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info. This compute can only be used for particles which interact via the updated Lagrangian or total Lagrangian SPH pair styles.

### 3.110.5 Related commands

*smd/plastic/strain*, *smd/tlsph/strain/rate*, *smd/tlsph/strain*

### 3.110.6 Default

none

## 3.111 compute smd/rho command

### 3.111.1 Syntax

```
compute ID group-ID smd/rho
```

- ID, group-ID are documented in *compute* command
- smd/rho = style name of this compute command

### 3.111.2 Examples

```
compute 1 all smd/rho
```

### 3.111.3 Description

Define a computation that calculates the per-particle mass density. The mass density is the mass of a particle which is constant during the course of a simulation, divided by its volume, which can change due to mechanical deformation.

See this PDF guide to use Smooth Mach Dynamics in LAMMPS.

### 3.111.4 Output info

This compute calculates a per-particle vector, which can be accessed by any command that uses per-particle values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-particle values will be in *units* of mass over volume.

### 3.111.5 Restrictions

This compute is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.111.6 Related commands

*compute smd/vol*

### 3.111.7 Default

none

## 3.112 compute smd/tlsph/defgrad command

### 3.112.1 Syntax

```
compute ID group-ID smd/tlsph/defgrad
```

- ID, group-ID are documented in *compute* command
- smd/tlsph/defgrad = style name of this compute command

### 3.112.2 Examples

```
compute 1 all smd/tlsph/defgrad
```

### 3.112.3 Description

Define a computation that calculates the deformation gradient. It is only meaningful for particles which interact according to the Total-Lagrangian SPH pair style.

See this PDF guide to use Smooth Mach Dynamics in LAMMPS.

### 3.112.4 Output info

This compute outputs a per-particle vector of vectors (tensors), which can be accessed by any command that uses per-particle values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

The per-particle vector values will be given dimensionless. See *units*. The per-particle vector has 10 entries. The first nine entries correspond to the xx, xy, xz, yx, yy, yz, zx, zy, zz components of the asymmetric deformation gradient tensor. The tenth entry is the determinant of the deformation gradient.

### 3.112.5 Restrictions

This compute is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info. TThis compute can only be used for particles which interact via the total Lagrangian SPH pair style.

### 3.112.6 Related commands

*smd/hourglass/error*

### 3.112.7 Default

none

## 3.113 compute smd/tlsph/dt command

### 3.113.1 Syntax

```
compute ID group-ID smd/tlsph/dt
```

- ID, group-ID are documented in *compute* command
- smd/tlsph/dt = style name of this compute command

### 3.113.2 Examples

```
compute 1 all smd/tlsph/dt
```

### 3.113.3 Description

Define a computation that outputs the CFL-stable time increment per particle. This time increment is essentially given by the speed of sound, divided by the SPH smoothing length. Because both the speed of sound and the smoothing length typically change during the course of a simulation, the stable time increment needs to be re-computed every time step. This calculation is performed automatically in the relevant SPH pair styles and this compute only serves to make the stable time increment accessible for output purposes.

See this PDF guide to using Smooth Mach Dynamics in LAMMPS.

### 3.113.4 Output info

This compute calculates a per-particle vector, which can be accessed by any command that uses per-particle values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-particle values will be given in *units* of time.

### 3.113.5 Restrictions

This compute is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This compute can only be used for particles interacting with the Total-Lagrangian SPH pair style.

### 3.113.6 Related commands

*smd/adjust/dt*

### 3.113.7 Default

none

## 3.114 compute smd/tlsph/num/neighs command

### 3.114.1 Syntax

```
compute ID group-ID smd/tlsph/num/neighs
```

- ID, group-ID are documented in *compute* command
- smd/tlsph/num/neighs = style name of this compute command

### 3.114.2 Examples

```
compute 1 all smd/tlsph/num/neighs
```

### 3.114.3 Description

Define a computation that calculates the number of particles inside of the smoothing kernel radius for particles interacting via the Total-Lagrangian SPH pair style.

See this PDF guide to using Smooth Mach Dynamics in LAMMPS.

### 3.114.4 Output info

This compute calculates a per-particle vector, which can be accessed by any command that uses per-particle values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-particle values are dimensionless. See *units*.

### 3.114.5 Restrictions

This compute is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This quantity will be computed only for particles which interact with the Total-Lagrangian pair style.

### 3.114.6 Related commands

*smd/ulsph/num/neighs*

### 3.114.7 Default

none

## 3.115 compute smd/tlsph/shape command

### 3.115.1 Syntax

```
compute ID group-ID smd/tlsph/shape
```

- ID, group-ID are documented in *compute* command
- smd/tlsph/shape = style name of this compute command

### 3.115.2 Examples

```
compute 1 all smd/tlsph/shape
```

### 3.115.3 Description

Define a computation that outputs the current shape of the volume associated with a particle as a rotated ellipsoid. It is only meaningful for particles which interact according to the Total-Lagrangian SPH pair style.

See this PDF guide to use Smooth Mach Dynamics in LAMMPS.

### 3.115.4 Output info

This compute calculates a per-particle vector of vectors, which can be accessed by any command that uses per-particle values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-particle vector has 7 entries. The first three entries correspond to the lengths of the ellipsoid's axes and have units of length. These axis values are computed as the contact radius times the xx, yy, or zz components of the Green-Lagrange strain tensor associated with the particle. The next 4 values are quaternions (order: q, x, y, z) which describe the spatial rotation of the particle relative to its initial state.

### 3.115.5 Restrictions

This compute is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This quantity will be computed only for particles which interact with the Total-Lagrangian SPH pair style.

### 3.115.6 Related commands

*smd/contact/radius*

### 3.115.7 Default

none

# 3.116 compute smd/tlsph/strain command

## 3.116.1 Syntax

```
compute ID group-ID smd/tlsph/strain
```

- ID, group-ID are documented in *compute* command
- smd/tlsph/strain = style name of this compute command

## 3.116.2 Examples

```
compute 1 all smd/tlsph/strain
```

## 3.116.3 Description

Define a computation that calculates the Green-Lagrange strain tensor for particles interacting via the Total-Lagrangian SPH pair style.

See this PDF guide to using Smooth Mach Dynamics in LAMMPS.

## 3.116.4 Output info

This compute calculates a per-particle vector of vectors (tensors), which can be accessed by any command that uses per-particle values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

The per-particle tensor values will be given dimensionless. See *units*.

The per-particle vector has 6 entries, corresponding to the xx, yy, zz, xy, xz, yz components of the symmetric strain tensor.

## 3.116.5 Restrictions

This compute is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This quantity will be computed only for particles which interact with the Total-Lagrangian SPH pair style.

## 3.116.6 Related commands

*smd/tlsph/strain/rate*, *smd/tlsph/stress*

## 3.116.7 Default

none

# 3.117 compute smd/tlsph/strain/rate command

## 3.117.1 Syntax

```
compute ID group-ID smd/tlsph/strain/rate
```

- ID, group-ID are documented in *compute* command
- smd/tlsph/strain/rate = style name of this compute command

### 3.117.2 Examples

```
compute 1 all smd/tlsph/strain/rate
```

### 3.117.3 Description

Define a computation that calculates the rate of the strain tensor for particles interacting via the Total-Lagrangian SPH pair style.

See this PDF guide to using Smooth Mach Dynamics in LAMMPS.

### 3.117.4 Output info

This compute calculates a per-particle vector of vectors (tensors), which can be accessed by any command that uses per-particle values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

The values will be given in *units* of one over time.

The per-particle vector has 6 entries, corresponding to the xx, yy, zz, xy, xz, yz components of the symmetric strain rate tensor.

### 3.117.5 Restrictions

This compute is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This quantity will be computed only for particles which interact with Total-Lagrangian SPH pair style.

### 3.117.6 Related commands

*compute smd/tlsph/strain*, *compute smd/tlsph/stress*

### 3.117.7 Default

none

## 3.118 compute smd/tlsph/stress command

### 3.118.1 Syntax

```
compute ID group-ID smd/tlsph/stress
```

- ID, group-ID are documented in *compute* command
- smd/tlsph/stress = style name of this compute command

### 3.118.2 Examples

```
compute 1 all smd/tlsph/stress
```

### 3.118.3 Description

Define a computation that outputs the Cauchy stress tensor for particles interacting via the Total-Lagrangian SPH pair style.

See this PDF guide to using Smooth Mach Dynamics in LAMMPS.

### 3.118.4 Output info

This compute calculates a per-particle vector of vectors (tensors), which can be accessed by any command that uses per-particle values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

The values will be given in *units* of pressure.

The per-particle vector has 7 entries. The first six entries correspond to the xx, yy, zz, xy, xz and yz components of the symmetric Cauchy stress tensor. The seventh entry is the second invariant of the stress tensor, i.e., the von Mises equivalent stress.

### 3.118.5 Restrictions

This compute is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This quantity will be computed only for particles which interact with the Total-Lagrangian SPH pair style.

### 3.118.6 Related commands

*compute smd/tlsph/strain*, *cmopute smd/tlsph/strain/rate*

### 3.118.7 Default

none

## 3.119 compute smd/triangle/vertices command

### 3.119.1 Syntax

```
compute ID group-ID smd/triangle/vertices
```

- ID, group-ID are documented in *compute* command
- smd/triangle/vertices = style name of this compute command

### 3.119.2 Examples

```
compute 1 all smd/triangle/vertices
```

### 3.119.3 Description

Define a computation that returns the coordinates of the vertices corresponding to the triangle-elements of a mesh created by the *fix smd/wall_surface*.

See this PDF guide to using Smooth Mach Dynamics in LAMMPS.

### 3.119.4 Output info

This compute returns a per-particle vector of vectors, which can be accessed by any command that uses per-particle values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-particle vector has nine entries, (x1/y1/z1), (x2/y2/z2), and (x3/y3/z3) corresponding to the first, second, and third vertex of each triangle.

It is only meaningful to use this compute for a group of particles which is created via the *fix smd/wall_surface* command.

The output of this compute can be used with the dump2vtk_tris tool to generate a VTK representation of the smd/wall_surface mesh for visualization purposes.

The values will be given in *units* of distance.

### 3.119.5 Restrictions

This compute is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.119.6 Related commands

*fix smd/move/tri/surf* , *fix smd/wall_surface*

### 3.119.7 Default

none

## 3.120 compute smd/ulsph/effm command

### 3.120.1 Syntax

```
compute ID group-ID smd/ulsph/effm
```

- ID, group-ID are documented in *compute* command
- smd/ulsph/effm = style name of this compute command

### 3.120.2 Examples

```
compute 1 all smd/ulsph/effm
```

### 3.120.3 Description

Define a computation that outputs the effective shear modulus for particles interacting via the updated Lagrangian SPH pair style.

See this PDF guide to using Smooth Mach Dynamics in LAMMPS.

### 3.120.4 Output info

This compute calculates a per-particle vector, which can be accessed by any command that uses per-particle values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-particle vector contains the current effective per atom shear modulus as computed by the *pair smd/ulsph* pair style.

### 3.120.5 Restrictions

This compute is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info. This compute can only be used for particles which interact with the updated Lagrangian SPH pair style.

### 3.120.6 Related commands

*pair smd/ulsph*

### 3.120.7 Default

none

## 3.121 compute smd/ulsph/num/neighs command

### 3.121.1 Syntax

```
compute ID group-ID smd/ulsph/num/neighs
```

- ID, group-ID are documented in *compute* command
- smd/ulsph/num/neighs = style name of this compute command

### 3.121.2 Examples

```
compute 1 all smd/ulsph/num/neighs
```

### 3.121.3 Description

Define a computation that returns the number of neighbor particles inside of the smoothing kernel radius for particles interacting via the updated Lagrangian SPH pair style.

See this PDF guide to using Smooth Mach Dynamics in LAMMPS.

### 3.121.4 Output info

This compute returns a per-particle vector, which can be accessed by any command that uses per-particle values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-particle values will be given dimensionless, see *units*.

### 3.121.5 Restrictions

This compute is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info. This compute can only be used for particles which interact with the updated Lagrangian SPH pair style.

### 3.121.6 Related commands

*compute smd/tlsph/num/neighs*

### 3.121.7 Default

none

## 3.122 compute smd/ulsph/strain command

### 3.122.1 Syntax

```
compute ID group-ID smd/ulsph/strain
```

- ID, group-ID are documented in *compute* command
- smd/ulsph/strain = style name of this compute command

### 3.122.2 Examples

```
compute 1 all smd/ulsph/strain
```

### 3.122.3 Description

Define a computation that outputs the logarithmic strain tensor. for particles interacting via the updated Lagrangian SPH pair style.

See this PDF guide to using Smooth Mach Dynamics in LAMMPS.

### 3.122.4 Output info

This compute calculates a per-particle tensor, which can be accessed by any command that uses per-particle values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-particle vector has 6 entries, corresponding to the xx, yy, zz, xy, xz, yz components of the symmetric strain rate tensor.

The per-particle tensor values will be given dimensionless, see *units*.

### 3.122.5 Restrictions

This compute is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info. This compute can only be used for particles which interact with the updated Lagrangian SPH pair style.

### 3.122.6 Related commands

*compute smd/tlsph/strain*

### 3.122.7 Default

none

## 3.123 compute smd/ulsph/strain/rate command

### 3.123.1 Syntax

```
compute ID group-ID smd/ulsph/strain/rate
```

- ID, group-ID are documented in *compute* command
- smd/ulsph/strain/rate = style name of this compute command

### 3.123.2 Examples

```
compute 1 all smd/ulsph/strain/rate
```

### 3.123.3 Description

Define a computation that outputs the rate of the logarithmic strain tensor for particles interacting via the updated Lagrangian SPH pair style.

See this PDF guide to using Smooth Mach Dynamics in LAMMPS.

### 3.123.4 Output info

This compute calculates a per-particle vector of vectors (tensors), which can be accessed by any command that uses per-particle values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

The values will be given in *units* of one over time.

The per-particle vector has 6 entries, corresponding to the xx, yy, zz, xy, xz, yz components of the symmetric strain rate tensor.

### 3.123.5 Restrictions

This compute is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This compute can only be used for particles which interact with the updated Lagrangian SPH pair style.

### 3.123.6 Related commands

*compute smd/tlsph/strain/rate*

### 3.123.7 Default

none

## 3.124 compute smd/ulsph/stress command

### 3.124.1 Syntax

```
compute ID group-ID smd/ulsph/stress
```

- ID, group-ID are documented in *compute* command
- smd/ulsph/stress = style name of this compute command

### 3.124.2 Examples

```
compute 1 all smd/ulsph/stress
```

### 3.124.3 Description

Define a computation that outputs the Cauchy stress tensor.

See this PDF guide to using Smooth Mach Dynamics in LAMMPS.

### 3.124.4 Output info

This compute calculates a per-particle vector of vectors (tensors), which can be accessed by any command that uses per-particle values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

The values will be given in *units* of pressure.

The per-particle vector has 7 entries. The first six entries correspond to the xx, yy, zz, xy, xz, yz components of the symmetric Cauchy stress tensor. The seventh entry is the second invariant of the stress tensor, i.e., the von Mises equivalent stress.

### 3.124.5 Restrictions

This compute is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info. This compute can only be used for particles which interact with the updated Lagrangian SPH pair style.

### 3.124.6 Related commands

*compute smd/ulsph/strain*, *compute smd/ulsph/strain/rate compute smd/tlsph/stress*

### 3.124.7 Default

none

## 3.125 compute smd/vol command

### 3.125.1 Syntax

```
compute ID group-ID smd/vol
```

- ID, group-ID are documented in *compute* command
- smd/vol = style name of this compute command

### 3.125.2 Examples

```
compute 1 all smd/vol
```

### 3.125.3 Description

Define a computation that provides the per-particle volume and the sum of the per-particle volumes of the group for which the compute is defined.

See this PDF guide to using Smooth Mach Dynamics in LAMMPS.

### 3.125.4 Output info

This compute calculates a per-particle vector, which can be accessed by any command that uses per-particle values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-particle vector values will be given in *units* of volume.

Additionally, the compute returns a scalar, which is the sum of the per-particle volumes of the group for which the compute is defined.

### 3.125.5 Restrictions

This compute is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.125.6 Related commands

*compute smd/rho*

### 3.125.7 Default

none

## 3.126 compute sna/atom command

## 3.127 compute snad/atom command

## 3.128 compute snav/atom command

## 3.129 compute snap command

## 3.130 compute sna/grid command

## 3.131 compute sna/grid/local command

Accelerator Variants: *sna/grid/kk*, *sna/grid/local/kk*

### 3.131.1 Syntax

```
compute ID group-ID sna/atom rcutfac rfac0 twojmax R_1 R_2 ... w_1 w_2 ... keyword␣
↪values ...
compute ID group-ID snad/atom rcutfac rfac0 twojmax R_1 R_2 ... w_1 w_2 ... keyword␣
↪values ...
compute ID group-ID snav/atom rcutfac rfac0 twojmax R_1 R_2 ... w_1 w_2 ... keyword␣
↪values ...
compute ID group-ID snap rcutfac rfac0 twojmax R_1 R_2 ... w_1 w_2 ... keyword values ...
compute ID group-ID snap rcutfac rfac0 twojmax R_1 R_2 ... w_1 w_2 ... keyword values ...
compute ID group-ID sna/grid grid nx ny nz rcutfac rfac0 twojmax R_1 R_2 ... w_1 w_2 ...␣
↪keyword values ...
compute ID group-ID sna/grid/local grid nx ny nz rcutfac rfac0 twojmax R_1 R_2 ... w_1 w_
↪2 ... keyword values ...
```

- ID, group-ID are documented in *compute* command

- sna/atom = style name of this compute command
- *rcutfac* = scale factor applied to all cutoff radii (positive real)
- *rfac0* = parameter in distance to angle conversion (0 < rcutfac < 1)
- *twojmax* = band limit for bispectrum components (non-negative integer)
- *R_1, R_2,...* = list of cutoff radii, one for each type (distance units)
- *w_1, w_2,...* = list of neighbor weights, one for each type
- *grid* values = nx, ny, nz, number of grid points in x, y, and z directions (positive integer)
- zero or more keyword/value pairs may be appended
- keyword = *rmin0* or *switchflag* or *bzeroflag* or *quadraticflag* or *chem* or *bnormflag* or *wselfallflag* or *bikflag* or *switchinnerflag* or *sinner* or *dinner* or *dgradflag* or *nnn* or *wmode* or *delta*

```
rmin0 value = parameter in distance to angle conversion (distance units)
switchflag value = 0 or 1
   0 = do not use switching function
   1 = use switching function
bzeroflag value = 0 or 1
   0 = do not subtract B0
   1 = subtract B0
quadraticflag value = 0 or 1
   0 = do not generate quadratic terms
   1 = generate quadratic terms
chem values = nelements elementlist
   nelements = number of SNAP elements
   elementlist = ntypes integers in range [0, nelements)
bnormflag value = 0 or 1
   0 = do not normalize
   1 = normalize bispectrum components
wselfallflag value = 0 or 1
   0 = self-contribution only for element of central atom
   1 = self-contribution for all elements
switchinnerflag value = 0 or 1
   0 = do not use inner switching function
   1 = use inner switching function
sinner values = sinnerlist
   sinnerlist = ntypes values of Sinner (distance units)
dinner values = dinnerlist
   dinnerlist = ntypes values of Dinner (distance units)
bikflag value = 0 or 1 (only implemented for compute snap)
   0 = descriptors are summed over atoms of each type
   1 = descriptors are listed separately for each atom
dgradflag value = 0 or 1 (only implemented for compute snap)
   0 = descriptor gradients are summed over atoms of each type
   1 = descriptor gradients are listed separately for each atom pair
```

- additional keyword = *nnn* or *wmode* or *delta*

```
nnn value = number of considered nearest neighbors to compute the bispectrum over a␣
↪target specific number of neighbors (only implemented for compute sna/atom)
wmode value = weight function for finding optimal cutoff to match the target number␣
↪of neighbors (required if nnn used, only implemented for compute sna/atom)
   0 = heavyside weight function
```

```
   1 = hyperbolic tangent weight function
delta value = transition interval centered at cutoff distance for hyperbolic␣
↪tangent weight function (ignored if wmode=0, required if wmode=1, only␣
↪implemented for compute sna/atom)
```

### 3.131.2 Examples

```
compute b all sna/atom 1.4 0.99363 6 2.0 2.4 0.75 1.0 rmin0 0.0
compute db all sna/atom 1.4 0.95 6 2.0 1.0
compute vb all sna/atom 1.4 0.95 6 2.0 1.0
compute snap all snap 1.4 0.95 6 2.0 1.0
compute snap all snap 1.0 0.99363 6 3.81 3.83 1.0 0.93 chem 2 0 1
compute snap all snap 1.0 0.99363 6 3.81 3.83 1.0 0.93 switchinnerflag 1 sinner 1.35 1.6␣
↪dinner 0.25 0.3
compute bgrid all sna/grid/local grid 200 200 200 1.4 0.95 6 2.0 1.0
compute bnnn all sna/atom 9.0 0.99363 8 0.5 1.0 rmin0 0.0 nnn 24 wmode 1 delta 0.2
```

### 3.131.3 Description

Define a computation that calculates a set of quantities related to the bispectrum components of the atoms in a group. These computes are used primarily for calculating the dependence of energy, force, and stress components on the linear coefficients in the *snap pair_style*, which is useful when training a SNAP potential to match target data.

Bispectrum components of an atom are order parameters characterizing the radial and angular distribution of neighbor atoms. The detailed mathematical definition is given in the paper by Thompson et al. *(Thompson)*

The position of a neighbor atom *i'* relative to a central atom *i* is a point within the 3D ball of radius $R_{ii'}$ = *rcutfac* $(R_i + R'_i)$

Bartok et al. *(Bartok)*, proposed mapping this 3D ball onto the 3-sphere, the surface of the unit ball in a four-dimensional space. The radial distance *r* within *R_ii'* is mapped on to a third polar angle $\theta_0$ defined by,

$$\theta_0 = \mathrm{rfac0}\frac{r - r_{min0}}{R_{ii'} - r_{min0}}\pi$$

In this way, all possible neighbor positions are mapped on to a subset of the 3-sphere. Points south of the latitude $\theta_0 = \mathrm{rfac0}\pi$ are excluded.

The natural basis for functions on the 3-sphere is formed by the representatives of *SU(2)*, the matrices $U^j_{m,m'}(\theta, \phi, \theta_0)$. These functions are better known as $D^j_{m,m'}$, the elements of the Wigner *D*-matrices *(Meremianin, Varshalovich, Mason)* The density of neighbors on the 3-sphere can be written as a sum of Dirac-delta functions, one for each neighbor, weighted by species and radial distance. Expanding this density function as a generalized Fourier series in the basis functions, we can write each Fourier coefficient as

$$u^j_{m,m'} = U^j_{m,m'}(0,0,0) + \sum_{r_{ii'}<R_{ii'}} f_c(r_{ii'}) w_{\mu_{i'}} U^j_{m,m'}(\theta_0, \theta, \phi)$$

The $w_{\mu_{i'}}$ neighbor weights are dimensionless numbers that depend on $\mu_{i'}$, the SNAP element of atom *i'*, while the central atom is arbitrarily assigned a unit weight. The function $f_c(r)$ ensures that the contribution of each neighbor atom goes smoothly to zero at $R_{ii'}$:

$$\begin{aligned} f_c(r) =& \frac{1}{2}(\cos(\pi\frac{r - r_{min0}}{R_{ii'} - r_{min0}}) + 1), r \le R_{ii'} \\ =& 0, r > R_{ii'} \end{aligned}$$

The expansion coefficients $u^j_{m,m'}$ are complex-valued and they are not directly useful as descriptors, because they are not invariant under rotation of the polar coordinate frame. However, the following scalar triple products of expansion

coefficients can be shown to be real-valued and invariant under rotation *(Bartok)*.

$$B_{j_1,j_2,j} = \sum_{m_1,m_1'=-j_1}^{j_1} \sum_{m_2,m_2'=-j_2}^{j_2} \sum_{m,m'=-j}^{j} (u^{j}_{m,m'})^* H \begin{matrix} jmm' \\ j_1 m_1 m_1' \\ j_2 m_2 m_2' \end{matrix} u^{j_1}_{m_1,m_1'} u^{j_2}_{m_2,m_2'}$$

The constants $H^{jmm'}_{j_1 m_1 m_{1'}, j_2 m_2 m_{2'}}$ are coupling coefficients, analogous to Clebsch-Gordan coefficients for rotations on the 2-sphere. These invariants are the components of the bispectrum and these are the quantities calculated by the compute *sna/atom*. They characterize the strength of density correlations at three points on the 3-sphere. The j2=0 subset form the power spectrum, which characterizes the correlations of two points. The lowest-order components describe the coarsest features of the density function, while higher-order components reflect finer detail. Each bispectrum component contains terms that depend on the positions of up to 4 atoms (3 neighbors and the central atom).

Compute *snad/atom* calculates the derivative of the bispectrum components summed separately for each LAMMPS atom type:

$$-\sum_{i' \in I} \frac{\partial B^{i'}_{j_1,j_2,j}}{\partial \mathbf{r}_i}$$

The sum is over all atoms *i'* of atom type *I*. For each atom *i*, this compute evaluates the above expression for each direction, each atom type, and each bispectrum component. See section below on output for a detailed explanation.

Compute *snav/atom* calculates the virial contribution due to the derivatives:

$$-\mathbf{r}_i \otimes \sum_{i' \in I} \frac{\partial B^{i'}_{j_1,j_2,j}}{\partial \mathbf{r}_i}$$

Again, the sum is over all atoms *i'* of atom type *I*. For each atom *i*, this compute evaluates the above expression for each of the six virial components, each atom type, and each bispectrum component. See section below on output for a detailed explanation.

Compute *snap* calculates a global array containing information related to all three of the above per-atom computes *sna/atom*, *snad/atom*, and *snav/atom*. The first row of the array contains the summation of *sna/atom* over all atoms, but broken out by type. The last six rows of the array contain the summation of *snav/atom* over all atoms, broken out by type. In between these are 3**N* rows containing the same values computed by *snad/atom* (these are already summed over all atoms and broken out by type). The element in the last column of each row contains the potential energy, force, or stress, according to the row. These quantities correspond to the user-specified reference potential that must be subtracted from the target data when fitting SNAP. The potential energy calculation uses the built in compute *thermo_pe*. The stress calculation uses a compute called *snap_press* that is automatically created behind the scenes, according to the following command:

```
compute snap_press all pressure NULL virial
```

See section below on output for a detailed explanation of the data layout in the global array.

Added in version 3Aug2022.

The compute *sna/grid* and *sna/grid/local* commands calculate bispectrum components for a regular grid of points. These are calculated from the local density of nearby atoms *i'* around each grid point, as if there was a central atom *i* at the grid point. This is useful for characterizing fine-scale structure in a configuration of atoms, and it is used in the MALA package to build machine-learning surrogates for finite-temperature Kohn-Sham density functional theory (*Ellis et al.*) Neighbor atoms not in the group do not contribute to the bispectrum components of the grid points. The distance cutoff $R_{ii'}$ assumes that *i* has the same type as the neighbor atom *i'*. Both computes can be hardware accelerated with Kokkos by using the *sna/grid/kk* and *sna/grid/local/kk* commands, respectively.

Compute *sna/grid* calculates a global array containing bispectrum components for a regular grid of points. The grid is aligned with the current box dimensions, with the first point at the box origin, and forming a regular 3d array with *nx*, *ny*, and *nz* points in the x, y, and z directions. For triclinic boxes, the array is congruent with the periodic lattice

vectors a, b, and c. The array contains one row for each of the $nx \times ny \times nz$ grid points, looping over the index for *ix* fastest, then *iy*, and *iz* slowest. Each row of the array contains the *x*, *y*, and *z* coordinates of the grid point, followed by the bispectrum components. See section below on output for a detailed explanation of the data layout in the global array.

Compute *sna/grid/local* calculates bispectrum components of a regular grid of points similarly to compute *sna/grid* described above. However, because the array is local, it contains only rows for grid points that are local to the processor subdomain. The global grid of $nx \times ny \times nz$ points is still laid out in space the same as for *sna/grid*, but grid points are strictly partitioned, so that every grid point appears in one and only one local array. The array contains one row for each of the local grid points, looping over the global index *ix* fastest, then *iy*, and *iz* slowest. Each row of the array contains the global indexes *ix*, *iy*, and *iz* first, followed by the *x*, *y*, and *z* coordinates of the grid point, followed by the bispectrum components. See section below on output for a detailed explanation of the data layout in the global array.

The value of all bispectrum components will be zero for atoms not in the group. Neighbor atoms not in the group do not contribute to the bispectrum of atoms in the group.

The neighbor list needed to compute this quantity is constructed each time the calculation is performed (i.e. each time a snapshot of atoms is dumped). Thus it can be inefficient to compute/dump this quantity too frequently.

The argument *rcutfac* is a scale factor that controls the ratio of atomic radius to radial cutoff distance.

The argument *rfac0* and the optional keyword *rmin0* define the linear mapping from radial distance to polar angle $theta_0$ on the 3-sphere, given above.

The argument *twojmax* defines which bispectrum components are generated. See section below on output for a detailed explanation of the number of bispectrum components and the ordered in which they are listed.

The keyword *switchflag* can be used to turn off the switching function $f_c(r)$.

The keyword *bzeroflag* determines whether or not *B0*, the bispectrum components of an atom with no neighbors, are subtracted from the calculated bispectrum components. This optional keyword normally only affects compute *sna/atom*. However, when *quadraticflag* is on, it also affects *snad/atom* and *snav/atom*.

The keyword *quadraticflag* determines whether or not the quadratic combinations of bispectrum quantities are generated. These are formed by taking the outer product of the vector of bispectrum components with itself. See section below on output for a detailed explanation of the number of quadratic terms and the ordered in which they are listed.

The keyword *chem* activates the explicit multi-element variant of the SNAP bispectrum components. The argument *nelements* specifies the number of SNAP elements that will be handled. This is followed by *elementlist*, a list of integers of length *ntypes*, with values in the range [0, *nelements* ), which maps each LAMMPS type to one of the SNAP elements. Note that multiple LAMMPS types can be mapped to the same element, and some elements may be mapped by no LAMMPS type. However, in typical use cases (training SNAP potentials) the mapping from LAMMPS types to elements is one-to-one.

The explicit multi-element variant invoked by the *chem* keyword partitions the density of neighbors into partial densities for each chemical element. This is described in detail in the paper by Cusentino et al. The bispectrum components are indexed on ordered triplets of elements:

$$B^{\kappa\lambda\mu}_{j_1,j_2,j} = \sum_{m_1,m_1'=-j_1}^{j_1} \sum_{m_2,m_2'=-j_2}^{j_2} \sum_{m,m'=-j}^{j} (u^{\mu}_{j,m,m'})^* H \begin{matrix} jmm' \\ j_1 m_1 m_1' \\ j_2 m_2 m_2' \end{matrix} u^{\kappa}_{j_1,m_1,m_1'} u^{\lambda}_{j_2,m_2,m_2'}$$

where $u^{\mu}_{j,m,m'}$ is an expansion coefficient for the partial density of neighbors of element $\mu$

$$u^{\mu}_{j,m,m'} = w^{self}_{\mu_i\mu} U^{j,m,m'}(0,0,0) + \sum_{r_{ii'}<R_{ii'}} \delta_{\mu\mu_{i'}} f_c(r_{ii'}) w_{\mu_{i'}} U^{j,m,m'}(\theta_0,\theta,\phi)$$

where $w^{self}_{\mu_i\mu}$ is the self-contribution, which is either 1 or 0 (see keyword *wselfallflag* below), $\delta_{\mu\mu_{i'}}$ indicates that the sum is only over neighbor atoms of element $\mu$, and all other quantities are the same as those appearing in the original equation for $u^{j}_{m,m'}$ given above.

The keyword *wselfallflag* defines the rule used for the self-contribution. If *wselfallflag* is on, then $w^{self}_{\mu_i\mu} = 1$. If it is off then $w^{self}_{\mu_i\mu} = 0$, except in the case of $\mu_i = \mu$, when $w^{self}_{\mu_i\mu} = 1$. When the *chem* keyword is not used, this keyword has no effect.

The keyword *bnormflag* determines whether or not the bispectrum component $B_{j_1,j_2,j}$ is divided by a factor of $2j+1$. This normalization simplifies force calculations because of the following symmetry relation

$$\frac{B_{j_1,j_2,j}}{2j+1} = \frac{B_{j,j_2,j_1}}{2j_1+1} = \frac{B_{j_1,j,j_2}}{2j_2+1}$$

This option is typically used in conjunction with the *chem* keyword, and LAMMPS will generate a warning if both *chem* and *bnormflag* are not both set or not both unset.

The keyword *switchinnerflag* with value 1 activates an additional radial switching function similar to $f_c(r)$ above, but acting to switch off smoothly contributions from neighbor atoms at short separation distances. This is useful when SNAP is used in combination with a simple repulsive potential. For a neighbor atom at distance $r$, its contribution is scaled by a multiplicative factor $f_{inner}(r)$ defined as follows:

$$\begin{aligned} &=0, r \le S_{inner} - D_{inner} \\ f_{inner}(r) &=\frac{1}{2}(1 - \cos(\frac{\pi}{2}(1 + \frac{r - S_{inner}}{D_{inner}})), S_{inner} - D_{inner} < r \le S_{inner} + D_{inner} \\ &=1, r > S_{inner} + D_{inner} \end{aligned}$$

where the switching region is centered at $S_{inner}$ and it extends a distance $D_{inner}$ to the left and to the right of this. With this option, additional keywords *sinner* and *dinner* must be used, each followed by *ntypes* values for $S_{inner}$ and $D_{inner}$, respectively. When the central atom and the neighbor atom have different types, the values of $S_{inner}$ and $D_{inner}$ are the arithmetic means of the values for both types.

The keywords *bikflag* and *dgradflag* are only used by compute *snap*. The keyword *bikflag* determines whether or not to list the descriptors of each atom separately, or sum them together and list in a single row. If *bikflag* is set to *0* then a single bispectrum row is used, which contains the per-atom bispectrum descriptors $B_{i,k}$ summed over all atoms $i$ to produce $B_k$. If *bikflag* is set to *1* this is replaced by a separate per-atom bispectrum row for each atom. In this case, the entries in the final column for these rows are set to zero.

The keyword *dgradflag* determines whether to sum atom gradients or list them separately. If *dgradflag* is set to 0, the bispectrum descriptor gradients w.r.t. atom $j$ are summed over all atoms $i$' of type $I$ (similar to *snad/atom* above). If *dgradflag* is set to 1, gradients are listed separately for each pair of atoms. Each row corresponds to a single term $\frac{\partial B_{i,k}}{\partial r^a_j}$ where $r^a_j$ is the *a-th* position coordinate of the atom with global index $j$. This also changes the number of columns to be equal to the number of bispectrum components, with 3 additional columns representing the indices $i$, $j$, and $a$, as explained more in the Output info section below. The option *dgradflag=1* requires that *bikflag=1*.

**Note**

Using *dgradflag* = 1 produces a global array with $N + 3N^2 + 1$ rows which becomes expensive for systems with more than 1000 atoms.

**Note**

If you have a bonded system, then the settings of *special_bonds* command can remove pairwise interactions between atoms in the same bond, angle, or dihedral. This is the default setting for the *special_bonds* command, and means those pairwise interactions do not appear in the neighbor list. Because this fix uses the neighbor list, it also means those pairs will not be included in the calculation. One way to get around this, is to write a dump file, and use the *rerun* command to compute the bispectrum components for snapshots in the dump file. The rerun script can use a *special_bonds* command that includes all pairs in the neighbor list.

The keyword *nnn* allows for the calculation of the bispectrum over a specific target number of neighbors. This option is only implemented for the compute *sna/atom*. An optimal cutoff radius for defining the neighborhood of the central atom is calculated by means of a dichotomy algorithm. This iterative process allows to assign weights to neighboring atoms in order to match the total sum of weights with the target number of neighbors. Depending on the radial weight function used in that process, the cutoff radius can fluctuate a lot in the presence of thermal noise. Therefore, in addition to the *nnn* keyword, the keyword *wmode* allows to choose whether a Heaviside (*wmode* = 0) function or a Hyperbolic tangent function (*wmode* = 1) should be used. If the Heaviside function is used, the cutoff radius exactly matches the distance between the central atom an its *nnn*'th neighbor. However, in the case of the hyperbolic tangent function, the dichotomy algorithm allows to span the weights over a distance *delta* in order to reduce fluctuations in the resulting local atomic environment fingerprint. The detailed formalism is given in the paper by Lafourcade et al. *(Lafourcade)*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 3.131.4 Output info

Compute *sna/atom* calculates a per-atom array, each column corresponding to a particular bispectrum component. The total number of columns and the identity of the bispectrum component contained in each column depend of the value of *twojmax*, as described by the following piece of python code:

```
for j1 in range(0,twojmax+1):
    for j2 in range(0,j1+1):
        for j in range(j1-j2,min(twojmax,j1+j2)+1,2):
            if (j>=j1): print j1/2.,j2/2.,j/2.
```

There are $m(m+1)/2$ descriptors with last index $j$, where $m = \lfloor j \rfloor + 1$. Hence, for even *twojmax* = 2($m$-1), $K = m(m+1)(2m+1)/6$, the $m$-th pyramidal number, and for odd *twojmax* = 2 $m$-1, $K = m(m+1)(m+2)/3$, twice the $m$-th tetrahedral number.

**Note**

the *diagonal* keyword allowing other possible choices for the number of bispectrum components was removed in 2019, since all potentials use the value of 3, corresponding to the above set of bispectrum components.

Compute *snad/atom* evaluates a per-atom array. The columns are arranged into *ntypes* blocks, listed in order of atom type $I$. Each block contains three sub-blocks corresponding to the $x$, $y$, and $z$ components of the atom position. Each of these sub-blocks contains $K$ columns for the $K$ bispectrum components, the same as for compute *sna/atom*

Compute *snav/atom* evaluates a per-atom array. The columns are arranged into *ntypes* blocks, listed in order of atom type $I$. Each block contains six sub-blocks corresponding to the $xx$, $yy$, $zz$, $yz$, $xz$, and $xy$ components of the virial tensor in Voigt notation. Each of these sub-blocks contains $K$ columns for the $K$ bispectrum components, the same as for compute *sna/atom*

Compute *snap* evaluates a global array. The columns are arranged into *ntypes* blocks, listed in order of atom type *I*. Each block contains one column for each bispectrum component, the same as for compute *sna/atom*. A final column contains the corresponding energy, force component on an atom, or virial stress component. The rows of the array appear in the following order:

- 1 row: *sna/atom* quantities summed for all atoms of type *I*
- 3*$N$ rows: *snad/atom* quantities, with derivatives w.r.t. x, y, and z coordinate of atom *i* appearing in consecutive rows. The atoms are sorted based on atom ID.
- 6 rows: *snav/atom* quantities summed for all atoms of type *I*

For example, if $K$ =30 and ntypes=1, the number of columns in the per-atom arrays generated by *sna/atom*, *snad/atom*, and *snav/atom* are 30, 90, and 180, respectively. With *quadratic* value=1, the numbers of columns are 930, 2790, and 5580, respectively. The number of columns in the global array generated by *snap* are 31, and 931, respectively, while the number of rows is 1+3*$N$+6, where $N$ is the total number of atoms.

Compute *sna/grid* evaluates a global array. The array contains one row for each of the $nx \times ny \times nz$ grid points, looping over the index for *ix* fastest, then *iy*, and *iz* slowest. Each row of the array contains the $x$, $y$, and $z$ coordinates of the grid point, followed by the bispectrum components.

Compute *sna/grid/local* evaluates a local array. The array contains one row for each of the local grid points, looping over the global index *ix* fastest, then *iy*, and *iz* slowest. Each row of the array contains the global indexes *ix*, *iy*, and *iz* first, followed by the $x$, $y$, and $z$ coordinates of the grid point, followed by the bispectrum components.

If the *quadratic* keyword value is set to 1, then additional columns are generated, corresponding to the products of all distinct pairs of bispectrum components. If the number of bispectrum components is $K$, then the number of distinct pairs is $K(K+1)/2$. For compute *sna/atom* these columns are appended to existing $K$ columns. The ordering of quadratic terms is upper-triangular, (1,1),(1,2)...(1,$K$),(2,1)...($K$-1,$K$-1),($K$-1,$K$),($K$,$K$). For computes *snad/atom* and *snav/atom* each set of $K(K+1)/2$ additional columns is inserted directly after each of sub-block of linear terms i.e. linear and quadratic terms are contiguous. So the nesting order from inside to outside is bispectrum component, linear then quadratic, vector/tensor component, type.

If the *chem* keyword is used, then the data is arranged into $N_{elem}^3$ sub-blocks, each sub-block corresponding to a particular chemical labeling $\kappa\lambda\mu$ with the last label changing fastest. Each sub-block contains $K$ bispectrum components. For the purposes of handling contributions to force, virial, and quadratic combinations, these $N_{elem}^3$ sub-blocks are treated as a single block of $KN_{elem}^3$ columns.

If the *bik* keyword is set to 1, the structure of the snap array is expanded. The first $N$ rows of the snap array correspond to $B_{i,k}$ instead of a single row summed over atoms $i$. In this case, the entries in the final column for these rows are set to zero. Also, each row contains only non-zero entries for the columns corresponding to the type of that atom. This is not true in the case of *dgradflag* keyword = 1 (see below).

If the *dgradflag* keyword is set to 1, this changes the structure of the global array completely. Here the *snad/atom* quantities are replaced with rows corresponding to descriptor gradient components on single atoms:

$$\frac{\partial B_{i,k}}{\partial r_j^a}$$

where $r_j^a$ is the *a-th* position coordinate of the atom with global index $j$. The rows are organized in chunks, where each chunk corresponds to an atom with global index $j$. The rows in an atom $j$ chunk correspond to atoms with global index $i$. The total number of rows for these descriptor gradients is therefore $3N^2$. The number of columns is equal to the number of bispectrum components, plus 3 additional left-most columns representing the global atom indices $i$, $j$, and Cartesian direction $a$ (0, 1, 2, for x, y, z). The first 3 columns of the first $N$ rows belong to the reference potential force components. The remaining K columns contain the $B_{i,k}$ per-atom descriptors corresponding to the non-zero entries obtained when *bikflag* = 1. The first column of the last row, after the first $N + 3N^2$ rows, contains the reference potential energy. The virial components are not used with this option. The total number of rows is therefore $N + 3N^2 + 1$ and the number of columns is $K + 3$.

These values can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options. To see how this command can be used within a Python workflow to train SNAP potentials, see the examples in FitSNAP.

### 3.131.5 Restrictions

These computes are part of the ML-SNAP package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.131.6 Related commands

*pair_style snap compute slcsa/atom*

### 3.131.7 Default

The optional keyword defaults are *rmin0* = 0, *switchflag* = 1, *bzeroflag* = 1, *quadraticflag* = 0, *bnormflag* = 0, *wselfallflag* = 0, *switchinnerflag* = 0, *nnn* = -1, *wmode* = 0, *delta* = 1.e-3

---

**(Thompson)** Thompson, Swiler, Trott, Foiles, Tucker, J Comp Phys, 285, 316, (2015).

**(Bartok)** Bartok, Payne, Risi, Csanyi, Phys Rev Lett, 104, 136403 (2010).

**(Meremianin)** Meremianin, J. Phys. A, 39, 3099 (2006).

**(Varshalovich)** Varshalovich, Moskalev, Khersonskii, Quantum Theory of Angular Momentum, World Scientific, Singapore (1987).

**(Mason)** J. K. Mason, Acta Cryst A65, 259 (2009).

**(Cusentino)** Cusentino, Wood, Thompson, J Phys Chem A, 124, 5456, (2020)

**(Ellis)** Ellis, Fiedler, Popoola, Modine, Stephens, Thompson, Cangi, Rajamanickam, Phys. Rev. B, 104, 035120, (2021)

**(Lafourcade)** Lafourcade, Maillet, Denoual, Duval, Allera, Goryaeva, and Marinica, Comp. Mat. Science, 230, 112534 (2023)

## 3.132 compute sph/e/atom command

### 3.132.1 Syntax

```
compute ID group-ID sph/e/atom
```

- ID, group-ID are documented in *compute* command
- sph/e/atom = style name of this compute command

### 3.132.2 Examples

```
compute 1 all sph/e/atom
```

### 3.132.3 Description

Define a computation that calculates the per-atom internal energy for each atom in a group.

The internal energy is the energy associated with the internal degrees of freedom of an SPH particle, i.e. a Smooth-Particle Hydrodynamics particle.

See this PDF guide to using SPH in LAMMPS.

> **Note**
>
> Please note that the SPH PDF guide file has not been updated for many years and thus does not reflect the current *syntax* of the SPH package commands. For that please refer to the LAMMPS manual.

The value of the internal energy will be 0.0 for atoms not in the specified compute group.

### 3.132.4 Output info

This compute calculates a per-atom vector, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-atom vector values will be in energy *units*.

### 3.132.5 Restrictions

This compute is part of the SPH package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.132.6 Related commands

*dump custom*

### 3.132.7 Default

none

## 3.133 compute sph/rho/atom command

### 3.133.1 Syntax

```
compute ID group-ID sph/rho/atom
```

- ID, group-ID are documented in *compute* command
- sph/rho/atom = style name of this compute command

### 3.133.2 Examples

```
compute 1 all sph/rho/atom
```

### 3.133.3 Description

Define a computation that calculates the per-atom SPH density for each atom in a group, i.e. a Smooth-Particle Hydrodynamics density.

The SPH density is the mass density of an SPH particle, calculated by kernel function interpolation using "pair style sph/rhosum".

See this PDF guide to using SPH in LAMMPS.

> **Note**
>
> Please note that the SPH PDF guide file has not been updated for many years and thus does not reflect the current *syntax* of the SPH package commands. For that please refer to the LAMMPS manual.

The value of the SPH density will be 0.0 for atoms not in the specified compute group.

### 3.133.4 Output info

This compute calculates a per-atom vector, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-atom vector values will be in mass/volume *units*.

### 3.133.5 Restrictions

This compute is part of the SPH package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.133.6 Related commands

*dump custom*

### 3.133.7 Default

none

## 3.134 compute sph/t/atom command

### 3.134.1 Syntax

```
compute ID group-ID sph/t/atom
```

- ID, group-ID are documented in *compute* command
- sph/t/atom = style name of this compute command

### 3.134.2 Examples

```
compute 1 all sph/t/atom
```

### 3.134.3 Description

Define a computation that calculates the per-atom internal temperature for each atom in a group.

The internal temperature is the ratio of internal energy over the heat capacity associated with the internal degrees of freedom of an SPH particles, i.e. a Smooth-Particle Hydrodynamics particle.

$$T_{int} = E_{int}/C_{V,int}$$

See this PDF guide to using SPH in LAMMPS.

**Note**

Please note that the SPH PDF guide file has not been updated for many years and thus does not reflect the current *syntax* of the SPH package commands. For that please refer to the LAMMPS manual.

The value of the internal energy will be 0.0 for atoms not in the specified compute group.

### 3.134.4 Output info

This compute calculates a per-atom vector, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-atom vector values will be in temperature *units*.

### 3.134.5 Restrictions

This compute is part of the SPH package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.134.6 Related commands

*dump custom*

### 3.134.7 Default

none

## 3.135 compute spin command

### 3.135.1 Syntax

```
compute ID group-ID spin
```

- ID, group-ID are documented in *compute* command
- spin = style name of this compute command

### 3.135.2 Examples

```
compute out_mag all spin
```

### 3.135.3 Description

Define a computation that calculates magnetic quantities for a system of atoms having spins.

This compute calculates the following 6 magnetic quantities:

- the three first quantities are the x,y and z coordinates of the total magnetization,
- the fourth quantity is the norm of the total magnetization,
- The fifth quantity is the magnetic energy (in eV),
- The sixth one is referred to as the spin temperature, according to the work of *(Nurdin)*.

The simplest way to output the results of the compute spin calculation is to define some of the quantities as variables, and to use the thermo and thermo_style commands, for example:

```
compute out_mag          all spin

variable mag_z           equal c_out_mag[3]
variable mag_norm        equal c_out_mag[4]
variable temp_mag        equal c_out_mag[6]

thermo                   10
thermo_style             custom step v_mag_z v_mag_norm v_temp_mag
```

This series of commands evaluates the total magnetization along z, the norm of the total magnetization, and the magnetic temperature. Three variables are assigned to those quantities. The thermo and thermo_style commands print them every 10 timesteps.

### 3.135.4 Output info

The array values are "intensive". The array values will be in metal units (*units*).

### 3.135.5 Restrictions

The *spin* compute is part of the SPIN package. This compute is only enabled if LAMMPS was built with this package. See the *Build package* page for more info. The atom_style has to be "spin" for this compute to be valid.

**Related commands:**

none

### 3.135.6 Default

none

---

**(Nurdin)** Nurdin and Schotte Phys Rev E, 61(4), 3579 (2000)

## 3.136 compute stress/atom command

## 3.137 compute centroid/stress/atom command

### 3.137.1 Syntax

```
compute ID group-ID style temp-ID keyword ...
```

- ID, group-ID are documented in *compute* command
- style = *stress/atom* or *centroid/stress/atom*
- temp-ID = ID of compute that calculates temperature, can be NULL if not needed
- zero or more keywords may be appended
- keyword = *ke* or *pair* or *bond* or *angle* or *dihedral* or *improper* or *kspace* or *fix* or *virial*

### 3.137.2 Examples

```
compute 1 mobile stress/atom NULL
compute 1 mobile stress/atom myRamp
compute 1 all stress/atom NULL pair bond
compute 1 all centroid/stress/atom NULL bond dihedral improper
```

### 3.137.3 Description

Define a computation that computes per-atom stress tensor for each atom in a group. In case of compute *stress/atom*, the tensor for each atom is symmetric with 6 components and is stored as a 6-element vector in the following order: *xx*, *yy*, *zz*, *xy*, *xz*, *yz*. In case of compute *centroid/stress/atom*, the tensor for each atom is asymmetric with 9 components and is stored as a 9-element vector in the following order: *xx*, *yy*, *zz*, *xy*, *xz*, *yz*, *yx*, *zx*, *zy*. See the *compute pressure* command if you want the stress tensor (pressure) of the entire system.

The stress tensor for atom $I$ is given by the following formula, where $a$ and $b$ take on values $x$, $y$, $z$ to generate the components of the tensor:

$$S_{ab} = -mv_a v_b - W_{ab}$$

The first term is a kinetic energy contribution for atom $I$. See details below on how the specified *temp-ID* can affect the velocities used in this calculation. The second term is the virial contribution due to intra and intermolecular interactions, where the exact computation details are determined by the compute style.

In case of compute *stress/atom*, the virial contribution is:

$$\begin{aligned} W_{ab} = & \frac{1}{2}\sum_{n=1}^{N_p}(r_{1_a}F_{1_b} + r_{2_a}F_{2_b}) + \frac{1}{2}\sum_{n=1}^{N_b}(r_{1_a}F_{1_b} + r_{2_a}F_{2_b}) \\ & + \frac{1}{3}\sum_{n=1}^{N_a}(r_{1_a}F_{1_b} + r_{2_a}F_{2_b} + r_{3_a}F_{3_b}) + \frac{1}{4}\sum_{n=1}^{N_d}(r_{1_a}F_{1_b} + r_{2_a}F_{2_b} + r_{3_a}F_{3_b} + r_{4_a}F_{4_b}) \\ & + \frac{1}{4}\sum_{n=1}^{N_i}(r_{1_a}F_{1_b} + r_{2_a}F_{2_b} + r_{3_a}F_{3_b} + r_{4_a}F_{4_b}) + \mathrm{Kspace}(r_{i_a}, F_{i_b}) + \sum_{n=1}^{N_f} r_{i_a}F_{i_b} \end{aligned}$$

The first term is a pairwise energy contribution where $n$ loops over the $N_p$ neighbors of atom $I$, $\mathbf{r}_1$ and $\mathbf{r}_2$ are the positions of the two atoms in the pairwise interaction, and $\mathbf{F}_1$ and $\mathbf{F}_2$ are the forces on the two atoms resulting from the pairwise interaction. The second term is a bond contribution of similar form for the $N_b$ bonds which atom $I$ is part of. There are similar terms for the $N_a$ angle, $N_d$ dihedral, and $N_i$ improper interactions atom $I$ is part of. There is also a term for the KSpace contribution from long-range Coulombic interactions, if defined. Finally, there is a term for the $N_f$ *fixes* that apply internal constraint forces to atom $I$. Currently, only the *fix shake* and *fix rigid* commands contribute to this term. As the coefficients in the formula imply, a virial contribution produced by a small set of atoms (e.g. 4 atoms in a dihedral or 3 atoms in a Tersoff 3-body interaction) is assigned in equal portions to each atom in the set. E.g. 1/4 of the dihedral virial to each of the 4 atoms, or 1/3 of the fix virial due to SHAKE constraints applied to atoms in a water molecule via the *fix shake* command. As an exception, the virial contribution from constraint forces in *fix rigid* on each atom is computed from the constraint force acting on the corresponding atom and its position, i.e. the total virial is not equally distributed.

In case of compute *centroid/stress/atom*, the virial contribution is:

$$W_{ab} = \sum_{n=1}^{N_p} r_{I0_a} F_{I_b} + \sum_{n=1}^{N_b} r_{I0_a} F_{I_b} + \sum_{n=1}^{N_a} r_{I0_a} F_{I_b} + \sum_{n=1}^{N_d} r_{I0_a} F_{I_b} + \sum_{n=1}^{N_i} r_{I0_a} F_{I_b}$$
$$+ \text{Kspace}(r_{i_a}, F_{i_b}) + \sum_{n=1}^{N_f} r_{i_a} F_{i_b}$$

As with compute *stress/atom*, the first, second, third, fourth and fifth terms are pairwise, bond, angle, dihedral and improper contributions, but instead of assigning the virial contribution equally to each atom, only the force $\mathbf{F}_I$ acting on atom $I$ due to the interaction and the relative position $\mathbf{r}_{I0}$ of the atom $I$ to the geometric center of the interacting atoms, i.e. centroid, is used. As the geometric center is different for each interaction, the $\mathbf{r}_{I0}$ also differs. The sixth term, Kspace contribution, is computed identically to compute *stress/atom*. The seventh term is handed differently depending on if the constraint forces are due to *fix shake* or *fix rigid*. In case of SHAKE constraints, each distance constraint is handed as a pairwise interaction. E.g. in case of a water molecule, two OH and one HH distance constraints are treated as three pairwise interactions. In case of *fix rigid*, all constraint forces in the molecule are treated as a single many-body interaction with a single centroid position. In case of water molecule, the formula expression would become identical to that of the three-body angle interaction. Although the total system virial is the same as compute *stress/atom*, compute *centroid/stress/atom* is know to result in more consistent heat flux values for angle, dihedrals, improper and constraint force contributions when computed via *compute heat/flux*.

If no extra keywords are listed, the kinetic contribution *and* all of the virial contribution terms are included in the per-atom stress tensor. If any extra keywords are listed, only those terms are summed to compute the tensor. The *virial* keyword means include all terms except the kinetic energy *ke*.

Note that the stress for each atom is due to its interaction with all other atoms in the simulation, not just with other atoms in the group.

Details of how compute *stress/atom* obtains the virial for individual atoms for either pairwise or many-body potentials, and including the effects of periodic boundary conditions is discussed in *(Thompson)*. The basic idea for many-body potentials is to treat each component of the force computation between a small cluster of atoms in the same manner as in the formula above for bond, angle, dihedral, etc interactions. Namely the quantity $\mathbf{r} \cdot \mathbf{F}$ is summed over the atoms in the interaction, with the $r$ vectors unwrapped by periodic boundaries so that the cluster of atoms is close together. The total contribution for the cluster interaction is divided evenly among those atoms.

Details of how compute *centroid/stress/atom* obtains the virial for individual atoms are given in *(Surblys2019)* and *(Surblys2021)*, where the idea is that the virial of the atom $I$ is the result of only the force $\mathbf{F}_I$ on the atom due to the interaction and its positional vector $\mathbf{r}_{I0}$, relative to the geometric center of the interacting atoms, regardless of the number of participating atoms. The periodic boundary treatment is identical to that of compute *stress/atom*, and both of them reduce to identical expressions for two-body interactions, i.e. computed values for contributions from bonds and two-body pair styles, such as *Lennard-Jones*, will be the same, while contributions from angles, dihedrals and impropers will be different.

The *dihedral_style charmm* style calculates pairwise interactions between 1-4 atoms. The virial contribution of these terms is included in the pair virial, not the dihedral virial.

The KSpace contribution is calculated using the method in *(Heyes)* for the Ewald method and by the methodology described in *(Sirk)* for PPPM. The choice of KSpace solver is specified by the *kspace_style pppm* command. Note that for PPPM, the calculation requires 6 extra FFTs each timestep that per-atom stress is calculated. Thus it can significantly increase the cost of the PPPM calculation if it is needed on a large fraction of the simulation timesteps.

The *temp-ID* argument can be used to affect the per-atom velocities used in the kinetic energy contribution to the total stress. If the kinetic energy is not included in the stress, than the temperature compute is not used and can be specified as NULL. If the kinetic energy is included and you wish to use atom velocities as-is, then *temp-ID* can also be specified as NULL. If desired, the specified temperature compute can be one that subtracts off a bias to leave each atom with only a thermal velocity to use in the formula above, e.g. by subtracting a background streaming velocity. See the doc pages for individual *compute commands* to determine which ones include a bias.

Note that as defined in the formula, per-atom stress is the negative of the per-atom pressure tensor. It is also really a stress*volume formulation, meaning the computed quantity is in units of pressure*volume. It would need to be divided by a per-atom volume to have units of stress (pressure), but an individual atom's volume is not well defined or easy to compute in a deformed solid or a liquid. See the *compute voronoi/atom* command for one possible way to estimate a per-atom volume.

Thus, if the diagonal components of the per-atom stress tensor are summed for all atoms in the system and the sum is divided by $dV$, where $d$ = dimension and $V$ is the volume of the system, the result should be $-P$, where $P$ is the total pressure of the system.

These lines in an input script for a 3d system should yield that result. I.e. the last 2 columns of thermo output will be the same:

```
compute         peratom all stress/atom NULL
compute         p all reduce sum c_peratom[1] c_peratom[2] c_peratom[3]
variable        press equal -(c_p[1]+c_p[2]+c_p[3])/(3*vol)
thermo_style    custom step temp etotal press v_press
```

**Note**

The per-atom stress does not include any Lennard-Jones tail corrections to the pressure added by the *pair_modify tail yes* command, since those are contributions to the global system pressure.

The compute stress/atom can be used in a number of ways. Here is an example to compute a 1-d pressure profile in x-direction across the complete simulation box. You will need to adjust the number of bins and the selections for time averaging to your specific simulation. This assumes that the dimensions of the simulation cell does not change.

```
# set number of bins
variable nbins index 20
variable fraction equal 1.0/v_nbins
# define bins as chunks
compute cchunk all chunk/atom bin/1d x lower ${fraction} units reduced
compute stress all stress/atom NULL
# apply conversion to pressure early since we have no variable style for processing␣
↪chunks
variable press atom -(c_stress[1]+c_stress[2]+c_stress[3])/(3.0*vol*${fraction})
compute binpress all reduce/chunk cchunk sum v_press
fix avg all ave/time 10 40 400 c_binpress mode vector file ave_stress.txt
```

### 3.137.4 Output info

Compute *stress/atom* calculates a per-atom array with 6 columns, which can be accessed by indices 1-6 by any command that uses per-atom values from a compute as input. Compute *centroid/stress/atom* produces a per-atom array with 9 columns, but otherwise can be used in an identical manner to compute *stress/atom*. See the *Howto output* page for an overview of LAMMPS output options.

The ordering of the 6 columns for *stress/atom* is as follows: xx, yy, zz, xy, xz, yz. The ordering of the 9 columns for *centroid/stress/atom* is as follows: xx, yy, zz, xy, xz, yz, yx, zx, zy.

The per-atom array values will be in pressure*volume *units* as discussed above.

### 3.137.5 Restrictions

Currently, compute *centroid/stress/atom* does not support pair styles with many-body interactions (*EAM* is an exception, since its computations are performed pairwise), nor granular pair styles with pairwise forces which are not aligned with the vector between the pair of particles. All bond styles are supported. All angle, dihedral, improper styles are supported with the exception of INTEL and KOKKOS variants of specific styles. It also does not support models with long-range Coulombic or dispersion forces, i.e. the kspace_style command in LAMMPS. It also does not implement the following fixes which add rigid-body constraints: *fix rigid/** and the OpenMP accelerated version of *fix rigid/small*, while all other *fix rigid/*/small* are implemented.

LAMMPS will generate an error if one of these options is included in your model. Extension of centroid stress calculations to these force and fix styles is planned for the future.

### 3.137.6 Related commands

*compute pe*, *compute pressure*

### 3.137.7 Default

By default the compute includes contributions from the keywords: `ke pair bond angle dihedral improper kspace fix`

---

**(Heyes)** Heyes, Phys Rev B, 49, 755 (1994).

**(Sirk)** Sirk, Moore, Brown, J Chem Phys, 138, 064505 (2013).

**(Thompson)** Thompson, Plimpton, Mattson, J Chem Phys, 131, 154107 (2009).

**(Surblys2019)** Surblys, Matsubara, Kikugawa, Ohara, Phys Rev E, 99, 051301(R) (2019).

**(Surblys2021)** Surblys, Matsubara, Kikugawa, Ohara, J Appl Phys 130, 215104 (2021).

## 3.138 compute stress/cartesian command

### 3.138.1 Syntax

```
compute ID group-ID stress/cartesian args
```

- ID, group-ID are documented in *compute* command
- args = argument specific to the compute style

```
stress/cartesian args = dim1 bin_width1 dim2 bin_width2 keyword
  dim1 = x or y or z
  bin_width1 = width of the bin
  dim2 = x or y or z or NULL
  bin_width2 = width of the bin
  keyword = ke or pair or bond
```

### 3.138.2 Examples

```
compute 1 all stress/cartesian x 0.1 NULL 0
compute 1 all stress/cartesian y 0.1 z 0.1
compute 1 all stress/cartesian x 0.1 NULL 0 ke pair
```

### 3.138.3 Description

Compute *stress/cartesian* defines computations that calculate profiles of the diagonal components of the local stress tensor over one or two Cartesian dimensions, as described in *(Ikeshoji)*. The stress tensor is split into a kinetic contribution $P^k$ and a virial contribution $P^v$. The sum gives the total stress tensor $P = P^k + P^v$. This compute obeys momentum balance through fluid interfaces. They use the Irving–Kirkwood contour, which is the straight line between particle pairs.

Added in version 15Jun2023: Added support for bond styles

This compute only supports pair and bond (no angle, dihedral, improper, or kspace) forces. By default, if no extra keywords are specified, all supported contributions to the stress are included (ke, pair, bond). If any keywords are specified, then only those components are summed.

### 3.138.4 Output info

The output columns for *stress/cartesian* are the position of the center of the local volume in the first and second dimensions, number density, $P^k_{xx}$, $P^k_{yy}$, $P^k_{zz}$, $P^v_{xx}$, $P^v_{yy}$, and $P^v_{zz}$. There are 8 columns when one dimension is specified and 9 columns when two dimensions are specified. The number of bins (rows) is $(L_1/b_1)(L_2/b_2)$, where $L_1$ and $L_2$ are the lengths of the simulation box in the specified dimensions and $b_1$ and $b_2$ are the specified bin widths. When only one dimension is specified, the number of bins (rows) is $L_1/b_1$.

This array can be output with *fix ave/time*,

```
compute p all stress/cartesian x 0.1
fix 2 all ave/time 100 1 100 c_p[*] file dump_p.out mode vector
```

The values calculated by this compute are "intensive". The stress values will be in pressure *units*. The number density values are in inverse volume *units*.

NOTE 1: The local stress does not include any Lennard-Jones tail corrections to the stress added by the *pair_modify tail yes* command, since those are contributions to the global system pressure.

NOTE 2: The local stress profiles generated by these computes are similar to those obtained by the *method-of-planes (MOP)*. A key difference is that compute *stress/mop/profile* considers particles crossing a set of planes, while *stress/cartesian* computes averages for a set of small volumes. Moreover, this compute computes the diagonal components of the stress tensor $P_{xx}$, $P_{yy}$, and $P_{zz}$, while *stress/mop/profile* computes the components $P_{ix}$, $P_{iy}$, and $P_{iz}$, where $i$ is the direction normal to the plane.

More information on the similarities and differences can be found in *(Ikeshoji)*.

### 3.138.5 Restrictions

These computes calculate the stress tensor contributions for pair and bond forces only (no angle, dihedral, improper, or kspace force). It requires pairwise force calculations not available for most many-body pair styles.

These computes are part of the EXTRA-COMPUTE package. They are only enabled if LAMMPS was built with that package. See the *Build package* doc page for more info.

### 3.138.6 Related commands

*compute stress/atom*, *compute pressure*, *compute stress/mop/profile*, *compute stress/spherical*, *compute stress/cylinder*

---

**(Ikeshoji)** Ikeshoji, Hafskjold, Furuholt, Mol Sim, 29, 101-109, (2003).

# 3.139 compute stress/cylinder command

# 3.140 compute stress/spherical command

## 3.140.1 Syntax

```
compute ID group-ID style args
```

- ID, group-ID are documented in *compute* command
- style = stress/spherical or stress/cylinder
- args = argument specific to the compute style

```
stress/cylinder args = zlo zh Rmax bin_width keyword
  zlo = minimum z-boundary for cylinder
  zhi = maximum z-boundary for cylinder
  Rmax = maximum radius to perform calculation to
  bin_width = width of radial bins to use for calculation
  keyword = ke (zero or one can be specified)
    ke = yes or no
stress/spherical
  x0, y0, z0 = origin of the spherical coordinate system
  bin_width = width of spherical shells
  Rmax = maximum radius of spherical shells
```

## 3.140.2 Examples

```
compute 1 all stress/cylinder -10.0 10.0 15.0 0.25
compute 1 all stress/cylinder -10.0 10.0 15.0 0.25 ke no
compute 1 all stress/spherical 0 0 0 0.1 10
```

## 3.140.3 Description

Compute *stress/cylinder*, and compute *stress/spherical* define computations that calculate profiles of the diagonal components of the local stress tensor in the specified coordinate system. The stress tensor is split into a kinetic contribution $P^k$ and a virial contribution $P^v$. The sum gives the total stress tensor $P = P^k + P^v$. These computes can for example be used to calculate the diagonal components of the local stress tensor of surfaces with cylindrical or spherical symmetry. These computes obeys momentum balance through fluid interfaces. They use the Irving–Kirkwood contour, which is the straight line between particle pairs.

The compute *stress/cylinder* computes the stress profile along the radial direction in cylindrical coordinates, as described in *(Addington)*. The compute *stress/spherical* computes the stress profile along the radial direction in spherical coordinates, as described in *(Ikeshoji)*.

## 3.140.4 Output info

The default output columns for *stress/cylinder* are the radius to the center of the cylindrical shell, number density, $P^k_{rr}$, $P^k_{\phi\phi}$, $P^k_{zz}$, $P^v_{rr}$, $P^v_{\phi\phi}$, and $P^v_{zz}$. When the keyword *ke* is set to *no*, the kinetic contributions are not calculated, and consequently there are only 5 columns: the position of the center of the cylindrical shell, the number density, $P^v_{rr}$, $P^v_{\phi\phi}$, and $P^v_{zz}$. The number of bins (rows) is $R_{\max}/b$, where $b$ is the specified bin width.

The output columns for *stress/spherical* are the position of the center of the spherical shell, the number density, $P^k_{rr}$, $P^k_{\theta\theta}$, $P^k_{\phi\phi}$, $P^v_{rr}$, $P^v_{\theta\theta}$, and $P^v_{\phi\phi}$. There are 8 columns and the number of bins (rows) is $R_{\max}/b$, where $b$ is the specified bin width.

This array can be output with *fix ave/time*,

```
compute p all stress/spherical 0 0 0 0.1 10
fix 2 all ave/time 100 1 100 c_p[*] file dump_p.out mode vector
```

The values calculated by this compute are "intensive". The stress values will be in pressure *units*. The number density values are in inverse volume *units*.

NOTE 1: The local stress does not include any Lennard-Jones tail corrections to the stress added by the *pair_modify tail yes* command, since those are contributions to the global system pressure.

### 3.140.5 Restrictions

These computes calculate the stress tensor contributions for pair styles only (i.e., no bond, angle, dihedral, etc. contributions, and in the presence of bonded interactions, the result may be incorrect due to exclusions for *special bonds* excluding pairs of atoms completely). It requires pairwise force calculations not available for most many-body pair styles. Note that *k*-space calculations are also excluded.

These computes are part of the EXTRA-COMPUTE package. They are only enabled if LAMMPS was built with that package. See the *Build package* doc page for more info.

### 3.140.6 Related commands

*compute stress/atom*, *compute pressure*, *compute stress/mop/profile*, *compute stress/cartesian*

### 3.140.7 Default

The keyword default for ke in style *stress/cylinder* is yes.

---

**(Ikeshoji)** Ikeshoji, Hafskjold, Furuholt, Mol Sim, 29, 101-109, (2003).

**(Addington)** Addington, Long, Gubbins, J Chem Phys, 149, 084109 (2018).

## 3.141 compute stress/mop command

## 3.142 compute stress/mop/profile command

### 3.142.1 Syntax

```
compute ID group-ID style dir args keywords ...
```

- ID, group-ID are documented in *compute* command
- style = *stress/mop* or *stress/mop/profile*
- dir = *x* or *y* or *z* is the direction normal to the plane
- args = argument specific to the compute style
- keywords = *kin* or *conf* or *total* or *pair* or *bond* or *angle* or *dihedral* (one or more can be specified)

```
stress/mop args = pos
  pos = lower or center or upper or coordinate value (distance units) is the position of␣
↪the plane
stress/mop/profile args = origin delta
```

```
  origin = lower or center or upper or coordinate value (distance units) is the position␣
↪of the first plane
  delta = value (distance units) is the distance between planes
```

### 3.142.2 Examples

```
compute 1 all stress/mop x lower total
compute 1 liquid stress/mop z 0.0 kin conf
fix 1 all ave/time 10 1000 10000 c_1[*] file mop.time
fix 1 all ave/time 10 1000 10000 c_1[2] file mop.time

compute 1 all stress/mop/profile x lower 0.1 total
compute 1 liquid stress/mop/profile z 0.0 0.25 kin conf
fix 1 all ave/time 500 20 10000 c_1[*] ave running overwrite file mopp.time mode vector
```

### 3.142.3 Description

Compute *stress/mop* and compute *stress/mop/profile* calculate components of the local stress tensor using the method of planes *(Todd)*. Specifically, compute *stress/mop* calculates 3 components in directions *ix*, *iy*, and *iz* where *i* is the direction normal to the plane, while compute *stress/mop/profile* calculates the profile of the local stress along the *i* direction.

Contrary to methods based on histograms of atomic stress (i.e., using *compute stress/atom*), the method of planes is compatible with mechanical balance in heterogeneous systems and at interfaces *(Todd)*.

The stress tensor is the sum of a kinetic term and a configurational term, which are given respectively by Eq. (21) and Eq. (16) in *(Todd)*. For the kinetic part, the algorithm considers that atoms have crossed the plane if their positions at times $t - \Delta t$ and $t$ are one on either side of the plane, and uses the velocity at time $t - \Delta t/2$ given by the velocity Verlet algorithm.

Added in version 15Jun2023: contributions from bond, angle and dihedral potentials

Between one and seven keywords can be used to indicate which contributions to the stress must be computed: total stress (total), kinetic stress (kin), configurational stress (conf), stress due to bond stretching (bond), stress due to angle bending (angle), stress due to dihedral terms (dihedral) and/or due to pairwise non-bonded interactions (pair).

NOTE 1: The configurational stress is computed considering all pairs of atoms where at least one atom belongs to group group-ID.

NOTE 2: The local stress does not include any Lennard-Jones tail corrections to the stress added by the *pair_modify tail yes* command, since those are contributions to the global system pressure.

NOTE 3: The local stress profile generated by compute *stress/mop/profile* is similar to that obtained by compute *stress/cartesian*. A key difference is that compute *stress/mop/profile* considers particles crossing a set of planes, while *stress/cartesian* computes averages for a set of small volumes. Moreover, *stress/cartesian* compute computes the diagonal components of the stress tensor $P_{xx}$, $P_{yy}$, and $P_{zz}$, while *stress/mop/profile* computes the components $P_{ix}$, $P_{iy}$, and $P_{iz}$, where *i* is the direction normal to the plane.

### 3.142.4 Output info

Compute *stress/mop* calculates a global vector (indices starting at 1), with 3 values for each declared keyword (in the order the keywords have been declared). For each keyword, the stress tensor components are ordered as follows: $P_{ix}$, $P_{iy}$, and $P_{iz}$, where *i* is the direction normal to the plane.

Compute *stress/mop/profile* instead calculates a global array, with 1 column giving the position of the planes where the stress tensor was computed, and with 3 columns of values for each declared keyword (in the order the keywords have been declared). For each keyword, the profiles of stress tensor components are ordered as follows: $P_{ix}$, $P_{iy}$, and $P_{iz}$.

The values are in pressure *units*.

The values produced by this compute can be accessed by various *output commands*. For instance, the results can be written to a file using the *fix ave/time* command. Please see the example in the examples/PACKAGES/mop folder.

### 3.142.5 Restrictions

These styles are part of the EXTRA-COMPUTE package. They are only enabled if LAMMPS is built with that package. See the *Build package* doc page on for more info.

The method is implemented for orthogonal simulation boxes whose size does not change in time, and axis-aligned planes.

Compute *stress/mop* and *stress/mop/profile* do not work with manybody non-bonded interactions, long range (kspace) interactions and improper intramolecular interactions. The reason is that the current implementation requires the class method `Pair::single()` to be implemented, which is not possible for manybody potentials.

The impact of fixes that affect the stress (e.g. fix langevin) is also not included in the stress computed here.

### 3.142.6 Related commands

*compute stress/atom*, *compute pressure*, *compute stress/cartesian*, *compute stress/cylinder*, *compute stress/spherical*

### 3.142.7 Default

none

---

**(Todd)** B. D. Todd, Denis J. Evans, and Peter J. Daivis: "Pressure tensor for inhomogeneous fluids", Phys. Rev. E 52, 1627 (1995).

**(Ikeshoji)** Ikeshoji, Hafskjold, Furuholt, Mol Sim, 29, 101-109, (2003).

## 3.143 compute force/tally command

## 3.144 compute heat/flux/tally command

## 3.145 compute heat/flux/virial/tally command

## 3.146 compute pe/tally command

## 3.147 compute pe/mol/tally command

## 3.148 compute stress/tally command

### 3.148.1 Syntax

```
compute ID group-ID style group2-ID
```

- ID, group-ID are documented in *compute* command
- style = *force/tally* or *heat/flux/tally* or *heat/flux/virial/tally* or *pe/tally* or *pe/mol/tally* or *stress/tally*
- group2-ID = group ID of second (or same) group

### 3.148.2 Examples

```
compute 1 lower force/tally upper
compute 1 left pe/tally right
compute 1 lower stress/tally lower
compute 1 subregion heat/flux/tally all
compute 1 liquid heat/flux/virial/tally solid
```

### 3.148.3 Description

Define a computation that calculates properties between two groups of atoms by accumulating them from pairwise non-bonded computations. Except for *heat/flux/virial/tally*, the two groups can be the same. This is similar to *compute group/group* only that the data is accumulated directly during the non-bonded force computation. The computes *force/tally*, *pe/tally*, *stress/tally*, and *heat/flux/tally* are primarily provided as example how to program additional, more sophisticated computes using the tally callback mechanism. Compute *pe/mol/tally* is one such style, that can—through using this mechanism—separately tally intermolecular and intramolecular energies. Something that would otherwise be impossible without integrating this as a core functionality into the base classes of LAMMPS.

---

Compute *heat/flux/tally* obtains the heat flux (strictly speaking, heat flow) inside the first group, which is the sum of the convective contribution due to atoms in the first group and the virial contribution due to interaction between the first and second groups:

$$\mathbf{Q} = \sum_{i \in \text{group 1}} e_i \mathbf{v}_i + \frac{1}{2} \sum_{i \in \text{group 1}} \sum_{\substack{j \in \text{group 2} \\ j \neq i}} \left( \mathbf{F}_{ij} \cdot \mathbf{v}_j \right) \mathbf{r}_{ij}$$

When the second group in *heat/flux/tally* is set to "all", the resulting values will be identical to that obtained by *compute heat/flux*, provided only pairwise interactions exist.

Compute *heat/flux/virial/tally* obtains the total virial heat flux (strictly speaking, heat flow) into the first group due to interaction with the second group, and is defined as:

$$Q = \frac{1}{2} \sum_{i \in \text{group 1}} \sum_{j \in \text{group 2}} \mathbf{F}_{ij} \cdot \left( \mathbf{v}_i + \mathbf{v}_j \right)$$

Although, the *heat/flux/virial/tally* compute does not include the convective term, it can be used to obtain the total heat flux over control surfaces, when there are no particles crossing over, such as is often in solid–solid and solid–liquid interfaces. This would be identical to the method of planes method. Note that the *heat/flux/virial/tally* compute is distinctly different from the *heat/flux* and *heat/flux/tally* computes, that are essentially volume averaging methods. The following example demonstrates the difference:

```
# System with only pairwise interactions.
# Non-periodic boundaries in the x direction.
# Has LeftLiquid and RightWall groups along x direction.

# Heat flux over the solid-liquid interface
compute hflow_hfvt RightWall heat/flux/virial/tally LeftLiquid
variable hflux_hfvt equal c_hflow_hfvt/(ly*lz)

# x component of approximate heat flux vector inside the liquid region,
# two approaches.
#
compute myKE all ke/atom
compute myPE all pe/atom
```

(continues on next page)

(continued from previous page)

```
compute myStress all stress/atom NULL virial
compute hflow_hf LeftLiquid heat/flux myKE myPE myStress
variable hflux_hf equal c_hflow_hf[1]/${volLiq}
#
compute hflow_hft LeftLiquid heat/flux/tally all
variable hflux_hft equal c_hflow_hft[1]/${volLiq}

# Pressure over the solid-liquid interface, three approaches.
#
compute force_gg RightWall group/group LeftLiquid
variable press_gg equal c_force_gg[1]/(ly*lz)
#
compute force_ft RightWall force/tally LeftLiquid
compute rforce_ft RightWall reduce sum c_force_ft[1]
variable press_ft equal c_rforce_ft/(ly*lz)
#
compute rforce_hfvt all reduce sum c_hflow_hfvt[1]
variable press_hfvt equal c_rforce_hfvt/(ly*lz)
```

---

The pairwise contributions are computing via a callback that the compute registers with the non-bonded pairwise force computation. This limits the use to systems that have no bonds, no Kspace, and no many-body interactions. On the other hand, the computation does not have to compute forces or energies a second time and thus can be much more efficient. The callback mechanism allows to write more complex pairwise property computations.

---

### 3.148.4 Output info

- Compute *pe/tally* calculates a global scalar (the energy) and a per atom scalar (the contributions of the single atom to the global scalar).
- Compute *pe/mol/tally* calculates a global four-element vector containing (in this order): *evdwl* and *ecoul* for intramolecular pairs and *evdwl* and *ecoul* for intermolecular pairs. Since molecules are identified by their molecule IDs, the partitioning does not have to be related to molecules, but the energies are tallied into the respective slots depending on whether the molecule IDs of a pair are the same or different.
- Compute *force/tally* calculates a global scalar (the force magnitude) and a per atom 3-element vector (force contribution from each atom).
- Compute *stress/tally* calculates a global scalar (average of the diagonal elements of the stress tensor) and a per atom vector (the six elements of stress tensor contributions from the individual atom).
- As in *compute heat/flux*, compute *heat/flux/tally* calculates a global vector of length 6, where the first three components are the $x$, $y$, $z$ components of the full heat flow vector, and the next three components are the corresponding components of just the convective portion of the flow (i.e., the first term in the equation for $\mathbf{Q}$).
- Compute *heat/flux/virial/tally* calculates a global scalar (heat flow) and a per atom three-element vector (contribution to the force acting over atoms in the first group from individual atoms in both groups).

Both the scalar and vector values calculated by this compute are "extensive".

### 3.148.5 Restrictions

This compute is part of the TALLY package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Not all pair styles can be evaluated in a pairwise mode as required by this compute. For example, 3-body and other many-body potentials, such as *Tersoff* and *Stillinger-Weber* cannot be used. *EAM* potentials only include the pair potential portion of the EAM interaction when used by this compute, not the embedding term. Also bonded or Kspace interactions do not contribute to this compute.

These computes are not compatible with accelerated pair styles from the GPU, INTEL, KOKKOS, or OPENMP packages. They will either create an error or print a warning when required data was not tallied in the required way and thus the data acquisition functions from these computes not called.

When used with dynamic groups, a *run 0* command needs to be inserted in order to initialize the dynamic groups before accessing the computes.

### 3.148.6 Related commands

- *compute group/group*
- *compute heat/flux*

### 3.148.7 Default

none

## 3.149 compute tdpd/cc/atom command

### 3.149.1 Syntax

```
compute ID group-ID tdpd/cc/atom index
```

- ID, group-ID are documented in *compute* command
- tdpd/cc/atom = style name of this compute command
- index = index of chemical species (1 to Nspecies)

### 3.149.2 Examples

```
compute 1 all tdpd/cc/atom 2
```

### 3.149.3 Description

Define a computation that calculates the per-atom chemical concentration of a specified species for each tDPD particle in a group.

The chemical concentration of each species is defined as the number of molecules carried by a tDPD particle for dilute solution. For more details see *(Li2015)*.

### 3.149.4 Output info

This compute calculates a per-atom vector, which can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The per-atom vector values will be in the units of chemical species per unit mass.

### 3.149.5 Restrictions

This compute is part of the DPD-MESO package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.149.6 Related commands

*pair_style tdpd*

### 3.149.7 Default

none

---

**(Li2015)** Li, Yazdani, Tartakovsky, Karniadakis, J Chem Phys, 143: 014101 (2015). DOI: 10.1063/1.4923254

## 3.150 compute temp command

Accelerator Variants: *temp/kk*

### 3.150.1 Syntax

```
compute ID group-ID temp
```

- ID, group-ID are documented in *compute* command
- temp = style name of this compute command

### 3.150.2 Examples

```
compute 1 all temp
compute myTemp mobile temp
```

### 3.150.3 Description

Define a computation that calculates the temperature of a group of atoms. A compute of this style can be used by any command that computes a temperature, e.g. *thermo_modify*, *fix temp/rescale*, *fix npt*, etc.

The temperature is calculated by the formula

$$T = \frac{2E_{\mathrm{kin}}}{N_{\mathrm{DOF}}k_B} \quad \text{with} \quad E_{\mathrm{kin}} = \sum_{i=1}^{N_{\mathrm{atoms}}} \frac{1}{2} m_i v_i^2 \quad \text{and} \quad N_{\mathrm{DOF}} = n_{\mathrm{dim}} N_{\mathrm{atoms}} - n_{\mathrm{dim}} - N_{\mathrm{fixDOFs}}$$

where $E_{\mathrm{kin}}$ is the total kinetic energy of the group of atoms, $n_{\mathrm{dim}}$ is the dimensionality of the simulation (i.e. either 2 or 3), $N_{\mathrm{atoms}}$ is the number of atoms in the group, $N_{\mathrm{fixDOFs}}$ is the number of degrees of freedom removed by fix commands (see below), $k_B$ is the Boltzmann constant, and $T$ is the resulting computed temperature.

A symmetric tensor, stored as a six-element vector, is also calculated by this compute for use in the computation of a pressure tensor by the *compute pressue* command. The formula for the components of the tensor is the same as the above expression for $E_{\mathrm{kin}}$, except that the 1/2 factor is NOT included and the $v_i^2$ is replaced by $v_{i,x}v_{i,y}$ for the $xy$ component, and so on. Note that because it lacks the 1/2 factor, these tensor components are twice those of the traditional kinetic energy tensor. The six components of the vector are ordered $xx$, $yy$, $zz$, $xy$, $xz$, $yz$.

The number of atoms contributing to the temperature is assumed to be constant for the duration of the run; use the *dynamic* option of the *compute_modify* command if this is not the case.

This compute subtracts out degrees-of-freedom due to fixes that constrain molecular motion, such as *fix shake* and *fix rigid*. This means the temperature of groups of atoms that include these constraints will be computed correctly. If needed, the subtracted degrees-of-freedom can be altered using the *extra* option of the *compute_modify* command. By default this *extra* component is initialized to $n_{\mathrm{dim}}$ (as shown in the formula above) to represent the degrees of freedom removed from a system due to its translation invariance due to periodic boundary conditions.

A compute of this style with the ID of "thermo_temp" is created when LAMMPS starts up, as if this command were in the input script:

```
compute thermo_temp all temp
```

See the "thermo_style" command for more details.

See the *Howto thermostat* page for a discussion of different ways to compute temperature and perform thermostatting.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 3.150.4 Output info

This compute calculates a global scalar (the temperature) and a global vector of length six (symmetric tensor), which can be accessed by indices 1–6. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "intensive". The vector values are "extensive".

The scalar value is in temperature *units*. The vector values are in energy *units*.

### 3.150.5 Restrictions

none

### 3.150.6 Related commands

*compute temp/partial*, *compute temp/region*, *compute pressure*

### 3.150.7 Default

none

## 3.151 compute temp/asphere command

### 3.151.1 Syntax

```
compute ID group-ID temp/asphere keyword value ...
```

- ID, group-ID are documented in *compute* command
- temp/asphere = style name of this compute command
- zero or more keyword/value pairs may be appended
- keyword = *bias* or *dof*

```
bias value = bias-ID
  bias-ID = ID of a temperature compute that removes a velocity bias
dof value = all or rotate
  all = compute temperature of translational and rotational degrees of freedom
  rotate = compute temperature of just rotational degrees of freedom
```

### 3.151.2 Examples

```
compute 1 all temp/asphere
compute myTemp mobile temp/asphere bias tempCOM
compute myTemp mobile temp/asphere dof rotate
```

### 3.151.3 Description

Define a computation that calculates the temperature of a group of aspherical particles, including a contribution from both their translational and rotational kinetic energy. This differs from the usual *compute temp* command, which assumes point particles with only translational kinetic energy.

Only finite-size particles (aspherical or spherical) can be included in the group. For 3d finite-size particles, each has six degrees of freedom (three translational, three rotational). For 2d finite-size particles, each has three degrees of freedom (two translational, one rotational).

**Note**

This choice for degrees of freedom (DOF) assumes that all finite-size aspherical or spherical particles in your model will freely rotate, sampling all their rotational DOF. It is possible to use a combination of interaction potentials and fixes that induce no torque or otherwise constrain some of all of your particles so that this is not the case. Then there are fewer DOF and you should use the *compute_modify extra/dof* command to adjust the DOF accordingly.

For example, an aspherical particle with all three of its shape parameters the same is a sphere. If it does not rotate, then it should have 3 DOF instead of 6 in 3d (or two instead of three in 2d). A uniaxial aspherical particle has two of its three shape parameters the same. If it does not rotate around the axis perpendicular to its circular cross section, then it should have 5 DOF instead of 6 in 3d. The latter is the case for uniaxial ellipsoids in a *GayBerne model* since there is no induced torque around the optical axis. It will also be the case for biaxial ellipsoids when exactly two of the semiaxes have the same length and the corresponding relative well depths are equal.

The translational kinetic energy is computed the same as is described by the *compute temp* command. The rotational kinetic energy is computed as $\frac{1}{2} I \omega^2$, where $I$ is the inertia tensor for the aspherical particle and $\omega$ is its angular velocity, which is computed from its angular momentum.

**Note**

For *2d models*, particles are treated as ellipsoids, not ellipses, meaning their moments of inertia will be the same as in 3d.

A kinetic energy tensor, stored as a six-element vector, is also calculated by this compute. The formula for the components of the tensor is the same as the above formula, except that $v^2$ and $\omega^2$ are replaced by $v_x v_y$ and $\omega_x \omega_y$ for the $xy$ component, and the appropriate elements of the moment of inertia tensor are used. The six components of the vector are ordered $xx$, $yy$, $zz$, $xy$, $xz$, $yz$.

A symmetric tensor, stored as a six-element vector, is also calculated by this compute for use in the computation of a pressure tensor by the *compute pressue* command. The formula for the components of the tensor is the same as the above expression for $E_{\rm kin}$, except that the 1/2 factor is NOT included and the $v_i^2$ and $\omega^2$ are replaced by $v_x v_y$ and $\omega_x \omega_y$ for the $xy$ component, and so on. And the appropriate elements of the moment of inertia tensor are used. Note that because it lacks the 1/2 factor, these tensor components are twice those of the traditional kinetic energy tensor. The six components of the vector are ordered $xx$, $yy$, $zz$, $xy$, $xz$, $yz$.

The number of atoms contributing to the temperature is assumed to be constant for the duration of the run; use the *dynamic/dof* option of the *compute_modify* command if this is not the case.

This compute subtracts out translational degrees-of-freedom due to fixes that constrain molecular motion, such as *fix shake* and *fix rigid*. This means the temperature of groups of atoms that include these constraints will be computed correctly. If needed, the subtracted degrees-of-freedom can be altered using the *extra/dof* option of the *compute_modify* command.

See the *Howto thermostat* page for a discussion of different ways to compute temperature and perform thermostatting.

---

The keyword/value option pairs are used in the following ways.

For the *bias* keyword, *bias-ID* refers to the ID of a temperature compute that removes a "bias" velocity from each atom. This allows compute temp/sphere to compute its thermal temperature after the translational kinetic energy components have been altered in a prescribed way (e.g., to remove a flow velocity profile). Thermostats that use this compute will work with this bias term. See the doc pages for individual computes that calculate a temperature and the doc pages for fixes that perform thermostatting for more details.

For the *dof* keyword, a setting of *all* calculates a temperature that includes both translational and rotational degrees of freedom. A setting of *rotate* calculates a temperature that includes only rotational degrees of freedom.

---

### 3.151.4 Output info

This compute calculates a global scalar (the temperature) and a global vector of length 6 (symmetric tensor), which can be accessed by indices 1–6. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "intensive". The vector values are "extensive".

The scalar value is in temperature *units*. The vector values are in energy *units*.

### 3.151.5 Restrictions

This compute is part of the ASPHERE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This compute requires that atoms store angular momentum and a quaternion as defined by the *atom_style ellipsoid* command.

All particles in the group must be finite-size. They cannot be point particles, but they can be aspherical or spherical as defined by their shape attribute.

### 3.151.6 Related commands

*compute temp*

### 3.151.7 Default

none

## 3.152 compute temp/body command

### 3.152.1 Syntax

```
compute ID group-ID temp/body keyword value ...
```

- ID, group-ID are documented in *compute* command
- temp/body = style name of this compute command
- zero or more keyword/value pairs may be appended
- keyword = *bias* or *dof*

```
bias value = bias-ID
  bias-ID = ID of a temperature compute that removes a velocity bias
dof value = all or rotate
  all = compute temperature of translational and rotational degrees of freedom
  rotate = compute temperature of just rotational degrees of freedom
```

### 3.152.2 Examples

```
compute 1 all temp/body
compute myTemp mobile temp/body bias tempCOM
compute myTemp mobile temp/body dof rotate
```

### 3.152.3 Description

Define a computation that calculates the temperature of a group of body particles, including a contribution from both their translational and rotational kinetic energy. This differs from the usual *compute temp* command, which assumes point particles with only translational kinetic energy.

Only body particles can be included in the group. For 3d particles, each has 6 degrees of freedom (3 translational, 3 rotational). For 2d body particles, each has 3 degrees of freedom (2 translational, 1 rotational).

**Note**

This choice for degrees of freedom (DOF) assumes that all body particles in your model will freely rotate, sampling all their rotational DOF. It is possible to use a combination of interaction potentials and fixes that induce no torque or otherwise constrain some of all of your particles so that this is not the case. Then there are less DOF and you should use the *compute_modify extra/dof* command to adjust the DOF accordingly.

The translational kinetic energy is computed the same as is described by the *compute temp* command. The rotational kinetic energy is computed as $\frac{1}{2} I \omega^2$, where $I$ is the moment of inertia tensor for the aspherical particle and $\omega$ is its angular velocity, which is computed from its angular momentum.

A symmetric tensor, stored as a six-element vector, is also calculated by this compute for use in the computation of a pressure tensor by the *compute pressue* command. The formula for the components of the tensor is the same as the above expression for $E_{\rm kin}$, except that the 1/2 factor is NOT included and the $v_i^2$ and $\omega^2$ are replaced by $v_x v_y$ and $\omega_x \omega_y$ for the *xy* component, and so on. And the appropriate elements of the moment of inertia tensor are used. Note that because it lacks the 1/2 factor, these tensor components are twice those of the traditional kinetic energy tensor. The six components of the vector are ordered *xx*, *yy*, *zz*, *xy*, *xz*, *yz*.

The number of atoms contributing to the temperature is assumed to be constant for the duration of the run; use the *dynamic/dof* option of the *compute_modify* command if this is not the case.

This compute subtracts out translational degrees-of-freedom due to fixes that constrain molecular motion, such as *fix shake* and *fix rigid*. This means the temperature of groups of atoms that include these constraints will be computed correctly. If needed, the subtracted degrees-of-freedom can be altered using the *extra/dof* option of the *compute_modify* command.

See the *Howto thermostat* page for a discussion of different ways to compute temperature and perform thermostatting.

---

The keyword/value option pairs are used in the following ways.

For the *bias* keyword, *bias-ID* refers to the ID of a temperature compute that removes a "bias" velocity from each atom. This allows compute temp/sphere to compute its thermal temperature after the translational kinetic energy components have been altered in a prescribed way (e.g., to remove a flow velocity profile). Thermostats that use this compute will work with this bias term. See the doc pages for individual computes that calculate a temperature and the doc pages for fixes that perform thermostatting for more details.

For the *dof* keyword, a setting of *all* calculates a temperature that includes both translational and rotational degrees of freedom. A setting of *rotate* calculates a temperature that includes only rotational degrees of freedom.

---

### 3.152.4 Output info

This compute calculates a global scalar (the temperature) and a global vector of length 6 (symmetric tensor), which can be accessed by indices 1–6. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "intensive". The vector values are "extensive".

The scalar value is in temperature *units*. The vector values are in energy *units*.

### 3.152.5 Restrictions

This compute is part of the BODY package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This compute requires that atoms store angular momentum and a quaternion as defined by the *atom_style body* command.

### 3.152.6 Related commands

*compute temp*

### 3.152.7 Default

none

## 3.153 compute temp/chunk command

### 3.153.1 Syntax

```
compute ID group-ID temp/chunk chunkID value1 value2 ... keyword value ...
```

- ID, group-ID are documented in *compute* command
- temp/chunk = style name of this compute command
- chunkID = ID of *compute chunk/atom* command
- zero or more values can be listed as value1,value2,etc.
- value = *temp* or *kecom* or *internal*

```
temp = temperature of each chunk
kecom = kinetic energy of each chunk based on velocity of center of mass
internal = internal kinetic energy of each chunk
```

- zero or more keyword/value pairs may be appended
- keyword = *com* or *bias* or *adof* or *cdof*

```
com value = yes or no
  yes = subtract center-of-mass velocity from each chunk before calculating␣
→temperature
  no = do not subtract center-of-mass velocity
bias value = bias-ID
  bias-ID = ID of a temperature compute that removes a velocity bias
adof value = dof_per_atom
  dof_per_atom = define this many degrees-of-freedom per atom
cdof value = dof_per_chunk
  dof_per_chunk = define this many degrees-of-freedom per chunk
```

### 3.153.2 Examples

```
compute 1 fluid temp/chunk molchunk
compute 1 fluid temp/chunk molchunk temp internal
compute 1 fluid temp/chunk molchunk bias tpartial adof 2.0
```

### 3.153.3 Description

Define a computation that calculates the temperature of a group of atoms that are also in chunks, after optionally subtracting out the center-of-mass velocity of each chunk. By specifying optional values, it can also calculate the per-chunk temperature or energies of the multiple chunks of atoms.

In LAMMPS, chunks are collections of atoms defined by a *compute chunk/atom* command, which assigns each atom to a single chunk (or no chunk). The ID for this command is specified as chunkID. For example, a single chunk could be the atoms in a molecule or atoms in a spatial bin. See the *compute chunk/atom* and *Howto chunk* doc pages for details of how chunks can be defined and examples of how they can be used to measure properties of a system.

The temperature is calculated by the formula

$$\mathrm{KE} = \frac{\mathrm{DOF}}{2} k_B T,$$

where KE is the total kinetic energy of all atoms assigned to chunks (sum of $\frac{1}{2}mv^2$), DOF is the the total number of degrees of freedom for those atoms, $k_B$ is Boltzmann constant, and $T$ is the absolute temperature.

The DOF is calculated as $N \times adof + N_{\mathrm{chunk}} \times cdof$, where $N$ is the number of atoms contributing to the kinetic energy, *adof* is the number of degrees of freedom per atom, and *cdof* is the number of degrees of freedom per chunk. By default, *adof* = 2 or 3 = dimensionality of system, as set via the *dimension* command, and *cdof* = 0.0. This gives the usual formula for temperature.

A symmetric tensor, stored as a six-element vector, is also calculated by this compute. The formula for the components of the tensor is the same as the above expression for $E_{\mathrm{kin}}$, except that the 1/2 factor is NOT included and the $v_i^2$ is replaced by $v_{i,x}v_{i,y}$ for the *xy* component, and so on. Note that because it lacks the 1/2 factor, these tensor components are twice those of the traditional kinetic energy tensor. The six components of the vector are ordered *xx*, *yy*, *zz*, *xy*, *xz*, *yz*.

Note that the number of atoms contributing to the temperature is calculated each time the temperature is evaluated since it is assumed the atoms may be dynamically assigned to chunks. Thus there is no need to use the *dynamic* option of the *compute_modify* command for this compute style.

If any optional values are specified, then per-chunk quantities are also calculated and stored in a global array, as described below.

The *temp* value calculates the temperature for each chunk by the formula

$$\mathrm{KE} = \frac{\mathrm{DOF}}{2} k_B T,$$

where KE is the total kinetic energy of the chunk of atoms (sum of $\frac{1}{2}mv^2$), DOF is the total number of degrees of freedom for all atoms in the chunk, $k_B$ is the Boltzmann constant, and $T$ is the absolute temperature.

The number of degrees of freedom (DOF) in this case is calculated as $N \times adof + cdof$, where $N$ is the number of atoms in the chunk, *adof* is the number of degrees of freedom per atom, and *cdof* is the number of degrees of freedom per chunk. By default, *cdof* = 2 or 3 = dimensionality of system, as set via the *dimension* command, and *cdof* = 0.0. This gives the usual formula for temperature.

The *kecom* value calculates the kinetic energy of each chunk as if all its atoms were moving with the velocity of the center-of-mass of the chunk.

The *internal* value calculates the internal kinetic energy of each chunk. The interal KE is summed over the atoms in the chunk using an internal "thermal" velocity for each atom, which is its velocity minus the center-of-mass velocity of the chunk.

---

Note that currently the global and per-chunk temperatures calculated by this compute only include translational degrees of freedom for each atom. No rotational degrees of freedom are included for finite-size particles. Also no degrees of freedom are subtracted for any velocity bias or constraints that are applied, such as *compute temp/partial*, or *fix shake* or *fix rigid*. This is because those degrees of freedom (e.g., a constrained bond) could apply to sets of atoms that are both included and excluded from a specific chunk, and hence the concept is somewhat ill-defined. In some cases, you can use the *adof* and *cdof* keywords to adjust the calculated degrees of freedom appropriately, as explained below.

Note that the per-chunk temperature calculated by this compute and the *fix ave/chunk temp* command can be different. This compute calculates the temperature for each chunk for a single snapshot. Fix ave/chunk can do that but can also time average those values over many snapshots, or it can compute a temperature as if the atoms in the chunk on different timesteps were collected together as one set of atoms to calculate their temperature. This compute allows the center-of-mass velocity of each chunk to be subtracted before calculating the temperature; fix ave/chunk does not.

**Note**

Only atoms in the specified group contribute to the calculations performed by this compute. The *compute chunk/atom* command defines its own group; atoms will have a chunk ID = 0 if they are not in that group, signifying they are not assigned to a chunk, and will thus also not contribute to this calculation. You can specify the "all" group for this command if you simply want to include atoms with non-zero chunk IDs.

The simplest way to output the per-chunk results of the compute temp/chunk calculation to a file is to use the *fix ave/time* command, for example:

```
compute cc1 all chunk/atom molecule
compute myChunk all temp/chunk cc1 temp
fix 1 all ave/time 100 1 100 c_myChunk[1] file tmp.out mode vector
```

---

The keyword/value option pairs are used in the following ways.

The *com* keyword can be used with a value of *yes* to subtract the velocity of the center-of-mass (VCM) for each chunk from the velocity of the atoms in that chunk, before calculating either the global or per-chunk temperature. This can be useful if the atoms are streaming or otherwise moving collectively, and you wish to calculate only the thermal temperature. This per-chunk VCM bias can be used in other fixes and computes that can incorporate a temperature bias. If this compute is used as a temperature bias in other commands then this bias is subtracted from each atom, the command runs with the remaining thermal velocities, and then the bias is added back in. This includes thermostatting fixes like *fix nvt*, *fix temp/rescale*, *fix temp/berendsen*, and *fix langevin*, and computes like *compute stress/atom* and *compute pressure*. See the input script in examples/stress_vcm for an example of how to use the *com* keyword in conjunction with compute stress/atom to create a stress profile of a rigid body while removing the overall motion of the rigid body.

For the *bias* keyword, *bias-ID* refers to the ID of a temperature compute that removes a "bias" velocity from each atom. This also allows calculation of the global or per-chunk temperature using only the thermal temperature of atoms in each chunk after the translational kinetic energy components have been altered in a prescribed way (e.g., to remove a velocity profile). It also applies to the calculation of the other per-chunk values, such as *kecom* or *internal*, which involve the center-of-mass velocity of each chunk, which is calculated after the velocity bias is removed from each atom. Note that the temperature compute will apply its bias globally to the entire system, not on a per-chunk basis.

The *adof* and *cdof* keywords define the values used in the degree of freedom (DOF) formulas used for the global or per-chunk temperature, as described above. They can be used to calculate a more appropriate temperature for some kinds of chunks. Here are three examples:

If spatially binned chunks contain some number of water molecules and *fix shake* is used to make each molecule rigid, then you could calculate a temperature with six degrees of freedom (DOF) (three translational, three rotational) per molecule by setting *adof* to 2.0.

If *compute temp/partial* is used with the *bias* keyword to only allow the x component of velocity to contribute to the temperature, then *adof* = 1.0 would be appropriate.

If each chunk consists of a large molecule, with some number of its bonds constrained by *fix shake* or the entire molecule by *fix rigid/small*, *adof* = 0.0 and *cdof* could be set to the remaining degrees of freedom for the entire molecule (entire chunk in this case; i.e., 6 for 3d, or 3 for 2d, for a rigid molecule).

---

### 3.153.4 Output info

This compute calculates a global scalar (the temperature) and a global vector of length 6 (symmetric tensor), which can be accessed by indices 1–6. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

This compute also optionally calculates a global array, if one or more of the optional values are specified. The number of rows in the array is the number of chunks *Nchunk* as calculated by the specified *compute chunk/atom* command. The number of columns is the number of specified values (1 or more). These values can be accessed by any command that uses global array values from a compute as input. Again, see the *Howto output* doc page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "intensive". The vector values are "extensive". The array values are "intensive".

The scalar value is in temperature *units*. The vector values are in energy *units*. The array values will be in temperature *units* for the *temp* value, and in energy *units* for the *kecom* and *internal* values.

### 3.153.5 Restrictions

The *com* and *bias* keywords cannot be used together.

### 3.153.6 Related commands

*compute temp*, *fix ave/chunk temp*

### 3.153.7 Default

The option defaults are com no, no bias, adof = dimensionality of the system (2 or 3), and cdof = 0.0.

## 3.154 compute temp/com command

Accelerator Variants: *temp/com/kk*

### 3.154.1 Syntax

```
compute ID group-ID temp/com
```

- ID, group-ID are documented in *compute* command
- temp/com = style name of this compute command

### 3.154.2 Examples

```
compute 1 all temp/com
compute myTemp mobile temp/com
```

### 3.154.3 Description

Define a computation that calculates the temperature of a group of atoms, after subtracting out the center-of-mass velocity of the group. This is useful if the group is expected to have a non-zero net velocity for some reason. A compute of this style can be used by any command that computes a temperature, (e.g., *thermo_modify*, *fix temp/rescale*, *fix npt*).

After the center-of-mass velocity has been subtracted from each atom, the temperature is calculated by the formula

$$\mathrm{KE} = \frac{\dim}{2} N k_B T,$$

where KE is the total kinetic energy of the group of atoms (sum of $\frac{1}{2}mv^2$), dim = 2 or 3 is the dimensionality of the simulation, $N$ is number of atoms in the group, $k_B$ is the Boltzmann constant, and $T$ is the absolute temperature.

A symmetric tensor, stored as a six-element vector, is also calculated by this compute for use in the computation of a pressure tensor by the *compute pressue* command. The formula for the components of the tensor is the same as the above expression for $E_{\rm kin}$, except that the 1/2 factor is NOT included and the $v_i^2$ is replaced by $v_{i,x}v_{i,y}$ for the $xy$ component, and so on. Note that because it lacks the 1/2 factor, these tensor components are twice those of the traditional kinetic energy tensor. The six components of the vector are ordered $xx$, $yy$, $zz$, $xy$, $xz$, $yz$.

The number of atoms contributing to the temperature is assumed to be constant for the duration of the run; use the *dynamic* option of the *compute_modify* command if this is not the case.

The removal of the center-of-mass velocity by this fix is essentially computing the temperature after a "bias" has been removed from the velocity of the atoms. If this compute is used with a fix command that performs thermostatting then this bias will be subtracted from each atom, thermostatting of the remaining thermal velocity will be performed, and the bias will be added back in. Thermostatting fixes that work in this way include *fix nvt*, *fix temp/rescale*, *fix temp/berendsen*, and *fix langevin*.

This compute subtracts out degrees-of-freedom due to fixes that constrain molecular motion, such as *fix shake* and *fix rigid*. This means the temperature of groups of atoms that include these constraints will be computed correctly. If needed, the subtracted degrees-of-freedom can be altered using the *extra* option of the *compute_modify* command.

See the *Howto thermostat* page for a discussion of different ways to compute temperature and perform thermostatting.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 3.154.4 Output info

This compute calculates a global scalar (the temperature) and a global vector of length 6 (symmetric tensor), which can be accessed by indices 1–6. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "intensive". The vector values are "extensive".

The scalar value is in temperature *units*. The vector values is in energy *units*.

### 3.154.5 Restrictions

none

### 3.154.6 Related commands

*compute temp*

### 3.154.7 Default

none

## 3.155 compute temp/cs command

### 3.155.1 Syntax

```
compute ID group-ID temp/cs group1 group2
```

- ID, group-ID are documented in *compute* command
- temp/cs = style name of this compute command
- group1 = group-ID of either cores or shells
- group2 = group-ID of either shells or cores

### 3.155.2 Examples

```
compute oxygen_c-s all temp/cs O_core O_shell
compute core_shells all temp/cs cores shells
```

### 3.155.3 Description

Define a computation that calculates the temperature of a system based on the center-of-mass velocity of atom pairs that are bonded to each other. This compute is designed to be used with the adiabatic core/shell model of *(Mitchell and Fincham)*. See the *Howto coreshell* page for an overview of the model as implemented in LAMMPS. Specifically, this compute enables correct temperature calculation and thermostatting of core/shell pairs where it is desirable for the internal degrees of freedom of the core/shell pairs to not be influenced by a thermostat. A compute of this style can be used by any command that computes a temperature via *fix_modify* (e.g., *fix temp/rescale*, *fix npt*).

Note that this compute does not require all ions to be polarized, hence defined as core/shell pairs. One can mix core/shell pairs and ions without a satellite particle if desired. The compute will consider the non-polarized ions according to the physical system.

For this compute, core and shell particles are specified by two respective group IDs, which can be defined using the *group* command. The number of atoms in the two groups must be the same and there should be one bond defined between a pair of atoms in the two groups. Non-polarized ions which might also be included in the treated system should not be included into either of these groups, they are taken into account by the *group-ID* (second argument) of the compute.

The temperature is calculated by the formula

$$\mathrm{KE} = \frac{\mathrm{dim}}{2} N k_B T,$$

where KE is the total kinetic energy of the group of atoms (sum of $\frac{1}{2}mv^2$), dim = 2 or 3 is the dimensionality of the simulation, $N$ is the number of atoms in the group, $k_B$ is the Boltzmann constant, and $T$ is the absolute temperature. Note that the velocity of each core or shell atom used in the KE calculation is the velocity of the center-of-mass (COM) of the core/shell pair the atom is part of.

A symmetric tensor, stored as a six-element vector, is also calculated by this compute for use in the computation of a pressure tensor by the *compute pressue* command. The formula for the components of the tensor is the same as the above expression for $E_{\mathrm{kin}}$, except that the 1/2 factor is NOT included and the $v_i^2$ is replaced by $v_{i,x}v_{i,y}$ for the $xy$ component, and so on. Note that because it lacks the 1/2 factor, these tensor components are twice those of the traditional kinetic energy tensor. The six components of the vector are ordered $xx$, $yy$, $zz$, $xy$, $xz$, $yz$.

The change this fix makes to core/shell atom velocities is essentially computing the temperature after a "bias" has been removed from the velocity of the atoms. This "bias" is the velocity of the atom relative to the center-of-mass velocity of the core/shell pair. If this compute is used with a fix command that performs thermostatting then this bias will be subtracted from each atom, thermostatting of the remaining center-of-mass velocity will be performed, and the bias will be added back in. This means the thermostatting will effectively be performed on the core/shell pairs, instead of on the individual core and shell atoms. Thermostatting fixes that work in this way include *fix nvt*, *fix temp/rescale*, *fix temp/berendsen*, and *fix langevin*.

The internal energy of core/shell pairs can be calculated by the *compute temp/chunk* command, if chunks are defined as core/shell pairs. See the *Howto coreshell* doc page for more discussion on how to do this.

### 3.155.4 Output info

This compute calculates a global scalar (the temperature) and a global vector of length 6 (symmetric tensor), which can be accessed by indices 1–6. These values can be used by any command that uses global scalar or vector values from a compute as input.

The scalar value calculated by this compute is "intensive". The vector values are "extensive".

The scalar value is in temperature *units*. The vector values are in energy *units*.

### 3.155.5 Restrictions

The number of core/shell pairs contributing to the temperature is assumed to be constant for the duration of the run. No fixes should be used which generate new molecules or atoms during a simulation.

### 3.155.6 Related commands

*compute temp*, *compute temp/chunk*

### 3.155.7 Default

none

---

**(Mitchell and Fincham)** Mitchell, Fincham, J Phys Condensed Matter, 5, 1031-1038 (1993).

## 3.156 compute temp/deform command

Accelerator Variants: *temp/deform/kk*

### 3.156.1 Syntax

```
compute ID group-ID temp/deform keyword value ...
```

- ID, group-ID are documented in *compute* command
- temp/deform = style name of this compute command
- zero or more keyword/value pairs may be appended
- keyword = *temp*

  *temp* value = compute ID that calculates a temperature

### 3.156.2 Examples

```
compute myTemp all temp/deform
```

### 3.156.3 Description

Define a computation that calculates the temperature of a group of atoms, after subtracting out a streaming velocity induced by the simulation box changing size and/or shape, for example in a non-equilibrium MD (NEMD) simulation. The size/shape change is induced by use of the *fix deform* command. A compute of this style is created by the *fix nvt/sllod* command to compute the thermal temperature of atoms for thermostatting purposes. A compute of this style can also be used by any command that computes a temperature (e.g., *thermo_modify*, *fix temp/rescale*, *fix npt*).

The deformation fix changes the box size and/or shape over time, so each atom in the simulation box can be thought of as having a "streaming" velocity. For example, if the box is being sheared in $x$, relative to $y$, then atoms at the bottom of the box (low $y$) have a small $x$ velocity, while atoms at the top of the box (high $y$) have a large $x$ velocity. This position-dependent streaming velocity is subtracted from each atom's actual velocity to yield a thermal velocity, which is then used to compute the temperature.

Changed in version 11Feb2026.

The method for calculating temperature once the streaming velocity has been removed depends on the internal temperature compute. By default if the *temp* keyword is omitted, *compute temp* is used, and the internal temperature compute is created automatically as if the following command had been issued:

```
compute compute-ID_temp group-ID temp
```

The *temp* keyword allows an alternative internal temperature compute to be specified. For example, this can be used to exclude some directions with *compute temp/partial*, or account for additional bias beyond that due to deformation by using *compute temp/profile*. The group ID of compute temp/deform and the internal temperature compute must match. The internal temperature compute can be changed with the *compute_modify* command.

**Note**

If the internal temperature compute is evaluated directly, e.g. with "c_compute-ID_temp" or "c_compute-ID_temp[*]", then the streaming component of the velocity will NOT be removed for that calculation.

**Note**

*Fix deform* has an option for remapping either atom coordinates or velocities to the changing simulation box. When using this compute in conjunction with a deforming box, fix deform should NOT remap atom positions, but rather should let atoms respond to the changing box by adjusting their own velocities (or let *fix deform* remap the atom velocities; see its remap option). If fix deform does remap atom positions, then they appear to move with the box but their velocity is not changed, and thus they do NOT have the streaming velocity assumed by this compute. LAMMPS will warn you if fix deform is defined and its remap setting is not consistent with this compute. A similar situation arises if *fix nvt/sllod* is used with the "peculiar yes" option.

After the streaming velocity has been subtracted from each atom, the temperature is calculated as specified by the internal temperature compute. With *compute temp* as the internal temperature compute (the default), the temperature is calculated by the formula

$$\mathrm{KE} = \frac{\dim}{2} N k_B T,$$

where KE is the total kinetic energy of the group of atoms (sum of $\frac{1}{2}mv^2$, dim = 2 or 3 is the dimensionality of the simulation, $N$ is the number of atoms in the group, $k_B$ is the Boltzmann constant, and $T$ is the temperature. Note that $v$ in the kinetic energy formula is the atom's velocity with the streaming component removed.

A symmetric tensor, stored as a six-element vector, is also calculated by this compute for use in the computation of a pressure tensor by the *compute pressue* command. The formula for the components of the tensor is also dictated by the internal temperature compute. With *compute temp* it is the same as the above expression for $E_{\rm kin}$, except that the 1/2 factor is NOT included and the $v_i^2$ is replaced by $v_{i,x}v_{i,y}$ for the $xy$ component, and so on. Note that because it lacks the 1/2 factor, these tensor components are twice those of the traditional kinetic energy tensor. The six components of the vector are ordered $xx$, $yy$, $zz$, $xy$, $xz$, $yz$.

The number of atoms contributing to the temperature is assumed to be constant for the duration of the run; use the *dynamic/dof* option of the *compute_modify* command if this is not the case. This will also flag the internal temperature compute to treat the number of atoms as dynamic as if the following command had been issued:

```
compute_modify internal-tcompute-ID dynamic/dof value
```

**Warning**

Setting the dynamic/dof flag of the internal temperature compute but not compute temp/deform may lead to incorrect results.

The removal of the box deformation velocity component by this fix is essentially computing the temperature after a "bias" has been removed from the velocity of the atoms. If this compute is used with a fix command that performs thermostatting then this bias will be subtracted from each atom, thermostatting of the remaining thermal velocity will be performed, and the bias will be added back in. Thermostatting fixes that work in this way include *fix nvt*, *fix temp/rescale*, *fix temp/berendsen*, and *fix langevin*. If the internal temperature compute also subtracts a bias, then it will be calculated and subtracted after the box deformation component has been removed, and the velocity after both biases have been subtracted is the one to which the thermostat will be applied.

**Note**

With compute temp as the internal temperature compute, or other temperature computes which don't calculate an additional bias from the atom velocities, the temperature calculated by this compute is only accurate if the atoms are indeed moving with a stream velocity profile that matches the box deformation. If not, then the compute will subtract off an incorrect stream velocity, yielding a bogus thermal temperature. You should **not** assume that your atoms are streaming at the same rate the box is deforming. Rather, you should monitor their velocity profiles (e.g., via the *fix ave/chunk* command). You can also compare the results of this compute to *compute temp/profile*, which actually calculates the stream profile before subtracting it. If the two computes do not give roughly the same temperature, then your atoms are not streaming consistent with the box deformation. See the *fix deform* command for more details on ways to get atoms to stream consistently with the box deformation.

Subtracting out degrees-of-freedom due to fixes that constrain molecular motion, such as *fix shake* and *fix rigid* is handled by the internal temperature compute. This means the temperature of groups of atoms that include these constraints will be computed correctly. If needed, the subtracted degrees-of-freedom can be altered using the *extra/dof* option of the *compute_modify* command, which will have the same effect as if the following command were issued:

```
compute_modify internal-tcompute-ID extra/dof value
```

See the *Howto thermostat* page for a discussion of different ways to compute temperature and perform thermostatting.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix.

They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 3.156.4 Output info

This compute calculates a global scalar (the temperature) and a global vector of length 6 (symmetric tensor), which can be accessed by indices 1–6. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "intensive". The vector values are "extensive".

The scalar value is in temperature *units*. The vector values are in energy *units*.

### 3.156.5 Restrictions

none

### 3.156.6 Related commands

*compute temp/ramp*, *compute temp/profile*, *fix deform*, *fix nvt/sllod*

### 3.156.7 Default

none

## 3.157 compute temp/deform/eff command

### 3.157.1 Syntax

```
compute ID group-ID temp/deform/eff keyword value ...
```

- ID, group-ID are documented in *compute* command
- temp/deform/eff = style name of this compute command
- zero or more keyword/value pairs may be appended
- keyword = *temp*

  *temp* value = compute ID that calculates a temperature

### 3.157.2 Examples

```
compute myTemp all temp/deform/eff
```

### 3.157.3 Description

Define a computation that calculates the temperature of a group of nuclei and electrons in the *electron force field* model, after subtracting out a streaming velocity induced by the simulation box changing size and/or shape, for example in a non-equilibrium MD (NEMD) simulation. The size/shape change is induced by use of the *fix deform* command. A compute of this style is created by the *fix nvt/sllod/eff* command to compute the thermal temperature of atoms for thermostatting purposes. A compute of this style can also be used by any command that computes a temperature (e.g., *thermo_modify*, *fix npt/eff*).

The calculation performed by this compute is exactly like that described by the *compute temp/deform* command, except that the formulas for the temperature (scalar) and diagonal components of the symmetric tensor (vector) include the radial electron velocity contributions, as discussed by the *compute temp/eff* command. Note that only the translational degrees of freedom for each nuclei or electron are affected by the streaming velocity adjustment. The radial velocity component of the electrons is not affected.

Changed in version 11Feb2026.

By default, the internal temperature compute has the style *compute temp/eff*. If an internal temperature compute is used which does not have the /eff suffix, the contribution to the scalar and vector values due to the radial electron velocity will be added in by this compute.

### 3.157.4 Output info

This compute calculates a global scalar (the temperature) and a global vector of length 6 (symmetric tensor), which can be accessed by indices 1–6. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "intensive". The vector values are "extensive".

The scalar value is in temperature *units*. The vector values are in energy *units*.

### 3.157.5 Restrictions

This compute is part of the EFF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.157.6 Related commands

*compute temp/ramp*, *fix deform*, *fix nvt/sllod/eff*

### 3.157.7 Default

none

## 3.158 compute temp/drude command

### 3.158.1 Syntax

```
compute ID group-ID temp/drude
```

- ID, group-ID are documented in *compute* command
- temp/drude = style name of this compute command

### 3.158.2 Examples

```
compute TDRUDE all temp/drude
```

Example input scripts available: `examples/PACKAGES/drude`.

### 3.158.3 Description

Define a computation that calculates the temperatures of core–Drude pairs. This compute is designed to be used with the *thermalized Drude oscillator model*. Polarizable models in LAMMPS are described on the *Howto polarizable* doc page.

Drude oscillators consist of a core particle and a Drude particle connected by a harmonic bond, and the relative motion of these Drude oscillators is usually maintained cold by a specific thermostat that acts on the relative motion of the core–Drude particle pairs. Therefore, because LAMMPS considers Drude particles as normal atoms in its default temperature compute (*compute temp* command), the reduced temperature of the core–Drude particle pairs is not calculated correctly.

By contrast, this compute calculates the temperature of the cores using center-of-mass velocities of the core–Drude pairs, and the reduced temperature of the Drude particles using the relative velocities of the Drude particles with respect to their cores. Non-polarizable atoms are considered as cores. Their velocities contribute to the temperature of the cores.

### 3.158.4 Output info

This compute calculates a global scalar (the temperature) and a global vector of length 6, which can be accessed by indices 1–6, whose components are

1. temperature of the centers of mass (temperature units)
2. temperature of the dipoles (temperature units)
3. number of degrees of freedom of the centers of mass
4. number of degrees of freedom of the dipoles
5. kinetic energy of the centers of mass (energy units)
6. kinetic energy of the dipoles (energy units)

These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

Both the scalar value and the first two values of the vector calculated by this compute are “intensive”. The other four vector values are “extensive”.

### 3.158.5 Restrictions

The number of degrees of freedom contributing to the temperature is assumed to be constant for the duration of the run unless the *fix_modify command* sets the option *dynamic/dof yes*.

### 3.158.6 Related commands

*fix drude*, *fix langevin/drude*, *fix drude/transform*, *pair_style thole*, *compute temp*

### 3.158.7 Default

none

## 3.159 compute temp/eff command

### 3.159.1 Syntax

```
compute ID group-ID temp/eff
```

- ID, group-ID are documented in *compute* command
- temp/eff = style name of this compute command

### 3.159.2 Examples

```
compute 1 all temp/eff
compute myTemp mobile temp/eff
```

### 3.159.3 Description

Define a computation that calculates the temperature of a group of nuclei and electrons in the *electron force field* model. A compute of this style can be used by commands that compute a temperature (e.g., *thermo_modify*, *fix npt/eff*).

The temperature is calculated by the formula

$$\mathrm{KE} = \frac{\mathrm{dim}}{2} N k_B T,$$

where KE is the total kinetic energy of the group of atoms (sum of $\frac{1}{2}mv^2$ for nuclei and sum of $\frac{1}{2}(mv^2 + \frac{3}{4}ms^2)$ for electrons, where $s$ includes the radial electron velocity contributions), dim = 2 or 3 is the dimensionality of the simulation, $N$ is the number of atoms (only total number of nuclei in the eFF (see the *pair_eff* command) in the group, $k_B$ is the Boltzmann constant, and $T$ is the absolute temperature. This expression is summed over all nuclear and electronic degrees of freedom, essentially by setting the kinetic contribution to the heat capacity to $\frac{3}{2}k$ (where only nuclei contribute). This subtlety is valid for temperatures well below the Fermi temperature, which for densities two to five times the density of liquid hydrogen ranges from 86,000 to 170,000 K.

**Note**

For eFF models, in order to override the default temperature reported by LAMMPS in the thermodynamic quantities reported via the *thermo* command, the user should apply a *thermo_modify* command, as shown in the following example:

```
compute         effTemp all temp/eff
thermo_style    custom step etotal pe ke temp press
thermo_modify   temp effTemp
```

A six-component kinetic energy tensor is also calculated by this compute for use in the computation of a pressure tensor. The formula for the components of the tensor is the same as the above formula, except that $v^2$ is replaced by $v_x v_y$ for the $xy$ component, etc. For the eFF, again, the radial electronic velocities are also considered.

The number of atoms contributing to the temperature is assumed to be constant for the duration of the run; use the *dynamic* option of the *compute_modify* command if this is not the case.

This compute subtracts out degrees-of-freedom due to fixes that constrain molecular motion, such as *fix shake* and *fix rigid*. This means the temperature of groups of atoms that include these constraints will be computed correctly. If needed, the subtracted degrees-of-freedom can be altered using the *extra* option of the *compute_modify* command.

See the *Howto thermostat* page for a discussion of different ways to compute temperature and perform thermostatting.

### 3.159.4 Output info

The scalar value calculated by this compute is "intensive", meaning it is independent of the number of atoms in the simulation. The vector values are "extensive", meaning they scale with the number of atoms in the simulation.

### 3.159.5 Restrictions

This compute is part of the EFF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.159.6 Related commands

*compute temp/partial*, *compute temp/region*, *compute pressure*

### 3.159.7 Default

none

## 3.160 compute temp/partial command

### 3.160.1 Syntax

```
compute ID group-ID temp/partial xflag yflag zflag
```

- ID, group-ID are documented in *compute* command
- temp/partial = style name of this compute command
- xflag,yflag,zflag = 0/1 for whether to exclude/include this dimension

### 3.160.2 Examples

```
compute newT flow temp/partial 1 1 0
```

### 3.160.3 Description

Define a computation that calculates the temperature of a group of atoms, after excluding one or more velocity components. A compute of this style can be used by any command that computes a temperature (e.g. *thermo_modify*, *fix temp/rescale*, *fix npt*).

The temperature is calculated by the formula

$$\mathrm{KE} = \frac{\mathrm{dim}}{2} N k_B T,$$

where KE is the total kinetic energy of the group of atoms (sum of $\frac{1}{2}mv^2$), dim = 2 or 3 is the dimensionality of the simulation, $N$ is the number of atoms in the group, $k_B$ is the Boltzmann constant, and $T$ = temperature. The calculation of KE excludes the $x$, $y$, or $z$ dimensions if *xflag*, *yflag*, or *zflag* is 0. The dim parameter is adjusted to give the correct number of degrees of freedom.

A symmetric tensor, stored as a six-element vector, is also calculated by this compute for use in the computation of a pressure tensor by the *compute pressure* command. The formula for the components of the tensor is the same as the above expression for $E_{\rm kin}$, except that the 1/2 factor is NOT included and the $v_i^2$ is replaced by $v_{i,x}v_{i,y}$ for the $xy$ component, and so on. Note that because it lacks the 1/2 factor, these tensor components are twice those of the traditional kinetic energy tensor. The six components of the vector are ordered $xx$, $yy$, $zz$, $xy$, $xz$, $yz$.

The number of atoms contributing to the temperature is assumed to be constant for the duration of the run; use the *dynamic* option of the *compute_modify* command if this is not the case.

The removal of velocity components by this fix is essentially computing the temperature after a "bias" has been removed from the velocity of the atoms. If this compute is used with a fix command that performs thermostatting then this bias will be subtracted from each atom, thermostatting of the remaining thermal velocity will be performed, and the bias will be added back in. Thermostatting fixes that work in this way include *fix nvt*, *fix temp/rescale*, *fix temp/berendsen*, and *fix langevin*.

This compute subtracts out degrees-of-freedom due to fixes that constrain molecular motion, such as *fix shake* and *fix rigid*. This means the temperature of groups of atoms that include these constraints will be computed correctly. If needed, the subtracted degrees-of-freedom can be altered using the *extra* option of the *compute_modify* command.

See the *Howto thermostat* page for a discussion of different ways to compute temperature and perform thermostatting.

### 3.160.4 Output info

This compute calculates a global scalar (the temperature) and a global vector of length 6 (symmetric tensor), which can be accessed by indices 1–6. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "intensive". The vector values are "extensive".

The scalar value is in temperature *units*. The vector values are in energy *units*.

### 3.160.5 Restrictions

none

### 3.160.6 Related commands

*compute temp*, *compute temp/region*, *compute pressure*

### 3.160.7 Default

none

## 3.161 compute temp/profile command

### 3.161.1 Syntax

```
compute ID group-ID temp/profile xflag yflag zflag binstyle args
```

- ID, group-ID are documented in *compute* command
- temp/profile = style name of this compute command
- xflag,yflag,zflag = 0/1 for whether to exclude/include this dimension
- binstyle = $x$ or $y$ or $z$ or $xy$ or $yz$ or $xz$ or $xyz$

```
x arg = Nx
y arg = Ny
z arg = Nz
xy args = Nx Ny
yz args = Ny Nz
xz args = Nx Nz
xyz args = Nx Ny Nz
  Nx, Ny, Nz = number of velocity bins in x, y, z dimensions
```

- zero or more keyword/value pairs may be appended
- keyword = *out*

```
out value = tensor or bin
```

### 3.161.2 Examples

```
compute myTemp flow temp/profile 1 1 1 x 10
compute myTemp flow temp/profile 1 1 1 x 10 out bin
compute myTemp flow temp/profile 0 1 1 xyz 20 20 20
```

### 3.161.3 Description

Define a computation that calculates the temperature of a group of atoms, after subtracting out a spatially-averaged center-of-mass velocity field, before computing the kinetic energy. This can be useful for thermostatting a collection of atoms undergoing a complex flow (e.g. via a profile-unbiased thermostat (PUT) as described in *(Evans)*). A compute of this style can be used by any command that computes a temperature (e.g. *thermo_modify*, *fix temp/rescale*, *fix npt*).

The *xflag*, *yflag*, *zflag* settings determine which components of average velocity are subtracted out.

The *binstyle* setting and its *Nx*, *Ny*, *Nz* arguments determine how bins are setup to perform spatial averaging. "Bins" can be 1d slabs, 2d pencils, or 3d bricks depending on which *binstyle* is used. The simulation box is partitioned conceptually into $Nx \times Ny \times Nz$ bins. Depending on the *binstyle*, you may only specify one or two of these values; the others are effectively set to 1 (no binning in that dimension). For non-orthogonal (triclinic) simulation boxes, the bins are "tilted" slabs or pencils or bricks that are parallel to the tilted faces of the box. See the *region prism* command for a discussion of the geometry of tilted boxes in LAMMPS.

When a temperature is computed, the center-of-mass velocity for the set of atoms that are both in the compute group and in the same spatial bin is calculated. This bias velocity is then subtracted from the velocities of individual atoms in the bin to yield a thermal velocity for each atom. Note that if there is only one atom in the bin, its thermal velocity will thus be 0.0.

After the spatially-averaged velocity field has been subtracted from each atom, the temperature is calculated by the formula

$$\mathrm{KE} = \left( \frac{\mathrm{dim}}{N} - N_s N_x N_y N_z - \mathrm{extra} \right) \frac{k_B T}{2},$$

where KE is the total kinetic energy of the group of atoms (sum of $\frac{1}{2}mv^2$; dim = 2 or 3 is the dimensionality of the simulation; $N_s$ = 0, 1, 2, or 3 for streaming velocity subtracted in 0, 1, 2, or 3 dimensions, respectively; *extra* is the number of extra degrees of freedom; $N$ is the number of atoms in the group; $k_B$ is the Boltzmann constant, and $T$ is the absolute temperature. The $N_s N_x N_y N_z$ term is the number of degrees of freedom subtracted to adjust for the removal of the center-of-mass velocity in each direction of the *Nx*Ny*Nz* bins, as discussed in the *(Evans)* paper. The extra term defaults to dim $-$ $N_s$ and accounts for overall conservation of center-of-mass velocity across the group in directions where streaming velocity is *not* subtracted. This can be altered using the *extra* option of the *compute_modify* command.

If the *out* keyword is used with a *tensor* value, which is the default, then a symmetric tensor, stored as a six-element vector, is also calculated by this compute for use in the computation of a pressure tensor by the *compute pressue*

command. The formula for the components of the tensor is the same as the above expression for $E_{\rm kin}$, except that the 1/2 factor is NOT included and the $v_i^2$ is replaced by $v_{i,x}v_{i,y}$ for the *xy* component, and so on. Note that because it lacks the 1/2 factor, these tensor components are twice those of the traditional kinetic energy tensor. The six components of the vector are ordered *xx*, *yy*, *zz*, *xy*, *xz*, *yz*.

If the *out* keyword is used with a *bin* value, the count of atoms and computed temperature for each bin are stored for output, as an array of values, as described below. The temperature of each bin is calculated as described above, where the bias velocity is subtracted and only the remaining thermal velocity of atoms in the bin contributes to the temperature. See the note below for how the temperature is normalized by the degrees-of-freedom of atoms in the bin.

The number of atoms contributing to the temperature is assumed to be constant for the duration of the run; use the *dynamic* option of the *compute_modify* command if this is not the case.

The removal of the spatially-averaged velocity field by this fix is essentially computing the temperature after a "bias" has been removed from the velocity of the atoms. If this compute is used with a fix command that performs thermostatting then this bias will be subtracted from each atom, thermostatting of the remaining thermal velocity will be performed, and the bias will be added back in. Thermostatting fixes that work in this way include *fix nvt*, *fix temp/rescale*, *fix temp/berendsen*, and *fix langevin*.

This compute subtracts out degrees-of-freedom due to fixes that constrain molecular motion, such as *fix shake* and *fix rigid*. This means the temperature of groups of atoms that include these constraints will be computed correctly. If needed, the subtracted degrees-of-freedom can be altered using the *extra* option of the *compute_modify* command.

**Note**

When using the *out* keyword with a value of *bin*, the calculated temperature for each bin includes the degrees-of-freedom adjustment described in the preceding paragraph for fixes that constrain molecular motion, as well as the adjustment due to the *extra* option (which defaults to *dim* - *Ns* as described above), by fractionally applying them based on the fraction of atoms in each bin. As a result, the bin degrees-of-freedom summed over all bins exactly equals the degrees-of-freedom used in the scalar temperature calculation, $\Sigma N_{{\rm DOF}_i} = N_{\rm DOF}$ and the corresponding relation for temperature is also satisfied ($\Sigma N_{{\rm DOF}_i} T_i = N_{\rm DOF} T$). These relations will break down in cases for which the adjustment exceeds the actual number of degrees of freedom in a bin. This could happen if a bin is empty or in situations in which rigid molecules are non-uniformly distributed, in which case the reported temperature within a bin may not be accurate.

See the *Howto thermostat* page for a discussion of different ways to compute temperature and perform thermostatting. Using this compute in conjunction with a thermostatting fix, as explained there, will effectively implement a profile-unbiased thermostat (PUT), as described in *(Evans)*.

### 3.161.4 Output info

This compute calculates a global scalar (the temperature). Depending on the setting of the *out* keyword, it also calculates a global vector or array. For *out* = *tensor*, it calculates a vector of length 6 (symmetric tensor), which can be accessed by indices 1–6. For *out* = *bin* it calculates a global array which has 2 columns and $N$ rows, where $N$ is the number of bins. The first column contains the number of atoms in that bin. The second contains the temperature of that bin, calculated as described above. The ordering of rows in the array is as follows. Bins in $x$ vary fastest, then $y$, then $z$. Thus for a $10 \times 10 \times 10$ 3d array of bins, there will be 1000 rows. The bin with indices $(i_x, i_y, i_z) = (2, 3, 4)$ would map to row $M = 10^2(i_z - 1) + 10(i_y - 1) + i_x = 322$, where the rows are numbered from 1 to 1000 and the bin indices are numbered from 1 to 10 in each dimension.

These values can be used by any command that uses global scalar or vector or array values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "intensive". The vector values are "extensive". The array values are "intensive".

The scalar value us in temperature *units*. The vector values are in energy *units*. The first column of array values are counts; the values in the second column will be in temperature *units*.

### 3.161.5 Restrictions

You should not use too large a velocity-binning grid, especially in 3d. In the current implementation, the binned velocity averages are summed across all processors, so this will be inefficient if the grid is too large, and the operation is performed every timestep, as it will be for most thermostats.

### 3.161.6 Related commands

*compute temp*, *compute temp/ramp*, *compute temp/deform*, *compute pressure*

### 3.161.7 Default

The option default is out = tensor.

---

**(Evans)** Evans and Morriss, Phys Rev Lett, 56, 2172-2175 (1986).

## 3.162 compute temp/ramp command

### 3.162.1 Syntax

```
compute ID group-ID temp/ramp vdim vlo vhi dim clo chi keyword value ...
```

- ID, group-ID are documented in *compute* command
- temp/ramp = style name of this compute command
- vdim = *vx* or *vy* or *vz*
- vlo,vhi = subtract velocities between vlo and vhi (velocity units)
- dim = *x* or *y* or *z*
- clo,chi = lower and upper bound of domain to subtract from (distance units)
- zero or more keyword/value pairs may be appended
- keyword = *units*

*units* value = *lattice* or *box*

### 3.162.2 Examples

```
compute 2nd middle temp/ramp vx 0 8 y 2 12 units lattice
```

### 3.162.3 Description

Define a computation that calculates the temperature of a group of atoms, after subtracting out an ramped velocity profile before computing the kinetic energy. A compute of this style can be used by any command that computes a temperature (e.g. *thermo_modify*, *fix temp/rescale*, *fix npt*).

The meaning of the arguments for this command which define the velocity ramp are the same as for the *velocity ramp* command which was presumably used to impose the velocity.

After the ramp velocity has been subtracted from the specified dimension for each atom, the temperature is calculated by the formula

$$\mathrm{KE} = \frac{\mathrm{dim}}{2} N k_B T,$$

where KE is the total kinetic energy of the group of atoms (sum of $\frac{1}{2}mv^2$), dim = 2 or 3 is the dimensionality of the simulation, $N$ is the number of atoms in the group, $k_B$ is the Boltzmann constant, and $T$ is the absolute temperature.

The *units* keyword determines the meaning of the distance units used for coordinates (*clo*, *chi*) and velocities (*vlo*, *vhi*). A *box* value selects standard distance units as defined by the *units* command (e.g., Å for units = real or metal). A *lattice* value means the distance units are in lattice spacings (i.e., velocity in lattice spacings per unit time). The *lattice* command must have been previously used to define the lattice spacing.

A symmetric tensor, stored as a six-element vector, is also calculated by this compute for use in the computation of a pressure tensor by the *compute pressue* command. The formula for the components of the tensor is the same as the above expression for $E_{\mathrm{kin}}$, except that the 1/2 factor is NOT included and the $v_i^2$ is replaced by $v_{i,x}v_{i,y}$ for the *xy* component, and so on. Note that because it lacks the 1/2 factor, these tensor components are twice those of the traditional kinetic energy tensor. The six components of the vector are ordered *xx*, *yy*, *zz*, *xy*, *xz*, *yz*.

The number of atoms contributing to the temperature is assumed to be constant for the duration of the run; use the *dynamic* option of the *compute_modify* command if this is not the case.

The removal of the ramped velocity component by this fix is essentially computing the temperature after a "bias" has been removed from the velocity of the atoms. If this compute is used with a fix command that performs thermostatting then this bias will be subtracted from each atom, thermostatting of the remaining thermal velocity will be performed, and the bias will be added back in. Thermostatting fixes that work in this way include *fix nvt*, *fix temp/rescale*, *fix temp/berendsen*, and *fix langevin*.

This compute subtracts out degrees-of-freedom due to fixes that constrain molecular motion, such as *fix shake* and *fix rigid*. This means the temperature of groups of atoms that include these constraints will be computed correctly. If needed, the subtracted degrees-of-freedom can be altered using the *extra* option of the *compute_modify* command.

See the *Howto thermostat* page for a discussion of different ways to compute temperature and perform thermostatting.

### 3.162.4 Output info

This compute calculates a global scalar (the temperature) and a global vector of length 6 (symmetric tensor), which can be accessed by indices 1–6. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "intensive". The vector values are "extensive".

The scalar value is in temperature *units*. The vector values are in energy *units*.

### 3.162.5 Restrictions

none

### 3.162.6 Related commands

*compute temp*, *compute temp/profile*, *compute temp/deform*, *compute pressure*

### 3.162.7 Default

The option default is units = lattice.

# 3.163 compute temp/region command

## 3.163.1 Syntax

```
compute ID group-ID temp/region region-ID
```

- ID, group-ID are documented in *compute* command
- temp/region = style name of this compute command
- region-ID = ID of region to use for choosing atoms

## 3.163.2 Examples

```
compute mine flow temp/region boundary
```

## 3.163.3 Description

Define a computation that calculates the temperature of a group of atoms in a geometric region. This can be useful for thermostatting one portion of the simulation box. For example, a McDLT simulation where one side is cooled, and the other side is heated. A compute of this style can be used by any command that computes a temperature (e.g., *thermo_modify*, *fix temp/rescale*).

Note that a *region*-style temperature can be used to thermostat with *fix temp/rescale* or *fix langevin*, but should probably not be used with Nose–Hoover style fixes (*fix nvt*, *fix npt*, or *fix nph*) if the degrees of freedom included in the computed temperature vary with time.

The temperature is calculated by the formula

$$\mathrm{KE} = \frac{\mathrm{dim}}{2} N k_B T,$$

where KE = is the total kinetic energy of the group of atoms (sum of $\frac{1}{2}mv^2$), dim = 2 or 3 is the dimensionality of the simulation, $N$ is the number of atoms in both the group and region, $k_B$ is the Boltzmann constant, and $T$ temperature.

A symmetric tensor, stored as a six-element vector, is also calculated by this compute for use in the computation of a pressure tensor by the *compute pressue* command. The formula for the components of the tensor is the same as the above expression for $E_{\mathrm{kin}}$, except that the 1/2 factor is NOT included and the $v_i^2$ is replaced by $v_{i,x}v_{i,y}$ for the $xy$ component, and so on. Note that because it lacks the 1/2 factor, these tensor components are twice those of the traditional kinetic energy tensor. The six components of the vector are ordered $xx$, $yy$, $zz$, $xy$, $xz$, $yz$.

The number of atoms contributing to the temperature is calculated each time the temperature is evaluated since it is assumed atoms can enter/leave the region. Thus there is no need to use the *dynamic* option of the *compute_modify* command for this compute style.

The removal of atoms outside the region by this fix is essentially computing the temperature after a "bias" has been removed, which in this case is the velocity of any atoms outside the region. If this compute is used with a fix command that performs thermostatting then this bias will be subtracted from each atom, thermostatting of the remaining thermal velocity will be performed, and the bias will be added back in. Thermostatting fixes that work in this way include *fix nvt*, *fix temp/rescale*, *fix temp/berendsen*, and *fix langevin*. This means that when this compute is used to calculate the temperature for any of the thermostatting fixes via the *fix modify temp* command, the thermostat will operate only on atoms that are currently in the geometric region.

Unlike other compute styles that calculate temperature, this compute does not subtract out degrees-of-freedom due to fixes that constrain motion, such as *fix shake* and *fix rigid*. This is because those degrees of freedom (e.g., a constrained bond) could apply to sets of atoms that straddle the region boundary, and hence the concept is somewhat ill-defined. If needed the number of subtracted degrees of freedom can be set explicitly using the *extra* option of the *compute_modify* command.

See the *Howto thermostat* page for a discussion of different ways to compute temperature and perform thermostatting.

### 3.163.4 Output info

This compute calculates a global scalar (the temperature) and a global vector of length 6 (symmetric tensor), which can be accessed by indices 1–6. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "intensive". The vector values are "extensive".

The scalar value is in temperature *units*. The vector values are in energy *units*.

### 3.163.5 Restrictions

none

### 3.163.6 Related commands

*compute temp*, *compute pressure*

### 3.163.7 Default

none

## 3.164 compute temp/region/eff command

### 3.164.1 Syntax

```
compute ID group-ID temp/region/eff region-ID
```

- ID, group-ID are documented in *compute* command
- temp/region/eff = style name of this compute command
- region-ID = ID of region to use for choosing atoms

### 3.164.2 Examples

```
compute mine flow temp/region/eff boundary
```

### 3.164.3 Description

Define a computation that calculates the temperature of a group of nuclei and electrons in the *electron force field* model, within a geometric region using the electron force field. A compute of this style can be used by commands that compute a temperature (e.g., *thermo_modify*).

The operation of this compute is exactly like that described by the *compute temp/region* command, except that the formulas for the temperature (scalar) and diagonal components of the symmetric tensor (vector) include the radial electron velocity contributions, as discussed by the *compute temp/eff* command.

### 3.164.4 Output info

This compute calculates a global scalar (the temperature) and a global vector of length 6 (symmetric tensor), which can be accessed by indices 1–6. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "intensive". The vector values are "extensive".

The scalar value is in temperature *units*. The vector values are in energy *units*.

### 3.164.5 Restrictions

This compute is part of the EFF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.164.6 Related commands

*compute temp/region*, *compute temp/eff* , *compute pressure*

### 3.164.7 Default

none

## 3.165 compute temp/rotate command

### 3.165.1 Syntax

```
compute ID group-ID temp/rotate
```

- ID, group-ID are documented in *compute* command
- temp/rotate = style name of this compute command

### 3.165.2 Examples

```
compute Tbead bead temp/rotate
```

### 3.165.3 Description

Define a computation that calculates the temperature of a group of atoms, after subtracting out the center-of-mass velocity and angular velocity of the group. This is useful if the group is expected to have a non-zero net velocity and/or global rotation motion for some reason. A compute of this style can be used by any command that computes a temperature (e.g., *thermo_modify*, *fix temp/rescale*, *fix npt*).

After the center-of-mass velocity and angular velocity has been subtracted from each atom, the temperature is calculated by the formula

$$\mathrm{KE} = \frac{\mathrm{dim}}{2} N k_B T,$$

where KE is the total kinetic energy of the group of atoms (sum of $\frac{1}{2}mv^2$), dim = 2 or 3 is the dimensionality of the simulation, $N$ is the number of atoms in the group, $k_B$ is the Boltzmann constant, and $T$ is the absolute temperature.

A symmetric tensor, stored as a six-element vector, is also calculated by this compute for use in the computation of a pressure tensor by the *compute pressue* command. The formula for the components of the tensor is the same as the above expression for $E_{\mathrm{kin}}$, except that the 1/2 factor is NOT included and the $v_i^2$ is replaced by $v_{i,x}v_{i,y}$ for the $xy$ component, and so on. Note that because it lacks the 1/2 factor, these tensor components are twice those of the traditional kinetic energy tensor. The six components of the vector are ordered $xx$, $yy$, $zz$, $xy$, $xz$, $yz$.

The number of atoms contributing to the temperature is assumed to be constant for the duration of the run; use the *dynamic* option of the *compute_modify* command if this is not the case.

The removal of the center-of-mass velocity and angular velocity by this fix is essentially computing the temperature after a "bias" has been removed from the velocity of the atoms. If this compute is used with a fix command that performs thermostatting then this bias will be subtracted from each atom, thermostatting of the remaining thermal velocity will be performed, and the bias will be added back in. Thermostatting fixes that work in this way include *fix nvt*, *fix temp/rescale*, *fix temp/berendsen*, and *fix langevin*.

This compute subtracts out degrees-of-freedom due to fixes that constrain molecular motion, such as *fix shake* and *fix rigid*. This means the temperature of groups of atoms that include these constraints will be computed correctly. If needed, the subtracted degrees-of-freedom can be altered using the *extra* option of the *compute_modify* command.

See the *Howto thermostat* page for a discussion of different ways to compute temperature and perform thermostatting.

### 3.165.4 Output info

This compute calculates a global scalar (the temperature) and a global vector of length 6 (symmetric tensor), which can be accessed by indices 1-6. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "intensive". The vector values are "extensive".

The scalar value is in temperature *units*. The vector values are in energy *units*.

### 3.165.5 Restrictions

This compute is part of the EXTRA-COMPUTE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.165.6 Related commands

*compute temp*

### 3.165.7 Default

none

## 3.166 compute temp/sphere command

Accelerator Variants: *temp/sphere/kk*

### 3.166.1 Syntax

```
compute ID group-ID temp/sphere keyword value ...
```

- ID, group-ID are documented in *compute* command
- temp/sphere = style name of this compute command
- zero or more keyword/value pairs may be appended
- keyword = *bias* or *dof*

```
bias value = bias-ID
  bias-ID = ID of a temperature compute that removes a velocity bias
dof value = all or rotate
  all = compute temperature of translational and rotational degrees of freedom
  rotate = compute temperature of just rotational degrees of freedom
```

### 3.166.2 Examples

```
compute 1 all temp/sphere
compute myTemp mobile temp/sphere bias tempCOM
compute myTemp mobile temp/sphere dof rotate
```

### 3.166.3 Description

Define a computation that calculates the temperature of a group of spherical particles, including a contribution from both their translational and rotational kinetic energy. This differs from the usual *compute temp* command, which assumes point particles with only translational kinetic energy.

Both point and finite-size particles can be included in the group. Point particles do not rotate, so they have only three translational degrees of freedom. For 3d spherical particles, each has six degrees of freedom (three translational, three rotational). For 2d spherical particles, each has three degrees of freedom (two translational, one rotational).

**Note**

This choice for degrees of freedom (DOF) assumes that all finite-size spherical particles in your model will freely rotate, sampling all their rotational DOF. It is possible to use a combination of interaction potentials and fixes that induce no torque or otherwise constrain some of all of your particles so that this is not the case. Then there are less DOF and you should use the *compute_modify extra/dof* command to adjust the DOF accordingly.

The translational kinetic energy is computed the same as is described by the *compute temp* command. The rotational kinetic energy is computed as $\frac{1}{2}I\omega^2$, where $I$ is the moment of inertia for a sphere and $\omega$ is the particle's angular velocity.

**Note**

For *2d models*, particles are treated as spheres, not disks, meaning their moment of inertia will be the same as in 3d.

A kinetic energy tensor, stored as a six-element vector, is also calculated by this compute. The formula for the components of the tensor is the same as the above formulas, except that $v^2$ and $\omega^2$ are replaced by $v_x v_y$ and $\omega_x \omega_y$ for the $xy$ component. The six components of the vector are ordered $xx$, $yy$, $zz$, $xy$, $xz$, $yz$.

A symmetric tensor, stored as a six-element vector, is also calculated by this compute for use in the computation of a pressure tensor by the *compute pressue* command. The formula for the components of the tensor is the same as the above expression for $E_{\mathrm{kin}}$, except that the 1/2 factor is NOT included and the $v_i^2$ and $\omega^2$ are replaced by $v_x v_y$ and $\omega_x \omega_y$ for the $xy$ component, and so on. Note that because it lacks the 1/2 factor, these tensor components are twice those of the traditional kinetic energy tensor. The six components of the vector are ordered $xx$, $yy$, $zz$, $xy$, $xz$, $yz$.

The number of atoms contributing to the temperature is assumed to be constant for the duration of the run; use the *dynamic* option of the *compute_modify* command if this is not the case.

This compute subtracts out translational degrees-of-freedom due to fixes that constrain molecular motion, such as *fix shake* and *fix rigid*. This means the temperature of groups of atoms that include these constraints will be computed correctly. If needed, the subtracted degrees of freedom can be altered using the *extra/dof* option of the *compute_modify* command.

See the *Howto thermostat* page for a discussion of different ways to compute temperature and perform thermostatting.

---

The keyword/value option pairs are used in the following ways.

For the *bias* keyword, *bias-ID* refers to the ID of a temperature compute that removes a "bias" velocity from each atom. This allows compute temp/sphere to compute its thermal temperature after the translational kinetic energy components have been altered in a prescribed way (e.g., to remove a flow velocity profile). Thermostats that use this compute will work with this bias term. See the doc pages for individual computes that calculate a temperature and the doc pages for fixes that perform thermostatting for more details.

For the *dof* keyword, a setting of *all* calculates a temperature that includes both translational and rotational degrees of freedom. A setting of *rotate* calculates a temperature that includes only rotational degrees of freedom.

---

### 3.166.4 Output info

This compute calculates a global scalar (the temperature) and a global vector of length 6 (symmetric tensor), which can be accessed by indices 1–6. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "intensive". The vector values are "extensive".

The scalar value is in temperature *units*. The vector values are in energy *units*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 3.166.5 Restrictions

This fix requires that atoms store torque and angular velocity (omega) and a radius as defined by the *atom_style sphere* command.

All particles in the group must be finite-size spheres, or point particles with radius = 0.0.

### 3.166.6 Related commands

*compute temp*, *compute temp/asphere*

### 3.166.7 Default

The option defaults are no bias and dof = all.

## 3.167 compute temp/uef command

### 3.167.1 Syntax

```
compute ID group-ID temp/uef
```

- ID, group-ID are documented in *compute* command
- temp/uef = style name of this compute command

### 3.167.2 Examples

```
compute 1 all temp/uef
compute 2 sel temp/uef
```

### 3.167.3 Description

This command is used to compute the kinetic energy tensor in the reference frame of the applied flow field when *fix nvt/uef* or *fix npt/uef* is used. It is not necessary to use this command to compute the scalar value of the temperature. A *compute temp* may be used for that purpose.

Output information for this command can be found in the documentation for *compute temp*.

### 3.167.4 Restrictions

This fix is part of the UEF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This command can only be used when *fix nvt/uef* or *fix npt/uef* is active.

### 3.167.5 Related commands

*compute temp*, *fix nvt/uef* , *compute pressure/uef*

### 3.167.6 Default

none

## 3.168 compute ti command

### 3.168.1 Syntax

```
compute ID group ti keyword args ...
```

- ID, group-ID are documented in *compute* command
- ti = style name of this compute command
- one or more attribute/arg pairs may be appended
- keyword = pair style (lj/cut, gauss, born, etc.) or *tail* or *kspace*

```
pair style args = atype v_name1 v_name2
  atype = atom type (see asterisk form below)
  v_name1 = variable with name1 that is energy scale factor and function of lambda
  v_name2 = variable with name2 that is derivative of v_name1 with respect to lambda
tail args = atype v_name1 v_name2
  atype = atom type (see asterisk form below)
  v_name1 = variable with name1 that is energy tail correction scale factor and␣
↪function of lambda
  v_name2 = variable with name2 that is derivative of v_name1 with respect to lambda
kspace args = atype v_name1 v_name2
  atype = atom type (see asterisk form below)
  v_name1 = variable with name1 that is K-Space scale factor and function of lambda
  v_name2 = variable with name2 that is derivative of v_name1 with respect to lambda
```

### 3.168.2 Examples

```
compute 1 all ti lj/cut 1 v_lj v_dlj coul/long 2 v_c v_dc kspace 1 v_ks v_dks
compute 1 all ti lj/cut 1*3 v_lj v_dlj coul/long * v_c v_dc kspace * v_ks v_dks
```

### 3.168.3 Description

Define a computation that calculates the derivative of the interaction potential with respect to *lambda*, the coupling parameter used in a thermodynamic integration. This derivative can be used to infer a free energy difference resulting from an alchemical simulation, as described in *Eike*.

Typically this compute will be used in conjunction with the *fix adapt* command which can perform alchemical transformations by adjusting the strength of an interaction potential as a simulation runs, as defined by one or more *pair_style* or *kspace_style* commands. This scaling is done via a prefactor on the energy, forces, virial calculated by the pair or *k*-space style. The prefactor is often a function of a *lambda* parameter which may be adjusted from 0 to 1 (or vice versa) over the course of a *run*. The time-dependent adjustment is what the *fix adapt* command does.

Assume that the unscaled energy of a pair_style or kspace_style is given by $U$. Then the scaled energy is

$$U_s = f(\lambda)U$$

where $f$ is some function of $\lambda$. What this compute calculates is

$$\frac{dU_s}{d\lambda} = U\frac{df(\lambda)}{d\lambda} = \frac{U_s}{f(\lambda)}\frac{df(\lambda)}{d\lambda},$$

which is the derivative of the system's scaled potential energy $U_s$ with respect to $\lambda$.

To perform this calculation, you provide one or more atom types as *atype*. The variable *atype* can be specified in one of two ways. An explicit numeric value can be used, as in the first example above, or a wildcard asterisk can be used in place of or in conjunction with the *atype* argument to select multiple atom types. This takes the form "*" or "*n" or "m*" or "m*n". If $N$ is the number of atom types, then an asterisk with no numeric values means all types from 1 to $N$. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from m to N (inclusive). A middle asterisk means all types from m to n (inclusive).

You also specify two functions, as *equal-style variables*. The first is specified as *v_name1*, where *name1* is the name of the variable, and is $f(\lambda)$ in the notation above. The second is specified as *v_name2*, where *name2* is the name of the variable, and is $df(\lambda)/d\lambda$ in the notation above (i.e., it is the analytic derivative of $f$ with respect to $\lambda$). Note that the *name1* variable is also typically given as an argument to the *fix adapt* command.

An alchemical simulation may use several pair potentials together, invoked via the *pair_style hybrid or hybrid/overlay* command. The total $dU_s/d\lambda$ for the overall system is calculated as the sum of each contributing term as listed by the keywords in the *compute ti* command. Individual pair potentials can be listed, which will be sub-styles in the hybrid case. You can also include a *k*-space term via the *kspace* keyword. You can also include a pairwise long-range tail correction to the energy via the *tail* keyword.

For each term, you can specify a different (or the same) scale factor by the two variables that you list. Again, these will typically correspond toe the scale factors applied to these various potentials and the *k*-space contribution via the *fix adapt* command.

More details about the exact functional forms for the computation of $du/dl$ can be found in the paper by *Eike*.

---

### 3.168.4 Output info

This compute calculates a global scalar, namely $dU_s/d\lambda$. This value can be used by any command that uses a global scalar value from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "extensive".

The scalar value will be in energy *units*.

### 3.168.5 Restrictions

This compute is part of the EXTRA-COMPUTE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 3.168.6 Related commands

*fix adapt*

### 3.168.7 Default

none

---

**(Eike)** Eike and Maginn, Journal of Chemical Physics, 124, 164503 (2006).

## 3.169 compute torque/chunk command

### 3.169.1 Syntax

```
compute ID group-ID torque/chunk chunkID
```

- ID, group-ID are documented in *compute* command
- torque/chunk = style name of this compute command
- chunkID = ID of *compute chunk/atom* command

### 3.169.2 Examples

```
compute 1 fluid torque/chunk molchunk
```

### 3.169.3 Description

Define a computation that calculates the torque on multiple chunks of atoms.

In LAMMPS, chunks are collections of atoms defined by a *compute chunk/atom* command, which assigns each atom to a single chunk (or no chunk). The ID for this command is specified as chunkID. For example, a single chunk could be the atoms in a molecule or atoms in a spatial bin. See the *compute chunk/atom* and *Howto chunk* doc pages for details of how chunks can be defined and examples of how they can be used to measure properties of a system.

This compute calculates the three components of the torque vector for eqch chunk, due to the forces on the individual atoms in the chunk around the center-of-mass of the chunk. The calculation includes all effects due to atoms passing through periodic boundaries.

Note that only atoms in the specified group contribute to the calculation. The *compute chunk/atom* command defines its own group; atoms will have a chunk ID = 0 if they are not in that group, signifying they are not assigned to a chunk, and will thus also not contribute to this calculation. You can specify the "all" group for this command if you simply want to include atoms with non-zero chunk IDs.

> **Note**
>
> The coordinates of an atom contribute to the chunk's torque in "unwrapped" form, by using the image flags associated with each atom. See the *dump custom* command for a discussion of "unwrapped" coordinates. See the Atoms section of the *read_data* command for a discussion of image flags and how they are set for each atom. You can reset the image flags (e.g., to 0) before invoking this compute by using the *set image* command.

The simplest way to output the results of the compute torque/chunk calculation to a file is to use the *fix ave/time* command, for example:

```
compute cc1 all chunk/atom molecule
compute myChunk all torque/chunk cc1
fix 1 all ave/time 100 1 100 c_myChunk[*] file tmp.out mode vector
```

### 3.169.4 Output info

This compute calculates a global array where the number of rows is equal to the number of chunks *Nchunk* as calculated by the specified *compute chunk/atom* command. The number of columns is three for the $x$, $y$, and $z$ components of the torque for each chunk. These values can be accessed by any command that uses global array values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

The array values are "intensive". The array values will be in force-distance *units*.

### 3.169.5 Restrictions

none

### 3.169.6 Related commands

*variable torque() function*

### 3.169.7 Default

none

## 3.170 compute vacf command

### 3.170.1 Syntax

```
compute ID group-ID vacf
```

- ID, group-ID are documented in *compute* command
- vacf = style name of this compute command

### 3.170.2 Examples

```
compute 1 all vacf
compute 1 upper vacf
```

### 3.170.3 Description

Define a computation that calculates the velocity auto-correlation function (VACF), averaged over a group of atoms. Each atom's contribution to the VACF is its current velocity vector dotted into its initial velocity vector at the time the compute was specified.

A vector of four quantities is calculated by this compute. The first three elements of the vector are $v_x v_{x,0}$ (and similar for the $y$ and $z$ components), summed and averaged over atoms in the group, where $v_x$ is the current $x$-component of the velocity of the atom and $v_{x,0}$ is the initial $x$-component of the velocity of the atom. The fourth element of the vector is the total VACF (i.e., $(v_x v_{x,0} + v_y v_{y,0} + v_z v_{z,0})$), summed and averaged over atoms in the group.

The integral of the VACF versus time is proportional to the diffusion coefficient of the diffusing atoms. This can be computed in the following manner, using the *variable trap()* function:

```
compute         2 all vacf
fix             5 all vector 1 c_2[4]
variable        diff equal dt*trap(f_5)
thermo_style    custom step v_diff
```

**Note**

If you want the quantities calculated by this compute to be continuous when running from a *restart file*, then you should use the same ID for this compute, as in the original run. This is so that the fix this compute creates to store per-atom quantities will also have the same ID, and thus be initialized correctly with time=0 atom velocities from the restart file.

### 3.170.4 Output info

This compute calculates a global vector of length 4, which can be accessed by indices 1–4 by any command that uses global vector values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

The vector values are "intensive". The vector values will be in velocity$^2$ *units*.

### 3.170.5 Restrictions

none

### 3.170.6 Related commands

*compute msd*, *compute vacf/chunk*

### 3.170.7 Default

none

## 3.171 compute vacf/chunk command

### 3.171.1 Syntax

```
compute ID group-ID vacf/chunk chunkID
```

- ID, group-ID are documented in *compute* command
- vacf/chunk = style name of this compute command

- chunkID = ID of *compute chunk/atom* command

### 3.171.2 Examples

```
compute 1 all vacf/chunk molchunk
```

### 3.171.3 Description

Added in version 2Apr2025.

Define a computation that calculates the velocity auto-correlation function (VACF) for multiple chunks of atoms.

In LAMMPS, chunks are collections of atoms defined by a *compute chunk/atom* command, which assigns each atom to a single chunk (or no chunk). The ID for this command is specified as chunkID. For example, a single chunk could be the atoms in a molecule or atoms in a spatial bin. See the *compute chunk/atom* and *Howto chunk* doc pages for details of how chunks can be defined and examples of how they can be used to measure properties of a system.

Four quantities are calculated by this compute for each chunk. The first 3 quantities are the product of the initial center of mass velocity (VCM) for each chunk in $x$, $y$, and $z$ direction with the current center of mass velocity in the same direction. The fourth component is the total VACF, i.e. the sum of the three components.

Note that only atoms in the specified group contribute to the calculation. The *compute chunk/atom* command defines its own group; atoms will have a chunk ID = 0 if they are not in that group, signifying they are not assigned to a chunk, and will thus also not contribute to this calculation. You can specify the "all" group for this command if you simply want to include atoms with non-zero chunk IDs.

The integral of the VACF versus time is proportional to the diffusion coefficient of the diffusing chunks.

**Note**

The number of chunks *Nchunk* calculated by the *compute chunk/atom* command must remain constant each time this compute is invoked, so that the dot product for each chunk from its original position can be computed consistently. If *Nchunk* does not remain constant, an error will be generated. If needed, you can enforce a constant *Nchunk* by using the *nchunk once* or *ids once* options when specifying the *compute chunk/atom* command.

**Note**

This compute stores the original center-of-mass velocities of each chunk. When a VACF is calculated on a later timestep, it is assumed that the same atoms are assigned to the same chunk ID. However LAMMPS has no simple way to ensure this is the case, though you can use the *ids once* option when specifying the *compute chunk/atom* command. Note that if this is not the case, the VACF calculation does not have a sensible meaning.

**Note**

If you want the quantities calculated by this compute to be continuous when running from a *restart file*, then you should use the same ID for this compute, as in the original run. This is so that the fix this compute creates to store per-chunk quantities will also have the same ID, and thus be initialized correctly with chunk reference positions from the restart file.

The simplest way to output the results of the compute vacf/chunk calculation to a file is to use the *fix ave/time* command, for example:

```
compute cc1 all chunk/atom molecule
compute myChunk all vacf/chunk cc1
fix 1 all ave/time 100 1 100 c_myChunk[*] file tmp.out mode vector
```

### 3.171.4 Output info

This compute calculates a global array where the number of rows = the number of chunks *Nchunk* as calculated by the specified *compute chunk/atom* command. The number of columns = 4 for the $x$, $y$, $z$, component and the total VACF. These values can be accessed by any command that uses global array values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The array values are "intensive". The array values will be in distance$^2$ divided by time$^2$ *units*.

### 3.171.5 Restrictions

none

### 3.171.6 Related commands

*compute vacf* , *compute msd/chunk*

### 3.171.7 Default

none

## 3.172 compute vcm/chunk command

### 3.172.1 Syntax

```
compute ID group-ID vcm/chunk chunkID
```

- ID, group-ID are documented in *compute* command
- vcm/chunk = style name of this compute command
- chunkID = ID of *compute chunk/atom* command

### 3.172.2 Examples

```
compute 1 fluid vcm/chunk molchunk
```

### 3.172.3 Description

Define a computation that calculates the center-of-mass velocity for multiple chunks of atoms.

In LAMMPS, chunks are collections of atoms defined by a *compute chunk/atom* command, which assigns each atom to a single chunk (or no chunk). The ID for this command is specified as chunkID. For example, a single chunk could be the atoms in a molecule or atoms in a spatial bin. See the *compute chunk/atom* and *Howto chunk* doc pages for details of how chunks can be defined and examples of how they can be used to measure properties of a system.

This compute calculates the $(x, y, z)$ components of the center-of-mass velocity for each chunk. This is done by summing mass*velocity for each atom in the chunk and dividing the sum by the total mass of the chunk.

Note that only atoms in the specified group contribute to the calculation. The *compute chunk/atom* command defines its own group; atoms will have a chunk ID = 0 if they are not in that group, signifying they are not assigned to a chunk,

and will thus also not contribute to this calculation. You can specify the "all" group for this command if you simply want to include atoms with non-zero chunk IDs.

The simplest way to output the results of the compute vcm/chunk calculation to a file is to use the *fix ave/time* command, for example:

```
compute cc1 all chunk/atom molecule
compute myChunk all vcm/chunk cc1
fix 1 all ave/time 100 1 100 c_myChunk[*] file tmp.out mode vector
```

### 3.172.4 Output info

This compute calculates a global array where the number of rows is the number of chunks *Nchunk* as calculated by the specified *compute chunk/atom* command. The number of columns is 3 for the $(x, y, z)$ center-of-mass velocity coordinates of each chunk. These values can be accessed by any command that uses global array values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The array values are "intensive". The array values will be in velocity *units*.

### 3.172.5 Restrictions

none

### 3.172.6 Related commands

none

### 3.172.7 Default

none

## 3.173 compute viscosity/cos command

### 3.173.1 Syntax

```
compute ID group-ID viscosity/cos
```

- ID, group-ID are documented in *compute* command
- viscosity/cos = style name of this compute command

### 3.173.2 Examples

```
units    real
compute  cos all viscosity/cos
variable V equal c_cos[7]
variable A equal 0.02E-5  # A/fs^2
variable density equal density
variable lz equal lz
variable reciprocalViscosity equal v_V/${A}/v_density*39.4784/v_lz/v_lz*100  # 1/(Pa*s)
```

### 3.173.3 Description

Define a computation that calculates the velocity amplitude of a group of atoms with an cosine-shaped velocity profile and the temperature of them after subtracting out the velocity profile before computing the kinetic energy. A compute of this style can be used by any command that computes a temperature (e.g., *thermo_modify*, *fix npt*).

This command together with *fix_accelerate/cos* enables viscosity calculation with periodic perturbation method, as described by *Hess*. An acceleration along the $x$-direction is applied to the simulation system by using *fix_accelerate/cos* command. The acceleration is a periodic function along the $z$-direction:

$$a_x(z) = A \cos\left(\frac{2\pi z}{l_z}\right)$$

where $A$ is the acceleration amplitude, $l_z$ is the $z$-length of the simulation box. At steady state, the acceleration generates a velocity profile:

$$v_x(z) = V \cos\left(\frac{2\pi z}{l_z}\right)$$

The generated velocity amplitude $V$ is related to the shear viscosity $\eta$ by

$$V = \frac{A\rho}{\eta}\left(\frac{l_z}{2\pi}\right)^2,$$

and it can be obtained from ensemble average of the velocity profile via

$$V = \frac{\sum_i 2m_i v_{i,x} \cos\left(\frac{2\pi z_i}{l_z}\right)}{\sum_i m_i}$$

where $m_i$, $v_{i,x}$ and $z_i$ are the mass, $x$-component velocity, and $z$-coordinate of a particle, respectively.

After the cosine-shaped collective velocity in the $x$-direction has been subtracted for each atom, the temperature is calculated by the formula

$$\mathrm{KE} = \frac{\mathrm{dim}}{2} N k_B T,$$

where KE is the total kinetic energy of the group of atoms (sum of $\frac{1}{2}mv^2$), dim = 2 or 3 is the dimensionality of the simulation, $N$ is the number of atoms in the group, $k_B$ is the Boltzmann constant, and $T$ is the absolute temperature.

A symmetric tensor, stored as a six-element vector, is also calculated by this compute for use in the computation of a pressure tensor by the *compute pressue* command. The formula for the components of the tensor is the same as the above expression for $E_{\mathrm{kin}}$, except that the 1/2 factor is NOT included and the $v_i^2$ is replaced by $v_{i,x}v_{i,y}$ for the $xy$ component, and so on. Note that because it lacks the 1/2 factor, these tensor components are twice those of the traditional kinetic energy tensor. The six components of the vector are ordered $xx$, $yy$, $zz$, $xy$, $xz$, $yz$.

The number of atoms contributing to the temperature is assumed to be constant for the duration of the run; use the *dynamic* option of the *compute_modify* command if this is not the case. However, in order to get meaningful results, the group ID of this compute should be all.

The removal of the cosine-shaped velocity component by this command is essentially computing the temperature after a "bias" has been removed from the velocity of the atoms. If this compute is used with a fix command that performs thermostatting then this bias will be subtracted from each atom, thermostatting of the remaining thermal velocity will be performed, and the bias will be added back in. Thermostatting fixes that work in this way include *fix nvt*, *fix temp/rescale*, *fix temp/berendsen*, and *fix langevin*.

This compute subtracts out degrees of freedom due to fixes that constrain molecular motion, such as *fix shake* and *fix rigid*. This means that the temperature of groups of atoms that include these constraints will be computed correctly. If needed, the subtracted degrees of freedom can be altered using the *extra* option of the *compute_modify* command.

See the *Howto thermostat* page for a discussion of different ways to compute temperature and perform thermostatting.

### 3.173.4 Output info

This compute calculates a global scalar (the temperature) and a global vector of length 7, which can be accessed by indices 1–7. The first six elements of the vector are those of the symmetric tensor discussed above. The seventh is the cosine-shaped velocity amplitude $V$, which can be used to calculate the reciprocal viscosity, as shown in the example. These values can be used by any command that uses global scalar or vector values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The scalar value calculated by this compute is "intensive". The first six elements of vector values are "extensive", and the seventh element of vector values is "intensive".

The scalar value is in temperature *units*. The first six elements of vector values are in energy *units*. The seventh element of vector value us in velocity *units*.

### 3.173.5 Restrictions

This compute is part of the MISC package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Since this compute depends on *fix accelerate/cos* which can only work for 3d systems, it cannot be used for 2d systems.

### 3.173.6 Related commands

*fix accelerate/cos*

### 3.173.7 Default

none

---

**(Hess)** Hess, B. The Journal of Chemical Physics 2002, 116 (1), 209-217.

## 3.174 compute voronoi/atom command

### 3.174.1 Syntax

```
compute ID group-ID voronoi/atom keyword arg ...
```

- ID, group-ID are documented in *compute* command
- voronoi/atom = style name of this compute command
- zero or more keyword/value pairs may be appended
- keyword = *only_group* or *occupation* or *surface* or *radius* or *edge_histo* or *edge_threshold* or *face_threshold* or *neighbors*

```
only_group = no arg
occupation = no arg
surface arg = sgroup-ID
  sgroup-ID = compute the dividing surface between group-ID and sgroup-ID
    this keyword adds a third column to the compute output
radius arg = v_r
  v_r = radius atom style variable for a poly-disperse Voronoi tessellation
edge_histo arg = maxedge
  maxedge = maximum number of Voronoi cell edges to be accounted in the histogram
edge_threshold arg = minlength
  minlength = minimum length for an edge to be counted
```

```
face_threshold arg = minarea
  minarea = minimum area for a face to be counted
neighbors value = yes or no = store list of all neighbors or no
```

### 3.174.2 Examples

```
compute 1 all voronoi/atom
compute 2 precipitate voronoi/atom surface matrix
compute 3b precipitate voronoi/atom radius v_r
compute 4 solute voronoi/atom only_group
compute 5 defects voronoi/atom occupation
compute 6 all voronoi/atom neighbors yes
```

### 3.174.3 Description

Define a computation that calculates the Voronoi tessellation of the atoms in the simulation box. The tessellation is calculated using all atoms in the simulation, but non-zero values are only stored for atoms in the group.

Two per-atom quantities are calculated by this compute. The first is the volume of the Voronoi cell around each atom. Any point in an atom's Voronoi cell is closer to that atom than any other. The second is the number of faces of the Voronoi cell. This is equal to the number of nearest neighbors of the central atom, plus any exterior faces (see note below).

---

If the *only_group* keyword is specified the tessellation is performed only with respect to the atoms contained in the compute group. This is equivalent to deleting all atoms not contained in the group prior to evaluating the tessellation.

If the *surface* keyword is specified a third quantity per atom is computed: the Voronoi cell surface of the given atom. *surface* takes a group ID as an argument. If a group other than *all* is specified, only the Voronoi cell facets facing a neighbor atom from the specified group are counted towards the surface area.

In the example above, a precipitate embedded in a matrix, only atoms at the surface of the precipitate will have non-zero surface area, and only the outward facing facets of the Voronoi cells are counted (the hull of the precipitate). The total surface area of the precipitate can be obtained by running a "reduce sum" compute on c_2[3].

If the *radius* keyword is specified with an atom style variable as the argument, a poly-disperse Voronoi tessellation is performed. Examples for radius variables are

```
variable r1 atom (type==1)*0.1+(type==2)*0.4
compute radius all property/atom radius
variable r2 atom c_radius
```

Here v_r1 specifies a per-type radius of 0.1 units for type 1 atoms and 0.4 units for type 2 atoms, and v_r2 accesses the radius property present in atom_style sphere for granular models.

The *edge_histo* keyword activates the compilation of a histogram of number of edges on the faces of the Voronoi cells in the compute group. The argument *maxedge* of the this keyword is the largest number of edges on a single Voronoi cell face expected to occur in the sample. This keyword generates output of a global vector by this compute with *maxedge*+1 entries. The last entry in the vector contains the number of faces with more than *maxedge* edges. Since the polygon with the smallest amount of edges is a triangle, entries 1 and 2 of the vector will always be zero.

The *edge_threshold* and *face_threshold* keywords allow the suppression of edges below a given minimum length and faces below a given minimum area. Ultra short edges and ultra small faces can occur as artifacts of the Voronoi tessellation. These keywords will affect the neighbor count and edge histogram outputs.

If the *occupation* keyword is specified the tessellation is only performed for the first invocation of the compute and then stored. For all following invocations of the compute the number of atoms in each Voronoi cell in the stored tessellation

is counted. In this mode the compute returns a per-atom array with 2 columns. The first column is the number of atoms currently in the Voronoi volume defined by this atom at the time of the first invocation of the compute (note that the atom may have moved significantly). The second column contains the total number of atoms sharing the Voronoi cell of the stored tessellation at the location of the current atom. Numbers in column one can be any positive integer including zero, while column two values will always be greater than zero. Column one data can be used to locate vacancies (the coordinates are given by the atom coordinates at the time step when the compute was first invoked), while column two data can be used to identify interstitial atoms.

If the *neighbors* value is set to yes, then this compute also creates a local array with 3 columns. There is one row for each face of each Voronoi cell. The 3 columns are the atom ID of the atom that owns the cell, the atom ID of the atom in the neighboring cell (or zero if the face is external), and the area of the face. The array can be accessed by any command that uses local values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options. More specifically, the array can be accessed by a *dump local* command to write a file containing all the Voronoi neighbors in a system:

```
compute 6 all voronoi/atom neighbors yes
dump d2 all local 1 dump.neighbors index c_6[1] c_6[2] c_6[3]
```

If the *face_threshold* keyword is used, then only faces with areas greater than the threshold are stored.

---

The Voronoi calculation is performed by the freely available Voro++ package, written by Chris Rycroft at UC Berkeley and LBL, which must be installed on your system when building LAMMPS for use with this compute. See instructions on obtaining and installing the Voro++ software in the src/VORONOI/README file.

**Note**

The calculation of Voronoi volumes is performed by each processor for the atoms it owns, and includes the effect of ghost atoms stored by the processor. This assumes that the Voronoi cells of owned atoms are not affected by atoms beyond the ghost atom cut-off distance. This is usually a good assumption for liquid and solid systems, but may lead to underestimation of Voronoi volumes in low density systems. By default, the set of ghost atoms stored by each processor is determined by the cutoff used for *pair_style* interactions. The cutoff can be set explicitly via the *comm_modify cutoff* command. The Voronoi cells for atoms adjacent to empty regions will extend into those regions up to the communication cutoff in $x$, $y$, or $z$. In that situation, an exterior face is created at the cutoff distance normal to the $x$, $y$, or $z$ direction. For triclinic systems, the exterior face is parallel to the corresponding reciprocal lattice vector.

**Note**

The Voro++ package performs its calculation in 3d. This will still work for a 2d LAMMPS simulation, provided all the atoms have the same $z$-coordinate. The Voronoi cell of each atom will be a columnar polyhedron with constant cross-sectional area along the $z$-direction and two exterior faces at the top and bottom of the simulation box. If the atoms do not all have the same $z$-coordinate, then the columnar cells will be accordingly distorted. The cross-sectional area of each Voronoi cell can be obtained by dividing its volume by the $z$ extent of the simulation box. Note that you define the $z$ extent of the simulation box for 2d simulations when using the *create_box* or *read_data* commands.

### 3.174.4 Output info

Deprecated since version 21Nov2023: The *peratom* keyword was removed as it is no longer required.

This compute calculates a per-atom array with two columns. In regular dynamic tessellation mode the first column is the Voronoi volume, the second is the neighbor count, as described above (read above for the output data in case the

*occupation* keyword is specified). These values can be accessed by any command that uses per-atom values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

If the *edge_histo* keyword is used, then this compute generates a global vector of length *maxedge*+1, containing a histogram of the number of edges per face.

If the *neighbors* value is set to *yes*, then this compute calculates a local array with three columns. There is one row for each face of each Voronoi cell.

The Voronoi cell volume will be in distance *units* cubed. The Voronoi face area will be in distance *units* squared.

### 3.174.5 Restrictions

This compute is part of the VORONOI package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

It also requires you have a copy of the Voro++ library built and installed on your system. See instructions on obtaining and installing the Voro++ software in the src/VORONOI/README file.

### 3.174.6 Related commands

*dump custom*, *dump local*

### 3.174.7 Default

The default for the neighbors keyword is no.

## 3.175 compute xrd command

### 3.175.1 Syntax

```
compute ID group-ID xrd lambda type1 type2 ... typeN keyword value ...
```

- ID, group-ID are documented in *compute* command
- xrd = style name of this compute command
- lambda = wavelength of incident radiation (length units)
- type1 type2 ... typeN = chemical symbol of each atom type (see valid options below)
- zero or more keyword/value pairs may be appended
- keyword = *2Theta* or *c* or *LP* or *manual* or *echo*

```
2Theta values = Min2Theta Max2Theta
  Min2Theta,Max2Theta = minimum and maximum 2 theta range to explore
  (radians or degrees)
c values = c1 c2 c3
  c1,c2,c3 = parameters to adjust the spacing of the reciprocal
             lattice nodes in the h, k, and l directions respectively
LP value = switch to apply Lorentz-polarization factor
  0/1 = off/on
manual = flag to use manual spacing of reciprocal lattice points
           based on the values of the c parameters
echo = flag to provide extra output for debugging purposes
```

### 3.175.2 Examples

```
compute 1 all xrd 1.541838 Al O 2Theta 0.087 0.87 c 1 1 1 LP 1 echo
compute 2 all xrd 1.541838 Al O 2Theta 10 100 c 0.05 0.05 0.05 LP 1 manual

fix 1 all ave/histo/weight 1 1 1 0.087 0.87 250 c_1[1] c_1[2] mode vector file Rad2Theta.
↪xrd
fix 2 all ave/histo/weight 1 1 1 10 100 250 c_2[1] c_2[2] mode vector file Deg2Theta.xrd
```

### 3.175.3 Description

Define a computation that calculates X-ray diffraction intensity as described in *(Coleman)* on a mesh of reciprocal lattice nodes defined by the entire simulation domain (or manually) using a simulated radiation of wavelength *lambda*.

The X-ray diffraction intensity, $I$, at each reciprocal lattice point, $k$, is computed from the structure factor, $F$, using the equations:

$$I = L_p(\theta)\frac{F^*F}{N}$$

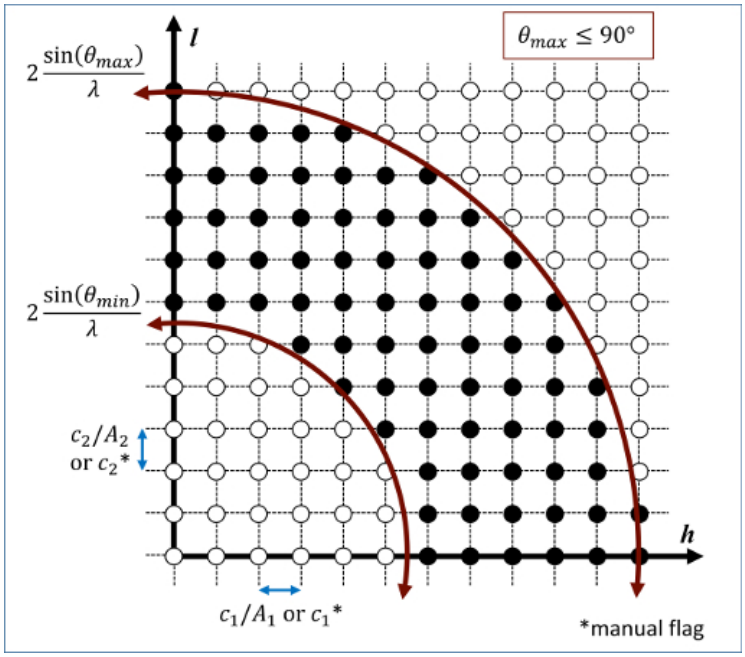
$$F(\mathbf{k}) = \sum_{j=1}^{N} f_j(\theta) exp(2\pi i \mathbf{k}\cdot\mathbf{r}_j)$$

$$L_p(\theta) = \frac{1+\cos^2(2\theta)}{\cos(\theta)\sin^2(\theta)}$$

$$\frac{\sin(\theta)}{\lambda} = \frac{\|\mathbf{k}\|}{2}$$

Here, $\mathbf{k}$ is the location of the reciprocal lattice node, $r_j$ is the position of each atom, $f_j$ are atomic scattering factors, $Lp$ is the Lorentz-polarization factor, and $\theta$ is the scattering angle of diffraction. The Lorentz-polarization factor can be turned off using the optional *LP* keyword.

Diffraction intensities are calculated on a three-dimensional mesh of reciprocal lattice nodes. The mesh spacing is defined either (a) by the entire simulation domain or (b) manually using selected values as shown in the 2D diagram below.



For a mesh defined by the simulation domain, a rectilinear grid is constructed with spacing $cA^{-1}$ along each reciprocal lattice axis, where $A$ is a matrix containing the vectors corresponding to the edges of the simulation cell. If one or two directions has non-periodic boundary conditions, then the spacing in these directions is defined from the average of the (inversed) box lengths with periodic boundary conditions. Meshes defined by the simulation domain must contain at least one periodic boundary.

If the *manual* flag is included, the mesh of reciprocal lattice nodes will be defined using the $c$ values for the spacing along each reciprocal lattice axis. Note that manual mapping of the reciprocal space mesh is good for comparing diffraction results from multiple simulations; however, it can reduce the likelihood that Bragg reflections will be satisfied unless small spacing parameters ($< 0.05$ Å$^{-1}$) are implemented. Meshes with manual spacing do not require a periodic boundary.

The limits of the reciprocal lattice mesh are determined by range of scattering angles explored. The *2Theta* parameter allows the user to reduce the scattering angle range to only the region of interest which reduces the cost of the computation.

The atomic scattering factor, $f_j$, accounts for the reduction in diffraction intensity due to Compton scattering. Compute xrd uses analytical approximations of the atomic scattering factors that vary for each atom type (type1 type2 ... typeN) and angle of diffraction. The analytic approximation is computed using the formula *(Colliex)*:

$$f_j\left(\frac{\sin(\theta)}{\lambda}\right) = \sum_{i=1}^{4} a_i \exp\left(-b_i \frac{\sin^2(\theta)}{\lambda^2}\right) + c$$

Coefficients parameterized by *(Peng)* are assigned for each atom type designating the chemical symbol and charge of each atom type. Valid chemical symbols for compute xrd are:

| | | | | |
|---|---|---|---|---|
| H | He1- | He | Li | Li1+ |
| Be | Be2+ | B | C | Cval |
| N | O | O1- | F | F1- |
| Ne | Na | Na1+ | Mg | Mg2+ |
| Al | Al3+ | Si | Sival | Si4+ |
| P | S | Cl | Cl1- | Ar |
| K | Ca | Ca2+ | Sc | Sc3+ |
| Ti | Ti2+ | Ti3+ | Ti4+ | V |
| V2+ | V3+ | V5+ | Cr | Cr2+ |
| Cr3+ | Mn | Mn2+ | Mn3+ | Mn4+ |
| Fe | Fe2+ | Fe3+ | Co | Co2+ |
| Co | Ni | Ni2+ | Ni3+ | Cu |
| Cu1+ | Cu2+ | Zn | Zn2+ | Ga |
| Ga3+ | Ge | Ge4+ | As | Se |
| Br | Br1- | Kr | Rb | Rb1+ |
| Sr | Sr2+ | Y | Y3+ | Zr |
| Zr4+ | Nb | Nb3+ | Nb5+ | Mo |
| Mo3+ | Mo5+ | Mo6+ | Tc | Ru |
| Ru3+ | Ru4+ | Rh | Rh3+ | Rh4+ |
| Pd | Pd2+ | Pd4+ | Ag | Ag1+ |
| Ag2+ | Cd | Cd2+ | In | In3+ |
| Sn | Sn2+ | Sn4+ | Sb | Sb3+ |
| Sb5+ | Te | I | I1- | Xe |
| Cs | Cs1+ | Ba | Ba2+ | La |
| La3+ | Ce | Ce3+ | Ce4+ | Pr |
| Pr3+ | Pr4+ | Nd | Nd3+ | Pm |
| Pm3+ | Sm | Sm3+ | Eu | Eu2+ |
| Eu3+ | Gd | Gd3+ | Tb | Tb3+ |

continues on next page

Table 1 – continued from previous page

| | | | | |
|---|---|---|---|---|
| Dy | Dy3+ | Ho | Ho3+ | Er |
| Er3+ | Tm | Tm3+ | Yb | Yb2+ |
| Yb3+ | Lu | Lu3+ | Hf | Hf4+ |
| Ta | Ta5+ | W | W6+ | Re |
| Os | Os4+ | Ir | Ir3+ | Ir4+ |
| Pt | Pt2+ | Pt4+ | Au | Au1+ |
| Au3+ | Hg | Hg1+ | Hg2+ | Tl |
| Tl1+ | Tl3+ | Pb | Pb2+ | Pb4+ |
| Bi | Bi3+ | Bi5+ | Po | At |
| Rn | Fr | Ra | Ra2+ | Ac |
| Ac3+ | Th | Th4+ | Pa | U |
| U3+ | U4+ | U6+ | Np | Np3+ |
| Np4+ | Np6+ | Pu | Pu3+ | Pu4+ |
| Pu6+ | Am | Cm | Bk | Cf |

If the *echo* keyword is specified, compute xrd will provide extra reporting information to the screen.

### 3.175.4 Output info

This compute calculates a global array. The number of rows in the array is the number of reciprocal lattice nodes that are explored which by the mesh. The global array has two columns.

The first column contains the diffraction angle in the units (radians or degrees) provided with the *2Theta* values. The second column contains the computed diffraction intensities as described above.

The array can be accessed by any command that uses global values from a compute as input. See the *Howto output* doc page for an overview of LAMMPS output options.

All array values calculated by this compute are "intensive".

### 3.175.5 Restrictions

This compute is part of the DIFFRACTION package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The compute_xrd command does not work for triclinic cells.

### 3.175.6 Related commands

*fix ave/histo*, *compute saed*

### 3.175.7 Default

The option defaults are *2Theta* = 1 179 (degrees), *c* = 1 1 1, *LP* = 1, no manual flag, no echo flag.

---

**(Coleman)** Coleman, Spearot, Capolungo, MSMSE, 21, 055020 (2013).

**(Colliex)** Colliex et al. International Tables for Crystallography Volume C: Mathematical and Chemical Tables, 249-429 (2004).

**(Peng)** Peng, Ren, Dudarev, Whelan, Acta Crystallogr. A, 52, 257-76 (1996).

CHAPTER

FOUR

# PAIR STYLES

## 4.1 pair_style adp command

Accelerator Variants: *adp/kk*, *adp/omp*

### 4.1.1 Syntax

```
pair_style adp
```

### 4.1.2 Examples

```
pair_style adp
pair_coeff * * Ta.adp Ta
pair_coeff * * ../potentials/AlCu.adp Al Al Cu
```

### 4.1.3 Description

Style *adp* computes pairwise interactions for metals and metal alloys using the angular dependent potential (ADP) of *(Mishin)*, which is a generalization of the *embedded atom method (EAM) potential*. The LAMMPS implementation is discussed in *(Singh)*. The total energy Ei of an atom I is given by

$$E_i = F_\alpha \left( \sum_{j \neq i} \rho_\beta(r_{ij}) \right) + \frac{1}{2} \sum_{j \neq i} \phi_{\alpha\beta}(r_{ij}) + \frac{1}{2} \sum_s (\mu_i^s)^2 + \frac{1}{2} \sum_{s,t} (\lambda_i^{st})^2 - \frac{1}{6} \nu_i^2$$

$$\mu_i^s = \sum_{j \neq i} u_{\alpha\beta}(r_{ij}) r_{ij}^s$$

$$\lambda_i^{st} = \sum_{j \neq i} w_{\alpha\beta}(r_{ij}) r_{ij}^s r_{ij}^t$$

$$\nu_i = \sum_s \lambda_i^{ss}$$

where $F$ is the embedding energy which is a function of the atomic electron density $\rho$, $\phi$ is a pair potential interaction, $\alpha$ and $\beta$ are the element types of atoms $I$ and $J$, and $s$ and $t = 1, 2, 3$ and refer to the cartesian coordinates. The $\mu$ and $\lambda$ terms represent the dipole and quadruple distortions of the local atomic environment which extend the original EAM framework by introducing angular forces.

Note that unlike for other potentials, cutoffs for ADP potentials are not set in the pair_style or pair_coeff command; they are specified in the ADP potential files themselves. Likewise, the ADP potential files list atomic masses; thus you do not need to use the *mass* command to specify them.

**ADP potentials are available from:**

- The NIST WWW site at https://www.ctcms.nist.gov/potentials/. Note that ADP potentials obtained from NIST must be converted into the extended DYNAMO *setfl* format discussed below.
- The OpenKIM Project at https://openkim.org/browse/models/by-type provides ADP potentials that can be used directly in LAMMPS with the *kim command* interface.

---

Only a single pair_coeff command is used with the *adp* style which specifies an extended DYNAMO *setfl* file, which contains information for *M* elements. These are mapped to LAMMPS atom types by specifying *N* additional arguments after the filename in the pair_coeff command, where *N* is the number of LAMMPS atom types:

- filename
- *N* element names = mapping of extended *setfl* elements to atom types

See the *pair_coeff* page for alternate ways to specify the path for the potential file.

As an example, the potentials/AlCu.adp file, included in the potentials directory of the LAMMPS distribution, is an extended *setfl* file which has tabulated ADP values for w elements and their alloy interactions: Cu and Al. If your LAMMPS simulation has 4 atoms types and you want the first 3 to be Al, and the fourth to be Cu, you would use the following pair_coeff command:

```
pair_coeff * * AlCu.adp Al Al Al Cu
```

The first 2 arguments must be * * so as to span all LAMMPS atom types. The first three Al arguments map LAMMPS atom types 1,2,3 to the Al element in the extended *setfl* file. The final Cu argument maps LAMMPS atom type 4 to the Al element in the extended *setfl* file. Note that there is no requirement that your simulation use all the elements specified by the extended *setfl* file.

If a mapping value is specified as NULL, the mapping is not performed. This can be used when an *adp* potential is used as part of the *hybrid* pair style. The NULL values are placeholders for atom types that will be used with other potentials.

*Adp* files in the *potentials* directory of the LAMMPS distribution have an ".adp" suffix. A DYNAMO *setfl* file extended for ADP is formatted as follows. Basically it is the standard *setfl* format with additional tabulated functions u and w added to the file after the tabulated pair potentials. See the *pair_eam* command for further details on the *setfl* format.

- lines 1,2,3 = comments (ignored)
- line 4: $N_{\text{elements}}$ Element1 Element2 ... ElementN
- line 5: $N_\rho$, $d_\rho$, $N_r$, $d_r$, cutoff

Following the 5 header lines are $N_{\text{elements}}$ sections, one for each element, each with the following format:

- line 1 = atomic number, mass, lattice constant, lattice type (e.g. FCC)
- embedding function $F(\rho)$ ($N_\rho$ values)
- density function $\rho(r)$ ($N_r$ values)

Following the $N_{\text{elements}}$ sections, $N_r$ values for each pair potential $\phi(r)$ array are listed for all $i, j$ element pairs in the same format as other arrays. Since these interactions are symmetric ($i, j = j, i$) only $\phi$ arrays with $i \geq j$ are listed, in the following order:

$$i, j = (1,1), (2,1), (2,2), (3,1), (3,2), (3,3), (4,1), ..., (N_{\text{elements}}, N_{\text{elements}}).$$

The tabulated values for each $\phi$ function are listed as $r * \phi$ (in units of eV-Angstroms), since they are for atom pairs, the same as for *other EAM files*.

After the $\phi(r)$ arrays, each of the $u(r)$ arrays are listed in the same order with the same assumptions of symmetry. Directly following the $u(r)$, the $w(r)$ arrays are listed. Note that $\phi(r)$ is the only array tabulated with a scaling by $r$.

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 4.1.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, where types I and J correspond to two different element types, no special mixing rules are needed, since the ADP potential files specify alloy interactions explicitly.

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in tabulated potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.1.5 Restrictions

This pair style is part of the MANYBODY package. It is only enabled if LAMMPS was built with that package.

### 4.1.6 Related commands

*pair_coeff*, *pair_eam*

### 4.1.7 Default

none

**(Mishin)** Mishin, Mehl, and Papaconstantopoulos, Acta Mater, 53, 4029 (2005).

**(Singh)** Singh and Warner, Acta Mater, 58, 5797-5805 (2010),

## 4.2 pair_style agni command

Accelerator Variants: *agni/omp*

### 4.2.1 Syntax

```
pair_style agni
```

### 4.2.2 Examples

```
pair_style      agni
pair_coeff      * * Al.agni Al
```

### 4.2.3 Description

Style *agni* style computes the many-body vectorial force components for an atom as

$$F_i^u = \sum_t^{N_t} \alpha_t \cdot \exp\left[-\frac{\left(d_{i,t}^u\right)^2}{2l^2}\right]$$

$$d_{i,t}^u = ||V_i^u(\eta) - V_t^u(\eta)||$$

$$V_i^u(\eta) = \sum_{j \neq i} \frac{r_{ij}^u}{r_{ij}} \cdot e^{-\left(\frac{r_{ij}}{\eta}\right)^2} \cdot f_d\left(r_{ij}\right)$$

$$f_d\left(r_{ij}\right) = \frac{1}{2}\left[\cos\left(\frac{\pi r_{ij}}{R_c}\right) + 1\right]$$

$u$ labels the individual components, i.e. $x$, $y$ or $z$, and $V$ is the corresponding atomic fingerprint. $d$ is the Euclidean distance between any two atomic fingerprints. A total of $N_t$ reference atomic environments are considered to construct the force field file. $\alpha_t$ and $l$ are the weight coefficients and length scale parameter of the non-linear regression model.

The method implements the recently proposed machine learning access to atomic forces as discussed extensively in the following publications - *(Botu1)* and *(Botu2)*. The premise of the method is to map the atomic environment numerically into a fingerprint, and use machine learning methods to create a mapping to the vectorial atomic forces.

Only a single pair_coeff command is used with the *agni* style which specifies an AGNI potential file containing the parameters of the force field for the needed elements. These are mapped to LAMMPS atom types by specifying $N$ additional arguments after the filename in the pair_coeff command, where $N$ is the number of LAMMPS atom types:

- filename
- $N$ element names = mapping of AGNI elements to atom types

See the *pair_coeff* page for alternate ways to specify the path for the force field file.

An AGNI force field is fully specified by the filename which contains the parameters of the force field, i.e., the reference training environments used to construct the machine learning force field. Example force field and input files are provided in the examples/PACKAGES/agni directory.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.2.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.2.5 Restrictions

Currently, only elemental systems are implemented. Also, the method only provides access to the forces and not energies or stresses. The lack of potential energy data makes this pair style incompatible with several of the *minimizer algorthms* like *cg* or *sd*. It should work with damped dynamics based minimizers like *fire* or *quickmin*. However, one can access the energy via thermodynamic integration of the forces as discussed in *(Botu3)*. This pair style is part of the MISC package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The AGNI force field files provided with LAMMPS (see the potentials directory) are parameterized for metal *units*. You can use the AGNI potential with any LAMMPS units, but you would need to create your own AGNI potential file with coefficients listed in the appropriate units if your simulation does not use "metal" units.

### 4.2.6 Related commands

*pair_coeff*

### 4.2.7 Default

none

---

**(Botu1)** V. Botu and R. Ramprasad, Int. J. Quant. Chem., 115(16), 1074 (2015).

**(Botu2)** V. Botu and R. Ramprasad, Phys. Rev. B, 92(9), 094306 (2015).

**(Botu3)** V. Botu, R. Batra, J. Chapman and R. Ramprasad, https://arxiv.org/abs/1610.02098 (2016).

## 4.3 pair_style aip/water/2dm command

Accelerator Variants: *aip/water/2dm/opt*

### 4.3.1 Syntax

```
pair_style [hybrid/overlay ...] aip/water/2dm cutoff tap_flag
```

- cutoff = global cutoff (distance units)
- tap_flag = 0/1 to turn off/on the taper function

### 4.3.2 Examples

```
pair_style  hybrid/overlay aip/water/2dm 16.0 1
pair_coeff  * * aip/water/2dm  CBNOH.aip.water.2dm C Ow Hw

pair_style  hybrid/overlay aip/water/2dm 16.0 lj/cut/tip4p/long 2 3 1 1 0.1546 10 8.5
pair_coeff  2 2   lj/cut/tip4p/long    8.0313e-3  3.1589      # O-O
pair_coeff  2 3   lj/cut/tip4p/long    0.0        0.0         # O-H
pair_coeff  3 3   lj/cut/tip4p/long    0.0        0.0         # H-H
pair_coeff  * *   aip/water/2dm        CBNOH.aip.water.2dm    C Ow Hw

pair_style  hybrid/overlay aip/water/2dm 16.0 lj/cut/tip4p/long 3 4 1 1 0.1546 10 8.5␣
↪coul/shield 16.0 1
pair_coeff  1*2 1*2   none
pair_coeff  3 3   lj/cut/tip4p/long    8.0313e-3  3.1589      # O-O
pair_coeff  3 4   lj/cut/tip4p/long    0.0        0.0         # O-H
pair_coeff  4 4   lj/cut/tip4p/long    0.0        0.0         # H-H
pair_coeff  * *   aip/water/2dm        CBNOH.aip.water.2dm  B N Ow Hw
pair_coeff  1 3   coul/shield          1.333
pair_coeff  1 4   coul/shield          1.333
pair_coeff  2 3   coul/shield          1.333
pair_coeff  2 4   coul/shield          1.333
```

### 4.3.3 Description

Added in version 15Jun2023.

The *aip/water/2dm* style computes the anisotropic interfacial potential (AIP) potential for interfaces of water with two-dimensional (2D) materials as described in *(Feng1)* and *(Feng2)*.

$$
\begin{aligned}
E =& \frac{1}{2}\sum_i \sum_{j \neq i} V_{ij} \\
V_{ij} =& \mathrm{Tap}(r_{ij}) \left\{ e^{-\alpha(r_{ij}/\beta - 1)} \left[\varepsilon + f(\rho_{ij}) + f(\rho_{ji})\right] - \frac{1}{1 + e^{-d\left[(r_{ij}/(s_R \cdot r^{eff})) - 1\right]}} \cdot \frac{C_6}{r_{ij}^6} \right\} \\
\rho_{ij}^2 =& r_{ij}^2 - (\mathbf{r}_{ij} \cdot \mathbf{n}_i)^2 \\
\rho_{ji}^2 =& r_{ij}^2 - (\mathbf{r}_{ij} \cdot \mathbf{n}_j)^2 \\
f(\rho) =& C e^{-(\rho/\delta)^2} \\
\mathrm{Tap}(r_{ij}) =& 20 \left(\frac{r_{ij}}{R_{cut}}\right)^7 - 70 \left(\frac{r_{ij}}{R_{cut}}\right)^6 + 84 \left(\frac{r_{ij}}{R_{cut}}\right)^5 - 35 \left(\frac{r_{ij}}{R_{cut}}\right)^4 + 1
\end{aligned}
$$

Where $\mathrm{Tap}(r_{ij})$ is the taper function which provides a continuous cutoff (up to third derivative) for interatomic separations larger than $r_c$ *pair_style ilp_graphene_hbn*.

**Note**

This pair style uses the atomic normal vector definition from *(Feng1)*, where the atomic normal vectors of the hydrogen atoms are assumed to lie along the corresponding oxygen-hydrogen bonds and the normal vector of the central oxygen atom is defined as their average.

The provided parameter file, `CBNOH.aip.water.2dm`, is intended for use with *metal* *units*, with energies in meV. Two additional parameters, *S*, and *rcut* are included in the parameter file. *S* is designed to facilitate scaling of energies; *rcut* is the cutoff for an internal, short distance neighbor list that is generated for speeding up the calculation of the normals for all atom pairs.

**Note**

The parameters presented in the provided parameter file, `CBNOH.aip.water.2dm`, are fitted with the taper function enabled by setting the cutoff equal to 16.0 Angstrom. Using a different cutoff or taper function setting should be carefully checked as they can lead to significant errors. These parameters provide a good description in both short- and long-range interaction regimes. This is essential for simulations in high pressure regime (i.e., the interlayer distance is smaller than the equilibrium distance).

This potential must be used in combination with hybrid/overlay. Other interactions can be set to zero using *pair_coeff settings* with the pair style set to *none*.

This pair style tallies a breakdown of the total interlayer potential energy into sub-categories, which can be accessed via the *compute pair* command as a vector of values of length 2. The 2 values correspond to the following sub-categories:

1. *E_vdW* = vdW (attractive) energy
2. *E_Rep* = Repulsive energy

To print these quantities to the log file (with descriptive column headings) the following commands could be included in an input script:

```
compute 0 all pair aip/water/2dm
variable Evdw  equal c_0[1]
variable Erep  equal c_0[2]
thermo_style custom step temp epair v_Erep v_Evdw
```

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.3.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support the pair_modify mix, shift, table, and tail options.

This pair style does not write their information to binary restart files, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

### 4.3.5 Restrictions

This pair style is part of the INTERLAYER package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires the newton setting to be *on* for pair interactions.

The `CBNOH.aip.water.2dm` potential file provided with LAMMPS is parameterized for *metal* units. You can use this pair style with any LAMMPS units, but you would need to create your own potential file with parameters in the appropriate units, if your simulation does not use *metal* units.

### 4.3.6 Related commands

*pair_coeff*, *pair_none*, *pair_style hybrid/overlay*, *pair_style drip*, *pair_style ilp_tmd*, *pair_style saip_metal*, *pair_style ilp_graphene_hbn*, *pair_style pair_kolmogorov_crespi_z*, *pair_style pair_kolmogorov_crespi_full*, *pair_style pair_lebedeva_z*, *pair_style pair_coul_shield*.

### 4.3.7 Default

tap_flag = 1

---

**(Feng1)** Z. Feng, . . . , and W. Ouyang, J. Phys. Chem. C. 127(18), 8704-8713 (2023).

**(Feng2)** Z. Feng, . . . , and W. Ouyang, Langmuir 39(50), 18198-18207 (2023).

## 4.4 pair_style airebo command

Accelerator Variants: *airebo/intel*, *airebo/omp*

## 4.5 pair_style airebo/morse command

Accelerator Variants: *airebo/morse/intel*, *airebo/morse/omp*

## 4.6 pair_style airebo/bc command

Accelerator Variants: *airebo/bc/omp*

## 4.7 pair_style rebo command

Accelerator Variants: *rebo/intel*, *rebo/omp*

### 4.7.1 Syntax

```
pair_style style cutoff LJ_flag TORSION_flag cutoff_min
```

- style = *airebo* or *airebo/morse* or *airebo/bc* or *rebo*
- cutoff = LJ or Morse cutoff ($\sigma$ scale factor) (AIREBO, AIREBO-M, and AIREBO-BC only)
- LJ_flag = 0/1 to turn off/on the LJ or Morse term (AIREBO, AIREBO-M, and AIREBO-BC only, optional)
- TORSION_flag = 0/1 to turn off/on the torsion term (AIREBO, AIREBO-M, and AIREBO-BC only, optional)
- cutoff_min = Start of the transition region of cutoff ($\sigma$ scale factor) (AIREBO, AIREBO-M, and AIREBO-BC only, optional)

### 4.7.2 Examples

```
pair_style airebo 3.0
pair_style airebo 2.5 1 0
pair_coeff * * ../potentials/CH.airebo H C

pair_style airebo/morse 3.0
```

(continues on next page)

(continued from previous page)

```
pair_coeff * * ../potentials/CH.airebo-m H C

pair_style airebo/bc 3.0
pair_coeff * * ../potentials/CH.airebo-bc H C

pair_style rebo
pair_coeff * * ../potentials/CH.rebo H C
```

### 4.7.3 Description

The *airebo* pair style computes the Adaptive Intermolecular Reactive Empirical Bond Order (AIREBO) Potential of *(Stuart)* for a system of carbon and/or hydrogen atoms. Note that this is the initial formulation of AIREBO from 2000, not the later formulation.

The *airebo/morse* pair style computes the AIREBO-M potential, which is equivalent to AIREBO, but replaces the LJ term with a Morse potential. The Morse potentials are parameterized by high-quality quantum chemistry (MP2) calculations and do not diverge as quickly as particle density increases. This allows AIREBO-M to retain accuracy to much higher pressures than AIREBO (up to 40 GPa for Polyethylene). Details for this potential and its parameterization are given in *(O'Conner)*.

Added in version 4Jul2026.

The *airebo/bc* pair style computes the bond-centric modification of AIREBO described in *(Hur)*. It is equivalent to AIREBO except that the $P_{ij}$ coordination correction to the bond order is made bond-centric rather than atom-centric. In AIREBO the $P_{ij}$ term depends only on the number and type of atoms bonded to atom $i$; in AIREBO-BC the $P_{CC}$ term instead depends on the bond-averaged coordination numbers

$$\bar{N}_{ij}^{t} = \frac{1}{2}\left(N_{ij}^{t} + N_{ji}^{t}\right)$$

so that the correction reflects the local environment on both sides of the bond. Because stable molecules can then take half-integer coordination values, $P_{CC}$ is tabulated on a half-integer spline grid and re-parameterized accordingly (Table III of *(Hur)*). This improves the description of heterogeneous bonding environments, such as the sp/sp2 boundaries found in fullerene fragments and in linear/graphitic carbon, where the atom-centric average is qualitatively inaccurate. All other terms of the potential (the angular, radical, and dihedral contributions to the bond order, the LJ term, and the torsion term) are identical to AIREBO. Evaluating the denser bond-centric spline makes *airebo/bc* somewhat more expensive than *airebo*.

The *rebo* pair style computes the Reactive Empirical Bond Order (REBO) Potential of *(Brenner)*. Note that this is the so-called second generation REBO from 2002, not the original REBO from 1990. As discussed below, second generation REBO is closely related to the initial AIREBO; it is just a subset of the potential energy terms with a few slightly different parameters

The AIREBO potential consists of three terms:

$$E = \frac{1}{2}\sum_{i}\sum_{j\neq i}\left[E_{ij}^{\mathrm{REBO}} + E_{ij}^{\mathrm{LJ}} + \sum_{k\neq i,j}\sum_{l\neq i,j,k} E_{kijl}^{\mathrm{TORSION}}\right]$$

By default, all three terms are included. For the *airebo* style, if the first two optional flag arguments to the pair_style command are included, the LJ and torsional terms can be turned off. Note that both or neither of the flags must be included. If both of the LJ an torsional terms are turned off, it becomes the second-generation REBO potential, with a small caveat on the spline fitting procedure mentioned below. This can be specified directly as pair_style *rebo* with no additional arguments.

The detailed formulas for this potential are given in *(Stuart)*; here we provide only a brief description.

The $E^{\mathrm{REBO}}$ term has the same functional form as the hydrocarbon REBO potential developed in *(Brenner)*. The coefficients for $E^{\mathrm{REBO}}$ in AIREBO are essentially the same as Brenner's potential, but a few fitted spline values are slightly

different. For most cases the $E^{\mathrm{REBO}}$ term in AIREBO will produce the same energies, forces and statistical averages as the original REBO potential from which it was derived. The $E^{\mathrm{REBO}}$ term in the AIREBO potential gives the model its reactive capabilities and only describes short-ranged C-C, C-H and H-H interactions ($r < 2$Å). These interactions have strong coordination-dependence through a bond order parameter, which adjusts the attraction between the I,J atoms based on the position of other nearby atoms and thus has 3- and 4-body dependence.

The $E^{\mathrm{LJ}}$ term adds longer-ranged interactions ($2 < r <$ cutoff) using a form similar to the standard *Lennard Jones potential*. The $E^{\mathrm{LJ}}$ term in AIREBO contains a series of switching functions so that the short-ranged LJ repulsion ($1/r^{12}$) does not interfere with the energetics captured by the $E^{\mathrm{REBO}}$ term. The extent of the $E^{\mathrm{LJ}}$ interactions is determined by the *cutoff* argument to the pair_style command which is a scale factor. For each type pair (C-C, C-H, H-H) the cutoff is obtained by multiplying the scale factor by the sigma value defined in the potential file for that type pair. In the standard AIREBO potential, $\sigma_{CC}$ = 3.4Å, so with a scale factor of 3.0 (the argument in pair_style), the resulting $E^{\mathrm{LJ}}$ cutoff would be 10.2Å.

By default, the longer-ranged interaction is smoothly switched off between 2.16 and 3.0 $\sigma$. By specifying *cutoff_min* in addition to *cutoff*, the switching can be configured to take place between *cutoff_min* and *cutoff*. *cutoff_min* can only be specified if all optional arguments are given.

The $E^{\mathrm{TORSION}}$ term is an explicit 4-body potential that describes various dihedral angle preferences in hydrocarbon configurations.

---

Only a single pair_coeff command is used with the *airebo*, *airebo/morse*, *airebo/bc*, or *rebo* style which specifies an AIREBO, AIREBO-M, AIREBO-BC, or REBO potential file with parameters for C and H. Note that as of LAMMPS version 15 May 2019 the *rebo* style in LAMMPS uses its own potential file (CH.rebo). These are mapped to LAMMPS atom types by specifying N additional arguments after the filename in the pair_coeff command, where N is the number of LAMMPS atom types:

- filename
- *N* element names = mapping of AIREBO elements to atom types

See the *pair_coeff* page for alternate ways to specify the path for the potential file.

As an example, if your LAMMPS simulation has 4 atom types and you want the first 3 to be C, and the fourth to be H, you would use the following pair_coeff command:

```
pair_coeff * * CH.airebo C C C H
```

The first 2 arguments must be * * so as to span all LAMMPS atom types. The first three C arguments map LAMMPS atom types 1,2,3 to the C element in the AIREBO file. The final H argument maps LAMMPS atom type 4 to the H element in the AIREBO file. If a mapping value is specified as NULL, the mapping is not performed. This can be used when a *airebo* potential is used as part of the *hybrid* pair style. The NULL values are placeholders for atom types that will be used with other potentials.

The parameters/coefficients for the AIREBO potentials are listed in the CH.airebo file to agree with the original *(Stuart)* paper. Thus the parameters are specific to this potential and the way it was fit, so modifying the file should be done cautiously.

Similarly the parameters/coefficients for the AIREBO-M potentials are listed in the CH.airebo-m file to agree with the *(O'Connor)* paper. Thus the parameters are specific to this potential and the way it was fit, so modifying the file should be done cautiously. The AIREBO-M Morse potentials were parameterized using a cutoff of 3.0 ($\sigma$). Modifying this cutoff may impact simulation accuracy.

The parameters/coefficients for the AIREBO-BC potential are listed in the CH.airebo-bc file to agree with the *(Hur)* paper. This file is identical to CH.airebo through the standard parameter sections and appends the bond-centric $P_{CC}$ spline knots of Table III; as such it can also be read by the *airebo* style. As with the other variants, the parameters are specific to this potential and the way it was fit, so modifying the file should be done cautiously.

This pair style tallies a breakdown of the total AIREBO potential energy into sub-categories, which can be accessed via the *compute pair* command as a vector of values of length 3. The 3 values correspond to the following sub-categories:

1. $E_{\mathrm{REBO}}$ = REBO energy
2. $E_{\mathrm{LJ}}$ = Lennard-Jones energy
3. $E_{\mathrm{TORSION}}$ = Torsion energy

To print these quantities to the log file (with descriptive column headings) the following commands could be included in an input script:

```
compute 0 all pair airebo
variable REBO     equal c_0[1]
variable LJ       equal c_0[2]
variable TORSION  equal c_0[3]
thermo_style custom step temp epair v_REBO v_LJ v_TORSION
```

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.7.4 Mixing, shift, table, tail correction, restart, rRESPA info

These pair styles do not support the *pair_modify* mix, shift, table, and tail options.

These pair styles do not write their information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

These pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

### 4.7.5 Restrictions

These pair styles are part of the MANYBODY package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

These pair potentials require the *newton* setting to be "on" for pair interactions.

The CH.airebo, CH.airebo-m, and CH.airebo-bc potential files provided with LAMMPS (see the potentials directory) are parameterized for metal *units*. You can use the pair styles with *any* LAMMPS units, but you would need to create your own AIREBO, AIREBO-M, or AIREBO-BC potential file with coefficients listed in the appropriate units, if your simulation does not use "metal" units.

The pair styles provided here **only** support potential files parameterized for the elements carbon and hydrogen (designated with "C" and "H" in the *pair_coeff* command. Using potential files for other elements will trigger an error.

### 4.7.6 Related commands

*pair_coeff*

### 4.7.7 Default

none

---

**(Stuart)** Stuart, Tutein, Harrison, J Chem Phys, 112, 6472-6486 (2000).

**(Brenner)** Brenner, Shenderova, Harrison, Stuart, Ni, Sinnott, J Physics: Condensed Matter, 14, 783-802 (2002).

**(O'Connor)** O'Connor et al., J. Chem. Phys. 142, 024903 (2015).

**(Hur)** Hur and Stuart, J. Chem. Phys. 137, 054102 (2012).

## 4.8 pair_style amoeba command

Accelerator Variants: *amoeba/gpu*

## 4.9 pair_style hippo command

Accelerator Variants: *hippo/gpu*

### 4.9.1 Syntax

```
pair_style style
```

- style = *amoeba* or *hippo*

### 4.9.2 Examples

```
pair_style amoeba
pair_coeff * * protein.prm.amoeba protein.key.amoeba
```

```
pair_style hippo
pair_coeff * * water.prm.hippo water.key.hippo
```

### 4.9.3 Additional info

- *Howto amoeba*
- examples/amoeba
- tools/amoeba
- potentials/*.amoeba
- potentials/*.hippo

### 4.9.4 Description

The *amoeba* style computes the AMOEBA polarizable field formulated by Jay Ponder's group at the U Washington at St Louis *(Ren)*, *(Shi)*. The *hippo* style computes the HIPPO polarizable force field, an extension to AMOEBA, formulated by Josh Rackers and collaborators in the Ponder group *(Rackers)*.

These force fields can be used when polarization effects are desired in simulations of water, organic molecules, and biomolecules including proteins, provided that parameterizations (Tinker PRM force field files) are available for the systems you are interested in. Files in the LAMMPS potentials directory with a "amoeba" or "hippo" suffix can be used. The Tinker distribution and website have additional force field files as well.

As discussed on the *Howto amoeba* doc page, the intermolecular (non-bonded) portion of the AMOEBA force field contains these terms:

$$U_{amoeba} = U_{multipole} + U_{polar} + U_{hal}$$

while the HIPPO force field contains these terms:

$$U_{hippo} = U_{multipole} + U_{polar} + U_{qxfer} + U_{repulsion} + U_{dispersion}$$

Conceptually, these terms compute the following interactions:

- $U_{hal}$ = buffered 14-7 van der Waals with offsets applied to hydrogen atoms
- $U_{repulsion}$ = Pauli repulsion due to rearrangement of electron density
- $U_{dispersion}$ = dispersion between correlated, instantaneous induced dipole moments
- $U_{multipole}$ = electrostatics between permanent point charges, dipoles, and quadrupoles
- $U_{polar}$ = electronic polarization between induced point dipoles
- $U_{qxfer}$ = charge transfer effects

Note that the AMOEBA versus HIPPO force fields typically compute the same term differently using their own formulas. The references on this doc page give full details for both force fields.

The formulas for the AMOEBA energy terms are:

$$\begin{aligned}
U_{hal} =& \varepsilon_{ij} \left( \frac{1.07}{\rho_{ij} + 0.07} \right)^7 \left( \frac{1.12}{\rho_{ij}^7 + 0.12} - 2 \right) \\
U_{multipole} =& \vec{M}_i T_{ij} \vec{M}_j, \quad \text{with} \quad \vec{M} = \left( q, \vec{\mu}_{perm}, \Theta \right) \\
U_{polar} =& \frac{1}{2} \vec{\mu}_i^{ind} \vec{E}_i^{perm}
\end{aligned}$$

The formulas for the HIPPO energy terms are:

$$\begin{aligned}
U_{multipole} =& Z_i \frac{1}{r_{ij}} Z_j + Z_i T_{ij}^{damp} \vec{M}_j + Z_j T_{ji}^{damp} \vec{M}_i + \vec{M}_i T_{ij}^{damp} \vec{M}_j, \quad \text{with} \quad \vec{M} = \left( q, \vec{\mu}_{perm}, \Theta \right) \\
U_{polar} =& \frac{1}{2} \vec{\mu}_i^{ind} \vec{E}_i^{perm} \\
U_{qxfer} =& \varepsilon_i e^{-\eta_j r_{ij}} + \varepsilon_j e^{-\eta_i r_{ij}} \\
U_{repulsion} =& \frac{K_i K_j}{r_{ij}} S^2 S^2 = \left( \int \phi_i \phi_j dv \right)^2 = \vec{M}_i T_{ij}^{repulsion} \vec{M}_j \\
U_{dispersion} =& - \frac{C_6^i C_6^j}{r_{ij}^6} \left( f_{damp}^{dispersion} \right)_{ij}^2
\end{aligned}$$

**Note**

The AMOEBA and HIPPO force fields compute long-range charge, dipole, and quadrupole interactions as well as long-range dispersion effects. However, unlike other models with long-range interactions in LAMMPS, this does not require use of a KSpace style via the *kspace_style* command. That is because for AMOEBA and HIPPO the long-range computations are intertwined with the pairwise computations. So these pair style include both short- and long-range computations. This means the energy and virial computed by the pair style as well as the "Pair" timing reported by LAMMPS will include the long-range calculations.

The implementation of the AMOEBA and HIPPO force fields in LAMMPS was done using F90 code provided by the Ponder group from their Tinker MD code.

The current implementation (July 2022) of AMOEBA in LAMMPS matches the version discussed in *(Ponder)*, *(Ren)*, and *(Shi)*. Likewise the current implementation of HIPPO in LAMMPS matches the version discussed in *(Rackers)*.

Added in version 8Feb2023.

Accelerator support via the GPU package is available.

---

Only a single pair_coeff command is used with either the *amoeba* and *hippo* styles which specifies two Tinker files, a PRM and KEY file.

```
pair_coeff * * ../potentials/protein.prm.amoeba ../potentials/protein.key.amoeba
pair_coeff * * ../potentials/water.prm.hippo ../potentials/water.key.hippo
```

Examples of the PRM files are in the potentials directory with an *.amoeba or *.hippo suffix. The examples/amoeba directory has examples of both PRM and KEY files.

A Tinker PRM file is composed of sections, each of which has multiple lines. A Tinker KEY file is composed of lines, each of which has a keyword followed by zero or more parameters.

The list of PRM sections and KEY keywords which LAMMPS recognizes are listed on the *Howto amoeba* doc page. If not recognized, the section or keyword is skipped.

Note that if the KEY file is specified as NULL, then no file is required; default values for various AMOEBA/HIPPO settings are used. The *Howto amoeba* doc page also gives the default settings.

---

Added in version 3Nov2022.

The *amoeba* and *hippo* pair styles support extraction of two per-atom quantities by the *fix pair* command. This allows the quantities to be output to files by the *dump* or otherwise processed by other LAMMPS commands.

The names of the two quantities are "uind" and "uinp" for the induced dipole moments for each atom. Neither quantity needs to be triggered by the *fix pair* command in order for these pair styles to calculate it.

---

### 4.9.5 Mixing, shift, table, tail correction, restart, rRESPA info

These pair styles do not support the *pair_modify* mix, shift, table, and tail options.

These pair styles do not write their information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

These pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

**Note**

Using the GPU accelerated pair styles 'amoeba/gpu' or 'hippo/gpu' when compiling the GPU package for OpenCL has a few known issues when running on integrated GPUs and the calculation may crash.

The GPU accelerated pair styles are also not (yet) compatible with single precision FFTs.

### 4.9.6 Restrictions

These pair styles are part of the AMOEBA package. They are only enabled if LAMMPS was built with that package. See the *Build package* doc page for more info.

The AMOEBA and HIPPO potential (PRM) and KEY files provided with LAMMPS in the potentials and examples/amoeba directories are Tinker files parameterized for Tinker units. Their numeric parameters are converted by LAMMPS to its real units *units*. Thus you can only use these pair styles with real units.

These potentials do not yet calculate per-atom energy or virial contributions.

As explained on the *AMOEBA and HIPPO howto* page, use of these pair styles to run a simulation with the AMOEBA or HIPPO force fields requires several things.

The first is a data file generated by the tools/tinker/tinker2lmp.py conversion script which uses Tinker file force field file input to create a data file compatible with LAMMPS.

The second is use of these commands:

- *atom_style amoeba*
- *fix property/atom*
- *special_bonds one/five*

And third, depending on the model being simulated, these commands for intramolecular interactions may also be required:

- *bond_style class2*
- *angle_style amoeba*
- *dihedral_style fourier*
- *improper_style amoeba*
- *fix amoeba/pitorsion*
- *fix amoeba/bitorsion*

### 4.9.7 Related commands

*atom_style amoeba*, *bond_style class2*, *angle_style amoeba*, *dihedral_style fourier*, *improper_style amoeba*, *fix amoeba/pitorsion*, *fix amoeba/bitorsion*, *special_bonds one/five*, *fix property/atom*

### 4.9.8 Default

none

---

**(Ponder)** Ponder, Wu, Ren, Pande, Chodera, Schnieders, Haque, Mobley, Lambrecht, DiStasio Jr, M. Head-Gordon, Clark, Johnson, T. Head-Gordon, J Phys Chem B, 114, 2549-2564 (2010).

**(Rackers)** Rackers, Silva, Wang, Ponder, J Chem Theory Comput, 17, 7056-7084 (2021).

**(Ren)** Ren and Ponder, J Phys Chem B, 107, 5933 (2003).

**(Shi)** Shi, Xia, Zhang, Best, Wu, Ponder, Ren, J Chem Theory Comp, 9, 4046, 2013.

## 4.10 pair_style atm command

Accelerator Variants: *atm/omp*

### 4.10.1 Syntax

```
pair_style atm cutoff cutoff_triple
```

- cutoff = cutoff for each pair in 3-body interaction (distance units)
- cutoff_triple = additional cutoff applied to product of 3 pairwise distances (distance units)

### 4.10.2 Examples

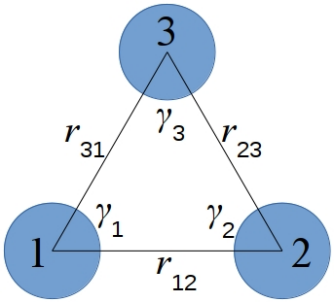
```
pair_style atm 4.5 2.5
pair_coeff * * * 0.072

pair_style hybrid/overlay lj/cut 6.5 atm 4.5 2.5
pair_coeff * * lj/cut 1.0 1.0
pair_coeff 1 1 atm 1 0.064
pair_coeff 1 1 atm 2 0.080
pair_coeff 1 2 atm 2 0.100
pair_coeff 2 2 atm 2 0.125
```

### 4.10.3 Description

The *atm* style computes a 3-body *Axilrod-Teller-Muto* potential for the energy E of a system of atoms as

$$E = \nu \frac{1 + 3\cos\gamma_1 \cos\gamma_2 \cos\gamma_3}{r_{12}^3 r_{23}^3 r_{31}^3}$$

where $\nu$ is the three-body interaction strength. The distances between pairs of atoms $r_{12}$, $r_{23}$, $r_{31}$ and the angles $\gamma_1$, $\gamma_2$, $\gamma_3$ are as shown in this diagram:




Note that for the interaction between a triplet of atoms $I, J, K$, there is no "central" atom. The interaction is symmetric with respect to permutation of the three atoms. Thus the $\nu$ value is the same for all those permutations of the atom types of $I, J, K$ and needs to be specified only once, as discussed below.

The *atm* potential is typically used in combination with a two-body potential using the *pair_style hybrid/overlay* command as in the example above.

The potential for a triplet of atom is calculated only if all 3 distances $r_{12}$, $r_{23}$, $r_{31}$ between the three atoms satisfy $r_{IJ} <$ cutoff. In addition, the product of the 3 distances $r_{12}r_{23}r_{31}$ < cutoff_triple $^3$ is required, which excludes from calculation the triplets with small contribution to the interaction.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the restart files read by the *read_restart* commands:

- $K$ = atom type of the third atom (1 to $N_{\text{types}}$)
- $\nu$ = prefactor (energy/distance^9 units)

$K$ can be specified in one of two ways. An explicit numeric value or type label can be used, as in the second example above. LAMMPS sets the coefficients for the other 5 symmetric interactions to the same values. E.g. if $I = 1$, $J = 2$, $K = 3$, then these 6 values are set to the specified $\nu$: $\nu_{123}$, $\nu_{132}$, $\nu_{213}$, $\nu_{231}$, $\nu_{312}$, $\nu_{321}$. This enforces the symmetry discussed above.

A wildcard asterisk can be used for K to set the coefficients for multiple triplets of atom types. This takes the form "*" or "*n" or "n*" or "m*n". If $N$ equals the number of atom types, then an asterisk with no numeric values means all types from 1 to $N$. A leading asterisk means all types from 1 to $n$ (inclusive). A trailing asterisk means all types from $n$ to $N$ (inclusive). A middle asterisk means all types from $m$ to $n$ (inclusive). Note that only type triplets with $J \leq K$ are considered; if asterisks imply type triplets where $K < J$, they are ignored.

Note that a pair_coeff command can override a previous setting for the same $I, J, K$ triplet. For example, these commands set $\nu$ for all $I, J.K$ triplets, then overwrite nu for just the $I, J, K = 2, 3, 4$ triplet:

```
pair_coeff * * * 0.25
pair_coeff 2 3 4 0.1
```

Note that for a simulation with a single atom type, only a single entry is required, e.g.

```
pair_coeff 1 1 1 0.25
```

For a simulation with two atom types, four pair_coeff commands will specify all possible nu values:

```
pair_coeff 1 1 1 nu1
pair_coeff 1 1 2 nu2
pair_coeff 1 2 2 nu3
pair_coeff 2 2 2 nu4
```

For a simulation with three atom types, ten pair_coeff commands will specify all possible nu values:

```
pair_coeff 1 1 1 nu1
pair_coeff 1 1 2 nu2
pair_coeff 1 1 3 nu3
pair_coeff 1 2 2 nu4
pair_coeff 1 2 3 nu5
pair_coeff 1 3 3 nu6
pair_coeff 2 2 2 nu7
pair_coeff 2 2 3 nu8
pair_coeff 2 3 3 nu9
pair_coeff 3 3 3 nu10
```

By default the $\nu$ value for all triplets is set to 0.0. Thus it is not required to provide pair_coeff commands that enumerate triplet interactions for all $K$ types. If some $I, J, K$ combination is not specified, then there will be no 3-body ATM interactions for that combination and all its permutations. However, as with all pair styles, it is required to specify a pair_coeff command for all $I, J$ combinations, else an error will result.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.10.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style do not support the *pair_modify* mix, shift, table, and tail options.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file. However, if the *atm* potential is used in combination with other potentials using the *pair_style hybrid/overlay* command then pair_coeff commands need to be re-specified in the restart input script.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, and *outer* keywords.

---

### 4.10.5 Restrictions

This pair style is part of the MANYBODY package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.10.6 Related commands

*pair_coeff*

### 4.10.7 Default

none

---

**(Axilrod)** Axilrod and Teller, J Chem Phys, 11, 299 (1943); Muto, Nippon Sugaku-Buturigakkwaishi 17, 629 (1943).

## 4.11 pair_style beck command

Accelerator Variants: *beck/gpu*, *beck/kk*, *beck/omp*

### 4.11.1 Syntax

```
pair_style beck Rc
```

- Rc = cutoff for interactions (distance units)

### 4.11.2 Examples

```
pair_style beck 8.0
pair_coeff * * 399.671876712 0.0000867636112694 0.675 4.390 0.0003746
pair_coeff 1 1 399.671876712 0.0000867636112694 0.675 4.390 0.0003746 6.0
```

### 4.11.3 Description

Style *beck* computes interactions based on the potential by *(Beck)*, originally designed for simulation of Helium. It includes truncation at a cutoff distance $r_c$.

$$E(r) = A \exp\left[-\alpha r - \beta r^6\right] - \frac{B}{\left(r^2+a^2\right)^3}\left(1 + \frac{2.709 + 3a^2}{r^2+a^2}\right) \qquad r < r_c$$

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands.

- $A$ (energy units)
- $B$ (energy-distance^6 units)
- $a$ (distance units)
- $\alpha$ (1/distance units)
- $\beta$ (1/distance^6 units)
- cutoff (distance units)

The last coefficient is optional. If not specified, the global cutoff $r_c$ is used.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.11.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, coefficients must be specified. No default mixing rules are used.

This pair style does not support the *pair_modify* shift option for the energy of the pair interaction.

The *pair_modify* table option is not relevant for this pair style.

This pair style does not support the *pair_modify* tail option for adding long-range tail corrections.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.11.5 Restrictions

This pair style is part of the EXTRA-PAIR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.11.6 Related commands

*pair_coeff*

### 4.11.7 Default

none

---

**(Beck)** Beck, Molecular Physics, 14, 311 (1968).

## 4.12 pair_style body/nparticle command

Accelerator Variants: *body/nparticle/omp*

### 4.12.1 Syntax

```
pair_style body/nparticle cutoff
```

cutoff = global cutoff for interactions (distance units)

### 4.12.2 Examples

```
pair_style body/nparticle 3.0
pair_coeff * * 1.0 1.0
pair_coeff 1 1 1.0 1.5 2.5
```

### 4.12.3 Description

Style *body/nparticle* is for use with body particles and calculates pairwise body/body interactions as well as interactions between body and point-particles. See the *Howto body* doc page for more details on using body particles.

This pair style is designed for use with the "nparticle" body style, which is specified as an argument to the "atom-style body" command. See the *Howto body* page for more details about the body styles LAMMPS supports. The "nparticle" style treats a body particle as a rigid body composed of N sub-particles.

The coordinates of a body particle are its center-of-mass (COM). If the COMs of a pair of body particles are within the cutoff (global or type-specific, as specified above), then all interactions between pairs of sub-particles in the two body particles are computed. E.g. if the first body particle has 3 sub-particles, and the second has 10, then 30 interactions are computed and summed to yield the total force and torque on each body particle.

**Note**

In the example just described, all 30 interactions are computed even if the distance between a particular pair of sub-particles is greater than the cutoff. Likewise, no interaction between two body particles is computed if the two COMs are further apart than the cutoff, even if the distance between some pairs of their sub-particles is within the cutoff. Thus care should be used in defining the cutoff distances for body particles, depending on their shape and size.

Similar rules apply for a body particle interacting with a point particle. The distance between the two particles is calculated using the COM of the body particle and the position of the point particle. If the distance is within the cutoff and the body particle has N sub-particles, then N interactions with the point particle are computed and summed. If the distance is not within the cutoff, no interactions between the body and point particle are computed.

The interaction between two sub-particles, or a sub-particle and point particle, or between two point particles is computed as a Lennard-Jones interaction, using the standard formula

$$E = 4\varepsilon \left[ \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \qquad r < R_c$$

where $R_c$ is the cutoff. As explained above, an interaction involving one or two body sub-particles may be computed even for $r > R_c$.

For style *body*, the following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $\varepsilon$ (energy units)
- $\sigma$ (distance units)
- cutoff (distance units)

The last coefficient is optional. If not specified, the global cutoff is used.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 4.12.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon and sigma coefficients and cutoff distance for all of this pair style can be mixed. The default mix value is *geometric*. See the *pair_modify* command for details.

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write its information to *binary restart files*.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.12.5 Restrictions

This style is part of the BODY package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Defining particles to be bodies so they participate in body/body or body/particle interactions requires the use of the *atom_style body* command.

### 4.12.6 Related commands

*pair_coeff*, *fix rigid*

### 4.12.7 Default

none

## 4.13 pair_style body/rounded/polygon command

Accelerator Variants: *body/rounded/polygon/omp*

### 4.13.1 Syntax

```
pair_style body/rounded/polygon c_n c_t mu delta_ua cutoff
```

```
c_n = normal damping coefficient
c_t = tangential damping coefficient
mu = normal friction coefficient during gross sliding
delta_ua = multiple contact scaling factor
cutoff = global separation cutoff for interactions (distance units), see below for␣
↪definition
```

### 4.13.2 Examples

```
pair_style body/rounded/polygon 20.0 5.0 0.0 1.0 0.5
pair_coeff * * 100.0 1.0
pair_coeff 1 1 100.0 1.0
```

### 4.13.3 Description

Style *body/rounded/polygon* is for use with 2d models of body particles of style *rounded/polygon*. It calculates pairwise body/body interactions which can include body particles modeled as 1-vertex circular disks with a specified diameter. See the *Howto body* page for more details on using body rounded/polygon particles.

This pairwise interaction between rounded polygons is described in *Fraige*, where a polygon does not have sharp corners, but is rounded at its vertices by circles centered on each vertex with a specified diameter. The edges of the polygon are defined between pairs of adjacent vertices. The circle diameter for each polygon is specified in the data file read by the *read data* command. This is a 2d discrete element model (DEM) which allows for multiple contact points.

Note that when two particles interact, the effective surface of each polygon particle is displaced outward from each of its vertices and edges by half its circle diameter (as in the diagram below of a gray and yellow square particle). The interaction forces and energies between two particles are defined with respect to the separation of their respective rounded surfaces, not by the separation of the vertices and edges themselves.

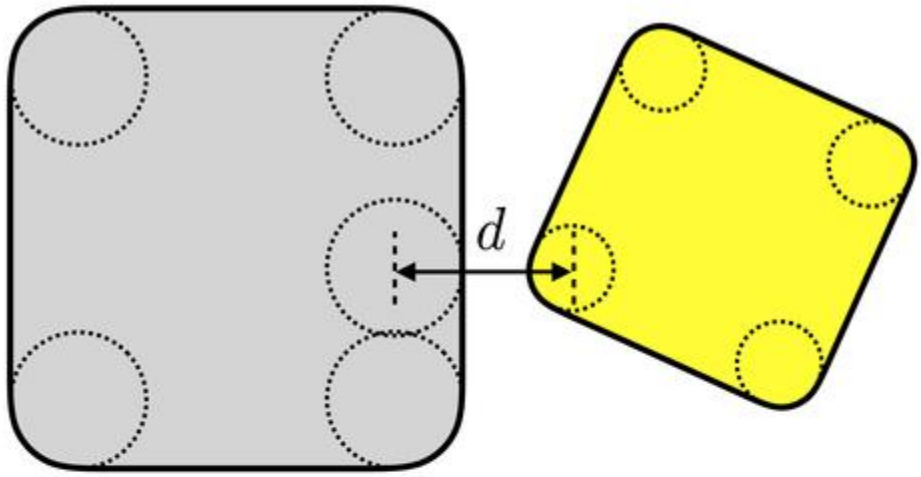
This means that the specified cutoff in the pair_style command is the cutoff distance, $r_c$, for the surface separation, $\delta_n$ (see figure below). This is the distance at which two particles no longer interact. If $r_c$ is specified as 0.0, then it is a contact-only interaction. I.e. the two particles must overlap in order to exert a repulsive force on each other. If $r_c > 0.0$, then the force between two particles will be attractive for surface separations from 0 to $r_c$, and repulsive once the particles overlap.

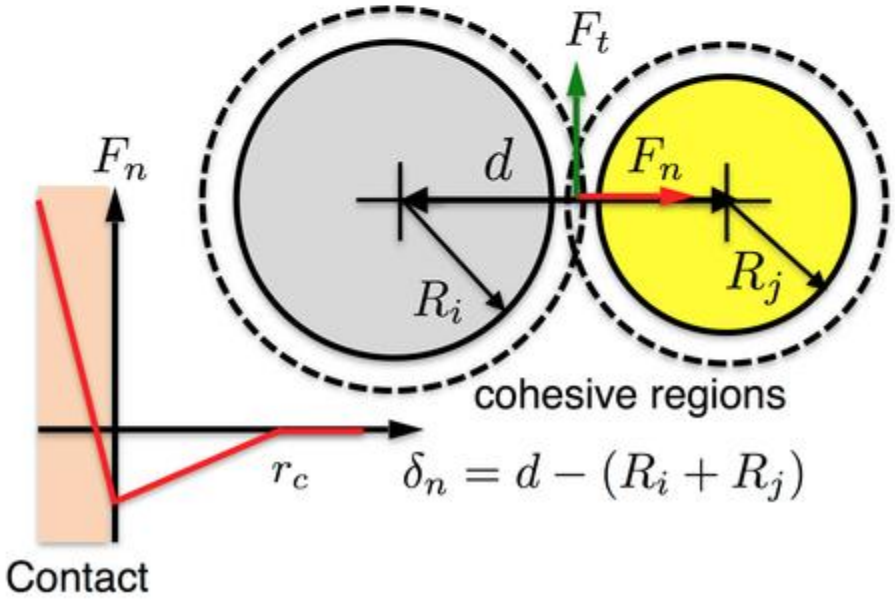
Note that unlike for other pair styles, the specified cutoff is not the distance between the centers of two particles at which they stop interacting. This center-to-center distance depends on the shape and size of the two particles and their relative orientation. LAMMPS takes that into account when computing the surface separation distance and applying the $r_c$ cutoff.

The forces between vertex-vertex, vertex-edge, and edge-edge overlaps are given by:

$$F_n = \begin{cases} k_n \delta_n - c_n v_n & \delta_n \leq 0 \\ -k_{na} \delta_n - c_n v_n & 0 < \delta_n \leq r_c \\ 0 & \delta_n > r_c \end{cases}$$

$$F_t = \begin{cases} \mu k_n \delta_n - c_t v_t & \delta_n \leq 0 \\ 0 & \delta_n > 0 \end{cases}$$




Note that $F_n$ and $F_t$ are functions of the surface separation $\delta_n = d - (R_i + R_j)$. In this model, when $(R_i + R_j) < d < (R_i + R_j) + r_c$, that is, $0 < \delta_n < r_c$, the cohesive region of the two surfaces overlap and the two surfaces are attractive to each other.

In *Fraige*, the tangential friction force between two particles that are in contact is modeled differently prior to gross sliding (i.e. static friction) and during gross-sliding (kinetic friction). The latter takes place when the tangential deformation exceeds the Coulomb frictional limit. In the current implementation, however, we do not take into account frictional history, i.e. we do not keep track of how many time steps the two particles have been in contact nor calculate the tangential deformation. Instead, we assume that gross sliding takes place as soon as two particles are in contact.

The following coefficients must be defined for each pair of atom types via the *pair_coeff* command as in the examples above, or in the data file read by the *read_data* command:

- $k_n$ (energy/distance^2 units)
- $k_{na}$ (energy/distance^2 units)

Effectively, $k_n$ and $k_{na}$ are the slopes of the red lines in the plot above for force versus surface separation, for $\delta_n < 0$ and $0 < \delta_n < r_c$ respectively.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.13.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support the *pair_modify* mix, shift, table, and tail options.

This pair style does not write its information to *binary restart files*. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.13.5 Restrictions

These pair styles are part of the BODY package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires the *newton* setting to be "on" for pair interactions.

### 4.13.6 Related commands

*pair_coeff*

### 4.13.7 Default

none

**(Fraige)** F. Y. Fraige, P. A. Langston, A. J. Matchett, J. Dodds, Particuology, 6, 455 (2008).

## 4.14 pair_style body/rounded/polyhedron command

Accelerator Variants: *body/rounded/polyhedron/omp*

### 4.14.1 Syntax

```
pair_style body/rounded/polyhedron c_n c_t mu delta_ua cutoff
```

```
c_n = normal damping coefficient
c_t = tangential damping coefficient
mu = normal friction coefficient during gross sliding
delta_ua = multiple contact scaling factor
cutoff = global separation cutoff for interactions (distance units), see below for␣
↪definition
```

### 4.14.2 Examples

```
pair_style body/rounded/polyhedron 20.0 5.0 0.0 1.0 0.5
pair_coeff * * 100.0 1.0
pair_coeff 1 1 100.0 1.0
```

### 4.14.3 Description

Style *body/rounded/polygon* is for use with 3d models of body particles of style *rounded/polyhedron*. It calculates pairwise body/body interactions which can include body particles modeled as 1-vertex spheres with a specified diameter. See the *Howto body* page for more details on using body rounded/polyhedron particles.

This pairwise interaction between the rounded polyhedra is described in *Wang*, where a polyhedron does not have sharp corners and edges, but is rounded at its vertices and edges by spheres centered on each vertex with a specified diameter. The edges of the polyhedron are defined between pairs of adjacent vertices. Its faces are defined by a loop of edges. The sphere diameter for each polygon is specified in the data file read by the *read data* command. This is a discrete element model (DEM) which allows for multiple contact points.

Note that when two particles interact, the effective surface of each polyhedron particle is displaced outward from each of its vertices, edges, and faces by half its sphere diameter. The interaction forces and energies between two particles are defined with respect to the separation of their respective rounded surfaces, not by the separation of the vertices, edges, and faces themselves.

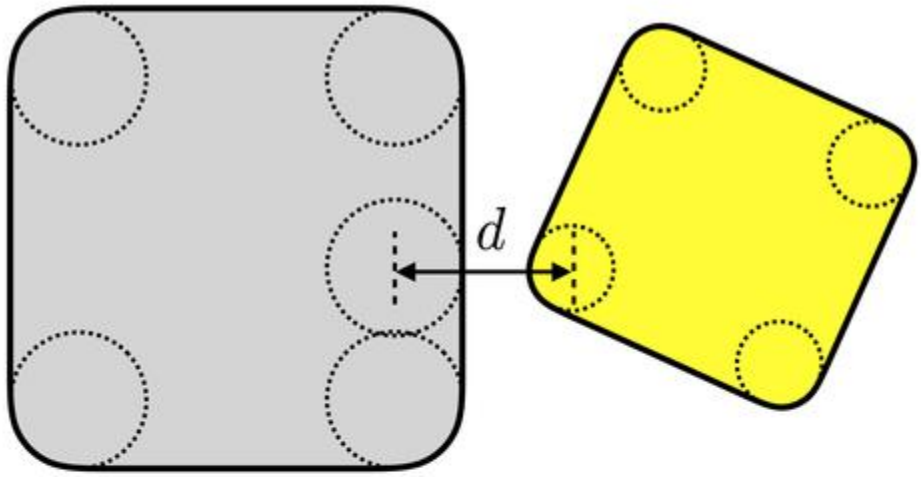
This means that the specified cutoff in the pair_style command is the cutoff distance, $r_c$, for the surface separation, $\delta_n$ (see figure below). This is the distance at which two particles no longer interact. If $r_c$ is specified as 0.0, then it is a contact-only interaction. I.e. the two particles must overlap in order to exert a repulsive force on each other. If $r_c > 0.0$, then the force between two particles will be attractive for surface separations from 0 to $r_c$, and repulsive once the particles overlap.

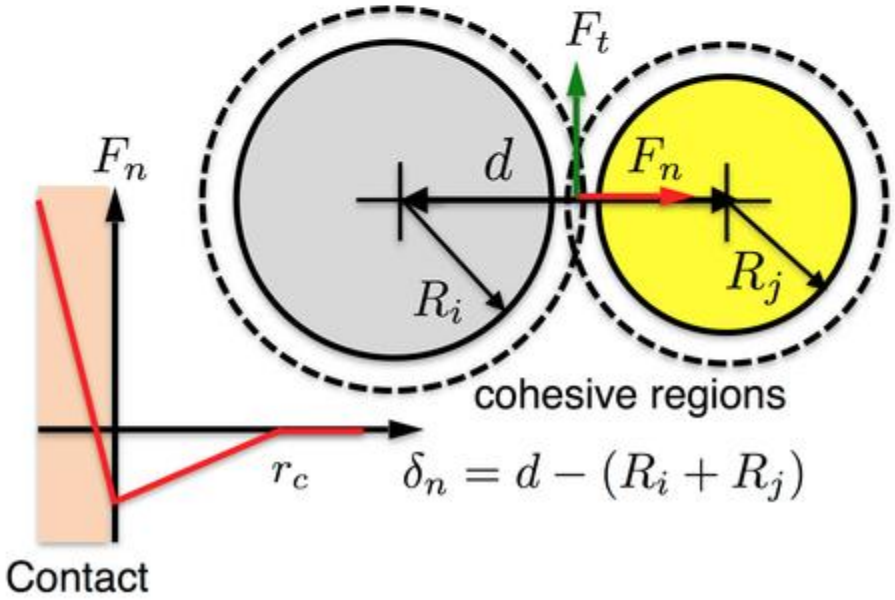
Note that unlike for other pair styles, the specified cutoff is not the distance between the centers of two particles at which they stop interacting. This center-to-center distance depends on the shape and size of the two particles and their relative orientation. LAMMPS takes that into account when computing the surface separation distance and applying the $r_c$ cutoff.

The forces between vertex-vertex, vertex-edge, vertex-face, edge-edge, and edge-face overlaps are given by:

$$F_n = \begin{cases} k_n\delta_n - c_n v_n, & \delta_n \leq 0 \\ -k_{na}\delta_n - c_n v_n & 0 < \delta_n \leq r_c \\ 0 & \delta_n > r_c \end{cases}$$

$$F_t = \begin{cases} \mu k_n \delta_n - c_t v_t & \delta_n \leq 0 \\ 0 & \delta_n > 0 \end{cases}$$




In *Wang*, the tangential friction force between two particles that are in contact is modeled differently prior to gross sliding (i.e. static friction) and during gross-sliding (kinetic friction). The latter takes place when the tangential deformation exceeds the Coulomb frictional limit. In the current implementation, however, we do not take into account frictional history, i.e. we do not keep track of how many time steps the two particles have been in contact nor calculate the tangential deformation. Instead, we assume that gross sliding takes place as soon as two particles are in contact.

The following coefficients must be defined for each pair of atom types via the *pair_coeff* command as in the examples above, or in the data file read by the *read_data* command:

- $k_n$ (energy/distance^2 units)
- $k_{na}$ (energy/distance^2 units)

Effectively, $k_n$ and $k_{na}$ are the slopes of the red lines in the plot above for force versus surface separation, for $\delta_n$ < 0 and $0 < \delta_n < r_c$ respectively.

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.14.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support the *pair_modify* mix, shift, table, and tail options.

This pair style does not write its information to *binary restart files*. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.14.5 Restrictions

These pair styles are part of the BODY package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires the *newton* setting to be "on" for pair interactions.

### 4.14.6 Related commands

*pair_coeff*

### 4.14.7 Default

none

**(Wang)** J. Wang, H. S. Yu, P. A. Langston, F. Y. Fraige, Granular Matter, 13, 1 (2011).

## 4.15 pair_style bondval command

Accelerator Variants: *bondval/kk*

## 4.16 pair_style bondval/vec command

Accelerator Variants: *bondval/vec/kk*

### 4.16.1 Syntax

```
pair_style bondval power cutoff
pair_style bondval/vec power cutoff
```

- *power* = exponent for the bond-valence force (typically 2.0)
- *cutoff* = global cutoff for bond-valence interactions (distance units)

### 4.16.2 Examples

A three-element perovskite BaTiO3 system:

```
pair_style hybrid/overlay lj/cut/coul/long 8.0 8.0 bondval 2.0 8.0 bondval/vec 2.0 8.0

# lj/cut/coul/long parameters: epsilon sigma
pair_coeff 1 1 lj/cut/coul/long 2.0 2.44805
pair_coeff 1 2 lj/cut/coul/long 2.0 2.32592
pair_coeff 1 3 lj/cut/coul/long 2.0 1.98792
pair_coeff 2 2 lj/cut/coul/long 2.0 2.73825
pair_coeff 2 3 lj/cut/coul/long 2.0 1.37741
pair_coeff 3 3 lj/cut/coul/long 2.0 1.99269

# bondval parameters: r0 alpha S V0
pair_coeff 1 1 bondval 0.0000000 5.0000000 0.5973900 2.0000000
pair_coeff 1 2 bondval 0.0000000 5.0000000 0.0000000 0.0000000
pair_coeff 1 3 bondval 2.2900000 8.9400000 0.0000000 0.0000000
pair_coeff 2 2 bondval 0.0000000 5.0000000 0.1653300 4.0000000
pair_coeff 2 3 bondval 1.7980000 5.2000000 0.0000000 0.0000000
pair_coeff 3 3 bondval 0.0000000 5.0000000 0.9306300 2.0000000

# bondval/vec parameters: r0 alpha D W0
pair_coeff 1 1 bondval/vec 0.0000000 5.0000000 0.0842900 0.1156100
pair_coeff 1 2 bondval/vec 0.0000000 5.0000000 0.0000000 0.0000000
pair_coeff 1 3 bondval/vec 2.1430000 8.9400000 0.0000000 0.0000000
pair_coeff 2 2 bondval/vec 0.0000000 5.0000000 0.8248400 0.3943700
pair_coeff 2 3 bondval/vec 1.7980000 5.2000000 0.0000000 0.0000000
pair_coeff 3 3 bondval/vec 0.0000000 5.0000000 0.2800600 0.3165100
```

### 4.16.3 Description

Added in version 4Jul2026.

The bond-valence potential is an empirical potential based on the conservation of the bond-valence (bondval) and bond-valence vector (bondval/vec), fitted to DFT calculations for a given bulk semiconductor. It is typically used with *pair_style hybrid/overlay* to combine Coulombic, Lennard-Jones repulsion, bond-valence, and bond-valence vector contributions.

The bond-valence for a given atom pair ($V_{ij}$) is a measure of the bonding strength calculated from the length of the bond ($r_{ij}$) by:

$$V_{ij} = \left( \frac{r0_{ij}}{r_{ij}} \right)^{\alpha_{ij}}$$

where $r0_{ij}$ and $\alpha_{ij}$ are Brown's empirical parameters for bond-valence. The bond-valence vector is a vector lying along the bond between atom i and j: $\vec{V}_{ij} = V_{ij}\hat{R}_{ij}$, where $\hat{R}_{ij}$ is the unit vector pointing from atom i toward atom j.

The total potential energy of the system consists of a Coulombic energy ($E_c$), a short-range repulsive energy ($E_r$), a bond-valence energy ($E_{BV}$), a bond-valence vector energy ($E_{BVV}$), and an optional angle potential ($E_a$):

$$E = E_c + E_r + E_{BV} + E_{BVV} + E_a$$

$$E_c = \sum_{i<j} \frac{q_i q_j}{r_{ij}}$$

$$E_r = \sum_{i<j} \left( \frac{B_{ij}}{r_{ij}} \right)^{12}$$

$$E_{BV} = \sum_i S_i \left(V_i - V_{0,i}\right)^2$$

$$E_{BVV} = \sum_i D_i \left(|\vec{W}_i|^2 - |\vec{W}_{0,i}|^2\right)^2$$

$$E_a = k \sum_i^{N_{\mathrm{oxygen}}} \left(\theta_i - 180^\circ\right)^2$$

where $V_i = \sum_{j \neq i} V_{ij}$ is the bond-valence sum (BVS), and $\vec{W}_i = \sum_{j \neq i} \vec{V}_{ij}$ is the bond-valence vector sum (BVVS), $q_i$ is the ionic charge, $B_{ij}$ is the short-range repulsion parameter, $S_i$ and $D_i$ are scaling parameters (energy units), $k$ is the angle spring constant, and $\theta_i$ is the O-O-O angle along the common axis of two adjacent oxygen octahedra.

The pair style *bondval* computes $E_{BV}$, the bond-valence energy term. The pair style *bondval/vec* computes $E_{BVV}$, the bond-valence vector energy term. The remaining energy contributions ($E_c$ and $E_r$) are typically provided by *pair_style lj/cut/coul/long*, and $E_a$ by an *angle_style*. The *power* argument to the pair_style command specifies the exponent used in computing the forces and is usually set to 2.0.

The quantities $r_{ij}$, $V_i$, and $\vec{W}_i$ are computed at each timestep from the current atom positions. All other parameters must be provided by the user.

The potential parameters are obtained by training against DFT energies for a given system. Trained and tested parameters for several bulk perovskite oxide materials are published by the group of Andrew M. Rappe (see references below). The published parameters use LAMMPS *metal* units. Atom charges are also fitted parameters; they can be set via *read_data* or *set*. The angle potential, if used, can be set with *angle_style harmonic* together with *angle_coeff*.

The following coefficients must be defined for each pair of atom types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands. When used within *pair_style hybrid/overlay*, the pair style name must be included as shown in the examples.

For *bondval*:

- $r0_{ij}$ = first of Brown's empirical bond-valence parameter (distance units)
- $\alpha_{ij}$ = second of Brown's empirical bond-valence exponent (dimensionless)
- $S_i$ = penalty for deviating from ideal bond valence (energy units)
- $V_{0,i}$ = ideal bond-valence for a given atom type (dimensionless)
- cutoff (distance units) – optional

The first two parameters ($r0_{ij}$ and $\alpha_{ij}$) are pair atom-type dependent and contribute to the bond-valence calculation for all atom-type pairs. The penalty constant $S_i$ and ideal value $V_{0,i}$ are single atom-type dependent. Thus only same-species values (I = J) are nonzero in input file.

For *bondval/vec*:

- $r0_{ij}$ = first of Brown's empirical bond-valence parameter (distance units)
- $\alpha_{ij}$ = second of Brown's empirical bond-valence exponent (dimensionless)
- $D_i$ = penalty for deviating from ideal bond valence vector (energy units)
- $W_{0,i}$ = ideal bond-valence vector for a given atom type (dimensionless)
- cutoff (distance units) – optional

The same distinction applies: $r0_{ij}$ and $\alpha_{ij}$ are pair atom-type dependent and used for all pairs, while $D_i$ and $W_{0,i}$ are atom-type dependent. Thus only same-species values (I = J) are nonzero in input file.

The final cutoff coefficient is optional for both styles. If not specified, the global cutoff given in the pair_style command is used.

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.16.4 Mixing, shift, table, tail correction, restart, rRESPA info

These pair styles do not support parameter mixing. Coefficients must be specified explicitly for all atom type pairs.

These pair styles do not support the *pair_modify* shift, table, or tail options.

These pair styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

These pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

### 4.16.5 Restrictions

These pair styles are part of the EXTRA-PAIR package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

These pair styles must be used with *pair_style hybrid/overlay*. They cannot be used as standalone pair styles.

For a physically correct simulation, *bondval*, *bondval/vec*, and a *pair_style lj/cut/coul/long* contribution for $E_c$ and $E_r$ must all be combined via `hybrid/overlay`. The published parameters for this potential are fitted to only include $r^{-12}$ repulsion term ($E_r$) in the Lennard-Jones potential, while the attractive $r^{-6}$ contribution is set to 0. To run with the published parameters correctly, users must manually initialize the internal variables `lj2` and `lj4` in the source code of *pair_style lj/cut/coul/long* to be zero in order to remove the attractive $r^{-6}$ contribution.

### 4.16.6 Related commands

- *pair_style hybrid/overlay*
- *pair_style lj/cut/coul/long*
- *angle_style harmonic*
- *kspace_style*

### 4.16.7 Default

none

---

**(Grinberg)** I. Grinberg, V. R. Cooper, and A. M. Rappe, *Nature*, **419**, 909 (2002).

**(Shin1)** Y.-H. Shin, J.-Y. Son, B.-J. Lee, I. Grinberg, and A. M. Rappe, *J. Phys.: Condens. Matter*, **20**, 015224 (2008).

**(Shin2)** Y.-H. Shin, V. R. Cooper, I. Grinberg, and A. M. Rappe, *Phys. Rev. B*, **71**, 054104 (2005).

**(Liu1)** S. Liu, I. Grinberg, and A. M. Rappe, *J. Phys.: Condens. Matter*, **25**, 102202 (2013).

**(Liu2)** S. Liu, I. Grinberg, H. Takenaka, and A. M. Rappe, *Phys. Rev. B*, **88**, 104102 (2013).

**(Brown1)** I. D. Brown, *Chem. Rev.*, **109**, 6858 (2009).

**(Brown2)** I. Brown and R. Shannon, *Acta Crystallogr. A*, **29**, 266 (1973).

**(Brown3)** I. Brown and K. K. Wu, *Acta Crystallogr. B*, **32**, 1957 (1976).

## 4.17 pair_style bop command

### 4.17.1 Syntax

```
pair_style bop keyword ...
```

- zero or more keywords may be appended
- keyword = *save*

```
save = pre-compute and save some values
```

### 4.17.2 Examples

```
pair_style bop
pair_coeff * * ../potentials/CdTe_bop Cd Te
pair_style bop save
pair_coeff * * ../potentials/CdTe.bop.table Cd Te Te
comm_modify cutoff 14.70
```

### 4.17.3 Description

The *bop* pair style computes Bond-Order Potentials (BOP) based on quantum mechanical theory incorporating both $\sigma$ and $\pi$ bonding. By analytically deriving the BOP from quantum mechanical theory its transferability to different phases can approach that of quantum mechanical methods. This potential is similar to the original BOP developed by Pettifor (*Pettifor_1*, *Pettifor_2*, *Pettifor_3*) and later updated by Murdick, Zhou, and Ward (*Murdick*, *Ward*). Currently, BOP potential files for these systems are provided with LAMMPS: AlCu, CCu, CdTe, CdTeSe, CdZnTe, CuH, GaAs. A system with only a subset of these elements, including a single element (e.g. C or Cu or Al or Ga or Zn or CdZn), can also be modeled by using the appropriate alloy file and assigning all atom types to the single element or subset of elements via the *pair_coeff command*, as discussed below.

The BOP potential consists of three terms:

$$E = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=i_1}^{i_N}\phi_{ij}\left(r_{ij}\right) - \sum_{i=1}^{N}\sum_{j=i_1}^{i_N}\beta_{\sigma,ij}\left(r_{ij}\right)\cdot\Theta_{\sigma,ij} - \sum_{i=1}^{N}\sum_{j=i_1}^{i_N}\beta_{\pi,ij}\left(r_{ij}\right)\cdot\Theta_{\pi,ij} + U_{prom}$$

where $\phi_{ij}(r_{ij})$ is a short-range two-body function representing the repulsion between a pair of ion cores, $\beta_{\sigma,ij}(r_{ij})$ and $\beta_{\sigma,ij}(r_{ij})$ are respectively sigma and $\pi$ bond integrals, $\Theta_{\sigma,ij}$ and $\Theta_{\pi,ij}$ are $\sigma$ and $\pi$ bond-orders, and U_prom is the promotion energy for sp-valent systems.

The detailed formulas for this potential are given in Ward (*Ward*); here we provide only a brief description.

The repulsive energy $\phi_{ij}(r_{ij})$ and the bond integrals $\beta_{\sigma,ij}(r_{ij})$ and $\beta_{\phi,ij}(r_{ij})$ are functions of the interatomic distance $r_{ij}$ between atom $i$ and $j$. Each of these potentials has a smooth cutoff at a radius of $r_{cut,ij}$. These smooth cutoffs ensure stable behavior at situations with high sampling near the cutoff such as melts and surfaces.

The bond-orders can be viewed as environment-dependent local variables that are ij bond specific. The maximum value of the $\sigma$ bond-order ($\Theta_\sigma$ is 1, while that of the $\pi$ bond-order ($\Theta_\pi$) is 2, attributing to a maximum value of the

total bond-order ($\Theta_\sigma + \Theta_\pi$) of 3. The $\sigma$ and $\pi$ bond-orders reflect the ubiquitous single-, double-, and triple- bond behavior of chemistry. Their analytical expressions can be derived from tight- binding theory by recursively expanding an inter-site Green's function as a continued fraction. To accurately represent the bonding with a computationally efficient potential formulation suitable for MD simulations, the derived BOP only takes (and retains) the first two levels of the recursive representations for both the $\sigma$ and the $\pi$ bond-orders. Bond-order terms can be understood in terms of molecular orbital hopping paths based upon the Cyrot-Lackmann theorem (*Pettifor_1*). The $\sigma$ bond-order with a half-full valence shell is used to interpolate the bond-order expression that incorporated explicit valance band filling. This $\pi$ bond-order expression also contains also contains a three-member ring term that allows implementation of an asymmetric density of states, which helps to either stabilize or destabilize close-packed structures. The $\pi$ bond-order includes hopping paths of length 4. This enables the incorporation of dihedral angles effects.

**Note**

Note that unlike for other potentials, cutoffs for BOP potentials are not set in the pair_style or pair_coeff command; they are specified in the BOP potential files themselves. Likewise, the BOP potential files list atomic masses; thus you do not need to use the *mass* command to specify them. Note that for BOP potentials with hydrogen, you will likely want to set the mass of H atoms to be 10x or 20x larger to avoid having to use a tiny timestep. You can do this by using the *mass* command after using the *pair_coeff* command to read the BOP potential file.

One option can be specified as a keyword with the pair_style command.

The *save* keyword gives you the option to calculate in advance and store a set of distances, angles, and derivatives of angles. The default is to not do this, but to calculate them on-the-fly each time they are needed. The former may be faster, but takes more memory. The latter requires less memory, but may be slower. It is best to test this option to optimize the speed of BOP for your particular system configuration.

---

Only a single pair_coeff command is used with the *bop* style which specifies a BOP potential file, with parameters for all needed elements. These are mapped to LAMMPS atom types by specifying N additional arguments after the filename in the pair_coeff command, where N is the number of LAMMPS atom types:

- filename
- N element names = mapping of BOP elements to atom types

As an example, imagine the CdTe.bop file has BOP values for Cd and Te. If your LAMMPS simulation has 4 atoms types and you want the first 3 to be Cd, and the fourth to be Te, you would use the following pair_coeff command:

```
pair_coeff * * CdTe Cd Cd Cd Te
```

The first 2 arguments must be * * so as to span all LAMMPS atom types. The first three Cd arguments map LAMMPS atom types 1,2,3 to the Cd element in the BOP file. The final Te argument maps LAMMPS atom type 4 to the Te element in the BOP file.

BOP files in the *potentials* directory of the LAMMPS distribution have a ".bop" suffix. The potentials are in tabulated form containing pre-tabulated pair functions for phi_ij(r_ij), beta_(sigma,ij)(r_ij), and beta_pi,ij)(r_ij).

The parameters/coefficients format for the different kinds of BOP files are given below with variables matching the formulation of Ward (*Ward*) and Zhou (*Zhou*). Each header line containing a ":" is preceded by a blank line.

---

**No angular table file format**:

The parameters/coefficients format for the BOP potentials input file containing pre-tabulated functions of g is given below with variables matching the formulation of Ward (*Ward*). This format also assumes the angular functions have the formulation of (*Ward*).

- Line 1: # elements N

The first line is followed by N lines containing the atomic number, mass, and element symbol of each element.

Following the definition of the elements several global variables for the tabulated functions are given.

- Line 1: nr, nBOt (nr is the number of divisions the radius is broken into for function tables and MUST be a factor of 5; nBOt is the number of divisions for the tabulated values of THETA_(S,ij)
- Line 2: delta_1-delta_7 (if all are not used in the particular
- formulation, set unused values to 0.0)

Following this N lines for e_1-e_N containing p_pi.

- Line 3: p_pi (for e_1)
- Line 4: p_pi (for e_2 and continues to e_N)

The next section contains several pair constants for the number of interaction types e_i-e_j, with i=1->N, j=i->N

- Line 1: r_cut (for e_1-e_1 interactions)
- Line 2: c_sigma, a_sigma, c_pi, a_pi
- Line 3: delta_sigma, delta_pi
- Line 4: f_sigma, k_sigma, delta_3 (This delta_3 is similar to that of the previous section but is interaction type dependent)

The next section contains a line for each three body interaction type e_j-e_i-e_k with i=0->N, j=0->N, k=j->N

- Line 1: g_(sigma0), g_(sigma1), g_(sigma2) (These are coefficients for g_(sigma,jik)(THETA_ijk) for e_1-e_1-e_1 interaction. *Ward* contains the full expressions for the constants as functions of b_(sigma,ijk), p_(sigma,ijk), u_(sigma,ijk))
- Line 2: g_(sigma0), g_(sigma1), g_(sigma2) (for e_1-e_1-e_2)

The next section contains a block for each interaction type for the phi_ij(r_ij). Each block has nr entries with 5 entries per line.

- Line 1: phi(r1), phi(r2), phi(r3), phi(r4), phi(r5) (for the e_1-e_1 interaction type)
- Line 2: phi(r6), phi(r7), phi(r8), phi(r9), phi(r10) (this continues until nr)
- ...
- Line nr/5_1: phi(r1), phi(r2), phi(r3), phi(r4), phi(r5), (for the e_1-e_1 interaction type)

The next section contains a block for each interaction type for the beta_(sigma,ij)(r_ij). Each block has nr entries with 5 entries per line.

- Line 1: beta_sigma(r1), beta_sigma(r2), beta_sigma(r3), beta_sigma(r4), beta_sigma(r5) (for the e_1-e_1 interaction type)
- Line 2: beta_sigma(r6), beta_sigma(r7), beta_sigma(r8), beta_sigma(r9), beta_sigma(r10) (this continues until nr)
- ...
- Line nr/5+1: beta_sigma(r1), beta_sigma(r2), beta_sigma(r3), beta_sigma(r4), beta_sigma(r5) (for the e_1-e_2 interaction type)

The next section contains a block for each interaction type for beta_(pi,ij)(r_ij). Each block has nr entries with 5 entries per line.

- Line 1: beta_pi(r1), beta_pi(r2), beta_pi(r3), beta_pi(r4), beta_pi(r5) (for the e_1-e_1 interaction type)

- Line 2: beta_pi(r6), beta_pi(r7), beta_pi(r8), beta_pi(r9), beta_pi(r10) (this continues until nr)
- …
- Line nr/5+1: beta_pi(r1), beta_pi(r2), beta_pi(r3), beta_pi(r4), beta_pi(r5) (for the e_1-e_2 interaction type)

The next section contains a block for each interaction type for the THETA_(S,ij)((THETA_(sigma,ij))^(1/2), f_(sigma,ij)). Each block has nBOt entries with 5 entries per line.

- Line 1: THETA_(S,ij)(r1), THETA_(S,ij)(r2), THETA_(S,ij)(r3), THETA_(S,ij)(r4), THETA_(S,ij)(r5) (for the e_1-e_2 interaction type)
- Line 2: THETA_(S,ij)(r6), THETA_(S,ij)(r7), THETA_(S,ij)(r8), THETA_(S,ij)(r9), THETA_(S,ij)(r10) (this continues until nBOt)
- …
- Line nBOt/5+1: THETA_(S,ij)(r1), THETA_(S,ij)(r2), THETA_(S,ij)(r3), THETA_(S,ij)(r4), THETA_(S,ij)(r5) (for the e_1-e_2 interaction type)

The next section contains a block of N lines for e_1-e_N

- Line 1: delta^mu (for e_1)
- Line 2: delta^mu (for e_2 and repeats to e_N)

The last section contains more constants for e_i-e_j interactions with i=0->N, j=i->N

- Line 1: (A_ij)^(mu*nu) (for e1-e1)
- Line 2: (A_ij)^(mu*nu) (for e1-e2 and repeats as above)

---

**Angular spline table file format**:

The parameters/coefficients format for the BOP potentials input file containing pre-tabulated functions of g is given below with variables matching the formulation of Ward (*Ward*). This format also assumes the angular functions have the formulation of (*Zhou*).

- Line 1: # elements N

The first line is followed by N lines containing the atomic number, mass, and element symbol of each element.

Following the definition of the elements several global variables for the tabulated functions are given.

- Line 1: nr, ntheta, nBOt (nr is the number of divisions the radius is broken into for function tables and MUST be a factor of 5; ntheta is the power of the power of the spline used to fit the angular function; nBOt is the number of divisions for the tabulated values of THETA_(S,ij)
- Line 2: delta_1-delta_7 (if all are not used in the particular
- formulation, set unused values to 0.0)

Following this N lines for e_1-e_N containing p_pi.

- Line 3: p_pi (for e_1)
- Line 4: p_pi (for e_2 and continues to e_N)

The next section contains several pair constants for the number of interaction types e_i-e_j, with i=1->N, j=i->N

- Line 1: r_cut (for e_1-e_1 interactions)
- Line 2: c_sigma, a_sigma, c_pi, a_pi
- Line 3: delta_sigma, delta_pi

- Line 4: f_sigma, k_sigma, delta_3 (This delta_3 is similar to that of the previous section but is interaction type dependent)

The next section contains a line for each three body interaction type e_j-e_i-e_k with i=0->N, j=0->N, k=j->N

- Line 1: g0, g1, g2… (These are coefficients for the angular spline of the g_(sigma,jik)(THETA_ijk) for e_1-e_1-e_1 interaction. The function can contain up to 10 term thus 10 constants. The first line can contain up to five constants. If the spline has more than five terms the second line will contain the remaining constants The following lines will then contain the constants for the remaining g0, g1, g2… (for e_1-e_1-e_2) and the other three body interactions

The rest of the table has the same structure as the previous section (see above).

---

**Angular no-spline table file format**:

The parameters/coefficients format for the BOP potentials input file containing pre-tabulated functions of g is given below with variables matching the formulation of Ward (*Ward*). This format also assumes the angular functions have the formulation of (*Zhou*).

- Line 1: # elements N

The first two lines are followed by N lines containing the atomic number, mass, and element symbol of each element.

Following the definition of the elements several global variables for the tabulated functions are given.

- Line 1: nr, ntheta, nBOt (nr is the number of divisions the radius is broken into for function tables and MUST be a factor of 5; ntheta is the number of divisions for the tabulated values of the g angular function; nBOt is the number of divisions for the tabulated values of THETA_(S,ij)
- Line 2: delta_1-delta_7 (if all are not used in the particular
- formulation, set unused values to 0.0)

Following this N lines for e_1-e_N containing p_pi.

- Line 3: p_pi (for e_1)
- Line 4: p_pi (for e_2 and continues to e_N)

The next section contains several pair constants for the number of interaction types e_i-e_j, with i=1->N, j=i->N

- Line 1: r_cut (for e_1-e_1 interactions)
- Line 2: c_sigma, a_sigma, c_pi, a_pi
- Line 3: delta_sigma, delta_pi
- Line 4: f_sigma, k_sigma, delta_3 (This delta_3 is similar to that of the previous section but is interaction type dependent)

The next section contains a line for each three body interaction type e_j-e_i-e_k with i=0->N, j=0->N, k=j->N

- Line 1: g(theta1), g(theta2), g(theta3), g(theta4), g(theta5) (for the e_1-e_1-e_1 interaction type)
- Line 2: g(theta6), g(theta7), g(theta8), g(theta9), g(theta10) (this continues until ntheta)
- …
- Line ntheta/5+1: g(theta1), g(theta2), g(theta3), g(theta4), g(theta5), (for the e_1-e_1-e_2 interaction type)

The rest of the table has the same structure as the previous section (see above).

---

### 4.17.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support the *pair_modify* mix, shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.17.5 Restrictions

These pair styles are part of the MANYBODY package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

These pair potentials require the *newtion* setting to be "on" for pair interactions.

Pair style bop is not compatible with being used as a sub-style with doc:*hybrid pair styles <pair_hybrid>*. Pair style bop is also not compatible with *multi-cutoff neighbor lists* or *multi-cutoff communitcation*.

The .bop.table potential files provided with LAMMPS (see the potentials directory) are parameterized for metal *units*. You can use the BOP potential with any LAMMPS units, but you would need to create your own BOP potential file with coefficients listed in the appropriate units if your simulation does not use "metal" units.

### 4.17.6 Related commands

*pair_coeff*

### 4.17.7 Default

non-tabulated potential file, a_0 is non-zero.

---

**(Pettifor_1)** D.G. Pettifor and I.I. Oleinik, Phys. Rev. B, 59, 8487 (1999).

**(Pettifor_2)** D.G. Pettifor and I.I. Oleinik, Phys. Rev. Lett., 84, 4124 (2000).

**(Pettifor_3)** D.G. Pettifor and I.I. Oleinik, Phys. Rev. B, 65, 172103 (2002).

**(Murdick)** D.A. Murdick, X.W. Zhou, H.N.G. Wadley, D. Nguyen-Manh, R. Drautz, and D.G. Pettifor, Phys. Rev. B, 73, 45206 (2006).

**(Ward)** D.K. Ward, X.W. Zhou, B.M. Wong, F.P. Doty, and J.A. Zimmerman, Phys. Rev. B, 85,115206 (2012).

**(Zhou)** X.W. Zhou, D.K. Ward, M. Foster (TBP).

## 4.18 pair_style born command

Accelerator Variants: *born/gpu*, *born/kk*, *born/omp*

## 4.19 pair_style born/coul/long command

Accelerator Variants: *born/coul/long/gpu*, *born/coul/long/kk*, *born/coul/long/omp*

## 4.20 pair_style born/coul/msm command

Accelerator Variants: *born/coul/msm/omp*

## 4.21 pair_style born/coul/wolf command

Accelerator Variants: *born/coul/wolf/gpu*, *born/coul/wolf/kk*, *born/coul/wolf/omp*

## 4.22 pair_style born/coul/dsf command

Accelerator Variants: *born/coul/dsf/omp*

### 4.22.1 Syntax

```
pair_style style args
```

- style = *born* or *born/coul/long* or *born/coul/msm* or *born/coul/wolf*
- args = list of arguments for a particular style

```
born args = cutoff
  cutoff = global cutoff for non-Coulombic interactions (distance units)
born/coul/long args = cutoff (cutoff2)
  cutoff = global cutoff for non-Coulombic (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
born/coul/msm args = cutoff (cutoff2)
  cutoff = global cutoff for non-Coulombic (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
born/coul/wolf args = alpha cutoff (cutoff2)
  alpha = damping parameter (inverse distance units)
  cutoff = global cutoff for non-Coulombic (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
born/coul/dsf args = alpha cutoff (cutoff2)
  alpha = damping parameter (inverse distance units)
  cutoff = global cutoff for non-Coulombic (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (distance units)
```

### 4.22.2 Examples

```
pair_style born 10.0
pair_coeff * * 6.08 0.317 2.340 24.18 11.51
pair_coeff 1 1 6.08 0.317 2.340 24.18 11.51

pair_style born/coul/long 10.0
pair_style born/coul/long 10.0 8.
pair_coeff * * 6.08 0.317 2.340 24.18 11.51
pair_coeff 1 1 6.08 0.317 2.340 24.18 11.51

pair_style born/coul/msm 10.0
pair_style born/coul/msm 10.0 8.0
pair_coeff * * 6.08 0.317 2.340 24.18 11.51
pair_coeff 1 1 6.08 0.317 2.340 24.18 11.51
```

(continues on next page)

(continued from previous page)

```
pair_style born/coul/wolf 0.25 10.0
pair_style born/coul/wolf 0.25 10.0 9.0
pair_coeff * * 6.08 0.317 2.340 24.18 11.51
pair_coeff 1 1 6.08 0.317 2.340 24.18 11.51

pair_style born/coul/dsf 0.1 10.0 12.0
pair_coeff * *   0.0 1.00 0.00 0.00 0.00
pair_coeff 1 1 480.0 0.25 0.00 1.05 0.50
```

### 4.22.3 Description

The *born* style computes the Born-Mayer-Huggins or Tosi/Fumi potential described in *(Fumi and Tosi)*, given by

$$E = A \exp\left(\frac{\sigma - r}{\rho}\right) - \frac{C}{r^6} + \frac{D}{r^8} \qquad r < r_c$$

where $\sigma$ is an interaction-dependent length parameter, $\rho$ is an ionic-pair dependent length parameter, and $r_c$ is the cutoff.

The styles with *coul/long* or *coul/msm* add a Coulombic term as described for the *lj/cut* pair styles. An additional damping factor is applied to the Coulombic term so it can be used in conjunction with the *kspace_style* command and its *ewald* or *pppm* of *msm* option. The Coulombic cutoff specified for this style means that pairwise interactions within this distance are computed directly; interactions outside that distance are computed in reciprocal space.

If one cutoff is specified for the *born/coul/long* and *born/coul/msm* style, it is used for both the A,C,D and Coulombic terms. If two cutoffs are specified, the first is used as the cutoff for the A,C,D terms, and the second is the cutoff for the Coulombic term.

The *born/coul/wolf* style adds a Coulombic term as described for the Wolf potential in the *coul/wolf* pair style.

The *born/coul/dsf* style computes the Coulomb contribution with the damped shifted force model as in the *coul/dsf* style.

Note that these potentials are related to the *Buckingham potential*.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- A (energy units)
- $\rho$ (distance units)
- $\sigma$ (distance units)
- C (energy units * distance units^6)
- D (energy units * distance units^8)
- cutoff (distance units)

The second coefficient, rho, must be greater than zero.

The last coefficient is optional. If not specified, the global A,C,D cutoff specified in the pair_style command is used.

For *born/coul/long*, *born/coul/wolf* and *born/coul/dsf* no Coulombic cutoff can be specified for an individual I,J type pair. All type pairs use the same global Coulombic cutoff specified in the pair_style command.

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.22.4 Mixing, shift, table, tail correction, restart, rRESPA info

These pair styles do not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

These styles support the *pair_modify* shift option for the energy of the exp(), 1/r^6, and 1/r^8 portion of the pair interaction.

The *born/coul/long* pair style supports the *pair_modify* table option to tabulate the short-range portion of the long-range Coulombic interaction.

These styles support the pair_modify tail option for adding long-range tail corrections to energy and pressure.

Thess styles writes thei information to binary *restart* files, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

These styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

---

### 4.22.5 Restrictions

The *born/coul/long* style is part of the KSPACE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The *born/coul/dsf* and *born/coul/wolf* pair styles are part of the EXTRA-PAIR package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.22.6 Related commands

*pair_coeff* , *pair_style buck*

### 4.22.7 Default

none

---

Fumi and Tosi, J Phys Chem Solids, 25, 31 (1964), Fumi and Tosi, J Phys Chem Solids, 25, 45 (1964).

## 4.23 pair_style born/gauss command

Accelerator Variants: *born/gauss/kk*, *born/gauss/omp*

### 4.23.1 Syntax

```
pair_style born/gauss cutoff
```

- born/gauss = name of the pair style
- cutoff = global cutoff (distance units)

### 4.23.2 Examples

```
pair_style born/gauss 10.0
pair_coeff 1 1 8.2464e13 12.48 0.042644277 0.44 3.56
```

### 4.23.3 Description

Added in version 28Mar2023.

Pair style *born/gauss* computes pairwise interactions from a combination of a Born-Mayer repulsive term and a Gaussian attractive term according to *(Bomont)*:

$$E = A_0 \exp\left(-\alpha r\right) - A_1 \exp\left[-\beta \left(r - r_0\right)^2\right] \qquad r < r_c$$

$r_c$ is the cutoff.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $A_0$ (energy units)
- $\alpha$ (1/distance units)
- $A_1$ (energy units)
- $\beta$ (1/(distance units)^2)
- $r_0$ (distance units)
- cutoff (distance units)

The last coefficient is optional. If not specified, the global cutoff is used.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.23.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

This pair style supports the *pair_modify* shift option for the energy of the pair interaction.

The *pair_modify* table options are not relevant for this pair style.

This pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.23.5 Restrictions

This pair style is only enabled if LAMMPS was built with the EXTRA-PAIR package. See the *Build package* page for more info.

### 4.23.6 Related commands

*pair_coeff*, *pair_style born*

### 4.23.7 Default

none

---

**(Bomont)** Bomont, Bretonnet, J. Chem. Phys. 124, 054504 (2006)

## 4.24 pair_style bpm/spring command

### 4.24.1 Syntax

```
pair_style bpm/spring keyword value  ...
```

- optional keyword = *anharmonic*

  *anharmonic* value = *yes* or *no*
     whether forces include the anharmonic term

### 4.24.2 Examples

```
pair_style bpm/spring
pair_coeff * * 1.0 1.0 1.0
pair_style bpm/spring anharmonic yes
pair_coeff 1 1 1.0 1.0 1.0 50.0
```

### 4.24.3 Description

Added in version 4May2022.

Style *bpm/spring* computes pairwise forces with the formula

$$F = k(r - r_c) + k_a(r - r_c)^3$$

where $k$ is a stiffness, $r_c$ is the cutoff length, and $k_a$ is an optional anharmonic cubic prefactor that can be enabled using the *anharmonic* keyword. The anharmonic term may be useful in scenarios that need to prevent large particle overlap.

An additional damping force is also applied to interacting particles. The force is proportional to the difference in the normal velocity of particles

$$F_D = -\gamma w(\hat{r} \bullet \vec{v})$$

where $\gamma$ is the damping strength, $\hat{r}$ is the radial normal vector, $\vec{v}$ is the velocity difference between the two particles, and $w$ is a smoothing factor. This smoothing factor is constructed such that damping forces go to zero as particles come out of contact to avoid discontinuities. It is given by

$$w = 1.0 - \left(\frac{r}{r_c}\right)^8.$$

This pair style is designed for use in a spring-based bonded particle model. It mirrors the construction of the *bpm/spring* bond style.

This pair interaction is always applied to pairs of non-bonded particles that are within the interaction distance. For pairs of bonded particles that are within the interaction distance, there is the option to either include this pair interaction and overlay the pair force over the bond force or to exclude this pair interaction such that the two particles only interact via the bond force. See discussion of the *overlay/pair* option for BPM bond styles and the *special_bonds* command in the *how to* page on BPMs for more details.

The following coefficients must be defined for each pair of atom types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- $k$ (force/distance units)
- $r_c$ (distance units)
- $\gamma$ (force/velocity units)

Added in version 4Feb2025.

Additionally, if *anharmonic* is set to *yes*, a fourth coefficient must be provided:

- $k_a$ (force/distance^3 units)

---

### 4.24.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the A coefficient and cutoff distance for this pair style can be mixed. A is always mixed via a *geometric* rule. The cutoff is mixed according to the pair_modify mix value. The default mix value is *geometric*. See the "pair_modify" command for details.

This pair style does not support the *pair_modify* shift option, since the pair interaction goes to 0.0 at the cutoff.

The *pair_modify* table and tail options are not relevant for this pair style.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

The potential energy and the single() function of this pair style returns $k(r - r_c)^2/2 + k_a(r - r_c)^4/4$ for a proxy of the energy of a pair interaction, ignoring any smoothing or dissipative forces.

---

### 4.24.5 Restrictions

This pair style is part of the BPM package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.24.6 Related commands

*pair_coeff*, *bond bpm/spring*

### 4.24.7 Default

The option defaults are *anharmonic* = *no*

## 4.25 pair_style brownian command

Accelerator Variants: *brownian/omp*, *brownian/kk*

## 4.26 pair_style brownian/poly command

Accelerator Variants: *brownian/poly/omp*

### 4.26.1 Syntax

```
pair_style style mu flaglog flagfld cutinner cutoff t_target seed flagHI flagVF
```

- style = *brownian* or *brownian/poly*
- mu = dynamic viscosity (dynamic viscosity units)
- flaglog = 0/1 log terms in the lubrication approximation on/off
- flagfld = 0/1 to include/exclude Fast Lubrication Dynamics effects
- cutinner = inner cutoff distance (distance units)
- cutoff = outer cutoff for interactions (distance units)
- t_target = target temp of the system (temperature units)
- seed = seed for the random number generator (positive integer)
- flagHI (optional) = 0/1 to include/exclude 1/r hydrodynamic interactions
- flagVF (optional) = 0/1 to include/exclude volume fraction corrections in the long-range isotropic terms

### 4.26.2 Examples

```
pair_style brownian 1.5 1 1 2.01 2.5 2.0 5878567 # (assuming radius = 1)
pair_coeff 1 1 2.05 2.8
pair_coeff * *
```

### 4.26.3 Description

Styles *brownian* and *brownian/poly* compute Brownian forces and torques on finite-size spherical particles. The former requires monodisperse spherical particles; the latter allows for polydisperse spherical particles.

These pair styles are designed to be used with either the *pair_style lubricate* or *pair_style lubricateU* commands to provide thermostatting when dissipative lubrication forces are acting. Thus the parameters *mu*, *flaglog*, *flagfld*, *cutinner*, and *cutoff* should be specified consistent with the settings in the lubrication pair styles. For details, refer to either of the lubrication pair styles.

The *t_target* setting is used to specify the target temperature of the system. The random number *seed* is used to generate random numbers for the thermostatting procedure.

The *flagHI* and *flagVF* settings are optional. Neither should be used, or both must be defined.

Changed in version 4Jul2026.

For *brownian/poly* the pairwise random Brownian force is now generated once per pair from a deterministic random number stream (keyed on the pair of atom IDs and the timestep) and applied equal and opposite to both particles, so that linear momentum is conserved exactly and the system is no longer heated spuriously. In addition the near-field resistance functions now use the symmetric Jeffrey & Onishi gap so the force magnitude is independent of which particle is taken as the reference, consistent with *pair_style lubricate/poly*. Previously the random force on the two members of a pair was drawn independently (violating Newton's third law) and the resistance was evaluated with one particle's radius as the reference length, both of which were incorrect for polydisperse systems.

---

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- cutinner (distance units)
- cutoff (distance units)

The two coefficients are optional. If neither is specified, the two cutoffs specified in the pair_style command are used. Otherwise both must be specified.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.26.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the two cutoff distances for these pair styles can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

These pair styles do not support the *pair_modify* shift option for the energy of the pair interaction.

The *pair_modify* table option is not relevant for these pair styles.

These pair styles do not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

These pair styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

These pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

---

### 4.26.5 Restrictions

These styles are part of the COLLOID package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Only spherical monodisperse particles are allowed for pair_style brownian.

Only spherical particles are allowed for pair_style brownian/poly. The volume fraction correction is not supported by pair_style brownian/poly.

These pair styles are only compatible with the following wall fixes: *fix wall/lj93, fix wall/lj126, fix wall/lj1043, fix wall/colloid, fix wall/harmonic, fix wall/lepton, fix wall/morse, fix wall/table*.

### 4.26.6 Related commands

*pair_coeff* , *pair_style lubricate*, *pair_style lubricateU*

### 4.26.7 Default

The default settings for the optional args are flagHI = 1 and flagVF = 1.

## 4.27 pair_style buck command

Accelerator Variants: *buck/gpu*, *buck/intel*, *buck/kk*, *buck/omp*

## 4.28 pair_style buck/coul/cut command

Accelerator Variants: *buck/coul/cut/gpu*, *buck/coul/cut/intel*, *buck/coul/cut/kk*, *buck/coul/cut/omp*

## 4.29 pair_style buck/coul/long command

Accelerator Variants: *buck/coul/long/gpu*, *buck/coul/long/intel*, *buck/coul/long/kk*, *buck/coul/long/omp*

## 4.30 pair_style buck/coul/msm command

Accelerator Variants: *buck/coul/msm/omp*

### 4.30.1 Syntax

```
pair_style style args
```

- style = *buck* or *buck/coul/cut* or *buck/coul/long* or *buck/coul/msm*
- args = list of arguments for a particular style

```
buck args = cutoff
  cutoff = global cutoff for Buckingham interactions (distance units)
buck/coul/cut args = cutoff (cutoff2)
  cutoff = global cutoff for Buckingham (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
buck/coul/long args = cutoff (cutoff2)
  cutoff = global cutoff for Buckingham (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
buck/coul/msm args = cutoff (cutoff2)
  cutoff = global cutoff for Buckingham (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
```

### 4.30.2 Examples

```
pair_style buck 2.5
pair_coeff * * 100.0 1.5 200.0
pair_coeff * * 100.0 1.5 200.0 3.0

pair_style buck/coul/cut 10.0
pair_style buck/coul/cut 10.0 8.0
pair_coeff * * 100.0 1.5 200.0
pair_coeff 1 1 100.0 1.5 200.0 9.0
pair_coeff 1 1 100.0 1.5 200.0 9.0 8.0

pair_style buck/coul/long 10.0
pair_style buck/coul/long 10.0 8.0
pair_coeff * * 100.0 1.5 200.0
pair_coeff 1 1 100.0 1.5 200.0 9.0

pair_style buck/coul/msm 10.0
pair_style buck/coul/msm 10.0 8.0
pair_coeff * * 100.0 1.5 200.0
pair_coeff 1 1 100.0 1.5 200.0 9.0
```

### 4.30.3 Description

The *buck* style computes a Buckingham potential (exp/6 instead of Lennard-Jones 12/6) given by

$$E = Ae^{-r/\rho} - \frac{C}{r^6} \qquad r < r_c$$

where $\rho$ is an ionic-pair dependent length parameter, and $r_c$ is the cutoff on both terms.

The styles with *coul/cut* or *coul/long* or *coul/msm* add a Coulombic term as described for the *lj/cut* pair styles. For *buck/coul/long* and *buc/coul/msm*, an additional damping factor is applied to the Coulombic term so it can be used in conjunction with the *kspace_style* command and its *ewald* or *pppm* or *msm* option. The Coulombic cutoff specified for this style means that pairwise interactions within this distance are computed directly; interactions outside that distance are computed in reciprocal space.

If one cutoff is specified for the *born/coul/cut* and *born/coul/long* and *born/coul/msm* styles, it is used for both the A,C and Coulombic terms. If two cutoffs are specified, the first is used as the cutoff for the A,C terms, and the second is the cutoff for the Coulombic term.

Note that these potentials are related to the *Born-Mayer-Huggins potential*.

**Note**

For all these pair styles, the terms with A and C are always cutoff. The additional Coulombic term can be cutoff or long-range (no cutoff) depending on whether the style name includes coul/cut or coul/long or coul/msm. If you wish the C/r^6 term to be long-range (no cutoff), then see the *pair_style buck/long/coul/long* command.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- A (energy units)
- $\rho$ (distance units)
- C (energy-distance^6 units)
- cutoff (distance units)
- cutoff2 (distance units)

The second coefficient, $\rho$, must be greater than zero. The coefficients A, $\rho$, and C can be written as analytical expressions of $\varepsilon$ and $\sigma$, in analogy to the Lennard-Jones potential *(Khrapak)*.

The latter 2 coefficients are optional. If not specified, the global A,C and Coulombic cutoffs are used. If only one cutoff is specified, it is used as the cutoff for both A,C and Coulombic interactions for this type pair. If both coefficients are specified, they are used as the A,C and Coulombic cutoffs for this type pair. You cannot specify 2 cutoffs for style *buck*, since it has no Coulombic terms. For *buck/coul/long* only the LJ cutoff can be specified since a Coulombic cutoff cannot be specified for an individual I,J type pair. All type pairs use the same global Coulombic cutoff specified in the pair_style command.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.30.4 Mixing, shift, table, tail correction, restart, rRESPA info

These pair styles do not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

These styles support the *pair_modify* shift option for the energy of the exp() and 1/r^6 portion of the pair interaction.

The *buck/coul/long* pair style supports the *pair_modify* table option to tabulate the short-range portion of the long-range Coulombic interaction.

These styles support the pair_modify tail option for adding long-range tail corrections to energy and pressure for the A,C terms in the pair interaction.

These styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

These styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

### 4.30.5 Restrictions

The *buck/coul/long* style is part of the KSPACE package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.30.6 Related commands

*pair_coeff*, *pair_style born*

### 4.30.7 Default

none

**(Khrapak)** Khrapak, Chaudhuri, and Morfill, J Chem Phys, 134, 054120 (2011).

## 4.31 pair_style buck6d/coul/gauss/dsf command

Accelerator Variants: *buck6d/coul/gauss/dsf/kk*, *buck6d/coul/gauss/dsf/omp*

## 4.32 pair_style buck6d/coul/gauss/long command

Accelerator Variants: *buck6d/coul/gauss/long/kk*, *buck6d/coul/gauss/long/omp*

### 4.32.1 Syntax

```
pair_style style args
```

- style = *buck6d/coul/gauss/dsf* or *buck6d/coul/gauss/long*
- args = list of arguments for a particular style

```
buck6d/coul/gauss/dsf args = smooth cutoff (cutoff2)
  smooth  = smoothing onset within Buckingham cutoff (ratio)
  cutoff  = global cutoff for Buckingham (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
buck6d/coul/gauss/long args = smooth smooth2 cutoff (cutoff2)
  smooth   = smoothing onset within Buckingham cutoff (ratio)
  smooth2  = smoothing onset within Coulombic cutoff (ratio)
  cutoff   = global cutoff for Buckingham (and Coulombic if only 1 arg) (distance units)
  cutoff2  = global cutoff for Coulombic (optional) (distance units)
```

### 4.32.2 Examples

```
pair_style buck6d/coul/gauss/dsf    0.9000    12.0000
pair_coeff 1  1  1030.  3.061  457.179  4.521  0.608

pair_style buck6d/coul/gauss/long   0.9000  1.0000  12.0000
pair_coeff 1  1  1030.  3.061  457.179  4.521  0.608
```

### 4.32.3 Description

The *buck6d/coul/gauss* styles evaluate vdW and Coulomb interactions following the MOF-FF force field after *(Schmid)*. The vdW term of the *buck6d* styles computes a dispersion damped Buckingham potential:

$$E = Ae^{-\kappa r} - \frac{C}{r^6} \cdot \frac{1}{1 + Dr^{14}} \qquad r < r_c$$

where A and C are a force constant, $\kappa$ is an ionic-pair dependent reciprocal length parameter, D is a dispersion correction parameter, and the cutoff $r_c$ truncates the interaction distance. The first term in the potential corresponds to the Buckingham repulsion term and the second term to the dispersion attraction with a damping correction analog to the Grimme correction used in DFT. The latter corrects for artifacts occurring at short distances which become an issue for soft vdW potentials.

The *buck6d* styles include a smoothing function which is invoked according to the global smoothing parameter within the specified cutoff. Hereby a parameter of i.e. 0.9 invokes the smoothing within 90% of the cutoff. No smoothing is applied at a value of 1.0. For the *gauss/dsf* style this smoothing is only applicable for the dispersion damped Buckingham potential. For the *gauss/long* styles the smoothing function can also be invoked for the real space coulomb interactions which enforce continuous energies and forces at the cutoff.

Both styles *buck6d/coul/gauss/dsf* and *buck6d/coul/gauss/long* evaluate a Coulomb potential using spherical Gaussian type charge distributions which effectively dampen electrostatic interactions for high charges at close distances. The electrostatic potential is thus evaluated as:

$$E = \frac{C_{q_i q_j}}{\varepsilon r_{ij}} \operatorname{erf}\left(\alpha_{ij} r_{ij}\right) \qquad r < r_c$$

where C is an energy-conversion constant, $q_i$ and $q_j$ are the charges on the two atoms, epsilon is the dielectric constant which can be set by the *dielectric* command, $\alpha$ is the ion pair dependent damping parameter and erf() is the error-function. The cutoff $r_c$ truncates the interaction distance.

The style *buck6d/coul/gauss/dsf* computes the Coulomb interaction via the damped shifted force model described in *(Fennell)* approximating an Ewald sum similar to the *pair coul/dsf* styles. In *buck6d/coul/gauss/long* an additional damping factor is applied to the Coulombic term so it can be used in conjunction with the *kspace_style* command and its *ewald* or *pppm* options. The Coulombic cutoff in this case separates the real and reciprocal space evaluation of the Ewald sum.

If one cutoff is specified it is used for both the vdW and Coulomb terms. If two cutoffs are specified, the first is used as the cutoff for the vdW terms, and the second is the cutoff for the Coulombic term.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- A (energy units)
- $\rho$ (distance^-1 units)
- C (energy-distance^6 units)
- D (distance^14 units)
- $\alpha$ (distance^-1 units)
- cutoff (distance units)

The second coefficient, $\rho$, must be greater than zero. The latter coefficient is optional. If not specified, the global vdW cutoff is used.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.32.4 Mixing, shift, table, tail correction, restart, rRESPA info

These pair styles do not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

These styles do not support the *pair_modify* shift option for the energy. Instead the smoothing function should be applied by setting the global smoothing parameter to a value < 1.0.

These styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

### 4.32.5 Restrictions

These styles are part of the MOFFF package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.32.6 Related commands

*pair_coeff*

### 4.32.7 Default

none

**(Schmid)** S. Bureekaew, S. Amirjalayer, M. Tafipolsky, C. Spickermann, T.K. Roy and R. Schmid, Phys. Status Solidi B, 6, 1128 (2013).

**(Fennell)** C. J. Fennell, J. D. Gezelter, J Chem Phys, 124, 234104 (2006).

## 4.33 pair_style buck/long/coul/long command

Accelerator Variants: *buck/long/coul/long/omp*

### 4.33.1 Syntax

```
pair_style buck/long/coul/long flag_buck flag_coul cutoff (cutoff2)
```

- flag_buck = *long* or *cut*

  *long* = use Kspace long-range summation for the dispersion term 1/r^6
  *cut* = use a cutoff

- flag_coul = *long* or *off*

  *long* = use Kspace long-range summation for the Coulombic term 1/r
  *off* = omit the Coulombic term

- cutoff = global cutoff for Buckingham (and Coulombic if only 1 cutoff) (distance units)
- cutoff2 = global cutoff for Coulombic (optional) (distance units)

### 4.33.2 Examples

```
pair_style buck/long/coul/long cut off 2.5
pair_style buck/long/coul/long cut long 2.5 4.0
pair_style buck/long/coul/long long long 4.0
pair_coeff * * 1 1
pair_coeff 1 1 1 3 4
```

### 4.33.3 Description

The *buck/long/coul/long* style computes a Buckingham potential (exp/6 instead of Lennard-Jones 12/6) and Coulombic potential, given by

$$E = Ae^{-r/\rho} - \frac{C}{r^6} \qquad r < r_c$$

$$E = \frac{Cq_iq_j}{\varepsilon r} \qquad r < r_c$$

$r_c$ is the cutoff. If one cutoff is specified in the pair_style command, it is used for both the Buckingham and Coulombic terms. If two cutoffs are specified, they are used as cutoffs for the Buckingham and Coulombic terms respectively.

The purpose of this pair style is to capture long-range interactions resulting from both attractive 1/r^6 Buckingham and Coulombic 1/r interactions. This is done by use of the *flag_buck* and *flag_coul* settings. The *Ismail* paper has more details on when it is appropriate to include long-range 1/r^6 interactions, using this potential.

If *flag_buck* is set to *long*, no cutoff is used on the Buckingham 1/r^6 dispersion term. The long-range portion can be calculated by using the *kspace_style ewald/disp or pppm/disp* commands. The specified Buckingham cutoff then determines which portion of the Buckingham interactions are computed directly by the pair potential versus which part is computed in reciprocal space via the Kspace style. If *flag_buck* is set to *cut*, the Buckingham interactions are simply cutoff, as with *pair_style buck*.

If *flag_coul* is set to *long*, no cutoff is used on the Coulombic interactions. The long-range portion can calculated by using any of several *kspace_style* command options such as *pppm* or *ewald*. Note that if *flag_buck* is also set to long, then the *ewald/disp* or *pppm/disp* Kspace style needs to be used to perform the long-range calculations for both the Buckingham and Coulombic interactions. If *flag_coul* is set to *off*, Coulombic interactions are not computed.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- A (energy units)
- rho (distance units)
- C (energy-distance^6 units)
- cutoff (distance units)
- cutoff2 (distance units)

The second coefficient, rho, must be greater than zero.

The latter 2 coefficients are optional. If not specified, the global Buckingham and Coulombic cutoffs specified in the pair_style command are used. If only one cutoff is specified, it is used as the cutoff for both Buckingham and Coulombic interactions for this type pair. If both coefficients are specified, they are used as the Buckingham and Coulombic cutoffs for this type pair. Note that if you are using *flag_buck* set to *long*, you cannot specify a Buckingham cutoff for an atom type pair, since only one global Buckingham cutoff is allowed. Similarly, if you are using *flag_coul* set to *long*, you cannot specify a Coulombic cutoff for an atom type pair, since only one global Coulombic cutoff is allowed.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages*

page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.33.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

This pair style supports the *pair_modify* shift option for the energy of the exp() and 1/r^6 portion of the pair interaction, assuming *flag_buck* is *cut*.

This pair style supports the *pair_modify* table and table/disp options since they can tabulate the short-range portion of the long-range Coulombic and dispersion interactions.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style supports the use of the *inner*, *middle*, and *outer* keywords of the *run_style respa* command, meaning the pairwise forces can be partitioned by distance at different levels of the rRESPA hierarchy. See the *run_style* command for details.

---

### 4.33.5 Restrictions

This style is part of the KSPACE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.33.6 Related commands

*pair_coeff*

### 4.33.7 Default

none

---

**(Ismail)** Ismail, Tsige, In 't Veld, Grest, Molecular Physics (accepted) (2007).

## 4.34 pair_style lj/charmm/coul/charmm command

Accelerator Variants: *lj/charmm/coul/charmm/gpu*, *lj/charmm/coul/charmm/intel*, *lj/charmm/coul/charmm/kk*, *lj/charmm/coul/charmm/omp*

## 4.35 pair_style lj/charmm/coul/charmm/implicit command

Accelerator Variants: *lj/charmm/coul/charmm/implicit/kk*, *lj/charmm/coul/charmm/implicit/omp*

## 4.36 pair_style lj/charmm/coul/long command

Accelerator Variants: *lj/charmm/coul/long/gpu*, *lj/charmm/coul/long/intel*, *lj/charmm/coul/long/kk*, *lj/charmm/coul/long/opt*, *lj/charmm/coul/long/omp*

## 4.37 pair_style lj/charmm/coul/msm command

Accelerator Variants: *lj/charmm/coul/msm/omp*

## 4.38 pair_style lj/charmmfsw/coul/charmmfsh command

Accelerator Variants: *lj/charmmfsw/coul/charmmfsh/omp*

## 4.39 pair_style lj/charmmfsw/coul/long command

Accelerator Variants: *lj/charmmfsw/coul/long/kk*, *lj/charmmfsw/coul/long/omp*

### 4.39.1 Syntax

```
pair_style style args
```

- style = *lj/charmm/coul/charmm* or *lj/charmm/coul/charmm/implicit* or *lj/charmm/coul/long* or *lj/charmm/coul/msm* or *lj/charmmfsw/coul/charmmfsh* or *lj/charmmfsw/coul/long*
- args = list of arguments for a particular style

```
lj/charmm/coul/charmm args = inner outer (inner2) (outer2)
  inner, outer = global switching cutoffs for Lennard Jones (and Coulombic if only 2␣
↪args)
  inner2, outer2 = global switching cutoffs for Coulombic (optional)
lj/charmm/coul/charmm/implicit args = inner outer (inner2) (outer2)
  inner, outer = global switching cutoffs for LJ (and Coulombic if only 2 args)
  inner2, outer2 = global switching cutoffs for Coulombic (optional)
lj/charmm/coul/long args = inner outer (cutoff)
  inner, outer = global switching cutoffs for LJ (and Coulombic if only 2 args)
  cutoff = global cutoff for Coulombic (optional, outer is Coulombic cutoff if only 2␣
↪args)
lj/charmm/coul/msm args = inner outer (cutoff)
  inner, outer = global switching cutoffs for LJ (and Coulombic if only 2 args)
  cutoff = global cutoff for Coulombic (optional, outer is Coulombic cutoff if only 2␣
↪args)
lj/charmmfsw/coul/charmmfsh args = inner outer (cutoff)
  inner, outer = global cutoffs for LJ (and Coulombic if only 2 args)
  cutoff = global cutoff for Coulombic (optional, outer is Coulombic cutoff if only 2␣
↪args)
lj/charmmfsw/coul/long args = inner outer (cutoff)
  inner, outer = global cutoffs for LJ (and Coulombic if only 2 args)
  cutoff = global cutoff for Coulombic (optional, outer is Coulombic cutoff if only 2␣
↪args)
```

### 4.39.2 Examples

```
pair_style lj/charmm/coul/charmm 8.0 10.0
pair_style lj/charmm/coul/charmm 8.0 10.0 7.0 9.0
pair_style lj/charmmfsw/coul/charmmfsh 10.0 12.0
pair_style lj/charmmfsw/coul/charmmfsh 10.0 12.0 9.0
pair_coeff * * 100.0 2.0
pair_coeff 1 1 100.0 2.0 150.0 3.5

pair_style lj/charmm/coul/charmm/implicit 8.0 10.0
pair_style lj/charmm/coul/charmm/implicit 8.0 10.0 7.0 9.0
pair_coeff * * 100.0 2.0
pair_coeff 1 1 100.0 2.0 150.0 3.5

pair_style lj/charmm/coul/long 8.0 10.0
pair_style lj/charmm/coul/long 8.0 10.0 9.0
pair_style lj/charmmfsw/coul/long 8.0 10.0
pair_style lj/charmmfsw/coul/long 8.0 10.0 9.0
pair_coeff * * 100.0 2.0
pair_coeff 1 1 100.0 2.0 150.0 3.5

pair_style lj/charmm/coul/msm 8.0 10.0
pair_style lj/charmm/coul/msm 8.0 10.0 9.0
pair_coeff * * 100.0 2.0
pair_coeff 1 1 100.0 2.0 150.0 3.5
```

### 4.39.3 Description

These pair styles compute Lennard Jones (LJ) and Coulombic interactions with additional switching or shifting functions that ramp the energy and/or force smoothly to zero between an inner and outer cutoff. They implement the widely used CHARMM force field, see *Howto discussion on biomolecular force fields* for details.

The styles with *charmm* (not *charmmfsw* or *charmmfsh*) in their name are the older, original LAMMPS implementations. They compute the LJ and Coulombic interactions with an energy switching function which ramps the energy smoothly to zero between the inner and outer cutoff. This can cause irregularities in pairwise forces (due to the discontinuous second derivative of energy at the boundaries of the switching region), which in some cases can result in detectable artifacts in an MD simulation.

The newer styles with *charmmfsw* or *charmmfsh* in their name replace the energy switching with force switching (fsw) and force shifting (fsh) functions, for LJ and Coulombic interactions respectively.

**Note**

The newer *charmmfsw* or *charmmfsh* styles were released in March 2017. We recommend they be used instead of the older *charmm* styles. This includes the newer *dihedral_style charmmfsw* command. Eventually code from the new styles will propagate into the related pair styles (e.g. implicit, accelerator, free energy variants).

**Note**

The newest CHARMM pair styles reset the Coulombic energy conversion factor used internally in the code, from the LAMMPS value to the CHARMM value, as if it were effectively a parameter of the force field. This is because the CHARMM code uses a slightly different value for the this conversion factor in *real units* (kcal/mol), namely

CHARMM = 332.0716, LAMMPS = 332.06371. This is to enable more precise agreement by LAMMPS with the CHARMM force field energies and forces, when using one of these two CHARMM pair styles.

When using the *lj/charmm/coul/charmm styles*, both the LJ and Coulombic terms require an inner and outer cutoff. They can be the same for both formulas or different depending on whether 2 or 4 arguments are used in the pair_style command. For the *lj/charmmfsw/coul/charmmfsh* style, the LJ term requires both an inner and outer cutoff, while the Coulombic term requires only one cutoff. If the Coulombic cutoff is not specified (2 instead of 3 arguments), the LJ outer cutoff is used for the Coulombic cutoff. In all cases where an inner and outer cutoff are specified, the inner cutoff distance must be less than the outer cutoff. It is typical to make the difference between the inner and outer cutoffs about 2.0 Angstroms.

Style *lj/charmm/coul/charmm/implicit* computes the same formulas as style *lj/charmm/coul/charmm* except that an additional 1/r term is included in the Coulombic formula. The Coulombic energy thus varies as 1/r^2. This is effectively a distance-dependent dielectric term which is a simple model for an implicit solvent with additional screening. It is designed for use in a simulation of an unsolvated biomolecule (no explicit water molecules).

Styles *lj/charmm/coul/long* and *lj/charmm/coul/msm* compute the same formulas as style *lj/charmm/coul/charmm* and style *lj/charmmfsw/coul/long* computes the same formulas as style *lj/charmmfsw/coul/charmmfsh*, except that an additional damping factor is applied to the Coulombic term, so it can be used in conjunction with the *kspace_style* command and its *ewald* or *pppm* or *msm* option. Only one Coulombic cutoff is specified for these styles; if only 2 arguments are used in the pair_style command, then the outer LJ cutoff is used as the single Coulombic cutoff. The Coulombic cutoff specified for these styles means that pairwise interactions within this distance are computed directly; interactions outside that distance are computed in reciprocal space.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- $\varepsilon$ (energy units)
- $\sigma$ (distance units)
- $\varepsilon_{14}$ (energy units)
- $\sigma_{14}$ (distance units)

Note that $\sigma$ is defined in the LJ formula as the zero-crossing distance for the potential, not as the energy minimum at $2^{1/6}\sigma$.

The latter 2 coefficients are optional. If they are specified, they are used in the LJ formula between two atoms of these types which are also first and fourth atoms in any dihedral. No cutoffs are specified because the CHARMM force field does not allow varying cutoffs for individual atom pairs; all pairs use the global cutoff(s) specified in the pair_style command.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.39.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon, sigma, epsilon_14, and sigma_14 coefficients for all of the lj/charmm pair styles can be mixed. The default mix value is *arithmetic* to coincide with the usual settings for the CHARMM force field. See the "pair_modify" command for details.

None of the *lj/charmm* or *lj/charmmfsw* pair styles support the *pair_modify* shift option, since the Lennard-Jones portion of the pair interaction is smoothed to 0.0 at the cutoff.

The *lj/charmm/coul/long* and *lj/charmmfsw/coul/long* styles support the *pair_modify* table option since they can tabulate the short-range portion of the long-range Coulombic interaction.

None of the *lj/charmm* or *lj/charmmfsw* pair styles support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure, since the Lennard-Jones portion of the pair interaction is smoothed to 0.0 at the cutoff.

All of the *lj/charmm* and *lj/charmmfsw* pair styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

The *lj/charmm/coul/long* and *lj/charmmfsw/coul/long* pair styles support the use of the *inner*, *middle*, and *outer* keywords of the *run_style respa* command, meaning the pairwise forces can be partitioned by distance at different levels of the rRESPA hierarchy. The other styles only support the *pair* keyword of run_style respa. See the *run_style* command for details.

---

### 4.39.5 Restrictions

All the styles with *coul/charmm* or *coul/charmmfsh* styles are part of the MOLECULE package. All the styles with *coul/long* style are part of the KSPACE package. They are only enabled if LAMMPS was built with those packages. See the *Build package* doc page for more info.

### 4.39.6 Related commands

*pair_coeff* , *angle_style charmm*, *dihedral_style charmm*, *dihedral_style charmmfsw*, *fix cmap*

### 4.39.7 Default

none

---

**(Brooks)** Brooks, et al, J Comput Chem, 30, 1545 (2009).

**(MacKerell)** MacKerell, Bashford, Bellott, Dunbrack, Evanseck, Field, Fischer, Gao, Guo, Ha, et al, J Phys Chem, 102, 3586 (1998).

**(Steinbach)** Steinbach, Brooks, J Comput Chem, 15, 667 (1994).

## 4.40 pair_style lj/class2 command

Accelerator Variants: *lj/class2/gpu*, *lj/class2/kk*, *lj/class2/omp*

## 4.41 pair_style lj/class2/coul/cut command

Accelerator Variants: *lj/class2/coul/cut/kk*, *lj/class2/coul/cut/omp*

## 4.42 pair_style lj/class2/coul/long command

Accelerator Variants: *lj/class2/coul/long/gpu*, *lj/class2/coul/long/kk*, *lj/class2/coul/long/omp*

### 4.42.1 Syntax

```
pair_style style args
```

- style = *lj/class2* or *lj/class2/coul/cut* or *lj/class2/coul/long*
- args = list of arguments for a particular style

```
lj/class2 args = cutoff
  cutoff = global cutoff for class 2 interactions (distance units)
lj/class2/coul/cut args = cutoff (cutoff2)
  cutoff = global cutoff for class 2 (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
lj/class2/coul/long args = cutoff (cutoff2)
  cutoff = global cutoff for class 2 (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
```

### 4.42.2 Examples

```
pair_style lj/class2 10.0
pair_coeff * * 100.0 2.5
pair_coeff 1 2* 100.0 2.5 9.0

pair_style lj/class2/coul/cut 10.0
pair_style lj/class2/coul/cut 10.0 8.0
pair_coeff * * 100.0 3.0
pair_coeff 1 1 100.0 3.5 9.0
pair_coeff 1 1 100.0 3.5 9.0 9.0

pair_style lj/class2/coul/long 10.0
pair_style lj/class2/coul/long 10.0 8.0
pair_coeff * * 100.0 3.0
pair_coeff 1 1 100.0 3.5 9.0
```

### 4.42.3 Description

The *lj/class2* styles compute a 6/9 Lennard-Jones potential given by

$$E = \varepsilon \left[ 2 \left( \frac{\sigma}{r} \right)^9 - 3 \left( \frac{\sigma}{r} \right)^6 \right] \qquad r < r_c$$

$r_c$ is the cutoff.

The *lj/class2/coul/cut* and *lj/class2/coul/long* styles add a Coulombic term as described for the *lj/cut* pair styles.

See *(Sun)* for a description of the COMPASS class2 force field.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- $\varepsilon$ (energy units)

- $\sigma$ (distance units)
- cutoff1 (distance units)
- cutoff2 (distance units)

The latter 2 coefficients are optional. If not specified, the global class 2 and Coulombic cutoffs are used. If only one cutoff is specified, it is used as the cutoff for both class 2 and Coulombic interactions for this type pair. If both coefficients are specified, they are used as the class 2 and Coulombic cutoffs for this type pair. You cannot specify 2 cutoffs for style *lj/class2*, since it has no Coulombic terms.

For *lj/class2/coul/long* only the class 2 cutoff can be specified since a Coulombic cutoff cannot be specified for an individual I,J type pair. All type pairs use the same global Coulombic cutoff specified in the pair_style command.

---

If the pair_coeff command is not used to define coefficients for a particular I != J type pair, the mixing rule for $\varepsilon$ and $\sigma$ for all class2 potentials is to use the *sixthpower* formulas documented by the *pair_modify* command. The *pair_modify mix* setting is thus ignored for class2 potentials for epsilon and sigma. However it is still followed for mixing the cutoff distance.

---

A version of these styles with a soft core, *lj/cut/soft*, suitable for use in free energy calculations, is part of the FEP package and is documented with the *pair_style */soft* styles. The version with soft core is only available if LAMMPS was built with that package. See the *Build package* page for more info.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.42.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon and sigma coefficients and cutoff distance for all of the lj/class2 pair styles can be mixed. Epsilon and sigma are always mixed with the value *sixthpower*. The cutoff distance is mixed by whatever option is set by the pair_modify command (default = geometric). See the "pair_modify" command for details.

All of the lj/class2 pair styles support the *pair_modify* shift option for the energy of the Lennard-Jones portion of the pair interaction.

The *lj/class2/coul/long* pair style does not support the *pair_modify* table option since a tabulation capability has not yet been added to this potential.

All of the lj/class2 pair styles support the *pair_modify* tail option for adding a long-range tail correction to the energy and pressure of the Lennard-Jones portion of the pair interaction.

All of the lj/class2 pair styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

Only the *lj/class2* and *lj/class2/coul/long* pair styles support the use of the *inner*, *middle*, and *outer* keywords of the *run_style respa* command, meaning the pairwise forces can be partitioned by distance at different levels of the rRESPA hierarchy. The other styles only support the *pair* keyword of run_style respa. See the *run_style* command for details.

### 4.42.5 Restrictions

These styles are part of the CLASS2 package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.42.6 Related commands

*pair_coeff*, *pair_style */soft*

### 4.42.7 Default

none

---

**(Sun)** Sun, J Phys Chem B 102, 7338-7364 (1998).

## 4.43 pair_style colloid command

Accelerator Variants: *colloid/gpu*, *colloid/kk*, *colloid/omp*

### 4.43.1 Syntax

```
pair_style colloid cutoff
```

- cutoff = global cutoff for colloidal interactions (distance units)

### 4.43.2 Examples

```
pair_style colloid 10.0
pair_coeff * *  25 1.0 10.0 10.0
pair_coeff 1 1 144 1.0 0.0 0.0 3.0
pair_coeff 1 2  75.398 1.0 0.0 10.0 9.0
pair_coeff 2 2  39.478 1.0 10.0 10.0 25.0
```

### 4.43.3 Description

Style *colloid* computes pairwise interactions between large colloidal particles and small solvent particles using 3 formulas. A colloidal particle has a size > sigma; a solvent particle is the usual Lennard-Jones particle of size sigma.

The colloid-colloid interaction energy is given by

$$U_A = -\frac{A_{cc}}{6}\left[\frac{2a_1a_2}{r^2-(a_1+a_2)^2}+\frac{2a_1a_2}{r^2-(a_1-a_2)^2}+\ln\left(\frac{r^2-(a_1+a_2)^2}{r^2-(a_1-a_2)^2}\right)\right]$$

$$\begin{aligned}U_R =& \frac{A_{cc}}{37800}\frac{\sigma^6}{r}\left[\frac{r^2-7r(a_1+a_2)+6\left(a_1^2+7a_1a_2+a_2^2\right)}{(r-a_1-a_2)^7}\right.\\ &+\frac{r^2+7r(a_1+a_2)+6\left(a_1^2+7a_1a_2+a_2^2\right)}{(r+a_1+a_2)^7}\\ &-\frac{r^2+7r(a_1-a_2)+6\left(a_1^2-7a_1a_2+a_2^2\right)}{(r+a_1-a_2)^7}\\ &\left.-\frac{r^2-7r(a_1-a_2)+6\left(a_1^2-7a_1a_2+a_2^2\right)}{(r-a_1+a_2)^7}\right]\end{aligned}$$

$$U = U_A + U_R, \qquad r < r_c$$

where $A_{cc}$ is the Hamaker constant, $a_1$ and $a_2$ are the radii of the two colloidal particles, and $r_c$ is the cutoff. This equation results from describing each colloidal particle as an integrated collection of Lennard-Jones particles of size sigma and is derived in *(Everaers)*.

The colloid-solvent interaction energy is given by

$$U = \frac{2\,a^3\,\sigma^3\,A_{cs}}{9\left(a^2-r^2\right)^3}\left[1-\frac{\left(5\,a^6+45\,a^4\,r^2+63\,a^2\,r^4+15\,r^6\right)\sigma^6}{15\,(a-r)^6\,(a+r)^6}\right], \quad r < r_c$$

where $A_{cs}$ is the Hamaker constant, $a$ is the radius of the colloidal particle, and $r_c$ is the cutoff. This formula is derived from the colloid-colloid interaction, letting one of the particle sizes go to zero.

The solvent-solvent interaction energy is given by the usual Lennard-Jones formula

$$U = \frac{A_{ss}}{36}\left[\left(\frac{\sigma}{r}\right)^{12}-\left(\frac{\sigma}{r}\right)^{6}\right], \quad r < r_c$$

with $A_{ss}$ set appropriately, which results from letting both particle sizes go to zero.

When used in combination with *pair_style yukawa/colloid*, the two terms become the so-called DLVO potential, which combines electrostatic repulsion and van der Waals attraction.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- A (energy units)
- $\sigma$ (distance units)
- d1 (distance units)
- d2 (distance units)
- cutoff (distance units)

A is the Hamaker energy prefactor and should typically be set as follows:

- $A_{cc}$ = colloid/colloid = $4\pi^2$ = 39.5
- $A_{cs}$ = colloid/solvent = $\sqrt{A_{cc}A_{ss}}$
- $A_{ss}$ = solvent/solvent = 144 (assuming epsilon = 1, so that 144/36 = 4)

$\sigma$ is the size of the solvent particle or the constituent particles integrated over in the colloidal particle and should typically be set as follows:

- $\sigma_{cc}$ = colloid/colloid = 1.0
- $\sigma_{cs}$ = colloid/solvent = arithmetic mixing between colloid $\sigma$ and solvent $\sigma$
- $\sigma_{ss}$ = solvent/solvent = 1.0 or whatever size the solvent particle is

Thus typically $\sigma_{cs}$ = 1.0, unless the solvent particle's size != 1.0.

D1 and d2 are particle diameters, so that d1 = 2*a1 and d2 = 2*a2 in the formulas above. Both d1 and d2 must be values >= 0. If d1 > 0 and d2 > 0, then the pair interacts via the colloid-colloid formula above. If d1 = 0 and d2 = 0, then the pair interacts via the solvent-solvent formula. I.e. a d value of 0 is a Lennard-Jones particle of size $\sigma$. If either d1 = 0 or d2 = 0 and the other is larger, then the pair interacts via the colloid-solvent formula.

Note that the diameter of a particular particle type may appear in multiple pair_coeff commands, as it interacts with other particle types. You should ensure the particle diameter is specified consistently each time it appears.

The last coefficient is optional. If not specified, the global cutoff specified in the pair_style command is used. However, you typically want different cutoffs for interactions between different particle sizes. E.g. if colloidal particles of diameter 10 are used with solvent particles of diameter 1, then a solvent-solvent cutoff of 2.5 would correspond to a colloid-colloid cutoff of 25. A good rule-of-thumb is to use a colloid-solvent cutoff that is half the big diameter + 4 times the small diameter. I.e. 9 = 5 + 4 for the colloid-solvent cutoff in this case.

**Note**

When using pair_style colloid for a mixture with 2 (or more) widely different particles sizes (e.g. sigma=10 colloids in a background sigma=1 LJ fluid), you will likely want to use these commands for efficiency: *neighbor multi* and *comm_modify multi*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.43.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the A, sigma, d1, and d2 coefficients and cutoff distance for this pair style can be mixed. A is an energy value mixed like a LJ epsilon. D1 and d2 are distance values and are mixed like sigma. The default mix value is *geometric*. See the "pair_modify" command for details.

This pair style supports the *pair_modify* shift option for the energy of the pair interaction.

The *pair_modify* table option is not relevant for this pair style.

This pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.43.5 Restrictions

This style is part of the COLLOID package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Normally, this pair style should be used with finite-size particles which have a diameter, e.g. see the *atom_style sphere* command. However, this is not a requirement, since the only definition of particle size is via the pair_coeff parameters for each type. In other words, the physical radius of the particle is ignored. Thus you should ensure that the d1,d2 parameters you specify are consistent with the physical size of the particles of that type.

Per-particle polydispersity is not yet supported by this pair style; only per-type polydispersity is enabled via the pair_coeff parameters.

### 4.43.6 Related commands

*pair_coeff*

### 4.43.7 Default

none

---

**(Everaers)** Everaers, Ejtehadi, Phys Rev E, 67, 041710 (2003).

## 4.44 pair_style comb command

Accelerator Variants: *comb/omp*

## 4.45 pair_style comb3 command

### 4.45.1 Syntax

```
pair_style comb
pair_style comb3 keyword
```

```
keyword = polar
  polar value = polar_on or polar_off = whether or not to include atomic polarization
```

### 4.45.2 Examples

```
pair_style comb
pair_coeff * * ../potentials/ffield.comb Si
pair_coeff * * ../potentials/ffield.comb Hf Si O

pair_style comb3 polar_off
pair_coeff * * ../potentials/ffield.comb3 O Cu N C O
```

### 4.45.3 Description

Style *comb* computes the second generation variable charge COMB (Charge-Optimized Many-Body) potential. Style *comb3* computes the third-generation COMB potential. These COMB potentials are described in *(COMB)* and *(COMB3)*. Briefly, the total energy $E_T$ of a system of atoms is given by

$$E_T = \sum_i [E_i^{self}(q_i) + \sum_{j>i} [E_{ij}^{short}(r_{ij}, q_i, q_j) + E_{ij}^{Coul}(r_{ij}, q_i, q_j)] + E^{polar}(q_i, r_{ij}) + E^{vdW}(r_{ij}) + E^{barr}(q_i) + E^{corr}(r_{ij}, \theta_{jik})]$$

where $E_i^{self}$ is the self-energy of atom $i$ (including atomic ionization energies and electron affinities), $E_{ij}^{short}$ is the bond-order potential between atoms $i$ and $j$, $E_{ij}^{Coul}$ is the Coulomb interactions, $E^{polar}$ is the polarization term for organic systems (style *comb3* only), $E^{vdW}$ is the van der Waals energy (style *comb3* only), $E^{barr}$ is a charge barrier function, and $E^{corr}$ are angular correction terms.

The COMB potentials (styles *comb* and *comb3*) are variable charge potentials. The equilibrium charge on each atom is calculated by the electronegativity equalization (QEq) method. See *Rick* for further details. This is implemented by the *fix qeq/comb* command, which should normally be specified in the input script when running a model with the COMB potential. The *fix qeq/comb* command has options that determine how often charge equilibration is performed, its convergence criterion, and which atoms are included in the calculation.

Only a single pair_coeff command is used with the *comb* and *comb3* styles which specifies the COMB potential file with parameters for all needed elements. These are mapped to LAMMPS atom types by specifying N additional arguments after the potential file in the pair_coeff command, where N is the number of LAMMPS atom types.

For example, if your LAMMPS simulation of a Si/SiO2/ HfO2 interface has 4 atom types, and you want the first and last to be Si, the second to be Hf, and the third to be O, and you would use the following pair_coeff command:

```
pair_coeff * * ../potentials/ffield.comb Si Hf O Si
```

The first two arguments must be * * so as to span all LAMMPS atom types. The first and last Si arguments map LAMMPS atom types 1 and 4 to the Si element in the *ffield.comb* file. The second Hf argument maps LAMMPS atom type 2 to the Hf element, and the third O argument maps LAMMPS atom type 3 to the O element in the potential file. If a mapping value is specified as NULL, the mapping is not performed. This can be used when a *comb* potential is used as part of the *hybrid* pair style. The NULL values are placeholders for atom types that will be used with other potentials.

For style *comb*, the provided potential file *ffield.comb* contains all currently-available second generation COMB parameterizations: for Si, Cu, Hf, Ti, O, their oxides and Zr, Zn and U metals. For style *comb3*, the potential file *ffield.comb3* contains all currently-available third generation COMB parameterizations: O, Cu, N, C, H, Ti, Zn and Zr. The status of the optimization of the compounds, for example Cu2O, TiN and hydrocarbons, are given in the following table:

| | O | Cu | N | C | H | Ti | Zn | Zr |
|---|---|---|---|---|---|---|---|---|
| O | F | F | F | F | F | F | F | F |
| Cu | F | F | P | F | F | P | F | P |
| N | F | P | F | M | F | P | P | P |
| C | F | F | M | F | F | M | M | M |
| H | F | F | F | F | F | M | M | F |
| Ti | F | P | P | M | M | F | P | P |
| Zn | F | F | P | M | M | P | F | P |
| Zr | F | P | P | M | F | P | P | F |

- F = Fully optimized
- M = Only optimized for dimer molecule
- P = in progress, but have it from mixing rule

For style *comb3*, in addition to ffield.comb3, a special parameter file, *lib.comb3*, that is exclusively used for C/O/H systems, will be automatically loaded if carbon atom is detected in LAMMPS input structure. This file must be in your working directory or in the directories listed in the environment variable `LAMMPS_POTENTIALS`, as described on the *pair_coeff* command doc page.

The keyword *polar* indicates whether the force field includes the atomic polarization. Since the equilibration of the polarization has not yet been implemented, it can only set polar_off at present.

**Note**

You can not use potential file *ffield.comb* with style *comb3*, nor file *ffield.comb3* with style *comb*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.45.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, where types I and J correspond to two different element types, mixing is performed by LAMMPS as described above from values in the potential file.

These pair styles does not support the *pair_modify* shift, table, and tail options.

These pair styles do not write its information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style, pair_coeff, and *fix qeq/comb* commands in an input script that reads a restart file.

These pair styles can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.45.5 Restrictions

These pair styles are part of the MANYBODY package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

These pair styles requires the *newton* setting to be "on" for pair interactions.

The COMB potentials in the *ffield.comb* and *ffield.comb3* files provided with LAMMPS (see the potentials directory) are parameterized for metal *units*. You can use the COMB potential with any LAMMPS units, but you would need to create your own COMB potential file with coefficients listed in the appropriate units if your simulation does not use "metal" units.

### 4.45.6 Related commands

*pair_style*, *pair_coeff*, *fix qeq/comb*

### 4.45.7 Default

none

---

**(COMB)** T.-R. Shan, B. D. Devine, T. W. Kemper, S. B. Sinnott, and S. R. Phillpot, Phys. Rev. B 81, 125328 (2010)

**(COMB3)** T. Liang, T.-R. Shan, Y.-T. Cheng, B. D. Devine, M. Noordhoek, Y. Li, Z. Lu, S. R. Phillpot, and S. B. Sinnott, Mat. Sci. & Eng: R 74, 255-279 (2013).

**(Rick)** S. W. Rick, S. J. Stuart, B. J. Berne, J Chem Phys 101, 6141 (1994).

## 4.46 pair_style cosine/squared command

Accelerator Variants: *cosine/squared/kk*, *cosine/squared/omp*

### 4.46.1 Syntax

```
pair_style cosine/squared cutoff
```

- cutoff = global cutoff for cosine-squared interactions (distance units)

```
pair_coeff I J eps sigma
pair_coeff I J eps sigma cutoff
pair_coeff I J eps sigma wca
pair_coeff I J eps sigma cutoff wca
```

- I, J = a particle type
- eps = interaction strength, i.e. the depth of the potential minimum (energy units)
- sigma = distance of the potential minimum from 0
- cutoff = the cutoff distance for this pair type, if different from global (distance units)
- wca = if specified a Weeks-Chandler-Andersen potential (with eps strength and minimum at sigma) is added, otherwise not

### 4.46.2 Examples

```
pair_style cosine/squared 3.0
pair_coeff * * 1.0 1.3
pair_coeff 1 3 1.0 1.3 2.0
pair_coeff 1 3 1.0 1.3 wca
pair_coeff 1 3 1.0 1.3 2.0 wca
```

### 4.46.3 Description

Style *cosine/squared* computes a potential of the form

$$E = \begin{cases} -\varepsilon & r < \sigma \\ -\varepsilon \cos\left(\frac{\pi(r-\sigma)}{2(r_c-\sigma)}\right)^2 & \sigma \le r < r_c \\ 0 & r \ge r_c \end{cases}$$

between two point particles, where $(\sigma, -\varepsilon)$ is the location of the (rightmost) minimum of the potential, as explained in the syntax section above.

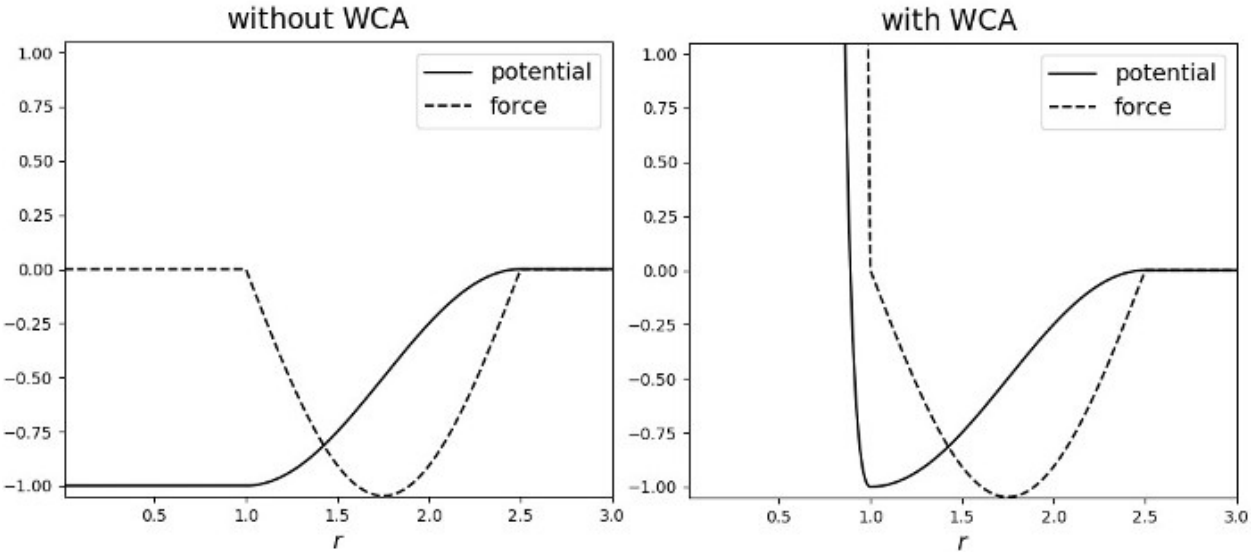
This potential was first used in *(Cooke)* for a coarse-grained lipid membrane model. It is generally very useful as a non-specific interaction potential because it is fully adjustable in depth and width while joining the minimum at (sigma, -epsilon) and zero at (cutoff, 0) smoothly, requiring no shifting and causing no related artifacts, tail energy calculations etc. This evidently requires *cutoff* to be larger than *sigma*.

If the *wca* option is used then a Weeks-Chandler-Andersen potential *(Weeks)* is added to the above specified cosine-squared potential, specifically the following:

$$E = \varepsilon \left[ \left(\frac{\sigma}{r}\right)^{12} - 2\left(\frac{\sigma}{r}\right)^{6} + 1 \right], \quad r < \sigma$$

In this case, and this case only, the $\sigma$ parameter can be equal to *cutoff* ($\sigma$ = cutoff) which will result in ONLY the WCA potential being used (and print a warning), so the minimum will be attained at (sigma, 0). This is a convenience feature that enables a purely repulsive potential to be used without a need to define an additional pair style and use the hybrid styles.

The energy and force of this pair style for parameters epsilon = 1.0, sigma = 1.0, cutoff = 2.5, with and without the WCA potential, are shown in the graphs below:



---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.46.4 Mixing, shift, table, tail correction, restart, rRESPA info

Mixing is not supported for this style.

The *shift*, *table* and *tail* options are not relevant for this style.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

These pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

---

### 4.46.5 Restrictions

The *cosine/squared* style is part of the EXTRA-PAIR package. It is only enabled if LAMMPS is build with that package. See the *Build package* page for more info.

### 4.46.6 Related commands

*pair_coeff* , *pair_style lj/cut*

### 4.46.7 Default

none

**(Cooke)** "Cooke, Kremer and Deserno, Phys. Rev. E, 72, 011506 (2005)"

**(Weeks)** "Weeks, Chandler and Andersen, J. Chem. Phys., 54, 5237 (1971)"

## 4.47 pair_style coul/cut command

Accelerator Variants: *coul/cut/gpu*, *coul/cut/kk*, *coul/cut/omp*

## 4.48 pair_style coul/cut/global command

Accelerator Variants: *coul/cut/global/kk*, *coul/cut/global/omp*

## 4.49 pair_style coul/ctip command

## 4.50 pair_style coul/debye command

Accelerator Variants: *coul/debye/gpu*, *coul/debye/kk*, *coul/debye/omp*

## 4.51 pair_style coul/dsf command

Accelerator Variants: *coul/dsf/gpu*, *coul/dsf/kk*, *coul/dsf/omp*

## 4.52 pair_style coul/exclude command

Accelerator Variants: *coul/exclude/omp*

## 4.53 pair_style coul/long command

Accelerator Variants: *coul/long/omp*, *coul/long/kk*, *coul/long/gpu*

## 4.54 pair_style coul/msm command

Accelerator Variants: *coul/msm/omp*

## 4.55 pair_style coul/streitz command

Accelerator Variants: *coul/streitz/omp*

## 4.56 pair_style coul/wolf command

Accelerator Variants: *coul/wolf/kk*, *coul/wolf/omp*

## 4.57 pair_style tip4p/cut command

Accelerator Variants: *tip4p/cut/omp*

## 4.58 pair_style tip4p/long command

Accelerator Variants: *tip4p/long/omp*

### 4.58.1 Syntax

```
pair_style coul/cut cutoff
pair_style coul/cut/global cutoff
pair_style coul/ctip alpha cutoff
pair_style coul/debye kappa cutoff
pair_style coul/dsf alpha cutoff
pair_style coul/esp cutoff
pair_style coul/exclude cutoff
```

(continues on next page)

(continued from previous page)

```
pair_style coul/long cutoff
pair_style coul/wolf alpha cutoff
pair_style coul/streitz cutoff keyword alpha

* cutoff = global cutoff for Coulombic interactions
* kappa = Debye length (inverse distance units)
* alpha = damping parameter (inverse distance units)
```

```
pair_style tip4p/cut otype htype btype atype qdist cutoff
pair_style tip4p/long otype htype btype atype qdist cutoff

* otype,htype = atom types (numeric or type label) for TIP4P O and H
* btype,atype = bond and angle types (numeric or type label) for TIP4P waters
* qdist = distance from O atom to massless charge (distance units)
```

### 4.58.2 Examples

```
pair_style coul/cut 2.5
pair_coeff * *
pair_coeff 2 2 3.5

pair_style coul/ctip 0.30 12.0
pair_coeff * * NiO.ctip Ni O

pair_style coul/debye 1.4 3.0
pair_coeff * *
pair_coeff 2 2 3.5

pair_style coul/dsf 0.05 10.0
pair_coeff * *

pair_style coul/esp 10.0
pair_coeff * *

pair_style hybrid/overlay coul/exclude 10.0 ...
pair_coeff * * coul/exclude

pair_style coul/long 10.0
pair_coeff * *

pair_style coul/msm 10.0
pair_coeff * *

pair_style coul/wolf 0.2 9.0
pair_coeff * *

pair_style coul/streitz 12.0 ewald
pair_style coul/streitz 12.0 wolf 0.30
pair_coeff * * AlO.streitz Al O

pair_style coul/streitz 12.0 wolf 0.3         # default taper width is zero
```

(continues on next page)

(continued from previous page)

```
pair_style coul/streitz 12.0 wolf 0.3 2.0
pair_style coul/streitz 12.0 dsf 0.3          # default taper width is zero
pair_style coul/streitz 12.0 dsf 0.3 2.0
pair_coeff * * GaN.streitz Ga N

pair_style tip4p/cut 1 2 7 8 0.15 12.0
pair_coeff * *

pair_style tip4p/long 1 2 7 8 0.15 10.0
pair_coeff * *

pair_style tip4p/cut OW HW HW-OW HW-OW-HW 0.15 12.0
labelmap atom 1 OW 2 HW
labelmap bond 1 HW-OW
labelmap angle 1 HW-OW-HW
pair_coeff * *
```

### 4.58.3 Description

The *coul/cut* style computes the standard Coulombic interaction potential given by

$$E = \frac{Cq_iq_j}{\varepsilon r} \qquad r < r_c$$

where C is an energy-conversion constant, Qi and Qj are the charges on the two atoms, and $\varepsilon$ is the dielectric constant which can be set by the *dielectric* command. The cutoff $r_c$ truncates the interaction distance.

Pair style *coul/cut/global* computes the same Coulombic interactions as style *coul/cut* except that it allows only a single global cutoff and thus makes it compatible for use in combination with long-range coulomb styles in *hybrid pair styles*.

---

Added in version 19Nov2024.

Style *coul/ctip* computes the Coulomb interactions as described in *Plummer*. It uses the the damped shifted model as in style *coul/dsf* but is further extended to the second derivative of the potential and incorporates empirical charge shielding meant to approximate the more expensive Coulomb integrals used in style *coul/streitz*. More details can be found in the referenced paper. Like the style *coul/streitz*, style *coul/ctip* is a variable charge potential and must be hybridized with a short-range potential via the *pair_style hybrid/overlay* command. Charge equilibration must be performed with the *fix qeq/ctip* command. For example:

```
pair_style hybrid/overlay eam/fs coul/ctip 0.30 12.0
pair_coeff * * eam/fs NiO.eam.fs Ni O
pair_coeff * * coul/ctip NiO.ctip Ni O
fix 1 all qeq/ctip 1 12.0 1.0e-8 100 coul/ctip cdamp 0.30 maxrepeat 10
```

See the examples/ctip directory for an example input script using the CTIP potential. An Ni-O CTIP and EAM/FS parameterization are included for use with the example.

---

Style *coul/debye* adds an additional exp() damping factor to the Coulombic term, given by

$$E = \frac{Cq_iq_j}{\varepsilon r} \exp(-\kappa r) \qquad r < r_c$$

where $\kappa$ is the Debye length. This potential is another way to mimic the screening effect of a polar solvent.

---

Style *coul/dsf* computes Coulombic interactions via the damped shifted force model described in *Fennell*, given by:

$$E = q_i q_j \left[ \frac{\mathrm{erfc}(\alpha r)}{r} - \frac{\mathrm{erfc}(\alpha r_c)}{r_c} + \left( \frac{\mathrm{erfc}(\alpha r_c)}{r_c^2} + \frac{2\alpha}{\sqrt{\pi}} \frac{\exp(-\alpha^2 r_c^2)}{r_c} \right) (r - r_c) \right] \qquad r < r_c$$

where $\alpha$ is the damping parameter and *erfc()* is the complementary error-function. The potential corrects issues in the Wolf model (described below) to provide consistent forces and energies (the Wolf potential is not differentiable at the cutoff) and smooth decay to zero.

---

Style *coul/wolf* computes Coulombic interactions via the Wolf summation method, described in *Wolf*, given by:

$$E_i = \frac{1}{2} \sum_{j \neq i} \frac{q_i q_j \mathrm{erfc}(\alpha r_{ij})}{r_{ij}} + \frac{1}{2} \sum_{j \neq i} \frac{q_i q_j \mathrm{erf}(\alpha r_{ij})}{r_{ij}} \qquad r < r_c$$

where $\alpha$ is the damping parameter, and *erf()* and *erfc()* are error-function and complementary error-function terms. This potential is essentially a short-range, spherically-truncated, charge-neutralized, shifted, pairwise *1/r* summation. With a manipulation of adding and subtracting a self term (for i = j) to the first and second term on the right-hand-side, respectively, and a small enough $\alpha$ damping parameter, the second term shrinks and the potential becomes a rapidly-converging real-space summation. With a long enough cutoff and small enough $\alpha$ parameter, the energy and forces calculated by the Wolf summation method approach those of the Ewald sum. So it is a means of getting effective long-range interactions with a short-range potential.

---

Style *coul/streitz* is the Coulomb pair interaction defined as part of the Streitz-Mintmire potential, as described in *this paper*, in which charge distribution about an atom is modeled as a Slater 1*s* orbital. More details can be found in the referenced paper. To fully reproduce the published Streitz-Mintmire potential, which is a variable charge potential, style *coul/streitz* must be used with *pair_style eam/alloy* (or some other short-range potential that has been parameterized appropriately) via the *pair_style hybrid/overlay* command. Likewise, charge equilibration must be performed via the *fix qeq/slater* command. For example:

```
pair_style hybrid/overlay coul/streitz 12.0 wolf 0.31 eam/alloy
pair_coeff * * coul/streitz Al0.streitz Al O
pair_coeff * * eam/alloy Al0.eam.alloy Al O
fix 1 all qeq/slater 1 12.0 1.0e-6 100 coul/streitz
```

The keyword *wolf* in the coul/streitz command denotes computing Coulombic interactions via Wolf summation. An additional damping parameter is required for the Wolf summation, as described for the coul/wolf potential above. Alternatively, Coulombic interactions can be computed via an Ewald summation. For example:

```
pair_style hybrid/overlay coul/streitz 12.0 ewald eam/alloy
kspace_style ewald 1e-6
```

Keyword *ewald* does not need a damping parameter, but a *kspace_style* must be defined, which can be style *ewald*, *esp* or *pppm*. The Ewald method was used in Streitz and Mintmire's original paper, but a Wolf summation offers a speed-up in some cases.

For the fix qeq/slater command, the *qfile* can be a filename that contains QEq parameters as discussed on the *fix qeq* command doc page. Alternatively *qfile* can be replaced by "coul/streitz", in which case the fix will extract QEq parameters from the coul/streitz pair style itself.

See the `examples/streitz` directory for an example input script that uses the Streitz-Mintmire potential. The potentials directory has the `Al0.eam.alloy` and `Al0.streitz` potential files used by the example.

Note that the Streitz-Mintmire potential is generally used for oxides, but there is no conceptual problem with extending it to nitrides and carbides (such as SiC, TiN). Pair coul/streitz used by itself or with any other pair style such as EAM, MEAM, Tersoff, or LJ in hybrid/overlay mode. To do this, you would need to provide a Streitz-Mintmire parameterization for the material being modeled.

Changed in version 11Feb2026.

In previous versions of LAMMPS, the real-space summations of Coulomb interactions were done by replacing *1/r* using a damped potential *erfc(alpha*r)/r* with the parameter *alpha* controlling the rate of decay. However, any finite value of *alpha* leads to a jump at the cutoff, which interferes with equilibration if atoms move across the cutoff. The charge-neutralized potential of *(Wolf et al.)* (*wolf*) and its extension by *(Fennell and Gezelter)* (*dsf*) solve this problem. An extension was implemented to specify the width of taper (see *(Mei et al.)*) to smoothly terminate the Coulomb integrals at the cutoff. This is done by specifying the optional arguments *wolf* and *dsf* with the value representing the width of taper that smoothly terminates the Coulomb integrals. For example, if the cutoff is 8 A and the taper width is 2 A, the Coulomb integrals are smoothly rescaled from their actual value at r=6 A to zero at r=8 A. For backward compatibility, the default taper width is zero.

An implementation of the Streitz-Mintmire potential for GaN due to *(Groger and Fikar)* can be found in the examples/streitz directory. The electrostatic parameters of Ga and N are stored in file GaN.streitz and the short-range tersoff/mod potential in the file GaN.streitz+tersoff.mod. The total potential must be specified as:

```
pair_style hybrid/overlay tersoff/mod coul/streitz 12.0 dsf 0.3 2.0
pair_coeff * * tersoff/mod GaN.streitz+tersoff.mod Ga N
pair_coeff * * coul/streitz GaN.streitz Ga N
```

where the last three parameters specify the method of calculation of Coulomb interactions (*ewald*, *wolf* or *dsf*), the value of *alpha*, and the width of taper to smoothly terminate the Coulomb integrals.

---

Pair style *coul/exclude* computes Coulombic interactions like *coul/cut* but **only** applies them to excluded pairs using a scaling factor of $\gamma - 1.0$ with $\gamma$ being the factor assigned to that excluded pair via the *special_bonds coul* setting. With this it is possible to treat Coulomb interactions for molecular systems with *kspace style scafacos*, which always computes the *full* Coulomb interactions without exclusions. Pair style *coul/exclude* will then *subtract* the excluded interactions accordingly. So to achieve the same forces as with `pair_style lj/cut/coul/long 12.0` with `kspace_style pppm 1.0e-6`, one would use `pair_style lj/cut/coul/esp 12.0` with `kspace_style esp 1.0e-6` or `pair_style hybrid/overlay lj/cut 12.0 coul/exclude 12.0` with `kspace_style scafacos p3m 1.0e-6`.

Styles *coul/esp*, *coul/long* and *coul/msm* compute the same Coulombic interactions as style *coul/cut* except that an additional damping factor is applied so it can be used in conjunction with the *kspace_style* command and its *ewald*, *esp* or *pppm* option. The Coulombic cutoff specified for this style means that pairwise interactions within this distance are computed directly; interactions outside that distance are computed in reciprocal space.

Styles *tip4p/cut* and *tip4p/long* implement the Coulomb part of the TIP4P water model of *(Jorgensen)*, which introduces a massless site located a short distance away from the oxygen atom along the bisector of the HOH angle. The atomic types of the oxygen and hydrogen atoms, the bond and angle types for OH and HOH interactions, and the distance to the massless charge site are specified as pair_style arguments. Style *tip4p/cut* uses a global cutoff for Coulomb interactions; style *tip4p/long* is for use with a long-range Coulombic solver (Ewald, ESP or PPPM).

**Note**

For each TIP4P water molecule in your system, the atom IDs for the O and 2 H atoms must be consecutive, with the O atom first. This is to enable LAMMPS to "find" the 2 H atoms associated with each O atom. For example, if the atom ID of an O atom in a TIP4P water molecule is 500, then its 2 H atoms must have IDs 501 and 502.

> **Note**
>
> If using type labels, the type labels must be defined before calling the *pair_coeff* command.

See the *Howto tip4p* page for more information on how to use the TIP4P pair styles and lists of parameters to set. Note that the neighbor list cutoff for Coulomb interactions is effectively extended by a distance 2*qdist when using the TIP4P pair style, to account for the offset distance of the fictitious charges on O atoms in water molecules. Thus it is typically best in an efficiency sense to use a LJ cutoff >= Coulombic cutoff + 2*qdist, to shrink the size of the neighbor list. This leads to slightly larger cost for the long-range calculation, so you can test the trade-off for your model.

---

Note that these potentials are designed to be combined with other pair potentials via the *pair_style hybrid/overlay* command. This is because they have no repulsive core. Hence if they are used by themselves, there will be no repulsion to keep two oppositely charged particles from moving arbitrarily close to each other.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- cutoff (distance units)

For *coul/cut* and *coul/debye* the cutoff coefficient is optional. If it is not used (as in some of the examples above), the default global value specified in the pair_style command is used.

For *coul/cut/global*, *coul/long* and *coul/msm* no cutoff can be specified for an individual I,J type pair via the pair_coeff command. All type pairs use the same global Coulomb cutoff specified in the pair_style command.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.58.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the cutoff distance for the *coul/cut* style can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

The *pair_modify* shift option is not relevant for these pair styles.

The *coul/long* style supports the *pair_modify* table option for tabulation of the short-range portion of the long-range Coulombic interaction.

These pair styles do not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

These pair styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

These pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

---

### 4.58.5 Restrictions

The *coul/long*, *coul/msm*, *coul/streitz*, and *tip4p/long* styles are part of the KSPACE package. The *coul/cut/global*, *coul/exclude*, and *coul/ctip* styles are part of the EXTRA-PAIR package. The *tip4p/cut* style is part of the MOLECULE package. A pair style is only enabled if LAMMPS was built with its corresponding package. See the *Build package* page for more info.

### 4.58.6 Related commands

*pair_coeff*, *pair_style hybrid/overlay*, *kspace_style*

### 4.58.7 Default

none

---

**(Wolf)** D. Wolf, P. Keblinski, S. R. Phillpot, J. Eggebrecht, J Chem Phys, 110, 8254 (1999).

**(Fennell)** C. J. Fennell, J. D. Gezelter, J Chem Phys, 124, 234104 (2006).

**(Streitz)** F. H. Streitz, J. W. Mintmire, Phys Rev B, 50, 11996-12003 (1994).

**(Plummer)** G. Plummer, J. P. Tavenner, M. I. Mendelev, Z. Wu, J. W. Lawson, J Chemical Physics, 162, 054709 (2025).

**(Jorgensen)** Jorgensen, Chandrasekhar, Madura, Impey, Klein, J Chem Phys, 79, 926 (1983).

**(Mei)** J. Mei, J. W. Davenport, G. W. Fernando, Phys. Rev. B 43, 4653 (1991).

**(Groger)** R. Groger, J. Fikar, Acta Mater. (2026) in review.

## 4.59 pair_style coul/diel command

Accelerator Variants: *coul/diel/kk*, *coul/diel/omp*

### 4.59.1 Syntax

```
pair_style coul/diel cutoff
```

cutoff = global cutoff (distance units)

### 4.59.2 Examples

```
pair_style coul/diel 3.5
pair_coeff 1 4 78. 1.375 0.112
```

### 4.59.3 Description

Style *coul/diel* computes a Coulomb correction for implicit solvent ion interactions in which the dielectric permittivity is distance dependent. The dielectric permittivity $\varepsilon_D(r)$ connects to limiting regimes: One limit is defined by a small dielectric permittivity (close to vacuum) at or close to contact separation between the ions. At larger separations the dielectric permittivity reaches a bulk value used in the regular Coulomb interaction coul/long or coul/cut. The transition is modeled by a hyperbolic function which is incorporated in the Coulomb correction term for small ion separations as follows

$$E = \frac{C q_i q_j}{\varepsilon r}\left(\frac{\varepsilon}{\varepsilon_D(r)} - 1\right) \qquad r < r_c$$

$$\varepsilon_D(r) = \frac{5.2+\varepsilon}{2} + \frac{\varepsilon - 5.2}{2}\tanh\left(\frac{r - r_{me}}{\sigma_e}\right)$$

where $r_{me}$ is the inflection point of $\varepsilon_D(r)$ and $\sigma_e$ is a slope defining length scale. C is the same Coulomb conversion factor as in the pair_styles coul/cut, coul/long, and coul/debye. In this way the Coulomb interaction between ions is corrected at small distances r. The lower limit of $\varepsilon_D(r \to 0) = 5.2$ due to dielectric saturation *(Stiles)* while the Coulomb interaction reaches its bulk limit by setting $\varepsilon_D(r \to \infty) = \varepsilon$, the bulk value of the solvent which is 78 for water at 298K.

Examples of the use of this type of Coulomb interaction include implicit solvent simulations of salt ions *(Lenart)* and of ionic surfactants *(Jusufi)*. Note that this potential is only reasonable for implicit solvent simulations and in combination with coul/cut or coul/long. It is also usually combined with gauss/cut, see *(Lenart)* or *(Jusufi)*.

The following coefficients must be defined for each pair of atom types via the *pair_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $\varepsilon$ (no units)
- $r_{me}$ (distance units)
- $\sigma_e$ (distance units)

The global cutoff ($r_c$) specified in the pair_style command is used.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.59.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support parameter mixing. Coefficients must be given explicitly for each type of particle pairs.

This pair style supports the *pair_modify* shift option for the energy of the Gauss-potential portion of the pair interaction.

The *pair_modify* table option is not relevant for this pair style.

This pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.59.5 Restrictions

This style is part of the EXTRA-PAIR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.59.6 Related commands

*pair_coeff* *pair_style gauss/cut*

### 4.59.7 Default

none

---

**(Stiles)** Stiles, Hubbard, and Kayser, J Chem Phys, 77, 6189 (1982).

**(Lenart)** Lenart, Jusufi, and Panagiotopoulos, J Chem Phys, 126, 044509 (2007).

**(Jusufi)** Jusufi, Hynninen, and Panagiotopoulos, J Phys Chem B, 112, 13783 (2008).

## 4.60 pair_style coul/shield command

Accelerator Variants: *coul/shield/kk*, *coul/shield/omp*

### 4.60.1 Syntax

```
pair_style coul/shield cutoff tap_flag
```

- cutoff = global cutoff (distance units)
- tap_flag = 0/1 to turn off/on the taper function

### 4.60.2 Examples

```
pair_style coul/shield 16.0 1
pair_coeff 1 2 0.70
```

### 4.60.3 Description

Style *coul/shield* computes a Coulomb interaction for boron and nitrogen atoms located in different layers of hexagonal boron nitride. This potential is designed be used in combination with the pair style *ilp/graphene/hbn*

**Note**

This potential is intended for electrostatic interactions between two different layers of hexagonal boron nitride. Therefore, to avoid interaction within the same layers, each layer should have a separate molecule id and is recommended to use the "full" atom style, so that charge and molecule ID information is included.

$$E = \frac{1}{2}\sum_i \sum_{j \neq i} V_{ij}$$

$$V_{ij} = \mathrm{Tap}(r_{ij}) \frac{\kappa q_i q_j}{\sqrt[3]{r_{ij}^3 + (1/\lambda_{ij})^3}}$$

$$\mathrm{Tap}(r_{ij}) = 20\left(\frac{r_{ij}}{R_{cut}}\right)^7 - 70\left(\frac{r_{ij}}{R_{cut}}\right)^6 + 84\left(\frac{r_{ij}}{R_{cut}}\right)^5 - 35\left(\frac{r_{ij}}{R_{cut}}\right)^4 + 1$$

Where $\mathrm{Tap}(r_{ij})$ is the taper function which provides a continuous cutoff (up to third derivative) for inter-atomic separations larger than $r_c$ *(Leven1)*, *(Leven2)* and *(Maaravi)*. Here $\lambda$ is the shielding parameter that eliminates the short-range singularity of the classical mono-polar electrostatic interaction expression *(Maaravi)*.

The shielding parameter $\lambda$ (1/distance units) must be defined for each pair of atom types via the *pair_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

The global cutoff ($r_c$) specified in the pair_style command is used.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.60.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support parameter mixing. Coefficients must be given explicitly for each type of particle pairs.

The *pair_modify table* option is not relevant for this pair style.

This pair style does not support the *pair_modify tail* option for adding long-range tail corrections to energy and pressure.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.60.5 Restrictions

This pair style is part of the INTERLAYER package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.60.6 Related commands

*pair_coeff* *pair_style ilp/graphene/hbn*

### 4.60.7 Default

tap_flag = 1

---

**(Leven1)** I. Leven, I. Azuri, L. Kronik and O. Hod, J. Chem. Phys. 140, 104106 (2014).

**(Leven2)** I. Leven et al, J. Chem.Theory Comput. 12, 2896-905 (2016).

**(Maaravi)** T. Maaravi et al, J. Phys. Chem. C 121, 22826-22835 (2017).

## 4.61 pair_style coul/slater command

## 4.62 pair_style coul/slater/cut command

Accelerator Variants: *coul/slater/cut/omp*

## 4.63 pair_style coul/slater/long command

Accelerator Variants: *coul/slater/long/gpu*, *coul/slater/long/kk*, *coul/slater/long/omp*

### 4.63.1 Syntax

```
pair_style coul/slater/cut lambda cutoff
pair_style coul/slater/long lambda cutoff
```

lambda = decay length of the charge (distance units) cutoff = cutoff (distance units)

### 4.63.2 Examples

```
pair_style coul/slater/cut 1.0 3.5
pair_coeff * *
pair_coeff 2 2 2.5

pair_style coul/slater/long 1.0 12.0
pair_coeff * *
pair_coeff 1 1 5.0
```

### 4.63.3 Description

Styles *coul/slater/** compute electrostatic interactions in mesoscopic models which employ potentials without explicit excluded-volume interactions. The goal is to prevent artificial ionic pair formation by including a charge distribution in the Coulomb potential, following the formulation of *(Melchor)*:

$$E = \frac{Cq_iq_j}{\varepsilon r}\left(1 - \left(1 + \frac{r_{ij}}{\lambda} exp\left(-2r_{ij}/\lambda\right)\right)\right) \qquad r < r_c$$

where $r_c$ is the cutoff distance and $\lambda$ is the decay length of the charge. C is the same Coulomb conversion factor as in the pair_styles coul/cut and coul/long. In this way the Coulomb interaction between ions is corrected at small distances r. For the *coul/slater/cut* style, the potential energy for distances larger than the cutoff is zero, while for the *coul/slater/long*, the long-range interactions are computed either by the Ewald or the PPPM technique.

Phenomena that can be captured at a mesoscopic level using this type of electrostatic interactions include the formation of polyelectrolyte-surfactant aggregates, charge stabilization of colloidal suspensions, and the formation of complexes driven by charged species in biological systems. *(Vaiwala)*.

The cutoff distance is optional. If it is not used, the default global value specified in the pair_style command is used. For each pair of atom types, a specific cutoff distance can be defined via the *pair_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $r_c$ (distance units)

The global decay length of the charge ($\lambda$) specified in the pair_style command is used for all pairs.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.63.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the cutoff distance for the *coul/slater* styles can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

The *pair_modify* shift and table options are not relevant for these pair styles.

These pair styles do not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

These pair styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.63.5 Restrictions

The *coul/slater/long* style requires the long-range solvers included in the KSPACE package.

These styles are part of the EXTRA-PAIR package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.63.6 Related commands

*pair_coeff*, *pair_style, hybrid/overlay*, *kspace_style*

### 4.63.7 Default

none

---

**(Melchor)** Gonzalez-Melchor, Mayoral, Velazquez, and Alejandre, J Chem Phys, 125, 224107 (2006).

**(Vaiwala)** Vaiwala, Jadhav, and Thaokar, J Chem Phys, 146, 124904 (2017).

## 4.64 pair_style coul/tt command

### 4.64.1 Syntax

```
pair_style style args
```

- style = *coul/tt*
- args = list of arguments for a particular style

```
coul/tt args = n cutoff
  n = degree of polynomial
  cutoff = global cutoff (distance units)
```

### 4.64.2 Examples

```
pair_style hybrid/overlay ... coul/tt 4 12.0
pair_coeff 1 2  coul/tt 4.5 1.0
pair_coeff 1 2  coul/tt 4.0 1.0 4 12.0
pair_coeff 1 3* coul/tt 4.5 1.0 4
```

Example input scripts available: examples/PACKAGES/drude

### 4.64.3 Description

The *coul/tt* pair style is meant to be used with force fields that include explicit polarization through Drude dipoles.

The *coul/tt* pair style should be used as a sub-style within in the *pair_style hybrid/overlay* command, in conjunction with a main pair style including Coulomb interactions and *thole* pair style, or with *lj/cut/thole/long* pair style that is equivalent to the combination of preceding two.

The *coul/tt* pair styles compute the charge-dipole Coulomb interaction damped at short distances by a function

$$f_{n,ij}(r) = 1 - c_{ij} \cdot e^{-b_{ij}r} \sum_{k=0}^{n} \frac{(b_{ij}r)^k}{k!}$$

This function results from an adaptation to the Coulomb interaction *(Salanne)* of the damping function originally proposed by *Tang Toennies* for van der Waals interactions.

The polynomial takes the degree 4 for damping the Coulomb interaction. The parameters $b_{ij}$ and $c_{ij}$ could be determined from first-principle calculations for small, mainly mono-atomic, ions *(Salanne)*, or else treated as empirical for large molecules.

In pair styles with Drude induced dipoles, this damping function is typically applied to the interactions between a Drude charge (either $q_{D,i}$ on a Drude particle or $-q_{D,i}$ on the respective Drude core)) and a charge on a non-polarizable atom, $q_j$.

The Tang-Toennies function could also be used to damp electrostatic interactions between the (non-polarizable part of the) charge of a core, $q_i - q_{D,i}$, and the Drude charge of another, $-q_{D,j}$. The $b_{ij}$ and $c_{ij}$ are equal to $b_{ji}$ and $c_{ji}$ in the case of core-core interactions.

For pair_style *coul/tt*, the following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the example above.

- $b_{ij}$
- $c_{ij}$
- degree of polynomial (positive integer)
- cutoff (distance units)

The last two coefficients are optional. If not specified the global degree of the polynomial or the global cutoff specified in the pair_style command are used. In order to specify a cutoff (forth argument), the degree of the polynomial (third argument) must also be specified.

---

### 4.64.4 Mixing, shift, table, tail correction, restart, rRESPA info

The *coul/tt* pair style does not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

### 4.64.5 Restrictions

These pair styles are part of the DRUDE package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair_style should currently not be used with the *charmm dihedral style* if the latter has non-zero 1-4 weighting factors. This is because the *coul/tt* pair style does not know which pairs are 1-4 partners of which dihedrals.

### 4.64.6 Related commands

*fix drude*, *fix langevin/drude*, *fix drude/transform*, *compute temp/drude*, *pair_style thole*

### 4.64.7 Default

none

---

**(Thole)** Chem Phys, 59, 341 (1981).

**(Salanne)** Salanne, Rotenberg, Jahn, Vuilleumier, Simon, Christian and Madden, Theor Chem Acc, 131, 1143 (2012).

**(Tang and Toennies)** J Chem Phys, 80, 3726 (1984).

## 4.65 pair_style born/coul/dsf/cs command

Accelerator Variants: *born/coul/dsf/cs/omp*

## 4.66 pair_style born/coul/long/cs command

Accelerator Variants: *born/coul/long/cs/gpu*, *born/coul/long/cs/omp*

## 4.67 pair_style born/coul/wolf/cs command

Accelerator Variants: *born/coul/wolf/cs/gpu*, *born/coul/wolf/cs/omp*

## 4.68 pair_style buck/coul/long/cs command

Accelerator Variants: *buck/coul/long/cs/omp*

## 4.69 pair_style coul/long/cs command

Accelerator Variants: *coul/long/cs/gpu*, *coul/long/cs/omp*

## 4.70 pair_style coul/wolf/cs command

Accelerator Variants: *coul/wolf/cs/omp*

## 4.71 pair_style lj/cut/coul/long/cs command

Accelerator Variants: *lj/cut/coul/long/cs/omp*

# 4.72 pair_style lj/class2/coul/long/cs command

Accelerator Variants: *lj/class2/coul/long/cs/omp*

## 4.72.1 Syntax

```
pair_style style args
```

- style = *born/coul/dsf/cs* or *born/coul/long/cs* or *born/coul/wolf/cs* or *buck/coul/long/cs* or *coul/long/cs* or *coul/wolf/cs* or *lj/cut/coul/long/cs* or *lj/class2/coul/long/cs*
- args = list of arguments for a particular style

*born/coul/dsf/cs* args = alpha cutoff (cutoff2)
  alpha = damping parameter (inverse distance units)
  cutoff = global cutoff for non-Coulombic (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (distance units)
*born/coul/long/cs* args = cutoff (cutoff2)
  cutoff = global cutoff for non-Coulombic (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
*born/coul/wolf/cs* args = alpha cutoff (cutoff2)
  alpha = damping parameter (inverse distance units)
  cutoff = global cutoff for Buckingham (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
*buck/coul/long/cs* args = cutoff (cutoff2)
  cutoff = global cutoff for Buckingham (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
*coul/long* args = cutoff
  cutoff = global cutoff for Coulombic (distance units)
*coul/wolf* args = alpha cutoff
  alpha = damping parameter (inverse distance units)
  cutoff = global cutoff for Coulombic (distance units)
*lj/cut/coul/long/cs* args = cutoff (cutoff2)
  cutoff = global cutoff for LJ (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
*lj/class2/coul/long/cs* args = cutoff (cutoff2)
  cutoff = global cutoff for LJ (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)

## 4.72.2 Examples

```
pair_style born/coul/dsf/cs 0.1 10.0 12.0
pair_coeff * *   0.0 1.00 0.00 0.00 0.00
pair_coeff 1 1 480.0 0.25 0.00 1.05 0.50

pair_style born/coul/long/cs 10.0 8.0
pair_coeff 1 1 6.08 0.317 2.340 24.18 11.51

pair_style born/coul/wolf/cs 0.25 10.0 12.0
pair_coeff * *   0.0 1.00 0.00 0.00 0.00
pair_coeff 1 1 480.0 0.25 0.00 1.05 0.50

pair_style buck/coul/long/cs 10.0
pair_style buck/coul/long/cs 10.0 8.0
```

(continues on next page)

(continued from previous page)

```
pair_coeff * * 100.0 1.5 200.0
pair_coeff 1 1 100.0 1.5 200.0 9.0

pair_style coul/long/cs 10.0
pair_coeff * *

pair_style coul/wolf/cs 0.2 9.0
pair_coeff * *

pair_style lj/cut/coul/long/cs 10.0
pair_style lj/cut/coul/long/cs 10.0 8.0
pair_coeff * * 100.0 3.0
pair_coeff 1 1 100.0 3.5 9.0
```

### 4.72.3 Description

These pair styles are designed to be used with the adiabatic core/shell model of *(Mitchell and Fincham)*. See the *Howto coreshell* page for an overview of the model as implemented in LAMMPS.

All the styles are identical to the corresponding pair style without the "/cs" in the name:

- *pair_style born/coul/dsf*
- *pair_style born/coul/long*
- *pair_style born/coul/wolf*
- *pair_style buck/coul/long*
- *pair_style coul/long*
- *pair_style coul/wolf*
- *pair_style lj/cut/coul/long*
- *pair_style lj/class2/coul/long*

except that they correctly treat the special case where the distance between two charged core and shell atoms in the same core/shell pair approach r = 0.0.

Styles with a "/long" in the name are used with a long-range solver for Coulombic interactions via the *kspace_style* command. They require special treatment of the short-range Coulombic interactions within the cor/shell model.

Specifically, the short-range Coulomb interaction between a core and its shell should be turned off using the *special_bonds* command by setting the 1-2 weight to 0.0, which works because the core and shell atoms are bonded to each other. This induces a long-range correction approximation which fails at small distances (~< 10e-8). Therefore, the Coulomb term which is used to calculate the correction factor is extended by a minimal distance (r_min = 1.0-6) when the interaction between a core/shell pair is treated, as follows

$$E = \frac{Cq_iq_j}{\varepsilon(r + r_{min})} \qquad r \to 0$$

where C is an energy-conversion constant, $q_i$ and $q_j$ are the charges on the core and shell, epsilon is the dielectric constant and $r_{min}$ is the minimal distance.

For styles that are not used with a long-range solver, i.e. those with "/dsf" or "/wolf" in the name, the only correction is the addition of a minimal distance to avoid the possible r = 0.0 case for a core/shell pair.

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.72.4 Mixing, shift, table, tail correction, restart, rRESPA info

See the corresponding doc pages for pair styles without the "cs" suffix to see how mixing, shifting, tabulation, tail correction, restarting, and rRESPA are handled by theses pair styles.

---

### 4.72.5 Restrictions

These pair styles are part of the CORESHELL package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.72.6 Related commands

*pair_coeff* , *pair_style born*, *pair_style buck*

### 4.72.7 Default

none

---

**(Mitchell and Fincham)** Mitchell, Fincham, J Phys Condensed Matter, 5, 1031-1038 (1993).

## 4.73 pair_style coul/cut/dielectric command

## 4.74 pair_style coul/long/dielectric command

## 4.75 pair_style lj/cut/coul/cut/dielectric command

Accelerator Variants: *lj/cut/coul/cut/dielectric/omp*

## 4.76 pair_style lj/cut/coul/debye/dielectric command

Accelerator Variants: *lj/cut/coul/debye/dielectric/omp*

## 4.77 pair_style lj/cut/coul/long/dielectric command

Accelerator Variants: *lj/cut/coul/long/dielectric/omp*

# 4.78 pair_style lj/cut/coul/msm/dielectric command

# 4.79 pair_style lj/long/coul/long/dielectric command

## 4.79.1 Syntax

```
pair_style style args
```

- style = *lj/cut/coul/cut/dielectric* or *lj/cut/coul/long/dielectric* or *lj/cut/coul/msm/dielectric* or *lj/long/coul/msm/dielectric*
- args = list of arguments for a particular style

## 4.79.2 Examples

```
pair_style coul/cut/dielectric 10.0
pair_coeff * *
pair_coeff 1 1 9.0

pair_style lj/cut/coul/cut/dielectric 10.0
pair_style lj/cut/coul/cut/dielectric 10.0 8.0
pair_coeff * * 100.0 3.0
pair_coeff 1 1 100.0 3.5 9.0

pair_style lj/cut/coul/long/dielectric 10.0
pair_style lj/cut/coul/long/dielectric 10.0 8.0
pair_coeff * * 100.0 3.0
pair_coeff 1 1 100.0 3.5 9.0
```

Used in input scripts:

```
examples/PACKAGES/dielectric/in.confined
examples/PACKAGES/dielectric/in.nopbc
```

## 4.79.3 Description

All these pair styles are derived from the corresponding pair styles without the *dielectric* suffix. In addition to computing atom forces and energies, these pair styles compute the electric field vector at each atom, which are intended to be used by the *fix polarize* commands to compute induced charges at interfaces between two regions of different dielectric constant.

These pair styles should be used with *atom_style dielectric*.

The styles lj/cut/coul/long/dielectric, lj/cut/coul/msm/dielectric, and lj/long/coul/long/dielectric should be used with their kspace style counterparts, namely, pppm/dielectric, pppm/disp/dielectric, and msm/dielectric, respectively.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.79.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon and sigma coefficients and cutoff distances for this pair style can be mixed. The default mix algorithm is *geometric*. See the *pair_modify*" command for details.

The *pair_modify* table option is not relevant for this pair style.

These pair styles write its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

These pair styles can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.79.5 Restrictions

These styles are part of the DIELECTRIC package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.79.6 Related commands

*pair_coeff* , *fix polarize*, *read_data*

### 4.79.7 Default

none

## 4.80 pair_style lj/cut/dipole/cut command

Accelerator Variants: *lj/cut/dipole/cut/gpu*, *lj/cut/dipole/cut/kk*, *lj/cut/dipole/cut/omp*

## 4.81 pair_style lj/sf/dipole/sf command

Accelerator Variants: *lj/sf/dipole/sf/gpu*, *lj/sf/dipole/sf/omp*

## 4.82 pair_style lj/cut/dipole/long command

Accelerator Variants: *lj/cut/dipole/long/gpu*, *lj/cut/dipole/long/omp*

## 4.83 pair_style lj/long/dipole/long command

### 4.83.1 Syntax

```
pair_style lj/cut/dipole/cut cutoff (cutoff2)
pair_style lj/sf/dipole/sf cutoff (cutoff2)
pair_style lj/cut/dipole/long cutoff (cutoff2)
pair_style lj/long/dipole/long flag_lj flag_coul cutoff (cutoff2)
```

- cutoff = global cutoff LJ (and Coulombic if only 1 arg) (distance units)
- cutoff2 = global cutoff for Coulombic and dipole (optional) (distance units)
- flag_lj = *long* or *cut* or *off*

  *long* = use long-range damping on dispersion 1/r^6 term
  *cut* = use a cutoff on dispersion 1/r^6 term
  *off* = omit disperion 1/r^6 term entirely

- flag_coul = *long* or *off*

  *long* = use long-range damping on Coulombic 1/r and point-dipole terms
  *off* = omit Coulombic and point-dipole terms entirely

### 4.83.2 Examples

```
pair_style lj/cut/dipole/cut 2.5 5.0
pair_coeff * * 1.0 1.0
pair_coeff 2 3 0.8 1.0 2.5 4.0

pair_style lj/sf/dipole/sf 9.0
pair_coeff * * 1.0 1.0
pair_coeff 2 3 1.0 1.0 2.5 4.0 scale 0.5
pair_coeff 2 3 0.8 1.0 2.5 4.0

pair_style lj/cut/dipole/long 2.5 3.5
pair_coeff * * 1.0 1.0
pair_coeff 2 3 0.8 1.0 3.0

pair_style lj/long/dipole/long long long 3.5
pair_coeff * * 1.0 1.0
pair_coeff 2 3 0.8 1.0

pair_style lj/long/dipole/long cut long 2.5 3.5
pair_coeff * * 1.0 1.0
pair_coeff 2 3 0.8 1.0 3.0
```

### 4.83.3 Description

Style *lj/cut/dipole/cut* computes interactions between pairs of particles that each have a charge and/or a point dipole moment. In addition to the usual Lennard-Jones interaction between the particles (Elj) the charge-charge (Eqq), charge-dipole (Eqp), and dipole-dipole (Epp) interactions are computed by these formulas for the energy (E), force (F), and

torque (T) between particles I and J.

$$
\begin{aligned}
E_{LJ} =& 4\varepsilon \left[ \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \\
E_{qq} =& \frac{q_i q_j}{r} \\
E_{qp} =& \frac{q}{r^3}(p \bullet \vec{r}) \\
E_{pp} =& \frac{1}{r^3}(\vec{p}_i \bullet \vec{p}_j) - \frac{3}{r^5}(\vec{p}_i \bullet \vec{r})(\vec{p}_j \bullet \vec{r})
\end{aligned}
$$

$$
\begin{aligned}
F_{qq} =& \frac{q_i q_j}{r^3}\vec{r} \\
F_{qp} =& -\frac{q}{r^3}\vec{p} + \frac{3q}{r^5}(\vec{p} \bullet \vec{r})\vec{r} \\
F_{pp} =& \frac{3}{r^5}(\vec{p}_i \bullet \vec{p}_j)\vec{r} - \frac{15}{r^7}(\vec{p}_i \bullet \vec{r})(\vec{p}_j \bullet \vec{r})\vec{r} + \frac{3}{r^5}\left[(\vec{p}_j \bullet \vec{r})\vec{p}_i + (\vec{p}_i \bullet \vec{r})\vec{p}_j\right]
\end{aligned}
$$

$$
\begin{aligned}
T_{pq} = T_{ij} =& \frac{q_j}{r^3}(\vec{p}_i \times \vec{r}) \\
T_{qp} = T_{ji} =& -\frac{q_i}{r^3}(\vec{p}_j \times \vec{r}) \\
T_{pp} = T_{ij} =& -\frac{1}{r^3}(\vec{p}_i \times \vec{p}_j) + \frac{3}{r^5}(\vec{p}_j \bullet \vec{r})(\vec{p}_i \times \vec{r}) \\
T_{pp} = T_{ji} =& -\frac{1}{r^3}(\vec{p}_j \times \vec{p}_i) + \frac{3}{r^5}(\vec{p}_i \bullet \vec{r})(\vec{p}_j \times \vec{r})
\end{aligned}
$$

where $q_i$ and $q_j$ are the charges on the two particles, $\vec{p}_i$ and $\vec{p}_j$ are the dipole moment vectors of the two particles, r is their separation distance, and the vector r = Ri - Rj is the separation vector between the two particles. Note that Eqq and Fqq are simply Coulombic energy and force, Fij = -Fji as symmetric forces, and Tij != -Tji since the torques do not act symmetrically. These formulas are discussed in *(AllenTildesley)* and in *(Toukmaji)*.

Also note, that in the code, all of these terms (except Elj) have a $C/\varepsilon$ prefactor, the same as the Coulombic term in the LJ + Coulombic pair styles discussed *here*. C is an energy-conversion constant and $\varepsilon$ is the dielectric constant which can be set by the *dielectric* command. The same is true of the equations that follow for other dipole pair styles.

Style *lj/sf/dipole/sf* computes "shifted-force" interactions between pairs of particles that each have a charge and/or a point dipole moment. In general, a shifted-force potential is a (slightly) modified potential containing extra terms that make both the energy and its derivative go to zero at the cutoff distance; this removes (cutoff-related) problems in energy conservation and any numerical instability in the equations of motion *(AllenTildesley)*. Shifted-force interactions for the Lennard-Jones (E_LJ), charge-charge (Eqq), charge-dipole (Eqp), dipole-charge (Epq) and dipole-dipole (Epp)

potentials are computed by these formulas for the energy (E), force (F), and torque (T) between particles I and J:

$$E_{LJ} = 4\varepsilon \left\{ \left[ \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] + \left[ 6\left(\frac{\sigma}{r_c}\right)^{12} - 3\left(\frac{\sigma}{r_c}\right)^{6} \right] \left(\frac{r}{r_c}\right)^{2} - 7\left(\frac{\sigma}{r_c}\right)^{12} + 4\left(\frac{\sigma}{r_c}\right)^{6} \right\}$$

$$E_{qq} = \frac{q_i q_j}{r} \left(1 - \frac{r}{r_c}\right)^{2}$$

$$E_{pq} = E_{ji} = -\frac{q}{r^3} \left[ 1 - 3\left(\frac{r}{r_c}\right)^{2} + 2\left(\frac{r}{r_c}\right)^{3} \right] (\vec{p} \bullet \vec{r})$$

$$E_{qp} = E_{ij} = \frac{q}{r^3} \left[ 1 - 3\left(\frac{r}{r_c}\right)^{2} + 2\left(\frac{r}{r_c}\right)^{3} \right] (\vec{p} \bullet \vec{r})$$

$$E_{pp} = \left[ 1 - 4\left(\frac{r}{r_c}\right)^{3} + 3\left(\frac{r}{r_c}\right)^{4} \right] \left[ \frac{1}{r^3}(\vec{p}_i \bullet \vec{p}_j) - \frac{3}{r^5}(\vec{p}_i \bullet \vec{r})(\vec{p}_j \bullet \vec{r}) \right]$$

$$F_{LJ} = \left\{ \left[ 48\varepsilon \left(\frac{\sigma}{r}\right)^{12} - 24\varepsilon \left(\frac{\sigma}{r}\right)^{6} \right] \frac{1}{r^2} - \left[ 48\varepsilon \left(\frac{\sigma}{r_c}\right)^{12} - 24\varepsilon \left(\frac{\sigma}{r_c}\right)^{6} \right] \frac{1}{r_c^2} \right\} \vec{r}$$

$$F_{qq} = \frac{q_i q_j}{r} \left(\frac{1}{r^2} - \frac{1}{r_c^2}\right) \vec{r}$$

$$F_{pq} = F_{ij} = -\frac{3q}{r^5} \left[ 1 - \left(\frac{r}{r_c}\right)^{2} \right] (\vec{p} \bullet \vec{r})\vec{r} + \frac{q}{r^3} \left[ 1 - 3\left(\frac{r}{r_c}\right)^{2} + 2\left(\frac{r}{r_c}\right)^{3} \right] \vec{p}$$

$$F_{qp} = F_{ij} = \frac{3q}{r^5} \left[ 1 - \left(\frac{r}{r_c}\right)^{2} \right] (\vec{p} \bullet \vec{r})\vec{r} - \frac{q}{r^3} \left[ 1 - 3\left(\frac{r}{r_c}\right)^{2} + 2\left(\frac{r}{r_c}\right)^{3} \right] \vec{p}$$

$$F_{pp} = \frac{3}{r^5} \left\{ \left[ 1 - \left(\frac{r}{r_c}\right)^{4} \right] \left[ (\vec{p}_i \bullet \vec{p}_j) - \frac{3}{r^2}(\vec{p}_i \bullet \vec{r})(\vec{p}_j \bullet \vec{r}) \right] \vec{r} + \left[ 1 - 4\left(\frac{r}{r_c}\right)^{3} + 3\left(\frac{r}{r_c}\right)^{4} \right] \left[ (\vec{p}_j \bullet \vec{r})\vec{p}_i + (\vec{p}_i \bullet \vec{r})\vec{p}_j - \frac{2}{r^2}(\vec{p}_i \bullet \vec{r})(\vec{p}_j \bullet \vec{r})\vec{r} \right] \right\}$$

$$T_{pq} = T_{ij} = \frac{q_j}{r^3} \left[ 1 - 3\left(\frac{r}{r_c}\right)^{2} + 2\left(\frac{r}{r_c}\right)^{3} \right] (\vec{p}_i \times \vec{r})$$

$$T_{qp} = T_{ji} = -\frac{q_i}{r^3} \left[ 1 - 3\left(\frac{r}{r_c}\right)^{2} + 2\left(\frac{r}{r_c}\right)^{3} \right] (\vec{p}_j \times \vec{r})$$

$$T_{pp} = T_{ij} = -\frac{1}{r^3} \left[ 1 - 4\left(\frac{r}{r_c}\right)^{3} + e3\left(\frac{r}{r_c}\right)^{4} \right] (\vec{p}_i \times \vec{p}_j) + \frac{3}{r^5} \left[ 1 - 4\left(\frac{r}{r_c}\right)^{3} + 3\left(\frac{r}{r_c}\right)^{4} \right] (\vec{p}_j \bullet \vec{r})(\vec{p}_i \times \vec{r})$$

$$T_{pp} = T_{ji} = -\frac{1}{r^3} \left[ 1 - 4\left(\frac{r}{r_c}\right)^{3} + 3\left(\frac{r}{r_c}\right)^{4} \right] (\vec{p}_j \times \vec{p}_i) + \frac{3}{r^5} \left[ 1 - 4\left(\frac{r}{r_c}\right)^{3} + 3\left(\frac{r}{r_c}\right)^{4} \right] (\vec{p}_i \bullet \vec{r})(\vec{p}_j \times \vec{r})$$

where $\varepsilon$ and $\sigma$ are the standard LJ parameters, $r_c$ is the cutoff, $q_i$ and $q_j$ are the charges on the two particles, $\vec{p}_i$ and $\vec{p}_j$ are the dipole moment vectors of the two particles, r is their separation distance, and the vector r = Ri - Rj is the

separation vector between the two particles. Note that Eqq and Fqq are simply Coulombic energy and force, Fij = -Fji as symmetric forces, and Tij != -Tji since the torques do not act symmetrically. The shifted-force formula for the Lennard-Jones potential is reported in *(Stoddard)*. The original (non-shifted) formulas for the electrostatic potentials, forces and torques can be found in *(Price)*. The shifted-force electrostatic potentials have been obtained by applying equation 5.13 of *(AllenTildesley)*. The formulas for the corresponding forces and torques have been obtained by applying the 'chain rule' as in appendix C.3 of *(AllenTildesley)*.

If one cutoff is specified in the pair_style command, it is used for both the LJ and Coulombic (q,p) terms. If two cutoffs are specified, they are used as cutoffs for the LJ and Coulombic (q,p) terms respectively. This pair style also supports an optional *scale* keyword as part of a pair_coeff statement, where the interactions can be scaled according to this factor. This scale factor is also made available for use with fix adapt.

Style *lj/cut/dipole/long* computes the short-range portion of point-dipole interactions as discussed in *(Toukmaji)*. Dipole-dipole, dipole-charge, and charge-charge interactions are all supported, along with the standard 12/6 Lennard-Jones interactions, which are computed with a cutoff. A *kspace_style* must be defined to use this pair style. If only dipoles (not point charges) are included in the model, the kspace style can be one of these 3 options, all of which compute the long-range portion of dipole-dipole interactions. If the model includes point charges (in addition to dipoles), then only the first of these kspace styles can be used:

- *kspace_style ewald/disp*
- *kspace_style ewald/dipole*
- *kspace_style pppm/dipole*

Style *lj/long/dipole/long* has the same functionality as style *lj/cut/dipole/long*, except it also has an option to compute 12/6 Lennard-Jones interactions for use with a long-range dispersion kspace style. This is done by setting its *flag_lj* argument to *long*. For long-range LJ interactions, the *kspace_style ewald/disp* command must be used.

---

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- $\varepsilon$ (energy units)
- $\sigma$ (distance units)
- cutoff1 (distance units)
- cutoff2 (distance units)

The latter 2 coefficients are optional. If not specified, the global LJ and Coulombic cutoffs specified in the pair_style command are used. If only one cutoff is specified, it is used as the cutoff for both LJ and Coulombic interactions for this type pair. If both coefficients are specified, they are used as the LJ and Coulombic cutoffs for this type pair. When using a long-rang Coulomb solver, only a global Coulomb cutoff may be used and only the LJ cutoff may be changed with the *pair_coeff* command. When using the *lj/long/dipole/long* pair style with *long long* setting, only a single global cutoff may be provided and no cutoff for the *pair_coeff* command.

---

Note that for systems using these pair styles, typically particles should be able to exert torque on each other via their dipole moments so that the particle and its dipole moment can rotate. This requires they not be point particles, but finite-size spheres. Thus you should use a command like *atom_style hybrid sphere dipole* to use particles with both attributes.

The magnitude and orientation of the dipole moment for each particle can be defined by the *set* command or in the "Atoms" section of the data file read in by the *read_data* command.

Rotating finite-size particles have 6 degrees of freedom (DOFs), translation and rotational. You can use the *compute temp/sphere* command to monitor a temperature which includes all these DOFs.

Finite-size particles with dipole moments should be integrated using one of these options:

- *fix nve/sphere update dipole*
- *fix nve/sphere update dipole* plus *fix langevin omega yes*
- *fix nvt/sphere update dipole*
- *fix npt/sphere update dipole*

In all cases the "update dipole" setting ensures the dipole moments are also rotated when the finite-size spheres rotate. The 2nd and 3rd bullets perform thermostatting; in the case of a Langevin thermostat the "omega yes" option also thermostats the rotational degrees of freedom (if desired). The 4th bullet performs thermostatting and barostatting.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.83.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon and sigma coefficients and cutoff distances for this pair style can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

For atom type pairs I,J and I != J, the A, sigma, d1, and d2 coefficients and cutoff distance for this pair style can be mixed. A is an energy value mixed like a LJ epsilon. D1 and d2 are distance values and are mixed like sigma. The default mix value is *geometric*. See the "pair_modify" command for details.

This pair style does not support the *pair_modify* shift option for the energy of the Lennard-Jones portion of the pair interaction; such energy goes to zero at the cutoff by construction.

The *pair_modify* table option is not relevant for this pair style.

This pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.83.5 Restrictions

The *lj/cut/dipole/cut*, *lj/cut/dipole/long*, *lj/long/dipole/long*, and lj/sf/dipole/sf* styles are part of the DIPOLE package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Using dipole pair styles with *electron units* is not currently supported.

### 4.83.6 Related commands

*pair_coeff*, *set*, *read_data*, *fix nve/sphere*, *fix nvt/sphere*

### 4.83.7 Default

none

---

**(AllenTildesley)** Allen and Tildesley, Computer Simulation of Liquids, Clarendon Press, Oxford, 1987.

**(Toukmaji)** Toukmaji, Sagui, Board, and Darden, J Chem Phys, 113, 10913 (2000).

**(Stoddard)** Stoddard and Ford, Phys Rev A, 8, 1504 (1973).

**(Price)** Price, Stone and Alderton, Mol Phys, 52, 987 (1984).

## 4.84 pair_style dispersion/d3 command

### 4.84.1 Syntax

```
pair_style dispersion/d3 damping functional cutoff cn_cutoff
```

- damping = damping function: *original*, *zerom*, *bj*, or *bjm*
- functional = XC functional form: *pbe*, *pbe0*, ... (see list below)
- cutoff = global cutoff (distance units)
- cn_cutoff = coordination number cutoff (distance units)

### 4.84.2 Examples

```
pair_style dispersion/d3 original pbe 30.0 20.0
pair_coeff * * C
```

### 4.84.3 Description

Added in version 4Feb2025.

Style *dispersion/d3* computes the dispersion energy-correction used in the DFT-D3 method of Grimme *(Grimme1)*. It would typically be used with a machine learning (ML) potential that was trained with results from plain DFT calculations without the dispersion correction through pair_style hybrid/overlay. ML potentials are often combined *a posteriori* with dispersion energy-correction schemes (see *e.g.* *(Qamar)* and *(Batatia)*).

The energy contribution $E_i$ for an atom $i$ is given by:

$$E_i = \frac{1}{2} \sum_{j \neq i} \left( s_6 \frac{C_{6,ij}}{r_{ij}^6} f_6^{damp}(r_{ij}) + s_8 \frac{C_{8,ij}}{r_{ij}^8} f_8^{damp}(r_{ij}) \right)$$

where $C_n$ is the averaged, geometry-dependent nth-order dispersion coefficient for atom pair $ij$, $r_{ij}$ their inter-nuclear distance, $s_n$ are XC functional-dependent scaling factor, and $f_n^{damp}$ are damping functions.

**Note**

It is currently *not* possible to calculate three-body dispersion contributions, according to, for example, the Axilrod-Teller-Muto model.

Changed in version 2Apr2025: renamed *zero* keyword to *original* to avoid conflicts with *pair style zero* when used as *hybrid sub-style*.

Available damping functions are the original "zero-damping" (*original*) *(Grimme1)*, Becke-Johnson damping (*bj*) *(Grimme2)*, and their revised forms (*zerom* and *bjm*, respectively) *(Sherrill)*.

Available XC functional scaling factors are listed in the table below, and depend on the selected damping function.

| Damping function | XC functional |
|---|---|
| original | slater-dirac-exchange, b-lyp, b-p, b97-d, revpbe, pbe, pbesol, rpw86-pbe, rpbe, tpss, b3-lyp, pbe0, hse06, revpbe38, pw6b95, tpss0, b2-plyp, pwpb95, b2gp-plyp, ptpss, hf, mpwlyp, bpbe, bh-lyp, tpssh, pwb6k, b1b95, bop, o-lyp, o-pbe, ssb, revssb, otpss, b3pw91, revpbe0, pbe38, mpw1b95, mpwb1k, bmk, cam-b3lyp, lc-wpbe, m05, m052x, m06l, m06, m062x, m06hf, hcth120 |
| zerom | b2-plyp, b3-lyp, b97-d, b-lyp, b-p, pbe, pbe0, lc-wpbe |
| bj | b-p, b-lyp, revpbe, rpbe, b97-d, pbe, rpw86-pbe, b3-lyp, tpss, hf, tpss0, pbe0, hse06, revpbe38, pw6b95, b2-plyp, dsd-blyp, dsd-blyp-fc, bop, mpwlyp, o-lyp, pbesol, bpbe, opbe, ssb, revssb, otpss, b3pw91, bh-lyp, revpbe0, tpssh, mpw1b95, pwb6k, b1b95, bmk, cam-b3lyp, lc-wpbe, b2gp-plyp, ptpss, pwpb95, hf/mixed, hf/sv, hf/minis, b3lyp/6-31gd, hcth120, pw1pw, pwgga, hsesol, hf3c, hf3cv, pbeh3c, pbeh-3c, mn15, r2scan |
| bjm | b2-plyp, b3-lyp, b97-d, b-lyp, b-p, pbe, pbe0, lc-wpbe |

This style is primarily supposed to be used combined with a machine-learned interatomic potential trained on a DFT dataset (the selected XC functional should be chosen accordingly) via the *pair_style hybrid* command.

### 4.84.4 Coefficients

All the required coefficients are already stored internally (in the `src/EXTRA-PAIR/d3_parameters.h` file). The only information to provide are the chemical symbols of the atoms. The number of chemical symbols given must be equal to the number of atom types used and must match their ordering as atom types.

### 4.84.5 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support mixing since all parameters are explicit for each pair of atom types.

This pair style does not support the *pair_modify command* shift, table, and tail options.

This pair style does not write its information to *binary restart files*.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.84.6 Restrictions

Style *dispersion/d3* is part of the EXTRA-PAIR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The compiled in parameters require the use of *metal units*.

It is currently *not* possible to calculate three-body dispersion contributions according to, for example, the Axilrod-Teller-Muto model.

### 4.84.7 Related commands

*pair_coeff*

### 4.84.8 Default

none

---

**(Grimme1)** S. Grimme, J. Antony, S. Ehrlich, and H. Krieg, J. Chem. Phys. 132, 154104 (2010).

**(Qamar)** M. Qamar, M. Mrovec, T. Lysogorskiy, A. Bochkarev, and R. Drautz, J. Chem. Theory Comput. 19, 5151 (2023).

**(Batatia)** I. Batatia, *et al.*, arXiv:2401.0096 (2023).

**(Grimme2)** S. Grimme, S. Ehrlich and L. Goerigk, J. Comput. Chem. 32, 1456 (2011).

**(Sherrill)** D. G. A. Smith, L. A. Burns, K. Patkowski, and C. D. Sherrill, J. Phys. Chem. Lett., 7, 2197, (2016).

## 4.85 pair_style dpd command

Accelerator Variants: *dpd/gpu*, *dpd/intel*, *dpd/kk*, *dpd/omp*

## 4.86 pair_style dpd/tstat command

Accelerator Variants: *dpd/tstat/gpu*, *dpd/tstat/kk*, *dpd/tstat/omp*

### 4.86.1 Syntax

```
pair_style dpd T cutoff seed
pair_style dpd/tstat Tstart Tstop cutoff seed
```

- T = temperature (temperature units) (dpd only)
- Tstart,Tstop = desired temperature at start/end of run (temperature units) (dpd/tstat only)
- cutoff = global cutoff for DPD interactions (distance units)
- seed = random # seed (positive integer)

### 4.86.2 Examples

```
pair_style dpd 1.0 2.5 34387
pair_coeff * * 3.0 1.0
pair_coeff 1 1 3.0 1.0 1.0
```

(continues on next page)

(continued from previous page)

```
pair_style hybrid/overlay lj/cut 2.5 dpd/tstat 1.0 1.0 2.5 34387
pair_coeff * * lj/cut 1.0 1.0
pair_coeff * * dpd/tstat 1.0
```

### 4.86.3 Description

Style *dpd* computes a force field for dissipative particle dynamics (DPD) following the exposition in *(Groot)*.

Style *dpd/tstat* invokes a DPD thermostat on pairwise interactions, which is equivalent to the non-conservative portion of the DPD force field. This pairwise thermostat can be used in conjunction with any *pair style*, and instead of per-particle thermostats like *fix langevin* or ensemble thermostats like Nose Hoover as implemented by *fix nvt*. To use *dpd/tstat* as a thermostat for another pair style, use the *pair_style hybrid/overlay* command to compute both the desired pair interaction and the thermostat for each pair of particles.

For style *dpd*, the force on atom I due to atom J is given as a sum of 3 terms

$$
\begin{aligned}
\vec{f} &= (F^C + F^D + F^R)\hat{r_{ij}} \qquad\qquad r < r_c \\
F^C &= A w(r) \\
F^D &= -\gamma w^2(r)(\hat{r_{ij}} \bullet \vec{v}_{ij}) \\
F^R &= \sigma w(r) \alpha (\Delta t)^{-1/2} \\
w(r) &= 1 - \frac{r}{r_c}
\end{aligned}
$$

where $F^C$ is a conservative force, $F^D$ is a dissipative force, and $F^R$ is a random force. $\hat{r_{ij}}$ is a unit vector in the direction $r_i - r_j$, $\vec{v}_{ij}$ is the vector difference in velocities of the two atoms $\vec{v}_i - \vec{v}_j$, $\alpha$ is a Gaussian random number with zero mean and unit variance, $dt$ is the timestep size, and $w(r)$ is a weighting factor that varies between 0 and 1. $r_c$ is the pairwise cutoff. $\sigma$ is set equal to $\sqrt{2 k_B T \gamma}$, where $k_B$ is the Boltzmann constant and $T$ is the temperature parameter in the pair_style command.

For style *dpd/tstat*, the force on atom I due to atom J is the same as the above equation, except that the conservative $F^C$ term is dropped. Also, during the run, $T$ is set each timestep to a ramped value from *Tstart* to *Tstop*.

For style *dpd*, the pairwise energy associated with style *dpd* is only due to the conservative force term $F^C$, and is shifted to be zero at the cutoff distance $r_c$. The pairwise virial is calculated using all 3 terms. For style *dpd/tstat* there is no pairwise energy, but the last two terms of the formula make a contribution to the virial.

For style *dpd*, the following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- A (force units)
- $\gamma$ (force/velocity units)
- cutoff (distance units)

The cutoff coefficient is optional. If not specified, the global DPD cutoff is used. Note that sigma is set equal to sqrt(2 T gamma), where T is the temperature set by the *pair_style* command so it does not need to be specified.

For style *dpd/tstat*, the coefficients defined for each pair of atoms types via the *pair_coeff* command are:

- $\gamma$ (force/velocity units)
- cutoff (distance units)

The cutoff coefficient is optional.

Styles with a *gpu* suffix are implemented based on the work of *(Afshar)* and *(Phillips)*.

**Note**

If you are modeling DPD polymer chains, you may want to use the *pair_style srp* command in conjunction with these pair styles. It is a soft segmental repulsive potential (SRP) that can prevent DPD polymer chains from crossing each other.

**Note**

The virial calculation for pressure when using these pair styles includes all the components of force listed above, including the random force. Since the random force depends on random numbers, everything that changes the order of atoms in the neighbor list (e.g. different number of MPI ranks or a different neighbor list skin distance) will also change the sequence in which the random numbers are applied and thus the individual forces and therefore also the virial/pressure.

**Note**

For more consistent time integration and force computation you may consider using *fix mvv/dpd* instead of *fix nve*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.86.4 Mixing, shift, table, tail correction, restart, rRESPA info

These pair styles do not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

These pair styles do not support the *pair_modify* shift option for the energy of the pair interaction. Note that as discussed above, the energy due to the conservative $F^C$ term is already shifted to be 0.0 at the cutoff distance $r_c$.

The *pair_modify* table option is not relevant for these pair styles.

These pair styles do not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

These pair styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file. Note that the user-specified random number seed is stored in the restart file, so when a simulation is restarted, each processor will re-initialize its random number generator the same way it did initially. This means the random forces will be random, but will not be the same as they would have been if the original simulation had continued past the restart time.

These pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

The *dpd/tstat* style can ramp its target temperature over multiple runs, using the *start* and *stop* keywords of the *run* command. See the *run* command for details of how to do this.

---

### 4.86.5 Restrictions

These styles are part of the DPD-BASIC package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The default frequency for rebuilding neighbor lists is every 10 steps (see the *neigh_modify* command). This may be too infrequent for style *dpd* simulations since particles move rapidly and can overlap by large amounts. If this setting yields a non-zero number of "dangerous" reneighborings (printed at the end of a simulation), you should experiment with forcing reneighboring more often and see if system energies/trajectories change.

These pair styles requires you to use the *comm_modify vel yes* command so that velocities are stored by ghost atoms.

These pair styles will not restart exactly when using the *read_restart* command, though they should provide statistically similar results. This is because the forces they compute depend on atom velocities. See the *read_restart* command for more details.

### 4.86.6 Related commands

*pair_style dpd/ext*, *pair_coeff*, *fix nvt*, *fix langevin*, *pair_style srp*, *fix mvv/dpd*.

### 4.86.7 Default

none

---

**(Groot)** Groot and Warren, J Chem Phys, 107, 4423-35 (1997).

**(Afshar)** Afshar, F. Schmid, A. Pishevar, S. Worley, Comput Phys Comm, 184, 1119-1128 (2013).

**(Phillips)** C. L. Phillips, J. A. Anderson, S. C. Glotzer, Comput Phys Comm, 230, 7191-7201 (2011).

## 4.87 pair_style dpd/coul/slater/long command

Accelerator Variants: *dpd/coul/slater/long/gpu*

### 4.87.1 Syntax

```
pair_style dpd/coul/slater/long T cutoff_DPD seed lambda cutoff_coul
```

- T = temperature (temperature units)
- cutoff_DPD = global cutoff for DPD interactions (distance units)
- seed = random # seed (positive integer)
- lambda = decay length of the charge (distance units)
- cutoff_coul = global cutoff for Coulombic interactions (distance units)

### 4.87.2 Examples

```
pair_style dpd/coul/slater/long 1.0 2.5 34387 0.25 3.0

pair_coeff 1 1 78.0 4.5          # not charged by default
pair_coeff 2 2 78.0 4.5 yes
```

### 4.87.3 Description

Added in version 27June2024.

Style *dpd/coul/slater/long* computes a force field for dissipative particle dynamics (DPD) following the exposition in *(Groot)*. It also allows for the use of charged particles in the model by adding a long-range Coulombic term to the DPD interactions. The short-range portion of the Coulombics is calculated by this pair style. The long-range Coulombics are computed by use of the *kspace_style* command, e.g. using the Ewald or PPPM styles.

Coulombic forces in mesoscopic models such as DPD employ potentials without explicit excluded-volume interactions. The goal is to prevent artificial ionic pair formation by including a charge distribution in the Coulomb potential, following the formulation in *(Melchor1)*.

**Note**

This pair style is effectively the combination of the *pair_style dpd* and *pair_style coul/slater/long* commands, but should be more efficient (especially on GPUs) than using *pair_style hybrid/overlay dpd coul/slater/long*. That is particularly true for the GPU package version of the pair style since this version is compatible with computing neighbor lists on the GPU instead of the CPU as is required for hybrid styles.

In the charged DPD model, the force on bead I due to bead J is given as a sum of 4 terms:

$$
\begin{aligned}
\vec{f} &= (F^C + F^D + F^R + F^E)\hat{r_{ij}} \\
F^C &= Aw(r) && r < r_{DPD} \\
F^D &= -\gamma w^2(r)(\hat{r_{ij}} \bullet \vec{v}_{ij}) && r < r_{DPD} \\
F^R &= \sigma w(r)\alpha(\Delta t)^{-1/2} && r < r_{DPD} \\
w(r) &= 1 - \frac{r}{r_{DPD}} \\
F^E &= \frac{Cq_iq_j}{\varepsilon r^2}\left(1 - exp\left(\frac{2r_{ij}}{\lambda}\right)\left(1 + \frac{2r_{ij}}{\lambda}\left(1 + \frac{r_{ij}}{\lambda}\right)\right)\right)
\end{aligned}
$$

where $F^C$ is a conservative force, $F^D$ is a dissipative force, $F^R$ is a random force, and $F^E$ is an electrostatic force. $\hat{r_{ij}}$ is a unit vector in the direction $r_i - r_j$, $\vec{v}_{ij}$ is the vector difference in velocities of the two atoms $\vec{v}_i - \vec{v}_j$, $\alpha$ is a Gaussian random number with zero mean and unit variance, $dt$ is the timestep size, and $w(r)$ is a weighting factor that varies between 0 and 1.

$\sigma$ is set equal to $\sqrt{2k_BT\gamma}$, where $k_B$ is the Boltzmann constant and $T$ is the temperature parameter in the pair_style command.

$r_{DPD}$ is the pairwise cutoff for the first 3 DPD terms in the formula as specified by *cutoff_DPD*. For the $F^E$ term, pairwise interactions within the specified *cutoff_coul* distance are computed directly; interactions beyond that distance are computed in reciprocal space. $C$ is the same Coulomb conversion factor used in the Coulombic formulas described on the *pair_coul* doc page.

The following parameters must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- A (force units)
- $\gamma$ (force/velocity units)

- is_charged (optional boolean, default = no)

The *is_charged* parameter is optional and can be specified as *yes* or *no*. *Yes* should be used for interactions between two types of charged particles. *No* is the default and should be used for interactions between two types of particles when one or both are uncharged.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.87.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

This pair style does not support the *pair_modify* shift option for the energy of the pair interaction.

The *pair_modify* table option is not relevant for this pair style.

This pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file. Note that the user-specified random number seed is stored in the restart file, so when a simulation is restarted, each processor will re-initialize its random number generator the same way it did initially. This means the random forces will be random, but will not be the same as they would have been if the original simulation had continued past the restart time.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.87.5 Restrictions

This style is part of the DPD-BASIC package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The default frequency for rebuilding neighbor lists is every 10 steps (see the *neigh_modify* command). This may be too infrequent since particles move rapidly and can overlap by large amounts. If this setting yields a non-zero number of "dangerous" reneighborings (printed at the end of a simulation), you should experiment with forcing reneighboring more often and see if system energies/trajectories change.

This pair style requires use of the *comm_modify vel yes* command so that velocities are stored by ghost atoms.

This pair style also requires use of a long-range solvers from the KSPACE package.

This pair style will not restart exactly when using the *read_restart* command, though they should provide statistically similar results. This is because the forces they compute depend on atom velocities. See the *read_restart* command for more details.

### 4.87.6 Related commands

*pair_style dpd*, *pair_style coul/slater/long*,

### 4.87.7 Default

For the pair_coeff command, the default is is_charged = no.

---

**(Groot)** Groot and Warren, J Chem Phys, 107, 4423-35 (1997).

**(Melchor)** Gonzalez-Melchor, Mayoral, Velazquez, and Alejandre, J Chem Phys, 125, 224107 (2006).

## 4.88 pair_style dpd/ext command

Accelerator Variants: *dpd/ext/kk*, *dpd/ext/omp*

## 4.89 pair_style dpd/ext/tstat command

Accelerator Variants: *dpd/ext/tstat/kk*, *dpd/ext/tstat/omp*

### 4.89.1 Syntax

```
pair_style dpd/ext T cutoff seed
pair_style dpd/ext/tstat Tstart Tstop cutoff seed
```

- T = temperature (temperature units)
- Tstart,Tstop = desired temperature at start/end of run (temperature units)
- cutoff = global cutoff for DPD interactions (distance units)
- seed = random # seed (positive integer)

### 4.89.2 Examples

```
pair_style dpd/ext 1.0 2.5 34387
pair_coeff 1 1 25.0 4.5 4.5 0.5 0.5 1.2
pair_coeff 1 2 40.0 4.5 4.5 0.5 0.5 1.2

pair_style hybrid/overlay lj/cut 2.5 dpd/ext/tstat 1.0 1.0 2.5 34387
pair_coeff * * lj/cut 1.0 1.0
pair_coeff * * 4.5 4.5 0.5 0.5 1.2
```

### 4.89.3 Description

The style *dpd/ext* computes an extended force field for dissipative particle dynamics (DPD) following the exposition in *(Groot)*, *(Junghans)*.

Style *dpd/ext/tstat* invokes an extended DPD thermostat on pairwise interactions, equivalent to the non-conservative portion of the extended DPD force field. To use *dpd/ext/tstat* as a thermostat for another pair style, use the *pair_style hybrid/overlay* command to compute both the desired pair interaction and the thermostat for each pair of particles.

For the style *dpd/ext*, the force on atom I due to atom J is given as a sum of 3 terms

$$
\begin{aligned}
\mathbf{f} &= f^C + f^D + f^R \qquad\qquad r < r_c \\
f^C &= A_{ij} w(r) \hat{\mathbf{r}}_{ij} \\
f^D &= -\gamma_{\parallel} w_{\parallel}^2(r)(\hat{\mathbf{r}}_{ij} \cdot \mathbf{v}_{ij})\hat{\mathbf{r}}_{ij} - \gamma_{\perp} w_{\perp}^2(r)(\mathbf{I} - \hat{\mathbf{r}}_{ij}\hat{\mathbf{r}}_{ij}^{\mathrm{T}})\mathbf{v}_{ij} \\
f^R &= \sigma_{\parallel} w_{\parallel}(r)\frac{\alpha}{\sqrt{\Delta t}}\hat{\mathbf{r}}_{ij} + \sigma_{\perp} w_{\perp}(r)(\mathbf{I} - \hat{\mathbf{r}}_{ij}\hat{\mathbf{r}}_{ij}^{\mathrm{T}})\frac{\xi_{ij}}{\sqrt{\Delta t}} \\
w(r) &= 1 - r/r_c
\end{aligned}
$$

where $\mathbf{f}^C$ is a conservative force, $\mathbf{f}^D$ is a dissipative force, and $\mathbf{f}^R$ is a random force. $A_{ij}$ is the maximum repulsion between the two atoms, $\hat{\mathbf{r}}_{ij}$ is a unit vector in the direction $\mathbf{r}_i - \mathbf{r}_j$, $\mathbf{v}_{ij} = \mathbf{v}_i - \mathbf{v}_j$ is the vector difference in velocities of the two atoms, $\alpha$ and $\xi_{ij}$ are Gaussian random numbers with zero mean and unit variance, $\Delta t$ is the timestep, $w(r) = 1 - r/r_c$ is a weight function for the conservative interactions that varies between 0 and 1, $r_c$ is the corresponding cutoff, $w_\alpha(r) = (1 - r/\bar{r}_c)^{s_\alpha}$, $\alpha \equiv (\parallel, \perp)$, are weight functions with coefficients $s_\alpha$ that vary between 0 and 1, $\bar{r}_c$ is the corresponding cutoff, $\mathbf{I}$ is the unit matrix, $\sigma_\alpha = \sqrt{2k_B T \gamma_\alpha}$, where $k_B$ is the Boltzmann constant and $T$ is the temperature in the pair_style command.

For the style *dpd/ext/tstat*, the force on atom I due to atom J is the same as the above equation, except that the conservative $\mathbf{f}^C$ term is dropped. Also, during the run, T is set each timestep to a ramped value from Tstart to Tstop.

For the style *dpd/ext*, the pairwise energy associated with style *dpd/ext* is only due to the conservative force term $\mathbf{f}^C$, and is shifted to be zero at the cutoff distance $r_c$. The pairwise virial is calculated using all three terms. There is no pairwise energy for style *dpd/ext/tstat*, but the last two terms of the formula contribute the virial.

For the style *dpd/ext/tstat*, the force on atom I due to atom J is the same as the above equation, except that the conservative $\mathbf{f}^C$ term is dropped. Also, during the run, T is set each timestep to a ramped value from Tstart to Tstop.

For the style *dpd/ext*, the pairwise energy associated with style *dpd/ext* is only due to the conservative force term $\mathbf{f}^C$, and is shifted to be zero at the cutoff distance $r_c$. The pairwise virial is calculated using all three terms. There is no pairwise energy for style *dpd/ext/tstat*, but the last two terms of the formula contribute the virial.

For the style *dpd/ext*, the following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above:

- A (force units)
- $\gamma_{\parallel}$ (force/velocity units)
- $\gamma_{\perp}$ (force/velocity units)
- $s_{\parallel}$ (unitless)
- $s_{\perp}$ (unitless)
- $r_c$ (distance units)

The last coefficient is optional. If not specified, the global DPD cutoff is used. Note that $\sigma$'s are set equal to $\sqrt{2k_B T \gamma}$, where $T$ is the temperature set by the *pair_style* command so it does not need to be specified.

For the style *dpd/ext/tstat*, the coefficients defined for each pair of atoms types via the *pair_coeff* command are:

- $\gamma_{\parallel}$ (force/velocity units)
- $\gamma_{\perp}$ (force/velocity units)
- $s_{\parallel}$ (unitless)
- $s_{\perp}$ (unitless)
- $r_c$ (distance units)

The last coefficient is optional.

**Note**

If you are modeling DPD polymer chains, you may want to use the *pair_style srp* command in conjunction with these pair styles. It is a soft segmental repulsive potential (SRP) that can prevent DPD polymer chains from crossing each other.

**Note**

The virial calculation for pressure when using these pair styles includes all the components of force listed above, including the random force. Since the random force depends on random numbers, everything that changes the order of atoms in the neighbor list (e.g. different number of MPI ranks or a different neighbor list skin distance) will also change the sequence in which the random numbers are applied and thus the individual forces and therefore also the virial/pressure.

**Note**

For more consistent time integration and force computation you may consider using *fix mvv/dpd* instead of *fix nve*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

**Mixing, shift, table, tail correction, restart, rRESPA info**:

The style *dpd/ext* does not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

The pair styles do not support the *pair_modify* shift option for the energy of the pair interaction. Note that as discussed above, the energy due to the conservative $\mathbf{f}^C$ term is already shifted to be zero at the cutoff distance $r_c$.

The *pair_modify* table option is not relevant for the style *dpd/ext*.

The style *dpd/ext* does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

The pair styles can only be used via the pair keyword of the *run_style respa* command. They do not support the *inner*, *middle*, and *outer*keywords.

The style *dpd/ext/tstat* can ramp its target temperature over multiple runs, using the start and stop keywords of the *run* command. See the *run* command for details of how to do this.

---

### 4.89.4 Restrictions

These styles are part of the DPD-BASIC package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The default frequency for rebuilding neighbor lists is every 10 steps (see the *neigh_modify* command). This may be too infrequent for style *dpd/ext* simulations since particles move rapidly and can overlap by large amounts. If this setting yields a non-zero number of say{dangerous} reneighborings (printed at the end of a simulation), you should experiment with forcing reneighboring more often and see if system energies/trajectories change.

The pair styles require to use the *comm_modify vel yes* command so that velocities are stored by ghost atoms.

The pair styles will not restart exactly when using the *read_restart* command, though they should provide statistically similar results. This is because the forces they compute depend on atom velocities. See the *read_restart* command for more details.

### 4.89.5 Related commands

*pair_style dpd*, *pair_coeff*, *fix nvt*, *fix langevin*, *pair_style srp*, *fix mvv/dpd*.

**Default:** none

---

**(Groot)** Groot and Warren, J Chem Phys, 107, 4423-35 (1997).

**(Junghans)** Junghans, Praprotnik and Kremer, Soft Matter 4, 156, 1119-1128 (2008).

## 4.90 pair_style dpd/fdt command

## 4.91 pair_style dpd/fdt/energy command

Accelerator Variants: *dpd/fdt/energy/kk*

### 4.91.1 Syntax

```
pair_style style args
```

- style = *dpd/fdt* or *dpd/fdt/energy*
- args = list of arguments for a particular style

```
dpd/fdt args = T cutoff seed
  T = temperature (temperature units)
  cutoff = global cutoff for DPD interactions (distance units)
  seed = random # seed (positive integer)
dpd/fdt/energy args = cutoff seed
  cutoff = global cutoff for DPD interactions (distance units)
  seed = random # seed (positive integer)
```

### 4.91.2 Examples

```
pair_style dpd/fdt 300.0 2.5 34387
pair_coeff * * 3.0 1.0 2.5

pair_style dpd/fdt/energy 2.5 34387
pair_coeff * * 3.0 1.0 0.1 2.5
```

### 4.91.3 Description

Styles *dpd/fdt* and *dpd/fdt/energy* compute the force for dissipative particle dynamics (DPD) simulations. The *dpd/fdt* style is used to perform DPD simulations under isothermal and isobaric conditions, while the *dpd/fdt/energy* style is used to perform DPD simulations under isoenergetic and isoenthalpic conditions (see *(Lisal)*). For DPD simulations in general, the force on atom I due to atom J is given as a sum of 3 terms

$$\begin{aligned} \vec{f} =&(F^C + F^D + F^R)\hat{r_{ij}} \qquad\qquad r < r_c \\ F^C =&Aw(r) \\ F^D =&-\gamma w^2(r)(\hat{r_{ij}} \bullet \vec{v}_{ij}) \\ F^R =&\sigma w(r)\alpha(\Delta t)^{-1/2} \\ w(r) =&1 - \frac{r}{r_c} \end{aligned}$$

where $F^C$ is a conservative force, $F^D$ is a dissipative force, and $F^R$ is a random force. $\hat{r_{ij}}$ is a unit vector in the direction $r_i - r_j$, $\vec{v}_{ij}$ is the vector difference in velocities of the two atoms, $\vec{v}_i - \vec{v}_j$, $\alpha$ is a Gaussian random number with zero mean and unit variance, $dt$ is the timestep size, and $w(r)$ is a weighting factor that varies between 0 and 1, $r_c$ is the pairwise cutoff. Note that alternative definitions of the weighting function exist, but would have to be implemented as a separate pair style command.

For style *dpd/fdt*, the fluctuation-dissipation theorem defines $\gamma$ to be set equal to $\sigma^2/(2T)$, where $T$ is the set point temperature specified as a pair style parameter in the above examples. The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- A (force units)
- $\sigma$ (force*time^(1/2) units)
- cutoff (distance units)

The last coefficient is optional. If not specified, the global DPD cutoff is used.

Style *dpd/fdt/energy* is used to perform DPD simulations under isoenergetic and isoenthalpic conditions. The fluctuation-dissipation theorem defines $\gamma$ to be set equal to $\sigma^2/(2\theta)$, where $\theta$ is the average internal temperature for the pair. The particle internal temperature is related to the particle internal energy through a mesoparticle equation of state (see *fix eos*). The differential internal conductive and mechanical energies are computed within style *dpd/fdt/energy* as:

$$\begin{aligned} du_i^{cond} =&\kappa_{ij}(\frac{1}{\theta_i} - \frac{1}{\theta_j})\omega_{ij}^2 + \alpha_{ij}\omega_{ij}\zeta_{ij}^q(\Delta t)^{-1/2} \\ du_i^{mech} =&-\frac{1}{2}\gamma_{ij}\omega_{ij}^2(\frac{\vec{r}_{ij}}{r_{ij}} \bullet \vec{v}_{ij})^2 - \frac{\sigma_{ij}^2}{4}(\frac{1}{m_i} + \frac{1}{m_j})\omega_{ij}^2 - \frac{1}{2}\sigma_{ij}\omega_{ij}(\frac{\vec{r}_{ij}}{r_{ij}} \bullet \vec{v}_{ij})\zeta_{ij}(\Delta t)^{-1/2} \end{aligned}$$

where

$$\begin{aligned} \alpha_{ij}^2 =&2k_B\kappa_{ij} \\ \sigma_{ij}^2 =&2\gamma_{ij}k_B\Theta_{ij} \\ \Theta_{ij}^{-1} =&\frac{1}{2}(\frac{1}{\theta_i} + \frac{1}{\theta_j}) \end{aligned}$$

$\zeta_i j^q$ is a second Gaussian random number with zero mean and unit variance that is used to compute the internal conductive energy. The fluctuation-dissipation theorem defines $alpha^2$ to be set equal to $2k_B\kappa$, where $\kappa$ is the mesoparticle thermal conductivity parameter. The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- A (force units)

- $\sigma$ (force*time^(1/2) units)
- $\kappa$ (energy*temperature/time units)
- cutoff (distance units)

The last coefficient is optional. If not specified, the global DPD cutoff is used.

The pairwise energy associated with styles *dpd/fdt* and *dpd/fdt/energy* is only due to the conservative force term $F^C$, and is shifted to be zero at the cutoff distance $r_c$. The pairwise virial is calculated using only the conservative term.

The forces computed through the *dpd/fdt* and *dpd/fdt/energy* styles can be integrated with the velocity-Verlet integration scheme or the Shardlow splitting integration scheme described by *(Lisal)*. In the cases when these pair styles are combined with the *fix shardlow*, these pair styles differ from the other dpd styles in that the dissipative and random forces are split from the force calculation and are not computed within the pair style. Thus, only the conservative force is computed by the pair style, while the stochastic integration of the dissipative and random forces are handled through the Shardlow splitting algorithm approach. The Shardlow splitting algorithm is advantageous, especially when performing DPD under isoenergetic conditions, as it allows significantly larger timesteps to be taken.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.91.4 Restrictions

These commands are part of the DPD-REACT package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Pair styles *dpd/fdt* and *dpd/fdt/energy* require use of the *comm_modify vel yes* option so that velocities are stored by ghost atoms.

Pair style *dpd/fdt/energy* requires *atom_style dpd* to be used in order to properly account for the particle internal energies and temperatures.

### 4.91.5 Related commands

*pair_coeff*, *fix shardlow*

### 4.91.6 Default

none

---

**(Lisal)** M. Lisal, J.K. Brennan, J. Bonet Avalos, J. Chem. Phys., 135, 204105 (2011).

# 4.92 pair_style drip command

Accelerator Variants: *drip/omp*

## 4.92.1 Syntax

```
pair_style hybrid/overlay drip [styles ...]
```

- styles = other styles to be overlayed with drip (optional)

## 4.92.2 Examples

```
pair_style hybrid/overlay drip
pair_coeff * * none
pair_coeff * * drip  C.drip  C

pair_style hybrid/overlay drip rebo
pair_coeff * * drip  C.drip     C
pair_coeff * * rebo  CH.airebo  C

pair_style hybrid/overlay drip rebo
pair_coeff * * drip  C.drip     C NULL
pair_coeff * * rebo  CH.airebo  C H
```

## 4.92.3 Description

Style *drip* computes the interlayer interactions of layered materials using the dihedral-angle-corrected registry-dependent (DRIP) potential as described in *(Wen)*, which is based on the *(Kolmogorov)* potential and provides an improved prediction for forces. The total potential energy of a system is

$$E = \frac{1}{2} \sum_i \sum_{j \notin \text{layer } i} \phi_{ij}$$

$$\phi_{ij} = f_c(x_r) \left[ e^{-\lambda(r_{ij}-z_0)} \left[ C + f(\rho_{ij}) + g(\rho_{ij}, \{\alpha_{ij}^{(m)}\}) \right] - A \left( \frac{z_0}{r_{ij}} \right)^6 \right]$$

where the $r^{-6}$ term models the attractive London dispersion, the exponential term is designed to capture the registry effect due to overlapping *pi* bonds, and *fc* is a cutoff function.

This potential (DRIP) only provides the interlayer interactions between graphene layers. So, to perform a realistic simulation, it should be used in combination with an intralayer potential such as *REBO* and *Tersoff*. To keep the intralayer interactions unaffected, we should avoid applying DRIP to contribute energy to intralayer interactions. This can be achieved by assigning different molecular IDs to atoms in different layers, and DRIP is implemented such that only atoms with different molecular ID can interact with each other. For this purpose, *atom style* "molecular" or "full" has to be used.

On the other way around, *REBO* (*Tersoff* or any other potential used to provide the intralayer interactions) should not interfere with the interlayer interactions described by DRIP. This is typically automatically achieved using the commands provided in the *Examples* section above, since the cutoff distance for carbon-carbon interaction in the intralayer potentials (e.g. 2 Angstrom for *REBO*) is much smaller than the equilibrium layer distance of graphene layers (about 3.4 Angstrom). If you want, you can enforce this by assigning different atom types to atoms in different layers, and apply an intralayer potential to one atom type. See *pair_hybrid* for details.

The *pair_coeff* command for DRIP takes *4+N* arguments, where *N* is the number of LAMMPS atom types. The fist three arguments must be fixed to be * * *drip*, the fourth argument is the path to the DRIP parameter file, and the remaining N arguments specifying the mapping between element in the parameter file and atom types. For example, if your LAMMPS simulation has 3 atom types and you want all of them to be C, you would use the following pair_coeff command:

```
pair_coeff * * drip  C.drip  C C C
```

If a mapping value is specified as NULL, the mapping is not performed. This could be useful when DRIP is used to model part of the system where other element exists. Suppose you have a hydrocarbon system, with C of atom type 1 and H of atom type 2, you can use the following command to inform DRIP not to model H atoms:

```
pair_style hybrid/overlay drip rebo
pair_coeff * * drip  C.drip     C NULL
pair_coeff * * rebo  CH.airebo  C H
```

**Note**

The potential parameters developed in *(Wen)* are provided with LAMMPS (see the "potentials" directory). Besides those in *Wen*, an additional parameter "normal_cutoff", specific to the LAMMPS implementation, is used to find the three nearest neighbors of an atom to construct the normal.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.92.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support the pair_modify mix, shift, table, and tail options.

This pair style does not write their information to binary restart files, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

### 4.92.5 Restrictions

This pair style is part of the INTERLAYER package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires the *newton* setting to be "on" for pair interactions.

The *C.drip* parameter file provided with LAMMPS (see the "potentials" directory) is parameterized for metal *units*. You can use the DRIP potential with any LAMMPS units, but you would need to create your own custom parameter file with coefficients listed in the appropriate units, if your simulation does not use "metal" units.

### 4.92.6 Related commands

*pair_style lebedeva_z*, *pair_style kolmogorov/crespi/z*, *pair_style kolmogorov/crespi/full*, *pair_style ilp/graphene/hbn*.

---

**(Wen)** M. Wen, S. Carr, S. Fang, E. Kaxiras, and E. B. Tadmor, Phys. Rev. B, 98, 235404 (2018)

**(Kolmogorov)** A. N. Kolmogorov, V. H. Crespi, Phys. Rev. B 71, 235415 (2005)

## 4.93 pair_style dsmc command

### 4.93.1 Syntax

```
pair_style dsmc max_cell_size seed weighting Tref Nrecompute Nsample
```

- max_cell_size = global maximum cell size for DSMC interactions (distance units)
- seed = random # seed (positive integer)
- weighting = macroparticle weighting
- Tref = reference temperature (temperature units)
- Nrecompute = re-compute v*sigma_max every this many timesteps (timesteps)
- Nsample = sample this many times in recomputing v*sigma_max

### 4.93.2 Examples

```
pair_style dsmc 2.5 34387 10 1.0 100 20
pair_coeff * * 1.0
pair_coeff 1 1 1.0
```

### 4.93.3 Description

Style *dsmc* computes collisions between pairs of particles for a direct simulation Monte Carlo (DSMC) model following the exposition in *(Bird)*. Each collision resets the velocities of the two particles involved. The number of pairwise collisions for each pair or particle types and the length scale within which they occur are determined by the parameters of the pair_style and pair_coeff commands.

Stochastic collisions are performed using the variable hard sphere (VHS) approach, with the user-defined *max_cell_size* value used as the maximum DSMC cell size, and reference cross-sections for collisions given using the pair_coeff command.

There is no pairwise energy or virial contributions associated with this pair style.

The following coefficient must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- sigma (area units, i.e. distance-squared)

The global DSMC *max_cell_size* determines the maximum cell length used in the DSMC calculation. A structured mesh is overlayed on the simulation box such that an integer number of cells are created in each direction for each processor's subdomain. Cell lengths are adjusted up to the user-specified maximum cell size.

---

To perform a DSMC simulation with LAMMPS, several additional options should be set in your input script, though LAMMPS does not check for these settings.

Since this pair style does not compute particle forces, you should use the "fix nve/noforce" time integration fix for the DSMC particles, e.g.

```
fix 1 all nve/noforce
```

This pair style assumes that all particles will communicated to neighboring processors every timestep as they move. This makes it possible to perform all collisions between pairs of particles that are on the same processor. To ensure this occurs, you should use these commands:

```
neighbor 0.0 bin
neigh_modify every 1 delay 0 check no
atom_modify sort 0 0.0
communicate single cutoff 0.0
```

These commands ensure that LAMMPS communicates particles to neighboring processors every timestep and that no ghost atoms are created. The output statistics for a simulation run should indicate there are no ghost particles or neighbors.

In order to get correct DSMC collision statistics, users should specify a Gaussian velocity distribution when populating the simulation domain. Note that the default velocity distribution is uniform, which will not give good DSMC collision rates. Specify "dist gaussian" when using the *velocity* command as in the following:

```
velocity all create 594.6 87287 loop geom dist gaussian
```

---

### 4.93.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

This pair style does not support the *pair_modify* shift option for the energy of the pair interaction.

The *pair_modify* table option is not relevant for this pair style.

This pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file. Note that the user-specified random number seed is stored in the restart file, so when a simulation is restarted, each processor will re-initialize its random number generator the same way it did initially. This means the random forces will be random, but will not be the same as they would have been if the original simulation had continued past the restart time.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.93.5 Restrictions

This pair style is part of the MC package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires an *atom style* with per atom type masses.

### 4.93.6 Related commands

*pair_coeff*, *fix nve/noforce*, *neigh_modify*, *neighbor*, *comm_modify*

### 4.93.7 Default

none

---

**(Bird)** G. A. Bird, "Molecular Gas Dynamics and the Direct Simulation of Gas Flows" (1994).

## 4.94 pair_style e3b command

Accelerator Variants: *e3b/omp*

### 4.94.1 Syntax

```
pair_style e3b Otype
```

- Otype = atom type (numeric or type label) for oxygen

```
pair_coeff * * keyword
```

- one or more keyword/value pairs must be appended.
- keyword = *preset* or *Ea* or *Eb* or *Ec* or *E2* or *K3* or *K2* or *Rs* or *Rc3* or *Rc2* or *bondL* or *neigh*
- If the *preset* keyword is given, no others are needed. Otherwise, all are mandatory except for *neigh*. The *neigh* keyword is always optional.

```
preset arg = 2011 or 2015 = which set of predefined parameters to use
         2011 = use the potential parameters from (Tainter 2011)
         2015 = use the potential parameters from (Tainter 2015)
Ea arg = three-body energy for type A hydrogen bonding interactions (energy units)
Eb arg = three-body energy for type B hydrogen bonding interactions (energy units)
Ec arg = three-body energy for type C hydrogen bonding interactions (energy units)
E2 arg = two-body energy correction (energy units)
K3 arg = three-body exponential constant (inverse distance units)
K2 arg = two-body exponential constant (inverse distance units)
Rc3 arg = three-body cutoff (distance units)
Rc2 arg = two-body cutoff (distance units)
Rs arg = three-body switching function cutoff (distance units)
bondL arg = intramolecular OH bond length (distance units)
neigh arg = approximate integer number of molecules within Rc3 of an oxygen atom
```

### 4.94.2 Examples

```
pair_style e3b 1
pair_coeff * * Ea 35.85 Eb -240.2 Ec 449.3 E2 108269.9 K3 1.907 K2 4.872 Rc3 5.2 Rc2 5.2␣
↪Rs 5.0 bondL 0.9572

pair_style hybrid/overlay e3b 1 lj/cut/tip4p/long 1 2 1 1 0.15 8.5
pair_coeff * * e3b preset 2011

pair_style e3b OW
labelmap atom 1 C 2 H 3 O 4 N 5 OW 6 HW
pair_coeff * * Ea 35.85 Eb -240.2 Ec 449.3 E2 108269.9 K3 1.907 K2 4.872 Rc3 5.2 Rc2 5.2␣
↪Rs 5.0 bondL 0.9572
```

Used in example input script:

```
examples/PACKAGES/e3b/in.e3b-tip4p2005
```

### 4.94.3 Description

The *e3b* style computes an "explicit three-body" (E3B) potential for water *(Kumar 2008)*.

$$E = E_2 \sum_{i,j} e^{-k_2 r_{ij}} + E_A \sum_{\substack{i,j,k,\ell \\ \in \text{type A}}} f(r_{ij}) f(r_{k\ell}) + E_B \sum_{\substack{i,j,k,\ell \\ \in \text{type B}}} f(r_{ij}) f(r_{k\ell}) + E_C \sum_{\substack{i,j,k,\ell \\ \in \text{type C}}} f(r_{ij}) f(r_{k\ell})$$

$$f(r) = e^{-k_3 r} s(r)$$

$$s(r) = \begin{cases} 1 & r < R_s \\ \dfrac{(R_f - r)^2 (R_f - 3R_s + 2r)}{(R_f - R_s)^3} & R_s \le r \le R_f \\ 0 & r > R_f \end{cases}$$

This potential was developed as a water model that includes the three-body cooperativity of hydrogen bonding explicitly. To use it in this way, it must be applied in conjunction with a conventional two-body water model, through pair style *hybrid/overlay*. The three body interactions are split into three types: A, B, and C. Type A corresponds to anti-cooperative double hydrogen bond donor interactions. Type B corresponds to the cooperative interaction of molecules that both donate and accept a hydrogen bond. Type C corresponds to anti-cooperative double hydrogen bond acceptor interactions. The three-body interactions are smoothly cutoff by the switching function s(r) between Rs and Rc3. The two-body interactions are designed to correct for the effective many-body interactions implicitly included in the conventional two-body potential. The two-body interactions are cut off sharply at Rc2, because K3 is typically significantly smaller than K2. See *(Kumar 2008)* for more details.

Only a single *pair_coeff* command is used with the *e3b* style and the first two arguments must be * *. The oxygen atom type for the pair style is passed as the only argument to the *pair_style* command, not in the *pair_coeff* command. The hydrogen atom type is inferred from the ordering of the atoms.

**Note**

Every atom of type Otype must be part of a water molecule. Each water molecule must have consecutive IDs with the oxygen first. This pair style does not test that this criteria is met.

**Note**

If using type labels, the type labels must be defined before calling the *pair_coeff* command.

The *pair_coeff* command must have at least one keyword/value pair, as described above. The *preset* keyword sets the potential parameters to the values used in *(Tainter 2011)* or *(Tainter 2015)*. To use the water models defined in those references, the *e3b* style should always be used in conjunction with an *lj/cut/tip4p/long* style through *pair_style hybrid/overlay*, as demonstrated in the second example above. The *preset 2011* option should be used with the *TIP4P water model*. The *preset 2015* option should be used with the *TIP4P/2005 water model*. If the *preset* keyword is used, no other keyword is needed. Changes to the preset parameters can be made by specifying the *preset* keyword followed by the specific parameter to change, like *Ea*. Note that the other keywords must come after *preset* in the pair_style command. The *e3b* style can also be used to implement any three-body potential of the same form by specifying all the keywords except *neigh*: *Ea*, *Eb*, *Ec*, *E2*, *K3*, *K2*, *Rc3*, *Rc2*, *Rs*, and *bondL*. The keyword *bondL* specifies the intramolecular OH bond length of the water model being used. This is needed to include H atoms that are within the cutoff even when the attached oxygen atom is not.

This pair style allocates arrays sized according to the number of pairwise interactions within Rc3. To do this it needs an estimate for the number of water molecules within Rc3 of an oxygen atom. This estimate defaults to 10 and can be changed using the *neigh* keyword, which takes an integer as an argument. If the neigh setting is too small, the simulation will fail with the error "neigh is too small". If the neigh setting is too large, the pair style will use more memory than necessary.

This pair style tallies a breakdown of the total E3B potential energy into sub-categories, which can be accessed via the *compute pair* command as a vector of values of length 4. The 4 values correspond to the terms in the first equation above: the E2 term, the Ea term, the Eb term, and the Ec term.

See the examples/PACKAGES/e3b directory for a complete example script.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.94.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write its information to *binary restart files*. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style is incompatible with *respa*.

---

### 4.94.5 Restrictions

This pair style is part of the EXTRA-PAIR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires the *newton* setting to be "on" for pair interactions.

This pair style requires a fixed number of atoms in the simulation, so it is incompatible with fixes like *fix deposit*. If the number of atoms changes between runs, this pair style must be re-initialized by calling the *pair_style* and *pair_coeffs* commands. This is not a fundamental limitation of the pair style, but the code currently does not support a variable number of atoms.

The *preset* keyword currently only works with real, metal, si, and cgs *units*.

### 4.94.6 Related commands

*pair_coeff*, *compute pair*

### 4.94.7 Default

The option default for the *neigh* keyword is 10.

---

**(Kumar)** Kumar and Skinner, J. Phys. Chem. B, 112, 8311 (2008)

**(Tainter 2011)** Tainter, Pieniazek, Lin, and Skinner, J. Chem. Phys., 134, 184501 (2011)

**(Tainter 2015)** Tainter, Shi, and Skinner, 11, 2268 (2015)

## 4.95 pair_style eam command

Accelerator Variants: *eam/gpu*, *eam/intel*, *eam/kk*, *eam/omp*, *eam/opt*

## 4.96 pair_style eam/alloy command

Accelerator Variants: *eam/alloy/gpu*, *eam/alloy/intel*, *eam/alloy/kk*, *eam/alloy/omp*, *eam/alloy/opt*

## 4.97 pair_style eam/cd command

## 4.98 pair_style eam/cd/old command

## 4.99 pair_style eam/fs command

Accelerator Variants: *eam/fs/gpu*, *eam/fs/intel*, *eam/fs/kk*, *eam/fs/omp*, *eam/fs/opt*

## 4.100 pair_style eam/he command

### 4.100.1 Syntax

```
pair_style style
```

- style = *eam* or *eam/alloy* or *eam/cd* or *eam/cd/old* or *eam/fs* or *eam/he*

### 4.100.2 Examples

```
pair_style eam
pair_coeff * * cuu3
pair_coeff 1*3 1*3 niu3.eam

pair_style eam/alloy
pair_coeff * * ../potentials/NiAlH_jea.eam.alloy Ni Al Ni Ni

pair_style eam/cd
pair_coeff * * ../potentials/FeCr.cdeam Fe Cr

pair_style eam/fs
pair_coeff * * NiAlH_jea.eam.fs Ni Al Ni Ni

pair_style eam/he
pair_coeff * * PdHHe.eam.he Pd H He
```

### 4.100.3 Description

Style *eam* computes pairwise interactions for metals and metal alloys using embedded-atom method (EAM) potentials *(Daw)*. The total energy Ei of an atom I is given by

$$E_i = F_\alpha \left( \sum_{j \neq i} \rho_\beta(r_{ij}) \right) + \frac{1}{2} \sum_{j \neq i} \phi_{\alpha\beta}(r_{ij})$$

where F is the embedding energy which is a function of the atomic electron density rho, phi is a pair potential interaction, and alpha and beta are the element types of atoms I and J. The multi-body nature of the EAM potential is a result of the embedding energy term. Both summations in the formula are over all neighbors J of atom I within the cutoff distance.

The cutoff distance and the tabulated values of the functionals F, rho, and phi are listed in one or more files which are specified by the *pair_coeff* command. These are ASCII text files in a DYNAMO-style format which is described below. DYNAMO was the original serial EAM MD code, written by the EAM originators. Several DYNAMO potential files for different metals are included in the "potentials" directory of the LAMMPS distribution. All of these files are parameterized in terms of LAMMPS *metal units*.

**Note**

The *eam* style reads single-element EAM potentials in the DYNAMO *funcfl* format. Either single element or alloy systems can be modeled using multiple *funcfl* files and style *eam*. For the alloy case LAMMPS mixes the single-element potentials to produce alloy potentials, the same way that DYNAMO does. Alternatively, a single DYNAMO *setfl* file or Finnis/Sinclair EAM file can be used by LAMMPS to model alloy systems by invoking the *eam/alloy* or *eam/cd* or *eam/fs* or *eam/he* styles as described below. These files require no mixing since they specify alloy interactions explicitly.

**Note**

Note that unlike for other potentials, cutoffs for EAM potentials are not set in the pair_style or pair_coeff command; they are specified in the EAM potential files themselves. Likewise, valid EAM potential files usually contain atomic masses; thus you may not need to use the *mass* command to specify them, unless the potential file uses a dummy value (e.g. 0.0). LAMMPS will print a warning, if this is the case.

There are web sites that distribute and document EAM potentials stored in DYNAMO or other formats:

- https://www.ctcms.nist.gov/potentials/
- https://openkim.org/

These potentials should be usable with LAMMPS, though the alternate formats would need to be converted to the DYNAMO format used by LAMMPS and described on this page. The NIST site is maintained by Chandler Becker (cbecker at nist.gov) who is good resource for info on interatomic potentials and file formats.

The OpenKIM Project at https://openkim.org/browse/models/by-type provides EAM potentials that can be used directly in LAMMPS with the *kim command* interface.

**Warning**

The EAM potential files tabulate the embedding energy as a function of the local electron density $\rho$. When atoms get too close, this electron density may exceed the range for which the embedding energy was tabulated for. To avoid crashes, LAMMPS will assume a linearly increasing embedding energy for electron densities beyond the maximum tabulated value. LAMMPS will print a warning when this happens. It may be acceptable at the beginning of an equilibration (e.g. when using randomized coordinates) but would be a big concern for accuracy if it happens during

production runs. The EAM potential file triggering the warning during production is thus not a good choice, and the EAM model in general not likely a good model for the kind of system under investigation.

---

For style *eam*, potential values are read from a file that is in the DYNAMO single-element *funcfl* format. If the DYNAMO file was created by a Fortran program, it cannot have "D" values in it for exponents. C only recognizes "e" or "E" for scientific notation.

For style *eam* a potential file must be assigned to each I,I pair of atom types by using one or more pair_coeff commands, each with a single argument:

- filename

Thus the following command

```
pair_coeff *2 1*2 cuu3.eam
```

will read the cuu3 potential file and use the tabulated Cu values for F, phi, rho that it contains for type pairs 1,1 and 2,2 (type pairs 1,2 and 2,1 are ignored). See the *pair_coeff* doc page for alternate ways to specify the path for the potential file. In effect, this makes atom types 1 and 2 in LAMMPS be Cu atoms. Different single-element files can be assigned to different atom types to model an alloy system. The mixing to create alloy potentials for type pairs with I != J is done automatically the same way that the serial DYNAMO code originally did it; you do not need to specify coefficients for these type pairs.

*Funcfl* files in the *potentials* directory of the LAMMPS distribution have an ".eam" suffix. A DYNAMO single-element *funcfl* file is formatted as follows:

- line 1: comment (ignored)
- line 2: atomic number, mass, lattice constant, lattice type (e.g. FCC)
- line 3: Nrho, drho, Nr, dr, cutoff

On line 2, all values but the mass are ignored by LAMMPS. The mass is in mass *units*, e.g. mass number or grams/mole for metal units. The cubic lattice constant is in Angstroms. On line 3, Nrho and Nr are the number of tabulated values in the subsequent arrays, drho and dr are the spacing in density and distance space for the values in those arrays, and the specified cutoff becomes the pairwise cutoff used by LAMMPS for the potential. The units of dr are Angstroms; I'm not sure of the units for drho - some measure of electron density.

Following the three header lines are three arrays of tabulated values:

- embedding function F(rho) (Nrho values)
- effective charge function Z(r) (Nr values)
- density function rho(r) (Nr values)

The values for each array can be listed as multiple values per line, so long as each array starts on a new line. For example, the individual Z(r) values are for r = 0,dr,2*dr, ... (Nr-1)*dr.

The units for the embedding function F are eV. The units for the density function rho are the same as for drho (see above, electron density). The units for the effective charge Z are "atomic charge" or sqrt(Hartree * Bohr-radii). For two interacting atoms i,j this is used by LAMMPS to compute the pair potential term in the EAM energy expression as r*phi, in units of eV-Angstroms, via the formula

$$r \cdot \phi = 27.2 \cdot 0.529 \cdot Z_i \cdot Z_j$$

where 1 Hartree = 27.2 eV and 1 Bohr = 0.529 Angstroms.

---

Style *eam/alloy* computes pairwise interactions using the same formula as style *eam*. However the associated *pair_coeff* command reads a DYNAMO *setfl* file instead of a *funcfl* file. *Setfl* files can be used to model a single-element or alloy system. In the alloy case, as explained above, *setfl* files contain explicit tabulated values for alloy interactions. Thus they allow more generality than *funcfl* files for modeling alloys.

For style *eam/alloy*, potential values are read from a file that is in the DYNAMO multi-element *setfl* format, except that element names (Ni, Cu, etc) are added to one of the lines in the file. If the DYNAMO file was created by a Fortran program, it cannot have "D" values in it for exponents. C only recognizes "e" or "E" for scientific notation.

Only a single pair_coeff command is used with the *eam/alloy* style which specifies a DYNAMO *setfl* file, which contains information for M elements. These are mapped to LAMMPS atom types by specifying N additional arguments after the filename in the pair_coeff command, where N is the number of LAMMPS atom types:

- filename
- N element names = mapping of *setfl* elements to atom types

As an example, the potentials/NiAlH_jea.eam.alloy file is a *setfl* file which has tabulated EAM values for 3 elements and their alloy interactions: Ni, Al, and H. See the *pair_coeff* doc page for alternate ways to specify the path for the potential file. If your LAMMPS simulation has 4 atoms types and you want the first 3 to be Ni, and the fourth to be Al, you would use the following pair_coeff command:

```
pair_coeff * * NiAlH_jea.eam.alloy Ni Ni Ni Al
```

The first 2 arguments must be * * so as to span all LAMMPS atom types. The first three Ni arguments map LAMMPS atom types 1,2,3 to the Ni element in the *setfl* file. The final Al argument maps LAMMPS atom type 4 to the Al element in the *setfl* file. Note that there is no requirement that your simulation use all the elements specified by the *setfl* file.

If a mapping value is specified as NULL, the mapping is not performed. This can be used when an *eam/alloy* potential is used as part of the *hybrid* pair style. The NULL values are placeholders for atom types that will be used with other potentials.

*Setfl* files in the *potentials* directory of the LAMMPS distribution have an ".eam.alloy" suffix. A DYNAMO multi-element *setfl* file is formatted as follows:

- lines 1,2,3 = comments (ignored)
- line 4: Nelements Element1 Element2 ... ElementN
- line 5: Nrho, drho, Nr, dr, cutoff

In a DYNAMO *setfl* file, line 4 only lists Nelements = the # of elements in the *setfl* file. For LAMMPS, the element name (Ni, Cu, etc) of each element must be added to the line, in the order the elements appear in the file.

The meaning and units of the values in line 5 is the same as for the *funcfl* file described above. Note that the cutoff (in Angstroms) is a global value, valid for all pairwise interactions for all element pairings.

Following the 5 header lines are Nelements sections, one for each element, each with the following format:

- line 1 = atomic number, mass, lattice constant, lattice type (e.g. FCC)
- embedding function F(rho) (Nrho values)
- density function rho(r) (Nr values)

As with the *funcfl* files, only the mass (in mass *units*, e.g. mass number or grams/mole for metal units) is used by LAMMPS from the first line. The cubic lattice constant is in Angstroms. The F and rho arrays are unique to a single element and have the same format and units as in a *funcfl* file.

Following the Nelements sections, Nr values for each pair potential phi(r) array are listed for all i,j element pairs in the same format as other arrays. Since these interactions are symmetric (i,j = j,i) only phi arrays with i >= j are listed, in the following order: i,j = (1,1), (2,1), (2,2), (3,1), (3,2), (3,3), (4,1), ..., (Nelements, Nelements). Unlike the effective

charge array Z(r) in *funcfl* files, the tabulated values for each phi function are listed in *setfl* files directly as r*phi (in units of eV-Angstroms), since they are for atom pairs.

---

Style *eam/cd* is similar to the *eam/alloy* style, except that it computes alloy pairwise interactions using the concentration-dependent embedded-atom method (CD-EAM). This model can reproduce the enthalpy of mixing of alloys over the full composition range, as described in *(Stukowski)*. Style *eam/cd/old* is an older, slightly different and slower two-site formulation of the model *(Caro)*.

The pair_coeff command is specified the same as for the *eam/alloy* style. However the DYNAMO *setfl* file must has two lines added to it, at the end of the file:

- line 1: Comment line (ignored)
- line 2: N Coefficient0 Coefficient1 ... CoefficientN

The last line begins with the degree *N* of the polynomial function *h(x)* that modifies the cross interaction between A and B elements. Then *N+1* coefficients for the terms of the polynomial are then listed.

Modified EAM *setfl* files used with the *eam/cd* style must contain exactly two elements, i.e. in the current implementation the *eam/cd* style only supports binary alloys. The first and second elements in the input EAM file are always taken as the *A* and *B* species.

*CD-EAM* files in the *potentials* directory of the LAMMPS distribution have a ".cdeam" suffix.

---

Style *eam/fs* computes pairwise interactions for metals and metal alloys using a generalized form of EAM potentials due to Finnis and Sinclair *(Finnis)*. Style *eam/he* is similar to *eam/fs* except that it allows for negative electron density in order to capture the behavior of helium in metals *(Zhou6)*.

The total energy Ei of an atom I is given by

$$E_i = F_\alpha \left( \sum_{j \neq i} \rho_{\alpha\beta}(r_{ij}) \right) + \frac{1}{2} \sum_{j \neq i} \phi_{\alpha\beta}(r_{ij})$$

where $\rho_{\alpha\beta}$ refers to the density contributed by a neighbor atom J of element $\beta$ at the site of atom I of element $\alpha$. This has the same form as the EAM formula above, except that rho is now a functional specific to the elements of both atoms I and J, so that different elements can contribute differently to the total electron density at an atomic site depending on the identity of the element at that atomic site.

The associated *pair_coeff* command for style *eam/fs* or *eam/he* reads a DYNAMO *setfl* file that has been extended to include additional $\rho_{\alpha\beta}$ arrays of tabulated values. A discussion of how FS EAM differs from conventional EAM alloy potentials is given in *(Ackland1)*. An example of such a potential is the same author's Fe-P FS potential *(Ackland2)*. Note that while FS potentials always specify the embedding energy with a square root dependence on the total density, the implementation in LAMMPS does not require that; the user can tabulate any functional form desired in the FS potential files.

For style *eam/fs* and *eam/he* the form of the pair_coeff command is exactly the same as for style *eam/alloy*, e.g.

```
pair_coeff * * NiAlH_jea.eam.fs Ni Ni Ni Al
```

with N additional arguments after the filename, where N is the number of LAMMPS atom types. See the *pair_coeff* doc page for alternate ways to specify the path for the potential file. The N values determine the mapping of LAMMPS atom types to EAM elements in the file, as described above for style *eam/alloy*. As with *eam/alloy*, if a mapping value is NULL, the mapping is not performed. This can be used when an *eam/fs* or *eam/he* potential is used as part of a *hybrid* pair style. The NULL values are used as placeholders for atom types that will be used with other potentials.

FS EAM and HE EAM files include more information than the DYNAMO *setfl* format files read by *eam/alloy*, in that i,j density functionals for all pairs of elements are included as needed by the Finnis/Sinclair formulation of the EAM.

FS EAM files in the *potentials* directory of the LAMMPS distribution have an ".eam.fs" suffix. They are formatted as follows:

- lines 1,2,3 = comments (ignored)
- line 4: Nelements Element1 Element2 ... ElementN
- line 5: Nrho, drho, Nr, dr, cutoff

The 5-line header section is identical to an EAM *setfl* file.

Following the header are Nelements sections, one for each element $\beta$, each with the following format:

- line 1 = atomic number, mass, lattice constant, lattice type (e.g. FCC)
- embedding function F(rho) (Nrho values)
- density function $\rho_{1\beta}(r)$ for element $\beta$ at element 1 (Nr values)
- density function $\rho_{2\beta}(r)$ for element $\beta$ at element 2
- ...
- density function $\rho_{N_{elem}\beta}(r)$ for element $\beta$ at element $N_{elem}$

The units of these quantities in line 1 are the same as for *setfl* files. Note that the rho(r) arrays in Finnis/Sinclair can be asymmetric ($\rho_{\alpha\beta}(r) \neq \rho_{\beta\alpha}(r)$ ) so there are Nelements^2 of them listed in the file.

Following the Nelements sections, Nr values for each pair potential phi(r) array are listed in the same manner (r*phi, units of eV-Angstroms) as in EAM *setfl* files. Note that in Finnis/Sinclair, the phi(r) arrays are still symmetric, so only phi arrays for i >= j are listed.

HE EAM files in the *potentials* directory of the LAMMPS distribution have an ".eam.he" suffix. They are formatted as follows:

- lines 1,2,3 = comments (ignored)
- line 4: Nelements Element1 Element2 ... ElementN
- line 5: Nrho, drho, Nr, dr, cutoff, rhomax

The 5-line header section is identical to an FS EAM file except that line 5 lists an additional value, rhomax. Unlike in FS EAM files where embedding energies F(rho) are always defined between rho = 0 and rho = (Nrho -1)drho, F(rho) in HE EAM files are defined between rho = rhomin and rho = rhomax. Since drho = (rhomax - rhomin)/(Nrho - 1), rhomin = rhomax - (Nrho - 1)drho. The embedding energies F(rho) are listed for rho = rhomin, rhomin + drho, rhomin + 2drho, ..., rhomax. This gives users additional flexibility to define a negative rhomin and therefore an embedding energy function that works for both positive and negative electron densities. The format and units of these sections are identical to the FS EAM files (see above).

---

Added in version 3Nov2022.

The *eam*, *eam/alloy*, *eam/fs*, and *eam/he* pair styles support extraction of two per-atom quantities by the *fix pair* command. This allows the quantities to be output to files by the *dump* or otherwise processed by other LAMMPS commands.

The names of the two quantities are "rho" and "fp" for the density and derivative of the embedding energy for each atom. Neither quantity needs to be triggered by the *fix pair* command in order for these pair styles to calculate it.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.100.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, where types I and J correspond to two different element types, mixing is performed by LAMMPS as described above with the individual styles. You never need to specify a pair_coeff command with I != J arguments for the eam styles.

This pair style does not support the *pair_modify* shift, table, and tail options.

The eam pair styles do not write their information to *binary restart files*, since it is stored in tabulated potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

The eam pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

---

### 4.100.5 Restrictions

All of these styles are part of the MANYBODY package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Two-band (s+d) and multi-band EAM potentials, such as the Fe-Cr models that add an extra "s-band" embedding term to the standard EAM energy, can be run with the existing *eam/fs* and *hybrid/overlay* styles without modifying LAMMPS. See the *Howto_eam_overlay* page for the construction and a helper script that builds the s-band file.

### 4.100.6 Related commands

*pair_coeff* , *pair_style hybrid/overlay*, *Howto_eam_overlay*

### 4.100.7 Default

none

---

**(Ackland1)** Ackland, Condensed Matter (2005).

**(Ackland2)** Ackland, Mendelev, Srolovitz, Han and Barashev, Journal of Physics: Condensed Matter, 16, S2629 (2004).

**(Daw)** Daw, Baskes, Phys Rev Lett, 50, 1285 (1983). Daw, Baskes, Phys Rev B, 29, 6443 (1984).

**(Finnis)** Finnis, Sinclair, Philosophical Magazine A, 50, 45 (1984).

**(Zhou6)** Zhou, Bartelt, Sills, Physical Review B, 103, 014108 (2021).

**(Stukowski)** Stukowski, Sadigh, Erhart, Caro; Modeling Simulation Materials Science & Engineering, 7, 075005 (2009).

**(Caro)** A Caro, DA Crowson, M Caro; Phys Rev Lett, 95, 075702 (2005)

# 4.101 pair_style eam/apip command

Constant precision variant: *eam*

# 4.102 pair_style eam/fs/apip command

Constant precision variant: *eam/fs*

## 4.102.1 Syntax

```
pair_style eam/apip
pair_style eam/fs/apip
```

## 4.102.2 Examples

```
pair_style hybrid/overlay eam/fs/apip pace/precise/apip lambda/input/csp/apip fcc cutoff␣
↪5.0 lambda/zone/apip 12.0
pair_coeff * * eam/fs/apip Cu.eam.fs Cu
pair_coeff * * pace/precise/apip Cu_precise.yace Cu
pair_coeff * * lambda/input/csp/apip
pair_coeff * * lambda/zone/apip
```

## 4.102.3 Description

Style *eam* computes pairwise interactions for metals and metal alloys using embedded-atom method (EAM) potentials *(Daw)*. The total energy $E_i$ of an atom $i$ is given by

$$E_i^{\mathrm{EAM}} = F_\alpha \left( \sum_{j \neq i} \rho_\beta(r_{ij}) \right) + \frac{1}{2} \sum_{j \neq i} \phi_{\alpha\beta}(r_{ij})$$

where $F$ is the embedding energy which is a function of the atomic electron density $\rho$, $\phi$ is a pair potential interaction, and $\alpha$ and $\beta$ are the element types of atoms $i$ and $j$. The multi-body nature of the EAM potential is a result of the embedding energy term. Both summations in the formula are over all neighbors $j$ of atom $i$ within the cutoff distance. EAM is documented in detail in *pair_style eam*.

The potential energy $E_i$ of an atom $i$ of an adaptive-precision interatomic potential (APIP) according to *(Immel)* is given by

$$E_i^{\mathrm{APIP}} = \lambda_i E_i^{(\mathrm{fast})} + (1 - \lambda_i) E_i^{(\mathrm{precise})},$$

whereas the switching parameter $\lambda_i$ is computed dynamically during a simulation by *fix lambda/apip* or set prior to a simulation via *set*.

The pair style *eam/fs/apip* computes the potential energy $\lambda_i E_i^{\mathrm{EAM}}$ and the corresponding force and should be combined with a precise potential like *pair_style pace/precise/apip* that computes the potential energy $(1 - \lambda_i) E_i^{(\mathrm{precise})}$ and the corresponding force via *pair_style hybrid/overlay*.

## 4.102.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, where types I and J correspond to two different element types, mixing is performed by LAMMPS as described above with the individual styles. You never need to specify a pair_coeff command with I != J arguments for the eam/apip styles.

This pair style does not support the *pair_modify* shift, table, and tail options.

The eam/apip pair styles do not write their information to *binary restart files*, since it is stored in tabulated potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

The eam/apip pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

---

### 4.102.5 Restrictions

These pair styles are part of the APIP package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.102.6 Related commands

*pair_style eam*, *pair_style hybrid/overlay*, *fix lambda/apip*, *fix lambda_thermostat/apip*, *pair_style lambda/zone/apip*, *pair_style lambda/input/apip*, *pair_style pace/apip*, *fix atom_weight/apip*

### 4.102.7 Default

none

---

**(Immel)** Immel, Drautz and Sutmann, J Chem Phys, 162, 114119 (2025)

**(Daw)** Daw, Baskes, Phys Rev Lett, 50, 1285 (1983). Daw, Baskes, Phys Rev B, 29, 6443 (1984).

## 4.103 pair_style edip command

Accelerator Variants: *edip/omp*

## 4.104 pair_style edip/multi command

### 4.104.1 Syntax

```
pair_style style
```

- style = *edip* or *edip/multi*

### 4.104.2 Examples

```
pair_style edip
pair_coeff * * Si.edip Si
```

### 4.104.3 Description

The *edip* and *edip/multi* styles compute a 3-body *EDIP* potential which is popular for modeling silicon materials where it can have advantages over other models such as the *Stillinger-Weber* or *Tersoff* potentials. The *edip* style has been programmed for single element potentials, while *edip/multi* supports multi-element EDIP runs.

In EDIP, the energy E of a system of atoms is

$$E = \sum_{j \neq i} \phi_2(R_{ij}, Z_i) + \sum_{j \neq i} \sum_{k \neq i, k > j} \phi_3(R_{ij}, R_{ik}, Z_i)$$

$$\phi_2(r, Z) = A \left[ \left( \frac{B}{r} \right)^{\rho} - e^{-\beta Z^2} \right] exp\left( \frac{\sigma}{r - a} \right)$$

$$\phi_3(R_{ij}, R_{ik}, Z_i) = exp\left( \frac{\gamma}{R_{ij} - a} \right) exp\left( \frac{\gamma}{R_{ik} - a} \right) h(cos\theta_{ijk}, Z_i)$$

$$Z_i = \sum_{m \neq i} f(R_{im}) \qquad f(r) = \begin{cases} 1 & r < c \\ \exp\left( \frac{\alpha}{1 - x^{-3}} \right) & c < r < a \\ 0 & r > a \end{cases}$$

$$h(l, Z) = \lambda [(1 - e^{-Q(Z)(l + \tau(Z))^2}) + \eta Q(Z)(l + \tau(Z))^2]$$

$$Q(Z) = Q_0 e^{-\mu Z} \qquad \tau(Z) = u_1 + u_2(u_3 e^{-u_4 Z} - e^{-2u_4 Z})$$

where $\phi_2$ is a two-body term and $\phi_3$ is a three-body term. The summations in the formula are over all neighbors J and K of atom I within a cutoff distance = a. Both terms depend on the local environment of atom I through its effective coordination number defined by Z, which is unity for a cutoff distance < c and gently goes to 0 at distance = a.

Only a single pair_coeff command is used with the *edip* style which specifies a EDIP potential file with parameters for all needed elements. These are mapped to LAMMPS atom types by specifying N additional arguments after the filename in the pair_coeff command, where N is the number of LAMMPS atom types:

- filename
- N element names = mapping of EDIP elements to atom types

See the *pair_coeff* page for alternate ways to specify the path for the potential file.

As an example, imagine a file Si.edip has EDIP values for Si.

EDIP files in the *potentials* directory of the LAMMPS distribution have a ".edip" suffix. Lines that are not blank or comments (starting with #) define parameters for a triplet of elements. The parameters in a single entry correspond to the two-body and three-body coefficients in the formula above:

- element 1 (the center atom in a 3-body interaction)
- element 2
- element 3
- A (energy units)
- B (distance units)
- cutoffA (distance units)
- cutoffC (distance units)
- $\alpha$
- $\beta$
- $\eta$
- $\gamma$ (distance units)
- *lambda* (energy units)
- $\mu$
- $\tau$

- $\sigma$ (distance units)
- Q0
- u1
- u2
- u3
- u4

The A, B, beta, sigma parameters are used only for two-body interactions. The eta, gamma, lambda, mu, Q0 and all u1 to u4 parameters are used only for three-body interactions. The alpha and cutoffC parameters are used for the coordination environment function only.

The EDIP potential file must contain entries for all the elements listed in the pair_coeff command. It can also contain entries for additional elements not being used in a particular simulation; LAMMPS ignores those entries.

For a single-element simulation, only a single entry is required (e.g. SiSiSi). For a two-element simulation, the file must contain 8 entries (for SiSiSi, SiSiC, SiCSi, SiCC, CSiSi, CSiC, CCSi, CCC), that specify EDIP parameters for all permutations of the two elements interacting in three-body configurations. Thus for 3 elements, 27 entries would be required, etc.

At the moment, only a single element parameterization is implemented. However, the author is not aware of other multi-element EDIP parameterization. If you know any and you are interest in that, please contact the author of the EDIP package.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.104.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.104.5 Restrictions

This pair style can only be used if LAMMPS was built with the MANYBODY package. See the *Build package* doc page for more info.

This pair style requires the *newton* setting to be "on" for pair interactions.

The EDIP potential files provided with LAMMPS (see the potentials directory) are parameterized for metal *units*. You can use the EDIP potential with any LAMMPS units, but you would need to create your own EDIP potential file with coefficients listed in the appropriate units if your simulation does not use "metal" units.

### 4.104.6 Related commands

*pair_coeff*

### 4.104.7 Default

none

---

**(EDIP)** J F Justo et al, Phys Rev B 58, 2539 (1998).

## 4.105 pair_style eff/cut command

### 4.105.1 Syntax

```
pair_style eff/cut cutoff keyword args ...
```

- cutoff = global cutoff for Coulombic interactions
- zero or more keyword/value pairs may be appended

```
keyword = limit/eradius or pressure/evirials or ecp
  limit/eradius args = none
  pressure/evirials args = none
  ecp args = type element type element ...
    type = LAMMPS atom type (1 to Ntypes)
    element = element symbol (e.g. H, Si)
```

### 4.105.2 Examples

```
pair_style eff/cut 39.7
pair_style eff/cut 40.0 limit/eradius
pair_style eff/cut 40.0 limit/eradius pressure/evirials
pair_style eff/cut 40.0 ecp 1 Si 3 C
pair_coeff * *
pair_coeff 2 2 20.0
pair_coeff 1 s 0.320852 2.283269 0.814857
pair_coeff 3 p 22.721015 0.728733 1.103199 17.695345 6.693621
```

### 4.105.3 Description

This pair style contains a LAMMPS implementation of the electron Force Field (eFF) potential currently under development at Caltech, as described in *(Jaramillo-Botero)*. The eFF for Z<6 was first introduced by *(Su)* in 2007. It has been extended to higher Zs by using effective core potentials (ECPs) that now cover up to second and third row p-block elements of the periodic table.

eFF can be viewed as an approximation to QM wave packet dynamics and Fermionic molecular dynamics, combining the ability of electronic structure methods to describe atomic structure, bonding, and chemistry in materials, and of plasma methods to describe nonequilibrium dynamics of large systems with a large number of highly excited electrons. Yet, eFF relies on a simplification of the electronic wave function in which electrons are described as floating Gaussian wave packets whose position and size respond to the various dynamic forces between interacting classical nuclear particles and spherical Gaussian electron wave packets. The wave function is taken to be a Hartree product of the wave packets. To compensate for the lack of explicit antisymmetry in the resulting wave function, a spin-dependent Pauli potential is included in the Hamiltonian. Substituting this wave function into the time-dependent Schrodinger equation produces equations of motion that correspond - to second order - to classical Hamiltonian relations between electron position and size, and their conjugate momenta. The N-electron wave function is described as a product of one-electron Gaussian functions, whose size is a dynamical variable and whose position is not constrained to a nuclear center. This form allows for straightforward propagation of the wave function, with time, using a simple formulation from which the equations of motion are then integrated with conventional MD algorithms. In addition to this spin-dependent Pauli repulsion potential term between Gaussians, eFF includes the electron kinetic energy from the Gaussians. These two terms are based on first-principles quantum mechanics. On the other hand, nuclei are described as point charges, which interact with other nuclei and electrons through standard electrostatic potential forms.

The full Hamiltonian (shown below), contains then a standard description for electrostatic interactions between a set of delocalized point and Gaussian charges which include, nuclei-nuclei (NN), electron-electron (ee), and nuclei-electron (Ne). Thus, eFF is a mixed QM-classical mechanics method rather than a conventional force field method (in which electron motions are averaged out into ground state nuclear motions, i.e a single electronic state, and particle interactions are described via empirically parameterized interatomic potential functions). This makes eFF uniquely suited to simulate materials over a wide range of temperatures and pressures where electronically excited and ionized states of matter can occur and coexist. Furthermore, the interactions between particles -nuclei and electrons- reduce to the sum of a set of effective pairwise potentials in the eFF formulation. The *eff/cut* style computes the pairwise Coulomb interactions between nuclei and electrons (E_NN,E_Ne,E_ee), and the quantum-derived Pauli (E_PR) and Kinetic energy interactions potentials between electrons (E_KE) for a total energy expression given as,

$$U\left(R,r,s\right)=E_{NN}\left(R\right)+E_{Ne}\left(R,r,s\right)+E_{ee}\left(r,s\right)+E_{KE}\left(r,s\right)+E_{PR}\left(\uparrow\downarrow,S\right)$$

The individual terms are defined as follows:

$$\begin{aligned}
E_{KE} =& \frac{\hbar^2}{m_e}\sum_i \frac{3}{2s_i^2} \\
E_{NN} =& \frac{1}{4\pi\varepsilon_0}\sum_{i<j}\frac{Z_i Z_j}{R_{ij}} \\
E_{Ne} =& -\frac{1}{4\pi\varepsilon_0}\sum_{i,j}\frac{Z_i}{R_{ij}} Erf\left(\frac{\sqrt{2}R_{ij}}{s_j}\right) \\
E_{ee} =& \frac{1}{4\pi\varepsilon_0}\sum_{i<j}\frac{1}{r_{ij}} Erf\left(\frac{\sqrt{2}r_{ij}}{\sqrt{s_i^2+s_j^2}}\right) \\
E_{Pauli} =& \sum_{\sigma_i=\sigma_j} E\left(\uparrow\uparrow\right)_{ij} + \sum_{\sigma_i\neq\sigma_j} E\left(\uparrow\downarrow\right)_{ij}
\end{aligned}$$

where, s_i correspond to the electron sizes, the sigmas i's to the fixed spins of the electrons, Z_i to the charges on the nuclei, R_ij to the distances between the nuclei or the nuclei and electrons, and r_ij to the distances between electrons. For additional details see *(Jaramillo-Botero)*.

The overall electrostatics energy is given in Hartree units of energy by default and can be modified by an energy-conversion constant, according to the units chosen (see *electron_units*). The cutoff Rc, given in Bohrs (by default), truncates the interaction distance. The recommended cutoff for this pair style should follow the minimum image criterion, i.e. half of the minimum unit cell length.

This potential is designed to be used with *atom_style electron* definitions, in order to handle the description of systems with interacting nuclei and explicit electrons.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- cutoff (distance units)

For *eff/cut*, the cutoff coefficient is optional. If it is not used (as in some of the examples above), the default global value specified in the pair_style command is used.

---

The *limit/eradius* and *pressure/evirials* keywords are optional. Neither or both must be specified. If not specified they are unset.

The *limit/eradius* keyword is used to restrain electron size from becoming excessively diffuse at very high temperatures were the Gaussian wave packet representation breaks down, and from expanding as free particles to infinite size. If unset, electron radius is free to increase without bounds. If set, a restraining harmonic potential of the form E = 1/2k_ss^2 for s > L_box/2, where k_s = 1 Hartrees/Bohr^2, is applied on the electron radius.

The *pressure/evirials* keyword is used to control between two types of pressure computation: if unset, the computed pressure does not include the electronic radial virials contributions to the total pressure (scalar or tensor). If set, the computed pressure will include the electronic radial virial contributions to the total pressure (scalar and tensor).

The *ecp* keyword is used to associate an ECP representation for a particular atom type. The ECP captures the orbital overlap between a core pseudo particle and valence electrons within the Pauli repulsion. A list of type:element-symbol pairs may be provided for all ECP representations, after the "ecp" keyword.

**Note**

Default ECP parameters are provided for C, N, O, Al, and Si. Users can modify these using the pair_coeff command as exemplified above. For this, the User must distinguish between two different functional forms supported, one that captures the orbital overlap assuming the s-type core interacts with an s-like valence electron (s-s) and another that assumes the interaction is s-p. For systems that exhibit significant p-character (e.g. C, N, O) the s-p form is recommended. The "s" ECP form requires 3 parameters and the "p" 5 parameters.

**Note**

There are two different pressures that can be reported for eFF when defining this pair_style, one (default) that considers electrons do not contribute radial virial components (i.e. electrons treated as incompressible 'rigid' spheres) and one that does. The radial electronic contributions to the virials are only tallied if the flexible pressure option is set, and this will affect both global and per-atom quantities. In principle, the true pressure of a system is somewhere in between the rigid and the flexible eFF pressures, but, for most cases, the difference between these two pressures will not be significant over long-term averaged runs (i.e. even though the energy partitioning changes, the total energy remains similar).

---

**Note**

This implementation of eFF gives a reasonably accurate description for systems containing nuclei from Z = 1-6 in "all electron" representations. For systems with increasingly non-spherical electrons, Users should use the ECP representations. ECPs are now supported and validated for most of the second and third row elements of the p-block. Predefined parameters are provided for C, N, O, Al, and Si. The ECP captures the orbital overlap between the core and valence electrons (i.e. Pauli repulsion) with one of the functional forms:

$$E_{Pauli(ECP_s)} = p_1 \exp\left(-\frac{p_2 r^2}{p_3 + s^2}\right)$$

$$E_{Pauli(ECP_p)} = p_1 \left(\frac{2}{p_2/s + s/p_2}\right)(r - p_3 s)^2 \exp\left[-\frac{p_4 (r - p_3 s)^2}{p_5 + s^2}\right]$$

Where the first form correspond to core interactions with s-type valence electrons and the second to core interactions with p-type valence electrons.

The current version adds full support for models with fixed-core and ECP definitions. to enable larger timesteps (i.e. by avoiding the high frequency vibrational modes -translational and radial- of the 2 s electrons), and in the ECP case to reduce the increased orbital complexity in higher Z elements (up to Z<18). A fixed-core should be defined with a mass that includes the corresponding nuclear mass plus the 2 s electrons in atomic mass units (2x5.4857990943e-4), and a radius equivalent to that of minimized 1s electrons (see examples under /examples/PACKAGES/eff/fixed-core). An pseudo-core should be described with a mass that includes the corresponding nuclear mass, plus all the core electrons (i.e no outer shell electrons), and a radius equivalent to that of a corresponding minimized full-electron system. The charge for a pseudo-core atom should be given by the number of outer shell electrons.

In general, eFF excels at computing the properties of materials in extreme conditions and tracing the system dynamics over multi-picosecond timescales; this is particularly relevant where electron excitations can change significantly the nature of bonding in the system. It can capture with surprising accuracy the behavior of such systems because it describes consistently and in an unbiased manner many different kinds of bonds, including covalent, ionic, multicenter, ionic, and plasma, and how they interconvert and/or change when they become excited. eFF also excels in computing the relative thermochemistry of isodemic reactions and conformational changes, where the bonds of the reactants are of the same type as the bonds of the products. eFF assumes that kinetic energy differences dominate the overall exchange energy, which is true when the electrons present are nearly spherical and nodeless and valid for covalent compounds such as dense hydrogen, hydrocarbons, and diamond; alkali metals (e.g. lithium), alkali earth metals (e.g. beryllium) and semimetals such as boron; and various compounds containing ionic and/or multicenter bonds, such as boron dihydride.

---

### 4.105.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the cutoff distance for the *eff/cut* style can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

The *pair_modify* shift option is not relevant for these pair styles.

The *eff/long* (not yet available) style supports the *pair_modify* table option for tabulation of the short-range portion of the long-range Coulombic interaction.

These pair styles do not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

These pair styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

These pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

---

### 4.105.5 Restrictions

These pair styles will only be enabled if LAMMPS is built with the EFF package. It will only be enabled if LAMMPS was built with that package. See the *Build package* page for more info.

These pair styles require that particles store electron attributes such as radius, radial velocity, and radial force, as defined by the *atom_style*. The *electron* atom style does all of this.

Thes pair styles require you to use the *comm_modify vel yes* command so that velocities are stored by ghost atoms.

### 4.105.6 Related commands

*pair_coeff*

### 4.105.7 Default

If not specified, limit_eradius = 0 and pressure_with_evirials = 0.

---

**(Su)** Su and Goddard, Excited Electron Dynamics Modeling of Warm Dense Matter, Phys Rev Lett, 99:185003 (2007).

**(Jaramillo-Botero)** Jaramillo-Botero, Su, Qi, Goddard, Large-scale, Long-term Non-adiabatic Electron Molecular Dynamics for Describing Material Properties and Phenomena in Extreme Environments, J Comp Chem, 32, 497-512 (2011).

## 4.106 pair_style eim command

Accelerator Variants: *eim/omp*

### 4.106.1 Syntax

```
pair_style style
```

- style = *eim*

### 4.106.2 Examples

```
pair_style eim
pair_coeff * * Na Cl ../potentials/ffield.eim Na Cl
pair_coeff * * Na Cl ffield.eim  Na Na Na Cl
pair_coeff * * Na Cl ../potentials/ffield.eim Cl NULL Na
```

### 4.106.3 Description

Style *eim* computes pairwise interactions for ionic compounds using embedded-ion method (EIM) potentials *(Zhou)*. The energy of the system E is given by

$$E = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=i_1}^{i_N}\phi_{ij}\left(r_{ij}\right) + \sum_{i=1}^{N}E_i\left(q_i,\sigma_i\right)$$

The first term is a double pairwise sum over the J neighbors of all I atoms, where $\phi_{ij}$ is a pair potential. The second term sums over the embedding energy $E_i$ of atom I, which is a function of its charge $q_i$ and the electrical potential $\sigma_i$ at its location. $E_i$, $q_i$, and $\sigma_i$ are calculated as

$$q_i = \sum_{j=i_1}^{i_N}\eta_{ji}\left(r_{ij}\right)$$

$$\sigma_i = \sum_{j=i_1}^{i_N} q_j \cdot \psi_{ij}\left(r_{ij}\right)$$

$$E_i\left(q_i,\sigma_i\right) = \frac{1}{2}\cdot q_i \cdot \sigma_i$$

where $\eta_{ji}$ is a pairwise function describing electron flow from atom I to atom J, and $\psi_{ij}$ is another pairwise function. The multi-body nature of the EIM potential is a result of the embedding energy term. A complete list of all the pair functions used in EIM is summarized below

$$\phi_{ij}(r) = \begin{cases} \left[\frac{E_{b,ij}\beta_{ij}}{\beta_{ij}-\alpha_{ij}} \exp\left(-\alpha_{ij}\frac{r-r_{e,ij}}{r_{e,ij}}\right) - \frac{E_{b,ij}\alpha_{ij}}{\beta_{ij}-\alpha_{ij}} \exp\left(-\beta_{ij}\frac{r-r_{e,ij}}{r_{e,ij}}\right)\right] f_c\left(r, r_{e,ij}, r_{c,\phi,ij}\right), & p_{ij} = 1 \\ \left[\frac{E_{b,ij}\beta_{ij}}{\beta_{ij}-\alpha_{ij}} \left(\frac{r_{e,ij}}{r}\right)^{\alpha_{ij}} - \frac{E_{b,ij}\alpha_{ij}}{\beta_{ij}-\alpha_{ij}} \left(\frac{r_{e,ij}}{r}\right)^{\beta_{ij}}\right] f_c\left(r, r_{e,ij}, r_{c,\phi,ij}\right), & p_{ij} = 2 \end{cases}$$

$$\eta_{ji} = A_{\eta,ij}\left(\chi_j - \chi_i\right) f_c\left(r, r_{s,\eta,ij}, r_{c,\eta,ij}\right)$$

$$\psi_{ij}(r) = A_{\psi,ij} \exp\left(-\zeta_{ij} r\right) f_c\left(r, r_{s,\psi,ij}, r_{c,\psi,ij}\right)$$

$$f_c\left(r, r_p, r_c\right) = 0.510204 \cdot \operatorname{erfc}\left[\frac{1.64498\left(2r - r_p - r_c\right)}{r_c - r_p}\right] - 0.010204$$

Here $E_b, r_e, r_(c,\phi), \alpha, \beta, A_(\psi), \zeta, r_(s,\psi), r_(c,\psi), A_(\eta), r_(s,\eta), r_(c,\eta), \chi$, and pair function type $p$ are parameters, with subscripts $ij$ indicating the two species of atoms in the atomic pair.

**Note**

Even though the EIM potential is treating atoms as charged ions, you should not use a LAMMPS *atom_style* that stores a charge on each atom and thus requires you to assign a charge to each atom, e.g. the *charge* or *full* atom styles. This is because the EIM potential infers the charge on an atom from the equation above for $q_i$; you do not assign charges explicitly.

---

All the EIM parameters are listed in a potential file which is specified by the *pair_coeff* command. This is an ASCII text file in a format described below. The "ffield.eim" file included in the "potentials" directory of the LAMMPS distribution currently includes nine elements Li, Na, K, Rb, Cs, F, Cl, Br, and I. A system with any combination of these elements can be modeled. This file is parameterized in terms of LAMMPS *metal units*.

Note that unlike other potentials, cutoffs for EIM potentials are not set in the pair_style or pair_coeff command; they are specified in the EIM potential file itself. Likewise, the EIM potential file lists atomic masses; thus you do not need to use the *mass* command to specify them.

Only a single pair_coeff command is used with the *eim* style which specifies an EIM potential file and the element(s) to extract information for. The EIM elements are mapped to LAMMPS atom types by specifying N additional arguments after the filename in the pair_coeff command, where N is the number of LAMMPS atom types:

- Elem1, Elem2, ...
- EIM potential file
- N element names = mapping of EIM elements to atom types

See the *pair_coeff* page for alternate ways to specify the path for the potential file.

As an example like one of those above, suppose you want to model a system with Na and Cl atoms. If your LAMMPS simulation has 4 atoms types and you want the first 3 to be Na, and the fourth to be Cl, you would use the following pair_coeff command:

```
pair_coeff * * Na Cl ffield.eim Na Na Na Cl
```

The first 2 arguments must be * * so as to span all LAMMPS atom types. The filename is the EIM potential file. The Na and Cl arguments (before the file name) are the two elements for which info will be extracted from the potential file. The first three trailing Na arguments map LAMMPS atom types 1,2,3 to the EIM Na element. The final Cl argument maps LAMMPS atom type 4 to the EIM Cl element.

If a mapping value is specified as NULL, the mapping is not performed. This can be used when an *eim* potential is used as part of the *hybrid* pair style. The NULL values are placeholders for atom types that will be used with other potentials.

The ffield.eim file in the *potentials* directory of the LAMMPS distribution is formatted as follows:

Lines starting with # are comments and are ignored by LAMMPS. Lines starting with "global:" include three global values. The first value divides the cations from anions, i.e., any elements with electronegativity above this value are viewed as anions, and any elements with electronegativity below this value are viewed as cations. The second and third values are related to the cutoff function - i.e. the 0.510204, 1.64498, and 0.010204 shown in the above equation can be derived from these values.

Lines starting with "element:" are formatted as follows: name of element, atomic number, atomic mass, electronic negativity, atomic radius (LAMMPS ignores it), ionic radius (LAMMPS ignores it), cohesive energy (LAMMPS ignores it), and q0 (must be 0).

Lines starting with "pair:" are entered as: element 1, element 2, r_(c,phi), r_(c,phi) (redundant for historical reasons), E_b, r_e, alpha, beta, r_(c,eta), A_(eta), r_(s,eta), r_(c,psi), A_(psi), zeta, r_(s,psi), and p.

The lines in the file can be in any order; LAMMPS extracts the info it needs.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.106.4 Restrictions

This style is part of the MANYBODY package. It is only enabled if LAMMPS was built with that package.

### 4.106.5 Related commands

*pair_coeff*

### 4.106.6 Default

none

---

**(Zhou)** Zhou, submitted for publication (2010). Please contact Xiaowang Zhou (Sandia) for details via email at xzhou at sandia.gov.

## 4.107 pair_style exp6/rx command

Accelerator Variants: *exp6/rx/kk*

### 4.107.1 Syntax

```
pair_style exp6/rx cutoff ...
```

- cutoff = global cutoff for DPD interactions (distance units)
- weighting = fractional or molecular (optional)

### 4.107.2 Examples

```
pair_style exp6/rx 10.0
pair_style exp6/rx 10.0 fractional
pair_style exp6/rx 10.0 molecular
pair_coeff * * exp6.params h2o h2o exponent 1.0 1.0 10.0
pair_coeff * * exp6.params h2o 1fluid exponent 1.0 1.0 10.0
pair_coeff * * exp6.params 1fluid 1fluid exponent 1.0 1.0 10.0
pair_coeff * * exp6.params 1fluid 1fluid none 10.0
pair_coeff * * exp6.params 1fluid 1fluid polynomial filename 10.0
```

### 4.107.3 Description

Style *exp6/rx* is used in reaction DPD simulations, where the coarse-grained (CG) particles are composed of $m$ species whose reaction rate kinetics are determined from a set of $n$ reaction rate equations through the *fix rx* command. The species of one CG particle can interact with a species in a neighboring CG particle through a site-site interaction potential model. The *exp6/rx* style computes an exponential-6 potential given by

$$U_{ij}(r) = \frac{\varepsilon}{\alpha - 6}\{6\exp[\alpha(1 - \frac{r_{ij}}{R_m})] - \alpha(\frac{R_m}{r_{ij}})^6\}$$

where the $\varepsilon$ parameter determines the depth of the potential minimum located at $R_m$, and $\alpha$ determines the softness of the repulsion.

The coefficients must be defined for each species in a given particle type via the *pair_coeff* command as in the examples above, where the first argument is the filename that includes the exponential-6 parameters for each species. The file includes the species tag followed by the $\alpha, \varepsilon$ and $R_m$ parameters. The format of the file is described below.

The second and third arguments specify the site-site interaction potential between two species contained within two different particles. The species tags must either correspond to the species defined in the reaction kinetics files specified with the *fix rx* command or they must correspond to the tag "1fluid", signifying interaction with a product species mixture determined through a one-fluid approximation. The interaction potential is weighted by the geometric average of either the mole fraction concentrations or the number of molecules associated with the interacting coarse-grained particles (see the *fractional* or *molecular* weighting pair style options). The coarse-grained potential is stored before and after the reaction kinetics solver is applied, where the difference is defined to be the internal chemical energy (uChem).

The fourth argument specifies the type of scaling that will be used to scale the EXP-6 parameters as reactions occur. Currently, there are three scaling options: *exponent*, *polynomial* and *none*.

Exponent scaling requires two additional arguments for scaling the $R_m$ and $\varepsilon$ parameters, respectively. The scaling factor is computed by phi^exponent, where phi is the number of molecules represented by the coarse-grain particle and exponent is specified as a pair coefficient argument for $R_m$ and $\varepsilon$, respectively. The $R_m$ and $\varepsilon$ parameters are multiplied by the scaling factor to give the scaled interaction parameters for the CG particle.

Polynomial scaling requires a filename to be specified as a pair coeff argument. The file contains the coefficients to a fifth order polynomial for the $\alpha$, $\varepsilon$ and $R_m$ parameters that depend upon phi (the number of molecules represented by the CG particle). The format of a polynomial file is provided below.

The *none* option to the scaling does not have any additional pair coeff arguments. This is equivalent to specifying the *exponent* option with $R_m$ and $\varepsilon$ exponents of 0.0 and 0.0, respectively.

The final argument specifies the interaction cutoff (optional).

---

The format of a tabulated file is as follows (without the parenthesized comments):

```
# exponential-6 parameters for various species      (one or more comment or blank lines)

h2o  exp6  11.00 0.02 3.50                        # (species, exp6, alpha, Rm, epsilon)
no2  exp6  13.60 0.01 3.70
...
co2  exp6  13.00 0.03 3.20
```

The format of the polynomial scaling file as follows (without the parenthesized comments):

```
# POLYNOMIAL FILE          (one or more comment or blank lines)

#  General Functional Form:
#  A*phi^5 + B*phi^4 + C*phi^3 + D*phi^2 + E*phi + F
#
#  Parameter  A         B         C         D         E         F
                          (blank)
alpha        0.0000   0.00000   0.00008  0.04955  -0.73804  13.63201
epsilon      0.0000   0.00478  -0.06283  0.24486  -0.33737   2.60097
rm           0.0001  -0.00118  -0.00253  0.05812  -0.00509   1.50106
```

A section begins with a non-blank line whose first character is not a "#"; blank lines or lines starting with "#" can be used as comments between sections.

Following a blank line, the next N lines list the species and their corresponding parameters. The first argument is the species tag, the second argument is the exp6 tag, the third argument is the $\alpha$ parameter (energy units), the fourth argument is the $\varepsilon$ parameter (energy-distance^6 units), and the fifth argument is the $R_m$ parameter (distance units). If a species tag of "1fluid" is listed as a pair coefficient, a one-fluid approximation is specified where a concentration-dependent combination of the parameters is computed through the following equations:

$$R_m^3 = \sum_a \sum_b x_a x_b R_{m,ab}^3$$

$$\varepsilon = \frac{1}{R_m^3} \sum_a \sum_b x_a x_b \varepsilon_{ab} R_{m,ab}^3$$

$$\alpha = \frac{1}{\varepsilon R_m^3} \sum_a \sum_b x_a x_b \alpha_{ab} \varepsilon_{ab} R_{m,ab}^3$$

where

$$\varepsilon_{ab} = \sqrt{\varepsilon_a \varepsilon_b}$$

$$R_{m,ab} = \frac{R_{m,a} + R_{m,b}}{2}$$

$$\alpha_{ab} = \sqrt{\alpha_a \alpha_b}$$

and $x_a$ and $x_b$ are the mole fractions of a and b, respectively, which comprise the gas mixture.

---

### 4.107.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

This style does not support the *pair_modify* shift option for the energy of the exp() and 1/r^6 portion of the pair interaction.

This style does not support the pair_modify tail option for adding long-range tail corrections to energy and pressure for the A,C terms in the pair interaction.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.107.5 Restrictions

This command is part of the DPD-REACT package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.107.6 Related commands

*pair_coeff*

### 4.107.7 Default

fractional weighting

## 4.108 pair_style extep command

Accelerator Variants: *extep/omp*

### 4.108.1 Syntax

```
pair_style extep
```

### 4.108.2 Examples

```
pair_style extep
pair_coeff * * BN.extep B N
```

### 4.108.3 Description

Style *extep* computes the Extended Tersoff Potential (ExTeP) interactions as described in *(Los2017)*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.108.4 Restrictions

none

### 4.108.5 Related commands

*pair_tersoff*

### 4.108.6 Default

none

---

**(Los2017)** J. H. Los et al. "Extended Tersoff potential for boron nitride: Energetics and elastic properties of pristine and defective h-BN", Phys. Rev. B 96 (184108), 2017.

## 4.109 pair_style lj/cut/soft command

Accelerator Variants: *lj/cut/soft/omp*

## 4.110 pair_style lj/cut/soft/gapsys command

## 4.111 pair_style lj/cut/coul/cut/soft command

Accelerator Variants: *lj/cut/coul/cut/soft/gpu*, *lj/cut/coul/cut/soft/omp*

## 4.112 pair_style lj/cut/coul/long/soft command

Accelerator Variants: *lj/cut/coul/long/soft/gpu*, *lj/cut/coul/long/soft/omp*

## 4.113 pair_style lj/cut/tip4p/long/soft command

Accelerator Variants: *lj/cut/tip4p/long/soft/omp*

## 4.114 pair_style lj/charmm/coul/long/soft command

Accelerator Variants: *lj/charmm/coul/long/soft/omp*

## 4.115 pair_style lj/class2/soft command

Accelerator Variants: *lj/class2/soft/omp*

## 4.116 pair_style lj/class2/coul/cut/soft command

Accelerator Variants: *lj/class2/coul/cut/soft/omp*

## 4.117 pair_style lj/class2/coul/long/soft command

Accelerator Variants: *lj/class2/coul/long/soft/omp*

## 4.118 pair_style coul/cut/soft command

Accelerator Variants: *coul/cut/soft/omp*

## 4.119 pair_style coul/cut/soft/gapsys command

Accelerator Variants: *coul/cut/soft/gapsys/omp*

## 4.120 pair_style coul/long/soft command

Accelerator Variants: *coul/long/soft/omp*

## 4.121 pair_style tip4p/long/soft command

Accelerator Variants: *tip4p/long/soft/omp*

## 4.122 pair_style morse/soft command

Accelerator Variants: *morse/soft/omp*

### 4.122.1 Syntax

```
pair_style style args
```

- style = *lj/cut/soft* or *lj/cut/soft/gapsys* or *lj/cut/coul/cut/soft* or *lj/cut/coul/long/soft* or *lj/cut/tip4p/long/soft* or *lj/charmm/coul/long/soft* or *lj/class2/soft* or *lj/class2/coul/cut/soft* or *lj/class2/coul/long/soft* or *coul/cut/soft* or *coul/cut/soft/gapsys* or *coul/long/soft* or *tip4p/long/soft* or *morse/soft*
- args = list of arguments for a particular style

```
lj/cut/soft args = n alpha_lj cutoff
  n, alpha_LJ = parameters of soft-core potential
  cutoff = global cutoff for Lennard-Jones interactions (distance units)
```

```
lj/cut/soft/gapsys args = alpha_lj cutoff
  alpha_LJ = parameters of soft-core potential
  cutoff = global cutoff for Lennard-Jones interactions (distance units)
lj/cut/coul/cut/soft args = n alpha_LJ alpha_C cutoff (cutoff2)
  n, alpha_LJ, alpha_C = parameters of soft-core potential
  cutoff = global cutoff for LJ (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
lj/cut/coul/long/soft args = n alpha_LJ alpha_C cutoff
  n, alpha_LJ, alpha_C = parameters of the soft-core potential
  cutoff = global cutoff for LJ (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
lj/cut/tip4p/long/soft args = otype htype btype atype qdist n alpha_LJ alpha_C cutoff␣
↪(cutoff2)
  otype,htype = atom types (numeric or type label) for TIP4P O and H
  btype,atype = bond and angle types (numeric or type label) for TIP4P waters
  qdist = distance from O atom to massless charge (distance units)
  n, alpha_LJ, alpha_C = parameters of the soft-core potential
  cutoff = global cutoff for LJ (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
lj/charmm/coul/long/soft args = n alpha_LJ alpha_C inner outer (cutoff)
  n, alpha_LJ, alpha_C = parameters of the soft-core potential
  inner, outer = global switching cutoffs for LJ (and Coulombic if only 5 args)
  cutoff = global cutoff for Coulombic (optional, outer is Coulombic cutoff if only 5␣
↪args)
lj/class2/soft args = n alpha_lj cutoff
  n, alpha_LJ = parameters of soft-core potential
  cutoff = global cutoff for Lennard-Jones interactions (distance units)
lj/class2/coul/cut/soft args = n alpha_LJ alpha_C cutoff (cutoff2)
  n, alpha_LJ, alpha_C = parameters of soft-core potential
  cutoff = global cutoff for LJ (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
lj/class2/coul/long/soft args = n alpha_LJ alpha_C cutoff (cutoff2)
  n, alpha_LJ, alpha_C = parameters of soft-core potential
  cutoff = global cutoff for LJ (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
coul/cut/soft args = n alpha_C cutoff
  n, alpha_C = parameters of the soft-core potential
  cutoff = global cutoff for Coulomb interactions (distance units)
coul/cut/soft/gapsys args = sigma_q alpha_q cutoff
  sigma_q, alpha_q = parameters of the soft-core potential
  cutoff = global cutoff for Coulomb interactions (distance units)
coul/long/soft args = n alpha_C cutoff
  n, alpha_C = parameters of the soft-core potential
  cutoff = global cutoff for Coulomb interactions (distance units)
tip4p/long/soft args = otype htype btype atype qdist n alpha_C cutoff
  otype,htype = atom types (numeric or type label) for TIP4P O and H
  btype,atype = bond and angle types (numeric or type label) for TIP4P waters
  qdist = distance from O atom to massless charge (distance units)
  n, alpha_C = parameters of the soft-core potential
  cutoff = global cutoff for Coulomb interactions (distance units)
morse/soft args = n lf cutoff
  n = soft-core parameter
  lf = transformation range is lf < lambda < 1
  cutoff = global cutoff for Morse interactions (distance units)
```

### 4.122.2 Examples

```
pair_style lj/cut/soft 2.0 0.5 9.5
pair_coeff * * 0.28 3.1 1.0
pair_coeff 1 1 0.28 3.1 1.0 9.5

pair_style lj/cut/soft/gapsys 1.0 9.5
pair_coeff * * 0.28 3.1 1.0

pair_style lj/cut/coul/cut/soft 2.0 0.5 10.0 9.5
pair_style lj/cut/coul/cut/soft 2.0 0.5 10.0 9.5 9.5
pair_coeff * * 0.28 3.1 1.0
pair_coeff 1 1 0.28 3.1 0.5 10.0
pair_coeff 1 1 0.28 3.1 0.5 10.0 9.5

pair_style lj/cut/coul/long/soft 2.0 0.5 10.0 9.5
pair_style lj/cut/coul/long/soft 2.0 0.5 10.0 9.5 9.5
pair_coeff * * 0.28 3.1 1.0
pair_coeff 1 1 0.28 3.1 0.0 10.0
pair_coeff 1 1 0.28 3.1 0.0 10.0 9.5

pair_style lj/cut/tip4p/long/soft 1 2 7 8 0.15 2.0 0.5 10.0 9.8
pair_style lj/cut/tip4p/long/soft 1 2 7 8 0.15 2.0 0.5 10.0 9.8 9.5
pair_coeff * * 0.155 3.1536 1.0
pair_coeff 1 1 0.155 3.1536 1.0 9.5

pair_style lj/cut/tip4p/long/soft OW HW HW-OW HW-OW-HW 0.15 2.0 0.5 10.0 9.8
labelmap atom 1 OW 2 HW
labelmap bond 1 HW-OW
labelmap angle 1 HW-OW-HW
pair_coeff * * 0.155 3.1536 1.0
pair_coeff OW OW 0.155 3.1536 1.0 9.5

pair_style lj/charmm/coul/long 2.0 0.5 10.0 8.0 10.0
pair_style lj/charmm/coul/long 2.0 0.5 10.0 8.0 10.0 9.0
pair_coeff * * 0.28 3.1 1.0
pair_coeff 1 1 0.28 3.1 1.0 0.14 3.1

pair_style lj/class2/coul/long/soft 2.0 0.5 10.0 9.5
pair_style lj/class2/coul/long/soft 2.0 0.5 10.0 9.5 9.5
pair_coeff * * 0.28 3.1 1.0
pair_coeff 1 1 0.28 3.1 0.0 10.0
pair_coeff 1 1 0.28 3.1 0.0 10.0 9.5

pair_style coul/cut/soft/gapsys 1.0 2.0 9.5
pair_coeff * * 1.0

pair_style coul/long/soft 1.0 10.0 9.5
pair_coeff * * 1.0
pair_coeff 1 1 1.0

pair_style tip4p/long/soft 1 2 7 8 0.15 2.0 0.5 10.0 9.8
pair_coeff * * 1.0
```

(continues on next page)

(continued from previous page)

```
pair_coeff 1 1 1.0

pair_style morse/soft 4 0.9 10.0
pair_coeff * * 100.0 2.0 1.5 1.0
pair_coeff 1 1 100.0 2.0 1.5 1.0 3.0
```

Example input scripts available: examples/PACKAGES/fep

### 4.122.3 Description

These pair styles have a soft repulsive core, tunable by a parameter lambda, in order to avoid singularities during free energy calculations when sites are created or annihilated *(Beutler)*. When lambda tends to 0 the pair interaction vanishes with a soft repulsive core. When lambda tends to 1, the pair interaction approaches the normal, non-soft potential. These pair styles are suited for "alchemical" free energy calculations using the *fix adapt/fep* and *compute fep* commands.

The *lj/cut/soft* style and related sub-styles compute the 12-6 Lennard-Jones and Coulomb potentials modified by a soft core, with the functional form

$$E = \lambda^n 4\varepsilon \left\{ \frac{1}{\left[\alpha_{\mathrm{LJ}}(1-\lambda)^2 + \left(\frac{r}{\sigma}\right)^6\right]^2} - \frac{1}{\alpha_{\mathrm{LJ}}(1-\lambda)^2 + \left(\frac{r}{\sigma}\right)^6} \right\} \qquad r < r_c$$

The *lj/class2/soft* style is a 9-6 potential with the exponent of the denominator of the first term in brackets taking the value 1.5 instead of 2 (other details differ, see the form of the potential in *pair_style lj/class2*).

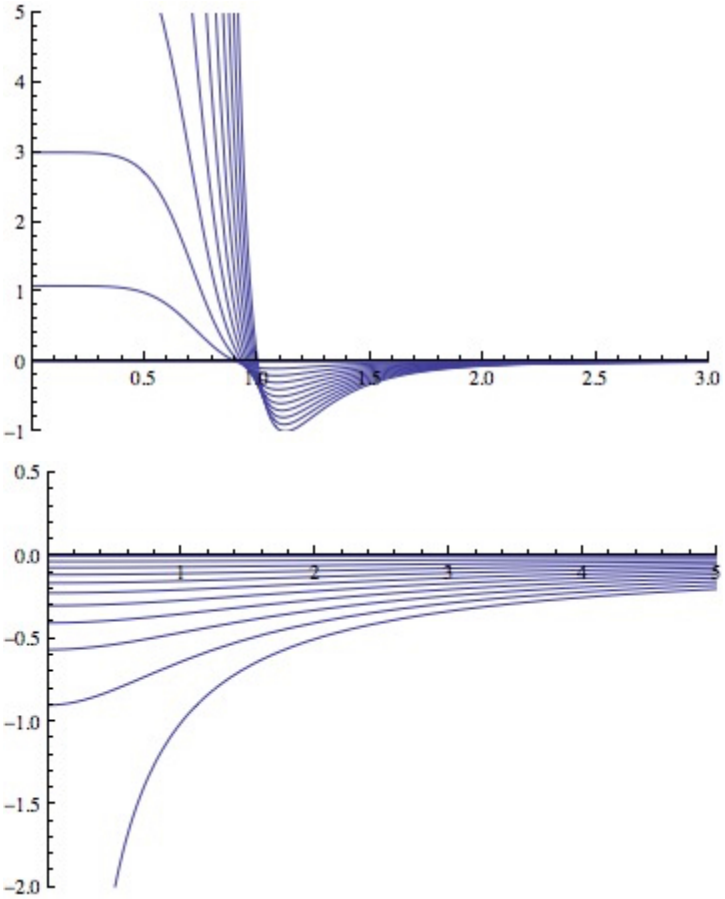
Coulomb interactions can also be damped with a soft core at short distance,

$$E = \lambda^n \frac{C q_i q_j}{\varepsilon \left[\alpha_{\mathrm{C}}(1-\lambda)^2 + r^2\right]^{1/2}} \qquad r < r_c$$

In the Coulomb part $C$ is an energy-conversion constant, $q_i$ and $q_j$ are the charges on the two atoms, and epsilon is the dielectric constant which can be set by the *dielectric* command.

The coefficient lambda is an activation parameter. When $\lambda = 1$ the pair potential is identical to a Lennard-Jones term or a Coulomb term or a combination of both. When $\lambda = 0$ the interactions are deactivated. The transition between these two extrema is smoothed by a soft repulsive core in order to avoid singularities in potential energy and forces when sites are created or annihilated and can overlap *(Beutler)*.

The parameters $n$, $\alpha_{\mathrm{LJ}}$ and $\alpha_{\mathrm{C}}$ are set in the *pair_style* command, before the cutoffs. Usual choices for the exponent are $n = 2$ or $n = 1$. For the remaining coefficients $\alpha_{\mathrm{LJ}} = 0.5$ and $\alpha_{\mathrm{C}} = 10$ A$^2$ are appropriate choices. Plots of the 12-6 LJ and Coulomb terms are shown below, for lambda ranging from 1 to 0 every 0.1.




For the *lj/cut/coul/cut/soft* or *lj/cut/coul/long/soft* pair styles, as well as for the equivalent *class2* versions, the following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- $\varepsilon$ (energy units)
- $\sigma$ (distance units)
- $\lambda$ (activation parameter, between 0 and 1)
- cutoff1 (distance units)
- cutoff2 (distance units)

The latter two coefficients are optional. If not specified, the global LJ and Coulombic cutoffs specified in the pair_style command are used. If only one cutoff is specified, it is used as the cutoff for both LJ and Coulombic interactions for this type pair. If both coefficients are specified, they are used as the LJ and Coulombic cutoffs for this type pair. You cannot specify 2 cutoffs for style *lj/cut/soft*, since it has no Coulombic terms. For the *coul/cut/soft* and *coul/long/soft* only lambda and the optional cutoff2 are to be specified.

Style *lj/cut/tip4p/long/soft* implements a soft-core version of the TIP4P water model. The usage of the TIP4P pair style is documented in the *pair_lj* styles. In the soft version the parameters $n$, $\alpha_{\mathrm{LJ}}$ and $\alpha_{\mathrm{C}}$ are set in the *pair_style* command, after the specific parameters of the TIP4P water model and before the cutoffs. The activation parameter lambda is supplied as an argument of the *pair_coeff* command, after epsilon and sigma and before the optional cutoffs.

**Note**

If using type labels, the type labels must be defined before calling the *pair_coeff* command.

Style *lj/charmm/coul/long/soft* implements a soft-core version of the modified 12-6 LJ potential used in CHARMM and documented in the *pair_style lj/charmm/coul/long* style. In the soft version the parameters $n$, $\alpha_{\mathrm{LJ}}$ and $\alpha_{\mathrm{C}}$ are set in the *pair_style* command, before the global cutoffs. The activation parameter lambda is introduced as an argument of the *pair_coeff* command, after $\varepsilon$ and $\sigma$ and before the optional eps14 and sigma14.

Style *lj/class2/soft* implements a soft-core version of the 9-6 potential in *pair_style lj/class2*. In the soft version the parameters $n$, $\alpha_{\mathrm{LJ}}$ and $\alpha_{\mathrm{C}}$ are set in the *pair_style* command, before the global cutoffs. The activation parameter lambda is introduced as an argument of the the *pair_coeff* command, after $\varepsilon$ and $\sigma$ and before the optional cutoffs.

The *coul/cut/soft*, *coul/long/soft* and *tip4p/long/soft* sub-styles are designed to be combined with other pair potentials via the *pair_style hybrid/overlay* command. This is because they have no repulsive core. Hence, if used by themselves, there will be no repulsion to keep two oppositely charged particles from overlapping each other. In this case, if $\lambda = 1$, a singularity may occur. These sub-styles are suitable to represent charges embedded in the Lennard-Jones radius of another site (for example hydrogen atoms in several water models). The $\lambda$ must be defined for each pair, and *coul/cut/soft* can accept an optional cutoff as the second coefficient.

**Note**

When using the soft-core Coulomb potentials with long-range solvers (*coul/long/soft*, *lj/cut/coul/long/soft*, etc.) in a free energy calculation in which sites holding electrostatic charges are being created or annihilated (using *fix adapt/fep* and *compute fep*) it is important to adapt both the $\lambda$ activation parameter (from 0 to 1, or the reverse) and the value of the charge (from 0 to its final value, or the reverse). This ensures that long-range electrostatic terms (kspace) are correct. It is not necessary to use soft-core Coulomb potentials if the van der Waals site is present during the free-energy route, thus avoiding overlap of the charges. Examples are provided in the LAMMPS source directory tree, under examples/PACKAGES/fep.

**Note**

To avoid division by zero do not set $\sigma = 0$ in the *lj/cut/soft* and related styles; use the lambda parameter instead to activate/deactivate interactions, or use $\varepsilon = 0$ and $\sigma = 1$. Alternatively, when sites do not interact though the Lennard-Jones term the *coul/long/soft* or similar sub-style can be used via the *pair_style hybrid/overlay* command.

---

The *morse/soft* variant modifies the *pair_morse* style at short range to have a soft core. The functional form differs from that of the *lj/soft* styles, and is instead given by:

$$
\begin{aligned}
s(\lambda) &= (1-\lambda)/(1-\lambda_f) \\
B &= -2De^{-2\alpha r_0}(e^{\alpha r_0}-1)/3 \\
E &= D_0\left[e^{-2\alpha(r-r_0)} - 2e^{-\alpha(r-r_0)}\right] + s(\lambda)Be^{-3\alpha(r-r_0)}, \qquad \lambda \geq \lambda_f, \quad r < r_c \\
E &= \left(D_0\left[e^{-2\alpha(r-r_0)} - 2e^{-\alpha(r-r_0)}\right] + Be^{-3\alpha(r-r_0)}\right)(\lambda/\lambda_f)^n, \qquad \lambda < \lambda_f, \quad r < r_c
\end{aligned}
$$

The *morse/soft* style requires the following pair coefficients:

- $D_0$ (energy units)
- $\alpha$ (1/distance units)
- $r_0$ (distance units)

- $\lambda$ (unitless, between 0.0 and 1.0)
- cutoff (distance units)

The last coefficient is optional. If not specified, the global morse cutoff is used.

---

Added in version 4Jul2026.

The pair style *coul/cut/soft/gapsys* implements the pair potential for Coulombic interactions which was proposed by Gapsys et al *(Gapsys)*. The main idea behind this potential is the definition of a distance $r_{inner}$, which is smaller than the cutoff distance $r_c$: for distances shorter than $r_{inner}$ the forces are computed based on a linearized expression while for distances larger than $r_{inner}$ the forces are computed based on the standard Coulombic potential. The linearized expression ensures continuity of forces as well as of the first derivative of the forces.

The distance $r_{inner}$ is given by

$$r_{inner} = \alpha_q \left(1 + \sigma_q \left|q_i q_j\right|\right) \lambda^{1/6}$$

where $q_i$ and $q_j$ are the charges on the two atoms. For $\lambda = 0$, $r_{inner} = 0$, which implies that the standard Coulombic potential is employed for all distances.

For distances larger than $r_{inner}$, the energy is computed by

$$E = \frac{C q_i q_j}{\varepsilon r} \qquad r_{inner} < r < r_c$$

where $C$ is an energy-conversion constant, and epsilon is the dielectric constant which can be set by the *dielectric* command.

For distances shorter than $r_{inner}$, the energy is computed by

$$E = \frac{q_i q_j}{r_{inner}^3} r^2 - \frac{3 q_i q_j}{r_{inner}^2} r + \frac{3 q_i q_j}{r_{inner}} \qquad r < r_{inner} < r_c$$

This pair style requires the following pair coefficients:

- $\sigma_q$ (inverse squared charge units, positive real number)
- $\alpha_q$ (distance units, positive real number)
- $\lambda$ (unitless, between 0.0 and 1.0)
- cutoff (distance units)

The recommended values for $\sigma_q$ and $\alpha_q$ are 1.0 and 0.3 $r_c$ respectively.

Added in version 4Jul2026.

Similarly, the pair style *lj/cut/soft/gapsys* implements the pair potential for Lennard-Jones interactions which was proposed by Gapsys et al *(Gapsys)*.

The distance $r_{inner}$ is given by

$$r_{inner} = \alpha_l j \left(26\lambda/7\right)^{1/6} \sigma$$

where $\sigma$ is the standard Lennard-Jones parameter. For $\lambda = 0$, $r_{inner} = 0$, which implies that the standard Lennard-Jones potential is employed for all distances.

For distances larger than $r_{inner}$, the energy is computed by

$$E = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \qquad r < r_c$$

$r_c$ is the cutoff.

For distances shorter than $r_{inner}$, the energy is computed by

$$E = 12\varepsilon c_1 r^2 - 96\varepsilon c_2 r + 28\varepsilon c_3 \qquad r < r_{inner} < r_c c_1 = 26\sigma^1 2/r_{inner}^1 4 - 7\sigma^6/r_{inner}^8 c_2 = 7\sigma^1 2/r_{inner}^1 3 - 2\sigma^6/r_{inner}^7 c_3 = 13\sigma^1 2/r_{inner}^1 2 - 4$$

This pair style requires the following pair coefficients:

- $\alpha_l j$ (distance units, positive real number)
- $\lambda$ (unitless, between 0.0 and 1.0)
- cutoff (distance units)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.122.4 Mixing, shift, table, tail correction, restart, rRESPA info

The different versions of the *lj/cut/soft* pair styles support mixing. For atom type pairs I,J and I != J, the $\varepsilon$ and $\sigma$ coefficients and cutoff distance for these pair styles can be mixed. The default mix value is *geometric* for 12-6 styles.

The mixing rule for epsilon and sigma for *lj/class2/soft* 9-6 potentials is to use the *sixthpower* formulas. The *pair_modify mix* setting is thus ignored for class2 potentials for $\varepsilon$ and $\sigma$. However it is still followed for mixing the cutoff distance. See the *pair_modify* command for details.

The *morse/soft* pair style does not support mixing. Thus, coefficients for all LJ pairs must be specified explicitly.

All of the pair styles with soft core support the *pair_modify* shift option for the energy of the Lennard-Jones portion of the pair interaction.

The different versions of the *lj/cut/soft* pair styles support the *pair_modify* tail option for adding a long-range tail correction to the energy and pressure for the Lennard-Jones portion of the pair interaction.

**Note**

The analytical form of the tail corrections for energy and pressure used in the *lj/cut/soft* potentials are approximate, being identical to that of the corresponding non-soft potentials scaled by a factor $\lambda^n$. The errors due to this approximation should be negligible. For example, for a cutoff of $2.5\sigma$ this approximation leads to maximum relative errors in tail corrections of the order of 1e-4 for energy and virial ($\alpha_{\mathrm{LJ}} = 0.5, n = 2$). The error vanishes when lambda approaches 0 or 1. Note that these are the errors affecting the long-range tail (itself a correction to the interaction energy) which includes other approximations, namely that the system is homogeneous (local density equal the average density) beyond the cutoff.

The *morse/soft* pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

All of these pair styles write information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

---

### 4.122.5 Restrictions

The pair styles with soft core are only enabled if LAMMPS was built with the FEP package. The *long* versions also require the KSPACE package to be installed. The soft *tip4p* versions also require the MOLECULE package to be installed. These styles are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

### 4.122.6 Related commands

*pair_coeff*, *fix adapt*, *fix adapt/fep*, *compute fep*

### 4.122.7 Default

none

---

**(Beutler)** Beutler, Mark, van Schaik, Gerber, van Gunsteren, Chem Phys Lett, 222, 529 (1994).

**(Gapsys)** Gapsys, Seeliger, de Groot, J Chem Theor Comput, 8, 2373 (2012).

## 4.123 pair_style gauss command

Accelerator Variants: *gauss/gpu*, *gauss/kk*, *gauss/omp*

## 4.124 pair_style gauss/cut command

Accelerator Variants: *gauss/cut/kk*, *gauss/cut/omp*

### 4.124.1 Syntax

```
pair_style gauss cutoff
pair_style gauss/cut cutoff
```

- cutoff = global cutoff for Gauss interactions (distance units)

### 4.124.2 Examples

```
pair_style gauss 12.0
pair_coeff * * 1.0 0.9
pair_coeff 1 4 1.0 0.9 10.0

pair_style gauss/cut 3.5
pair_coeff 1 4 0.2805 1.45 0.112
```

### 4.124.3 Description

Style *gauss* computes a tethering potential of the form

$$E = -A \exp(-Br^2) \qquad r < r_c$$

between an atom and its corresponding tether site which will typically be a frozen atom in the simulation. $r_c$ is the cutoff.

The following coefficients must be defined for each pair of atom types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- A (energy units)
- B (1/distance^2 units)
- cutoff (distance units)

The last coefficient is optional. If not specified, the global cutoff is used.

Style *gauss/cut* computes a generalized Gaussian interaction potential between pairs of particles:

$$E = \frac{H}{\sigma_h \sqrt{2\pi}} \exp\left[-\frac{(r - r_{mh})^2}{2\sigma_h^2}\right]$$

where H determines together with the standard deviation $\sigma_h$ the peak height of the Gaussian function, and $r_{mh}$ the peak position. Examples of the use of the Gaussian potentials include implicit solvent simulations of salt ions *(Lenart)* and of surfactants *(Jusufi)*. In these instances the Gaussian potential mimics the hydration barrier between a pair of particles. The hydration barrier is located at $r_{mh}$ and has a width of $\sigma_h$. The prefactor determines the height of the potential barrier.

The following coefficients must be defined for each pair of atom types via the *pair_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- H (energy * distance units)
- $r_{mh}$ (distance units)
- $\sigma_h$ (distance units)
- cutoff (distance units)

The last coefficient is optional. If not specified, the global cutoff is used.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.124.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the A, B, H, sigma_h, r_mh parameters, and the cutoff distance for these pair styles can be mixed:

- A (energy units)
- $\sqrt{\frac{1}{B}}$ (distance units, see below)
- H (energy units)
- $r_{mh}$ (distance units)
- $\sigma_h$ (distance units)
- cutoff (distance units)

The default mix value is *geometric*. Only *arithmetic* and *geometric* mix values are supported. See the "pair_modify" command for details.

The A and H parameters are mixed using the same rules normally used to mix the "epsilon" parameter in a Lennard Jones interaction. The sigma_h, r_mh, and the cutoff distance are mixed using the same rules used to mix the "sigma" parameter in a Lennard Jones interaction. The B parameter is converted to a distance (sigma), before mixing (using sigma=B^-0.5), and converted back to a coefficient afterwards (using B=sigma^2). Negative A values are converted to positive A values (using abs(A)) before mixing, and converted back after mixing (by multiplying by min(sign(Ai),sign(Aj))). This way, if either particle is repulsive (if Ai<0 or Aj<0), then the default interaction between both particles will be repulsive.

For the *gauss* style there is no effect due to the Gaussian well beyond the cutoff; hence reasonable cutoffs need to be specified.

The *gauss/cut* style supports the *pair_modify* shift option for the energy of the Gauss-potential portion of the pair interaction.

The *pair_modify* table and tail options are not relevant for these pair styles.

These pair styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

These pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

The *gauss* pair style tallies an "occupancy" count of how many Gaussian-well sites have an atom within the distance at which the force is a maximum = sqrt(0.5/b). This quantity can be accessed via the *compute pair* command as a vector of values of length 1.

To print this quantity to the log file (with a descriptive column heading) the following commands could be included in an input script:

```
compute gauss all pair gauss
variable occ equal c_gauss[1]
thermo_style custom step temp epair v_occ
```

---

### 4.124.5 Restrictions

The *gauss* and *gauss/cut* styles are part of the EXTRA-PAIR package. They are only enabled if LAMMPS is build with that package. See the *Build package* page for more info.

Changed in version 28Mar2023.

Prior to this version, the *gauss* pair style did not apply *special_bonds* factors.

### 4.124.6 Related commands

*pair_coeff*, *pair_style coul/diel*

### 4.124.7 Default

none

**(Lenart)** Lenart , Jusufi, and Panagiotopoulos, J Chem Phys, 126, 044509 (2007).

**(Jusufi)** Jusufi, Hynninen, and Panagiotopoulos, J Phys Chem B, 112, 13783 (2008).

## 4.125 pair_style gayberne command

Accelerator Variants: *gayberne/gpu*, *gayberne/intel*, *gayberne/omp*

### 4.125.1 Syntax

```
pair_style gayberne gamma upsilon mu cutoff
```

- gamma = shift for potential minimum (typically 1)
- upsilon = exponent for eta orientation-dependent energy function
- mu = exponent for chi orientation-dependent energy function
- cutoff = global cutoff for interactions (distance units)

### 4.125.2 Examples

```
pair_style gayberne 1.0 1.0 1.0 10.0
pair_coeff * * 1.0 1.7 1.7 3.4 3.4 1.0 1.0 1.0
```

### 4.125.3 Description

The *gayberne* styles compute a Gay-Berne anisotropic LJ interaction *(Berardi)* between pairs of ellipsoidal particles or an ellipsoidal and spherical particle via the formulas

$$
\begin{aligned}
U(\mathbf{A}_1, \mathbf{A}_2, \mathbf{r}_{12}) =& U_r(\mathbf{A}_1, \mathbf{A}_2, \mathbf{r}_{12}, \gamma) \cdot \eta_{12}(\mathbf{A}_1, \mathbf{A}_2, \upsilon) \cdot \chi_{12}(\mathbf{A}_1, \mathbf{A}_2, \mathbf{r}_{12}, \mu) \\
U_r =& 4\varepsilon(\rho^{12} - \rho^6) \\
\rho =& \frac{\sigma}{h_{12} + \gamma\sigma}
\end{aligned}
$$

where $\mathbf{A}_1$ and $\mathbf{A}_2$ are the transformation matrices from the simulation box frame to the body frame and $r_{12}$ is the center to center vector between the particles. $U_r$ controls the shifted distance dependent interaction based on the distance of closest approach of the two particles ($h_{12}$) and the user-specified shift parameter $\gamma$. When both particles are spherical, the formula reduces to the usual Lennard-Jones interaction (see details below for when Gay-Berne treats a particle as "spherical").

For large uniform molecules it has been shown that the energy parameters are approximately representable in terms of local contact curvatures *(Everaers)*:

$$
\varepsilon_a = \sigma \cdot \frac{a}{b \cdot c}; \varepsilon_b = \sigma \cdot \frac{b}{a \cdot c}; \varepsilon_c = \sigma \cdot \frac{c}{a \cdot b}
$$

The variable names utilized as potential parameters are for the most part taken from *(Everaers)* in order to be consistent with the *RE-squared pair potential*. Details on the upsilon and mu parameters are given here.

More details of the Gay-Berne formulation are given in the references listed below and in this supplementary document.

Use of this pair style requires the NVE, NVT, or NPT fixes with the *asphere* extension (e.g. *fix nve/asphere*) in order to integrate particle rotation. Additionally, *atom_style ellipsoid* should be used since it defines the rotational state and the size and shape of each ellipsoidal particle.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- $\varepsilon$ = well depth (energy units)
- $\sigma$ = minimum effective particle radii (distance units)
- $\varepsilon_{i,a}$ = relative well depth of type I for side-to-side interactions
- $\varepsilon_{i,b}$ = relative well depth of type I for face-to-face interactions
- $\varepsilon_{i,c}$ = relative well depth of type I for end-to-end interactions
- $\varepsilon_{j,a}$ = relative well depth of type J for side-to-side interactions
- $\varepsilon_{j,b}$ = relative well depth of type J for face-to-face interactions
- $\varepsilon_{j,c}$ = relative well depth of type J for end-to-end interactions
- cutoff (distance units)

The last coefficient is optional. If not specified, the global cutoff specified in the pair_style command is used.

It is typical with the Gay-Berne potential to define $\sigma$ as the minimum of the 3 shape diameters of the particles involved in an I,I interaction, though this is not required. Note that this is a different meaning for $\sigma$ than the *pair_style resquared* potential uses.

The $\varepsilon_i$ and $\varepsilon_j$ coefficients are actually defined for atom types, not for pairs of atom types. Thus, in a series of pair_coeff commands, they only need to be specified once for each atom type.

Specifically, if any of $\varepsilon_{i,a}$, $\varepsilon_{i,b}$, $\varepsilon_{i,c}$ are non-zero, the three values are assigned to atom type I. If all the $\varepsilon_i$ values are zero, they are ignored. If any of $\varepsilon_{j,a}$, $\varepsilon_{j,b}$, $\varepsilon_{j,c}$ are non-zero, the three values are assigned to atom type J. If all three epsilon_j values are zero, they are ignored. Thus the typical way to define the $\varepsilon_i$ and $\varepsilon_j$ coefficients is to list their values in "pair_coeff I J" commands when I = J, but set them to 0.0 when I != J. If you do list them when I != J, you should ensure they are consistent with their values in other pair_coeff commands, since only the last setting will be in effect.

Note that if this potential is being used as a sub-style of *pair_style hybrid*, and there is no "pair_coeff I I" setting made for Gay-Berne for a particular type I (because I-I interactions are computed by another hybrid pair potential), then you still need to ensure the $\varepsilon$ a,b,c coefficients are assigned to that type. e.g. in a "pair_coeff I J" command.

**Note**

If the $\varepsilon_a = \varepsilon_b = \varepsilon_c$ for an atom type, and if the shape of the particle itself is spherical, meaning its 3 shape parameters are all the same, then the particle is treated as an LJ sphere by the Gay-Berne potential. This is significant because if two LJ spheres interact, then the simple Lennard-Jones formula is used to compute their interaction energy/force using the specified epsilon and sigma as the standard LJ parameters. This is much cheaper to compute than the full Gay-Berne formula. To treat the particle as a LJ sphere with sigma = D, you should normally set $\varepsilon_a = \varepsilon_b = \varepsilon_c = 1.0$, set the pair_coeff $\sigma = D$, and also set the 3 shape parameters for the particle to D. The one exception is that if the 3 shape parameters are set to 0.0, which is a valid way in LAMMPS to specify a point particle, then the Gay-Berne potential will treat that as shape parameters of 1.0 (i.e. a LJ particle with $\sigma = 1$), since it requires finite-size particles. In this case you should still set the pair_coeff $\sigma$ to 1.0 as well.

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.125.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon and sigma coefficients and cutoff distance for this pair style can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

This pair style supports the *pair_modify* shift option for the energy of the Lennard-Jones portion of the pair interaction, but only for sphere-sphere interactions. There is no shifting performed for ellipsoidal interactions due to the anisotropic dependence of the interaction.

The *pair_modify* table option is not relevant for this pair style.

This pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.125.5 Restrictions

The *gayberne* style is part of the ASPHERE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

These pair styles require that atoms store torque and a quaternion to represent their orientation, as defined by the *atom_style*. It also require they store a per-type *shape*. The particles cannot store a per-particle diameter.

This pair style requires that atoms be ellipsoids as defined by the *atom_style ellipsoid* command.

Particles acted on by the potential can be finite-size aspherical or spherical particles, or point particles. Spherical particles have all 3 of their shape parameters equal to each other. Point particles have all 3 of their shape parameters equal to 0.0.

The Gay-Berne potential does not become isotropic as r increases *(Everaers)*. The distance-of-closest-approach approximation used by LAMMPS becomes less accurate when high-aspect ratio ellipsoids are used.

### 4.125.6 Related commands

*pair_coeff*, *fix nve/asphere*, *compute temp/asphere*, *pair_style resquared*

### 4.125.7 Default

none

---

**(Everaers)** Everaers and Ejtehadi, Phys Rev E, 67, 041710 (2003).

**(Berardi)** Berardi, Fava, Zannoni, Chem Phys Lett, 297, 8-14 (1998). Berardi, Muccioli, Zannoni, J Chem Phys, 128, 024905 (2008).

**(Perram)** Perram and Rasmussen, Phys Rev E, 54, 6565-6572 (1996).

**(Allen)** Allen and Germano, Mol Phys 104, 3225-3235 (2006).

## 4.126 pair_style gran/hooke command

Accelerator Variants: *gran/hooke/omp*

## 4.127 pair_style gran/hooke/history command

Accelerator Variants: *gran/hooke/history/omp*, *gran/hooke/history/kk*

## 4.128 pair_style gran/hertz/history command

Accelerator Variants: *gran/hertz/history/omp*

### 4.128.1 Syntax

```
pair_style style Kn Kt gamma_n gamma_t xmu dampflag keyword
```

- style = *gran/hooke* or *gran/hooke/history* or *gran/hertz/history*
- Kn = elastic constant for normal particle repulsion (force/distance units or pressure units - see discussion below)
- Kt = elastic constant for tangential contact (force/distance units or pressure units - see discussion below)
- gamma_n = damping coefficient for collisions in normal direction (1/time units or 1/time-distance units - see discussion below)
- gamma_t = damping coefficient for collisions in tangential direction (1/time units or 1/time-distance units - see discussion below)
- xmu = static yield criterion (unitless value between 0.0 and 1.0e4)
- dampflag = 0 or 1 if tangential damping force is excluded or included
- keyword = *limit_damping*

  *limit_damping* value = none
     limit damping to prevent attractive interaction

**Note**

Versions of LAMMPS before 9Jan09 had different style names for granular force fields. This is to emphasize the fact that the Hertzian equation has changed to model polydispersity more accurately. A side effect of the change is that the Kn, Kt, gamma_n, and gamma_t coefficients in the pair_style command must be specified with different

values in order to reproduce calculations made with earlier versions of LAMMPS, even for monodisperse systems. See the NOTE below for details.

### 4.128.2 Examples

```
pair_style gran/hooke/history 200000.0 NULL 50.0 NULL 0.5 1
pair_style gran/hooke 200000.0 70000.0 50.0 30.0 0.5 0
pair_style gran/hooke 200000.0 70000.0 50.0 30.0 0.5 0 limit_damping
```

### 4.128.3 Description

The *gran* styles use the following formulas for the frictional force between two granular particles, as described in *(Brilliantov)*, *(Silbert)*, and *(Zhang)*, when the distance r between two particles of radii Ri and Rj is less than their contact distance d = Ri + Rj. There is no force between the particles when r > d.

The two Hookean styles use this formula:

$$F_{hk} = (k_n \delta \mathbf{n}_{ij} - m_{eff} \gamma_n \mathbf{v}_n) - (k_t \Delta \mathbf{s}_t + m_{eff} \gamma_t \mathbf{v}_t)$$

The Hertzian style uses this formula:

$$F_{hz} = \sqrt{\delta} \sqrt{\frac{R_i R_j}{R_i + R_j}} F_{hk} = \sqrt{\delta} \sqrt{\frac{R_i R_j}{R_i + R_j}} \left[ (k_n \delta \mathbf{n}_{ij} - m_{eff}\ \gamma_n \mathbf{v}_n) - (k_t \Delta \mathbf{s}_t + m_{eff}\ \gamma_t \mathbf{v}_t) \right]$$

In both equations the first parenthesized term is the normal force between the two particles and the second parenthesized term is the tangential force. The normal force has 2 terms, a contact force and a damping force. The tangential force also has 2 terms: a shear force and a damping force. The shear force is a "history" effect that accounts for the tangential displacement between the particles for the duration of the time they are in contact. This term is included in pair styles *hooke/history* and *hertz/history*, but is not included in pair style *hooke*. The tangential damping force term is included in all three pair styles if *dampflag* is set to 1; it is not included if *dampflag* is set to 0.

The other quantities in the equations are as follows:

- $\delta$ = d - r = overlap distance of 2 particles
- $K_n$ = elastic constant for normal contact
- $K_t$ = elastic constant for tangential contact
- $\gamma_n$ = viscoelastic damping constant for normal contact
- $\gamma_t$ = viscoelastic damping constant for tangential contact
- $m_{eff} = M_i M_j / (M_i + M_j)$ = effective mass of 2 particles of mass M_i and M_j
- $\Delta \mathbf{s}_t$ = tangential displacement vector between 2 particles which is truncated to satisfy a frictional yield criterion
- $n_{ij}$ = unit vector along the line connecting the centers of the 2 particles
- $V_n$ = normal component of the relative velocity of the 2 particles
- $V_t$ = tangential component of the relative velocity of the 2 particles

The $K_n$, $K_t$, $\gamma_n$, and $\gamma_t$ coefficients are specified as parameters to the pair_style command. If a NULL is used for $K_t$, then a default value is used where $K_t = 2/7 K_n$. If a NULL is used for $\gamma_t$, then a default value is used where $\gamma_t = 1/2 \gamma_n$.

The interpretation and units for these 4 coefficients are different in the Hookean versus Hertzian equations.

The Hookean model is one where the normal push-back force for two overlapping particles is a linear function of the overlap distance. Thus the specified $K_n$ is in units of (force/distance). Note that this push-back force is independent of

absolute particle size (in the monodisperse case) and of the relative sizes of the two particles (in the polydisperse case). This model also applies to the other terms in the force equation so that the specified $\gamma_n$ is in units of (1/time), $K_t$ is in units of (force/distance), and $\gamma_t$ is in units of (1/time).

The Hertzian model is one where the normal push-back force for two overlapping particles is proportional to the area of overlap of the two particles, and is thus a non-linear function of overlap distance. Thus Kn has units of force per area and is thus specified in units of (pressure). The effects of absolute particle size (monodispersity) and relative size (polydispersity) are captured in the radii-dependent prefactors. When these prefactors are carried through to the other terms in the force equation it means that the specified $\gamma_n$ is in units of (1/(time*distance)), $K_t$ is in units of (pressure), and $\gamma_t$ is in units of (1/(time*distance)).

Note that in the Hookean case, $K_n$ can be thought of as a linear spring constant with units of force/distance. In the Hertzian case, $K_n$ is like a non-linear spring constant with units of force/area or pressure, and as shown in the *(Zhang)* paper, $K_n = 4G/(3(1-\nu))$ where $\nu$ = the Poisson ratio, G = shear modulus = $E/(2(1+\nu))$, and E = Young's modulus. Similarly, $K_t = 4G/(2-\nu)$. (NOTE: in an earlier version of the manual, we incorrectly stated that $K_t = 8G/(2-\nu)$.)

Thus in the Hertzian case $K_n$ and $K_t$ can be set to values that corresponds to properties of the material being modeled. This is also true in the Hookean case, except that a spring constant must be chosen that is appropriate for the absolute size of particles in the model. Since relative particle sizes are not accounted for, the Hookean styles may not be a suitable model for polydisperse systems.

**Note**

In versions of LAMMPS before 9Jan09, the equation for Hertzian interactions did not include the $\sqrt{r_i r_j/(r_i+r_j)}$ term and thus was not as accurate for polydisperse systems. For monodisperse systems, $\sqrt{r_i r_j/(r_i+r_j)}$ is a constant factor that effectively scales all 4 coefficients: $K_n, K_t, \gamma_n, \gamma_t$. Thus you can set the values of these 4 coefficients appropriately in the current code to reproduce the results of a previous Hertzian monodisperse calculation. For example, for the common case of a monodisperse system with particles of diameter 1, all 4 of these coefficients should now be set 2x larger than they were previously.

Xmu is also specified in the pair_style command and is the upper limit of the tangential force through the Coulomb criterion Ft = xmu*Fn, where Ft and Fn are the total tangential and normal force components in the formulas above. Thus in the Hookean case, the tangential force between 2 particles grows according to a tangential spring and dash-pot model until Ft/Fn = xmu and is then held at Ft = Fn*xmu until the particles lose contact. In the Hertzian case, a similar analogy holds, though the spring is no longer linear.

**Note**

Normally, xmu should be specified as a fractional value between 0.0 and 1.0, however LAMMPS allows large values (up to 1.0e4) to allow for modeling of systems which can sustain very large tangential forces.

The effective mass *m_eff* is given by the formula above for two isolated particles. If either particle is part of a rigid body, its mass is replaced by the mass of the rigid body in the formula above. This is determined by searching for a *fix rigid* command (or its variants).

For granular styles there are no additional coefficients to set for each pair of atom types via the *pair_coeff* command. All settings are global and are made via the pair_style command. However you must still use the *pair_coeff* for all pairs of granular atom types. For example the command

```
pair_coeff * *
```

should be used if all atoms in the simulation interact via a granular potential (i.e. one of the pair styles above is used). If a granular potential is used as a sub-style of *pair_style hybrid*, then specific atom types can be used in the pair_coeff command to determine which atoms interact via a granular potential.

If two particles are moving away from each other while in contact, there is a possibility that the particles could experience an effective attractive force due to damping. If the *limit_damping* keyword is used, this option will zero out the normal component of the force if there is an effective attractive force.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.128.4 Mixing, shift, table, tail correction, restart, rRESPA info

The *pair_modify* mix, shift, table, and tail options are not relevant for granular pair styles.

These pair styles write their information to *binary restart files*, so a pair_style command does not need to be specified in an input script that reads a restart file.

These pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

The single() function of these pair styles returns 0.0 for the energy of a pairwise interaction, since energy is not conserved in these dissipative potentials. It also returns only the normal component of the pairwise interaction force. However, the single() function also calculates 10 extra pairwise quantities. The first 3 are the components of the tangential force between particles I and J, acting on particle I. The fourth is the magnitude of this tangential force. The next 3 (5-7) are the components of the relative velocity in the normal direction (along the line joining the 2 sphere centers). The last 3 (8-10) the components of the relative velocity in the tangential direction.

These extra quantities can be accessed by the *compute pair/local* command, as $p1$, $p2$, ..., $p10$.

---

### 4.128.5 Restrictions

All the granular pair styles are part of the GRANULAR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

These pair styles require that atoms store torque and angular velocity (omega) as defined by the *atom_style*. They also require a per-particle radius is stored. The *sphere* atom style does all of this.

This pair style requires you to use the *comm_modify vel yes* command so that velocities are stored by ghost atoms.

These pair styles will not restart exactly when using the *read_restart* command, though they should provide statistically similar results. This is because the forces they compute depend on atom velocities. See the *read_restart* command for more details.

Accumulated values for individual contacts are saved to to restart files but are not saved to data files. Therefore, forces may differ significantly when a system is reloaded using A *read_data* command.

### 4.128.6 Related commands

*pair_coeff*

### 4.128.7 Default

none

---

**(Brilliantov)** Brilliantov, Spahn, Hertzsch, Poschel, Phys Rev E, 53, p 5382-5392 (1996).

**(Silbert)** Silbert, Ertas, Grest, Halsey, Levine, Plimpton, Phys Rev E, 64, p 051302 (2001).

**(Zhang)** Zhang and Makse, Phys Rev E, 72, p 011301 (2005).

## 4.129 pair_style granular command

### 4.129.1 Syntax

```
pair_style granular cutoff
```

- cutoff = global cutoff (optional). See discussion below.

### 4.129.2 Examples

```
pair_style granular
pair_coeff * * hooke 1000.0 50.0 tangential linear_nohistory 1.0 0.4 damping mass_
↪velocity

pair_style granular
pair_coeff * * hooke 1000.0 50.0 tangential linear_history 500.0 1.0 0.4 damping mass_
↪velocity

pair_style granular
pair_coeff * * hertz 1000.0 50.0 tangential mindlin 1000.0 1.0 0.4 limit_damping

pair_style granular
pair_coeff * * hertz/material 1e8 0.3 0.3 tangential mindlin_rescale NULL 1.0 0.4␣
↪damping tsuji

pair_style granular
pair_coeff * * hertz/material 1e8 0.3 0.3 tangential mindlin_rescale NULL 1.0 0.4␣
↪damping coeff_restitution synchronized_verlet

pair_style granular
pair_coeff 1 * jkr 1000.0 500.0 0.3 10 tangential mindlin 800.0 1.0 0.5 rolling sds 500.
↪0 200.0 0.5 twisting marshall
pair_coeff 2 2 hertz 200.0 100.0 tangential linear_history 300.0 1.0 0.1 rolling sds 200.
↪0 100.0 0.1 twisting marshall

pair_style granular
pair_coeff 1 1 dmt 1000.0 50.0 0.3 0.0 tangential mindlin NULL 0.5 0.5 rolling sds 500.0␣
↪200.0 0.5 twisting marshall
pair_coeff 2 2 dmt 1000.0 50.0 0.3 10.0 tangential mindlin NULL 0.5 0.1 rolling sds 500.
```

(continues on next page)

(continued from previous page)

```
↪0 200.0 0.1 twisting marshall

pair_style granular
pair_coeff * * hertz 1000.0 50.0 tangential mindlin 1000.0 1.0 0.4 heat area 0.1

pair_style granular
pair_coeff * * mdr 5e6 0.4 1.9e5 2.0 0.5 0.5 tangential linear_history 940.0 1.0 0.7␣
↪rolling sds 2.7e5 0.0 0.6 damping mdr 1
```

### 4.129.3 Description

The *granular* styles support a variety of options for the normal, tangential, rolling and twisting forces resulting from contact between two granular particles. This expands on the options offered by the *pair gran/** pair styles. The total computed forces and torques are the sum of various models selected for the normal, tangential, rolling and twisting modes of motion.

All model choices and parameters are entered in the *pair_coeff* command, as described below. Unlike e.g. *pair gran/hooke*, coefficient values are not global, but can be set to different values for different combinations of particle types, as determined by the *pair_coeff* command. If the contact model choice is the same for two particle types, the mixing for the cross-coefficients can be carried out automatically. This is shown in one of the examples, where model choices are the same for type 1 - type 1 as for type 2 - type2 interactions, but coefficients are different. In this case, the mixed coefficients for type 1 - type 2 interactions can be determined from mixing rules discussed below. For additional flexibility, coefficients as well as model forms can vary between particle types, as shown in the fourth example: type 1 - type 1 interactions are based on a Johnson-Kendall-Roberts normal contact model and 2-2 interactions are based on a DMT cohesive model (see below). In that example, 1-1 and 2-2 interactions have different model forms, in which case mixing of coefficients cannot be determined, so 1-2 interactions must be explicitly defined via the *pair_coeff 1 ** command, otherwise an error would result.

---

The first required keyword for the *pair_coeff* command is the normal contact model. Currently supported options for normal contact models and their required arguments are:

1. *hooke* : $k_n$, $\eta_{n0}$ (or $e$)
2. *hertz* : $k_n$, $\eta_{n0}$ (or $e$)
3. *hertz/material* : E, $\eta_{n0}$ (or $e$), $\nu$
4. *dmt* : E, $\eta_{n0}$ (or $e$), $\nu$, $\gamma$
5. *jkr* : E, $\eta_{n0}$ (or $e$), $\nu$, $\gamma$
6. *mdr* : $E$, $\nu$, $Y$, $\Delta\gamma$, $\psi_b$, $\eta_{n0}$

Here, $k_n$ is spring stiffness (with units that depend on model choice, see below); $\eta_{n0}$ is a damping prefactor (or, in its place a coefficient of restitution $e$, depending on the choice of damping mode, see below); E is Young's modulus in units of *force/length*^2, i.e. *pressure*; $\nu$ is Poisson's ratio and $\gamma$ is a surface energy density, in units of *energy/length*^2.

For the *hooke* model, the normal, elastic component of force acting on particle $i$ due to contact with particle $j$ is given by:

$$\mathbf{F}_{ne,Hooke} = k_n \delta_{ij} \mathbf{n}$$

Where $\delta_{ij} = R_i + R_j - \|\mathbf{r}_{ij}\|$ is the particle overlap, $R_i, R_j$ are the particle radii, $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ is the vector separating the two particle centers (note the i-j ordering so that $\mathbf{F}_{ne}$ is positive for repulsion), and $\mathbf{n} = \frac{\mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|}$. Therefore, for *hooke*, the units of the spring constant $k_n$ are *force/distance*, or equivalently *mass/time*^2.

For the *hertz* model, the normal component of force is given by:

$$\mathbf{F}_{ne,Hertz} = k_n R_{eff}^{1/2} \delta_{ij}^{3/2} \mathbf{n}$$

Here, $R_{eff} = R = \frac{R_i R_j}{R_i + R_j}$ is the effective radius, denoted for simplicity as $R$ from here on. For *hertz*, the units of the spring constant $k_n$ are *force/length*^2, or equivalently *pressure*.

For the *hertz/material* model, the force is given by:

$$\mathbf{F}_{ne,Hertz/material} = \frac{4}{3} E_{eff} R^{1/2} \delta_{ij}^{3/2} \mathbf{n}$$

Here, $E_{eff} = E = \left( \frac{1-\nu_i^2}{E_i} + \frac{1-\nu_j^2}{E_j} \right)^{-1}$ is the effective Young's modulus, with $\nu_i, \nu_j$ the Poisson ratios of the particles of types $i$ and $j$. $E_{eff}$ is denoted as $E$ from here on. Note that if the elastic modulus and the shear modulus of the two particles are the same, the *hertz/material* model is equivalent to the *hertz* model with $k_n = 4/3E$

The *dmt* model corresponds to the *(Derjaguin-Muller-Toporov)* cohesive model, where the force is simply Hertz with an additional attractive cohesion term:

$$\mathbf{F}_{ne,dmt} = \left( \frac{4}{3} E R^{1/2} \delta_{ij}^{3/2} - 4\pi\gamma R \right) \mathbf{n}$$

The *jkr* model is the *(Johnson-Kendall-Roberts)* model, where the force is computed as:

$$\mathbf{F}_{ne,jkr} = \left( \frac{4Ea^3}{3R} - 2\pi a^2 \sqrt{\frac{4\gamma E}{\pi a}} \right) \mathbf{n}$$

Here, $a$ is the radius of the contact zone, related to the overlap $\delta$ according to:

$$\delta = a^2/R - 2\sqrt{\pi\gamma a/E}$$

LAMMPS internally inverts the equation above to solve for $a$ in terms of $\delta$, then solves for the force in the previous equation. Additionally, note that the JKR model allows for a tensile force beyond contact (i.e. for $\delta < 0$), up to a maximum of $3\pi\gamma R$ (also known as the 'pull-off' force). Note that this is a hysteretic effect, where particles that are not contacting initially will not experience force until they come into contact $\delta \geq 0$; as they move apart and ($\delta < 0$), they experience a tensile force up to $3\pi\gamma R$, at which point they lose contact.

**Note**

Typically, neighbor lists are constructed for pair granular by testing whether finite sized particles overlap (using their radii). However, this is not the case for normal models which can interact beyond contact, e.g. *jkr*. Instead, the maximum radius for each particle type is first calculated then used to calculate a maximum per-type cutoff distance. For polydisperse systems, this affects the performance of the *multi neighbor* option where one should assign atoms of similar radii the same type. See the *pair lj/cut/sphere* page for a related discussion.

The *mdr* model is a mechanically-derived contact model designed to capture the contact response between adhesive elastic-plastic particles into large deformation. The theoretical foundations of the *mdr* model are detailed in the two-part series *Zunker and Kamrin Part I* and *Zunker and Kamrin Part II*. Further development and demonstrations of its application to industrially relevant powder compaction processes are presented in *Zunker et al.*. If you use the *mdr* normal model the only supported damping option is the *mdr* damping class described below.

The model requires the following inputs:

1. *Young's modulus* $E > 0$ : The Young's modulus is commonly reported for various powders.
2. *Poisson's ratio* $0 \leq \nu \leq 0.5$ : The Poisson's ratio is commonly reported for various powders.

3. *Yield stress* $Y \geq 0$ : The yield stress is often known for powders composed of materials such as metals but may be unreported for ductile organic materials, in which case it can be treated as a free parameter.

4. *Effective surface energy* $\Delta\gamma \geq 0$ : The effective surface energy for powder compaction applications is most easily determined through its relation to the more commonly reported critical stress intensity factor $K_{Ic} = \sqrt{2\Delta\gamma E/(1-\nu^2)}$.

5. *Critical confinement ratio* $0 \leq \psi_b \leq 1$ : The critical confinement ratio is a tunable parameter that determines when the bulk elastic response is triggered. Lower values of $\psi_b$ delay the onset of the bulk elastic response.

6. *Damping coefficent* $\eta_{n0} \geq 0$ : The damping coefficient is a tunable parameter that controls damping in the normal direction.

**Note**

The values for $E$, $\nu$, $Y$, and $\Delta\gamma$ (i.e., $K_{Ic}$) should be selected for zero porosity to reflect the intrinsic material property rather than the bulk powder property.

The *mdr* model produces a nonlinear force-displacement response, therefore the critical timestep $\Delta t$ depends on the inputs and level of deformation. As a conservative starting point the timestep can be assumed to be dictated by the bulk elastic response such that $\Delta t = 0.08\sqrt{m/k_{\text{bulk}}}$, where $m$ is the mass of the smallest particle and $k_{\text{bulk}} = \kappa R_{\text{min}}$ is an effective stiffness related to the bulk elastic response. Here, $\kappa = E/(3(1-2\nu))$ is the bulk modulus and $R_{\text{min}}$ is the radius of the smallest particle.

The *atom_style* must be set to *sphere 1* to enable dynamic particle radii. The *mdr* model is designed to respect the incompressibility of plastic deformation and inherently tracks free surface displacements induced by all particle contacts. In practice, this means that all particles begin with an initial radius, however as compaction occurs and plastic deformation is accumulated, a new enlarged apparent radius is defined to ensure that that volume change due to plastic deformation is not lost. This apparent radius is stored as the *atom radius* meaning it is used for subsequent neighbor list builds and contact detection checks. The advantage of this is that multi-neighbor dependent effects such as formation of secondary contacts caused by radial expansion are captured by the *mdr* model. Setting *atom_style sphere 1* ensures that updates to the particle radii are properly reflected throughout the simulation.

```
atom_style sphere 1
```

Newton's third law must be set to *off*. This ensures that the neighbor lists are constructed properly for the topological penalty algorithm used to screen for non-physical contacts occurring through obstructing particles, an issue prevalent under large deformation conditions. For more information on this algorithm see *Zunker et al.*.

```
newton off
```

The definition of multiple *mdr* models in the *pair_style* is currently not supported. Similarly, the *mdr* model cannot be combined with a different normal model in the *pair_style*. Physically this means that only one homogeneous collection of particles governed by a single *mdr* model is allowed.

The *mdr* model currently only supports *fix wall/gran/region*, not *fix wall/gran*. If the *mdr* model is specified for the *pair_style* any *fix wall/gran/region* commands must also use the *mdr* model. Additionally, the following *mdr* inputs must match between the *pair_style* and *fix wall/gran/region* definitions: $E$, $\nu$, $Y$, $\psi_b$, and $\eta_{n0}$. The exception is $\Delta\gamma$, which may vary, permitting different adhesive behaviors between particle-particle and particle-wall interactions.

**Note**

The *mdr* model has a number of custom *property/atom* and *pair/local* definitions that can be called in the input file. The useful properties for visualization and analysis are described below.

In addition to contact forces the *mdr* model also tracks the following quantities for each particle: elastic volume change, average normal stress components, total surface area involved in contact, and individual contact areas. In the input script, these quantities are initialized by calling *run 0* and can then be accessed using subsequent *compute* commands. The last *compute* command uses *pair/local p13* to calculate the pairwise contact areas for each active contact in the *group-ID*. Due to the use of an apparent radius in the *mdr* model, the keyword/arg pair *cutoff radius* must be specified for *pair/local* to properly detect existing contacts.

```
run 0
compute ID group-ID property/atom d_Velas
compute ID group-ID property/atom d_sigmaxx
compute ID group-ID property/atom d_sigmayy
compute ID group-ID property/atom d_sigmazz
compute ID group-ID property/atom d_Acon1
compute ID group-ID pair/local p13 cutoff radius
```

**Note**

The *mdr* model has two example input scripts within the *examples/granular* directory. The first is a die compaction simulation involving 200 particles named *in.tableting.200*. The second is a triaxial compaction simulation involving 12 particles named *in.triaxial.compaction.12*.

---

In addition, the normal force is augmented by a damping term of the following general form:

$$\mathbf{F}_{n,damp} = -\eta_n \mathbf{v}_{n,rel}$$

Here, $\mathbf{v}_{n,rel} = (\mathbf{v}_j - \mathbf{v}_i) \cdot \mathbf{n}\,\mathbf{n}$ is the component of relative velocity along $\mathbf{n}$.

The optional *damping* keyword to the *pair_coeff* command followed by a keyword determines the model form of the damping factor $\eta_n$, and the interpretation of the $\eta_{n0}$ or $e$ coefficients specified as part of the normal contact model settings. The *damping* keyword and corresponding model form selection may be appended anywhere in the *pair coeff* command. Note that the choice of damping model affects both the normal and tangential damping (and depending on other settings, potentially also the twisting damping). The options for the damping model currently supported are:

1. *velocity*
2. *mass_velocity*
3. *viscoelastic*
4. *tsuji*
5. *coeff_restitution*
6. *mdr* (class) : $d_{type}$

If the *damping* keyword is not specified, the *viscoelastic* model is used by default.

For *damping velocity*, the normal damping is simply equal to the user-specified damping coefficient in the *normal* model:

$$\eta_n = \eta_{n0}$$

Here, $\eta_{n0}$ is the damping coefficient specified for the normal contact model, in units of *mass*/*time*.

For *damping mass_velocity*, the normal damping is given by:

$$\eta_n = \eta_{n0} m_{eff}$$

Here, $\eta_{n0}$ is the damping coefficient specified for the normal contact model, in units of 1/*time* and $m_{eff} = m_i m_j/(m_i+m_j)$ is the effective mass. Use *damping mass_velocity* to reproduce the damping behavior of *pair gran/hooke/**.

The *damping viscoelastic* model is based on the viscoelastic treatment of *(Brilliantov et al)*, where the normal damping is given by:

$$\eta_n = \eta_{n0}\, a m_{eff}$$

Here, $a$ is the contact radius, given by $a = \sqrt{R\delta}$ for all models except *jkr*, for which it is given implicitly according to $\delta = a^2/R - 2\sqrt{\pi\gamma a/E}$. For *damping viscoelastic*, $\eta_{n0}$ is in units of 1/(*time***distance*).

The *tsuji* model is based on the work of *(Tsuji et al)*. Here, the damping coefficient specified as part of the normal model is interpreted as a restitution coefficient $e$. The damping constant $\eta_n$ is given by:

$$\eta_n = \alpha (m_{eff} k_{nd})^{1/2}$$

where $k_{nd}$ is an effective harmonic stiffness equal to the ratio of the normal force to the overlap. For example, $k_{nd} = 4/3Ea$ for a Hertz contact model based on material parameters with $a$ being the contact radius of $\sqrt{\delta R}$. For Hooke, $k_{nd}$ is simply the spring constant or $k_n$. This damping model is not compatible with cohesive normal models such as *JKR* or *DMT*. The parameter $\alpha$ is related to the restitution coefficient $e$ according to:

$$\alpha = 1.2728 - 4.2783e + 11.087e^2 - 22.348e^3 + 27.467e^4 - 18.022e^5 + 4.8218e^6$$

The dimensionless coefficient of restitution $e$ specified as part of the normal contact model parameters should be between 0 and 1, but no error check is performed on this.

The *coeff_restitution* model is useful when a specific normal coefficient of restitution $e$ is required. It operates much like the *Tsuji* model but, the normal coefficient of restitution $e$ is specified as an input in place of the usual $\eta_{n0}$ value in the normal model. Following the approach of *(Brilliantov et al)*, when using the *hooke* normal model, *coeff_restitution* then calculates the damping coefficient as:

$$\eta_n = \sqrt{\frac{4 m_{eff} k_{nd}}{1 + \left(\frac{\pi}{\log(e)}\right)^2}},$$

where $k_{nd}$ is the same stiffness defined in the above *Tsuji* model. For any other normal model, e.g. the *hertz* and *hertz/material* models, the damping coefficient is:

$$\eta_n = -2\sqrt{\frac{5}{6}}\frac{\log(e)}{\sqrt{\pi^2 + (\log(e))^2}}\sqrt{\frac{3}{2} k_{nd} m_{eff}},$$

Since *coeff_restitution* accounts for the effective mass, effective radius, and pairwise overlaps (except when used with the *hooke* normal model) when calculating the damping coefficient, it accurately reproduces the specified coefficient of restitution for both monodisperse and polydisperse particle pairs. This damping model is not compatible with cohesive normal models such as *JKR* or *DMT*.

The *mdr* damping class contains multiple damping models that can be toggled between by specifying different integer values for the $d_{type}$ input parameter. This damping option is only compatible with the normal *mdr* contact model.

Setting $d_{type} = 1$ is the suggested damping option. This specifies a damping model that takes into account the contact stiffness $k_{mdr}$ calculated by the normal *mdr* contact model to determine the damping coefficient:

$$\eta_n = \eta_{n0}(m_{eff} k_{mdr})^{1/2},$$

where $k_{mdr}$ is proportional to contact radius $a_{mdr}$ tracked by the normal *mdr* contact model:

$$k_{mdr} = 2E_{eff} a_{mdr}.$$

In this case, $\eta_{n0}$ is simply a dimensionless coefficient that scales the the overall damping coefficient.

The other supported option is $d_{type} = 2$, which defines a simple damping model similar to the *velocity* option

$$\eta_n = \eta_{n0},$$

but has additional checks to avoid non-physical damping after plastic deformation.

The total normal force is computed as the sum of the elastic and damping components:

$$\mathbf{F}_n = \mathbf{F}_{ne} + \mathbf{F}_{n,damp}$$

---

The *pair_coeff* command also requires specification of the tangential contact model. The required keyword *tangential* is expected, followed by the model choice and associated parameters. Currently supported tangential model choices and their expected parameters are as follows:

1. *linear_nohistory* : $x_{\gamma,t}$, $\mu_s$
2. *linear_history* : $k_t$, $x_{\gamma,t}$, $\mu_s$
3. *mindlin* : $k_t$ or NULL, $x_{\gamma,t}$, $\mu_s$
4. *mindlin/force* : $k_t$ or NULL, $x_{\gamma,t}$, $\mu_s$
5. *mindlin_rescale* : $k_t$ or NULL, $x_{\gamma,t}$, $\mu_s$
6. *mindlin_rescale/force* : $k_t$ or NULL, $x_{\gamma,t}$, $\mu_s$

Here, $x_{\gamma,t}$ is a dimensionless multiplier for the normal damping $\eta_n$ that determines the magnitude of the tangential damping, $\mu_t$ is the tangential (or sliding) friction coefficient, and $k_t$ is the tangential stiffness coefficient.

The general form of the tangential contact force for all models is:

$$\mathbf{F}_t = \min(\mu_t F_{n0}, \|\mathbf{F}_{te} + \mathbf{F}_{\text{t,damp}}\|)\mathbf{t}$$

where $\mathbf{F}_{te}$ and $\mathbf{F}_{\text{t,damp}}$ are the elastic and damping components of the tangential force, respectively, and $\mathbf{t}$ is the direction of the tangential force given by:

$$\mathbf{t} = \frac{\mathbf{F}_{te} + \mathbf{F}_{\text{t,damp}}}{\|\mathbf{F}_{te} + \mathbf{F}_{\text{t,damp}}\|}$$

The normal force value $F_{n0}$ used to compute the critical force depends on the form of the contact model. For non-cohesive models (*hertz*, *hertz/material*, *hooke*), it is given by the magnitude of the normal force:

$$F_{n0} = \|\mathbf{F}_n\|$$

For cohesive models such as *jkr* and *dmt*, the critical force is adjusted so that the critical tangential force approaches $\mu_t F_{pulloff}$, see *Marshall*, equation 43, and *Thornton*. For both models, $F_{n0}$ takes the form:

$$F_{n0} = \|\mathbf{F}_{ne} + 2F_{pulloff}\|$$

Where $F_{pulloff} = 3\pi\gamma R$ for *jkr*, and $F_{pulloff} = 4\pi\gamma R$ for *dmt*.

The tangential damping force is the same for all models and is given by:

$$\mathbf{F}_{\text{t,damp}} = -\eta_t \mathbf{v}_{t,rel}$$

The tangential damping prefactor $\eta_t$ is calculated by scaling the normal damping $\eta_n$ (see above):

$$\eta_t = -x_{\gamma,t}\eta_n$$

The normal damping prefactor $\eta_n$ is determined by the choice of the *damping* keyword, as discussed above. Thus, the *damping* keyword also affects the tangential damping. The parameter $x_{\gamma,t}$ is a scaling coefficient. Several works in the literature use $x_{\gamma,t} = 1$ (*Marshall*, *Tsuji et al*, *Silbert et al*). The relative tangential velocity at the point of contact is given by $\mathbf{v}_{t,rel} = \mathbf{v}_t - (R_i\Omega_i + R_j\Omega_j) \times \mathbf{n}$, where $\mathbf{v}_t = \mathbf{v}_r - \mathbf{v}_r \cdot \mathbf{n}\,\mathbf{n}$, $\mathbf{v}_r = \mathbf{v}_j - \mathbf{v}_i$ .

The elastic tangential force $\mathbf{F}_{te}$ depends on the tangential model and is detailed hereafter for each tangential model.

For *tangential linear_nohistory*, no elastic force is considered:

$$\mathbf{F}_{te} = \mathbf{0}$$

This is used to mimic the behavior of the *pair gran/hooke* style.

The remaining tangential options all use accumulated tangential displacement (i.e. contact history), except for the options *mindlin/force* and *mindlin_rescale/force*, that use accumulated tangential force instead, and are discussed further below. The accumulated tangential displacement is discussed in details below in the context of the *linear_history* option. The same treatment of the accumulated displacement applies to the other options as well.

For *tangential linear_history*, the elastic tangential force is given by:

$$\mathbf{F}_{te} = -k_t\xi$$

Here, $\xi$ is the tangential displacement accumulated during the entire duration of the contact:

$$\xi = \int_{t_0}^{t} \mathbf{v}_{t,rel}(\tau)\mathrm{d}\tau$$

This accumulated tangential displacement must be adjusted to account for changes in the frame of reference of the contacting pair of particles during contact. This occurs due to the overall motion of the contacting particles in a rigid-body-like fashion during the duration of the contact. There are two modes of motion that are relevant: the 'tumbling' rotation of the contacting pair, which changes the orientation of the plane in which tangential displacement occurs; and 'spinning' rotation of the contacting pair about the vector connecting their centers of mass ($\mathbf{n}$). Corrections due to the former mode of motion are made by rotating the accumulated displacement into the plane that is tangential to the contact vector at each step, or equivalently removing any component of the tangential displacement that lies along $\mathbf{n}$, and rescaling to preserve the magnitude. This follows the discussion in *Luding*, see equation 17 and relevant discussion in that work:

$$\xi = \left(\xi' - (\mathbf{n}\cdot\xi')\mathbf{n}\right)\frac{\|\xi'\|}{\|\xi' - (\mathbf{n}\cdot\xi')\mathbf{n}\|}$$

Here, $\xi'$ is the accumulated displacement prior to the current time step and $\xi$ is the corrected displacement. Corrections to the displacement due to the second mode of motion described above (rotations about $\mathbf{n}$) are not currently implemented, but are expected to be minor for most simulations.

Furthermore, when the tangential force exceeds the critical force, the tangential displacement is re-scaled to match the value for the critical force (see *Luding*, equation 20 and related discussion):

$$\xi = -\frac{1}{k_t}\left(\mu_t F_{n0}\mathbf{t} - \mathbf{F}_{t,damp}\right)$$

The tangential force is added to the total normal force (elastic plus damping) to produce the total force on the particle. The tangential force also acts at the contact point (defined as the center of the overlap region) to induce a torque on each particle according to:

$$\tau_i = -(R_i - 0.5\delta)\mathbf{n} \times \mathbf{F}_t$$

$$\tau_j = -(R_j - 0.5\delta)\mathbf{n} \times \mathbf{F}_t$$

For *tangential mindlin*, the *Mindlin* no-slip solution is used which differs from the *linear_history* option by an additional factor of $a$, the radius of the contact region. The elastic tangential force is given by:

$$\mathbf{F}_{te} = -k_t a \boldsymbol{\xi}$$

Here, $a$ is the radius of the contact region, given by $a = \sqrt{R\delta}$ for all normal contact models, except for *jkr*, where it is given implicitly by $\delta = a^2/R - 2\sqrt{\pi\gamma a/E}$, see discussion above. To match the Mindlin solution, one should set $k_t = 8G_{eff}$, where $G_{eff}$ is the effective shear modulus given by:

$$G_{eff} = \left( \frac{2-\nu_i}{G_i} + \frac{2-\nu_j}{G_j} \right)^{-1}$$

where $G_i$ is the shear modulus of a particle of type $i$, related to Young's modulus $E_i$ and Poisson's ratio $\nu_i$ by $G_i = E_i/(2(1+\nu_i))$. This can also be achieved by specifying *NULL* for $k_t$, in which case a normal contact model that specifies material parameters $E_i$ and $\nu_i$ is required (e.g. *hertz/material*, *dmt* or *jkr*). In this case, mixing of the shear modulus for different particle types $i$ and $j$ is done according to the formula above.

**Note**

The radius of the contact region $a$ depends on the normal overlap. As a result, the tangential force for *mindlin* can change due to a variation in normal overlap, even with no change in tangential displacement.

For *tangential mindlin/force*, the accumulated elastic tangential force characterizes the contact history, instead of the accumulated tangential displacement. This prevents the dependence of the tangential force on the normal overlap as noted above. The elastic tangential force is given by:

$$\mathbf{F}_{te} = \int_{t_0}^{t} -k_t a \mathbf{v}_{t,rel} \mathrm{d}\tau$$

The changes in frame of reference of the contacting pair of particles during contact are accounted for by the same formula as above, replacing the accumulated tangential displacement $\boldsymbol{\xi}$, by the accumulated tangential elastic force $F_{te}$. When the tangential force exceeds the critical force, the tangential force is directly re-scaled to match the value for the critical force:

$$\mathbf{F}_{te} = \mu_t F_{n0} \mathbf{t} - \mathbf{F}_{t,damp}$$

The same rules as those described for *mindlin* apply regarding the tangential stiffness and mixing of the shear modulus for different particle types.

The *mindlin_rescale* option uses the same form as *mindlin*, but the magnitude of the tangential displacement is re-scaled as the contact unloads, i.e. if $a < a_{t_{n-1}}$:

$$\boldsymbol{\xi} = \boldsymbol{\xi}_{\mathbf{t_{n-1}}} \frac{a}{a_{t_{n-1}}}$$

Here, $t_{n-1}$ indicates the value at the previous time step. This rescaling accounts for the fact that a decrease in the contact area upon unloading leads to the contact being unable to support the previous tangential loading, and spurious energy is created without the rescaling above (*Walton* ).

**Note**

For *mindlin*, a decrease in the tangential force already occurs as the contact unloads, due to the dependence of the tangential force on the normal force described above. By re-scaling $\boldsymbol{\xi}$, *mindlin_rescale* effectively re-scales the tangential force twice, i.e., proportionally to $a^2$. This peculiar behavior results from use of the accumulated tangential displacement to characterize the contact history. Although *mindlin_rescale* remains available for historic reasons and backward compatibility purposes, it should be avoided in favor of *mindlin_rescale/force*.

The *mindlin_rescale/force* option uses the same form as *mindlin/force*, but the magnitude of the tangential elastic force is re-scaled as the contact unloads, i.e. if $a < a_{t_{n-1}}$:

$$\mathbf{F}_{te} = \mathbf{F}_{te,t_{n-1}} \frac{a}{a_{t_{n-1}}}$$

This approach provides a better approximation of the *Mindlin-Deresiewicz* laws and is more consistent than *mindlin_rescale*. See discussions in *Thornton et al, 2013*, particularly equation 18(b) of that work and associated discussion, and *Agnolin and Roux, 2007*, particularly Appendix A.

---

The optional *rolling* keyword enables rolling friction, which resists pure rolling motion of particles. The options currently supported are:

1. *none*
2. *sds* : $k_{roll}$, $\gamma_{roll}$, $\mu_{roll}$

If the *rolling* keyword is not specified, the model defaults to *none*.

For *rolling sds*, rolling friction is computed via a spring-dashpot-slider, using a 'pseudo-force' formulation, as detailed by *Luding*. Unlike the formulation in *Marshall*, this allows for the required adjustment of rolling displacement due to changes in the frame of reference of the contacting pair. The rolling pseudo-force is computed analogously to the tangential force:

$$\mathbf{F}_{roll,0} = k_{roll}\xi_{roll} - \gamma_{roll}\mathbf{v}_{roll}$$

Here, $\mathbf{v}_{roll} = -R(\Omega_i - \Omega_j) \times \mathbf{n}$ is the relative rolling velocity, as given in *Wang et al* and *Luding*. This differs from the expressions given by *Kuhn and Bagi* and used in *Marshall*; see *Wang et al* for details. The rolling displacement is given by:

$$\xi_{roll} = \int_{t_0}^{t} \mathbf{v}_{roll}(\tau)\mathrm{d}\tau$$

A Coulomb friction criterion truncates the rolling pseudo-force if it exceeds a critical value:

$$\mathbf{F}_{roll} = \min(\mu_{roll}F_{n,0}, \|\mathbf{F}_{roll,0}\|)\mathbf{k}$$

Here, $\mathbf{k} = \mathbf{v}_{roll}/\|\mathbf{v}_{roll}\|$ is the direction of the pseudo-force. As with tangential displacement, the rolling displacement is rescaled when the critical force is exceeded, so that the spring length corresponds the critical force. Additionally, the displacement is adjusted to account for rotations of the frame of reference of the two contacting particles in a manner analogous to the tangential displacement.

The rolling pseudo-force does not contribute to the total force on either particle (hence 'pseudo'), but acts only to induce an equal and opposite torque on each particle, according to:

$$\tau_{roll,i} = R\mathbf{n} \times \mathbf{F}_{roll}$$

$$\tau_{roll,j} = -\tau_{roll,i}$$

---

The optional *twisting* keyword enables twisting friction, which resists rotation of two contacting particles about the vector **n** that connects their centers. The options currently supported are:

1. *none*
2. *sds* : $k_{twist}$, $\gamma_{twist}$, $\mu_{twist}$
3. *marshall*

If the *twisting* keyword is not specified, the model defaults to *none*.

For both *twisting sds* and *twisting marshall*, a history-dependent spring-dashpot-slider is used to compute the twisting torque. Because twisting displacement is a scalar, there is no need to adjust for changes in the frame of reference due to rotations of the particle pair. The formulation in *Marshall* therefore provides the most straightforward treatment:

$$\tau_{twist,0} = -k_{twist}\xi_{twist} - \gamma_{twist}\Omega_{twist}$$

Here $\xi_{twist} = \int_{t_0}^{t} \Omega_{twist}(\tau)\mathrm{d}\tau$ is the twisting angular displacement, and $\Omega_{twist} = (\blacksquare_i - \blacksquare_j)\cdot\mathbf{n}$ is the relative twisting angular velocity. The torque is then truncated according to:

$$\tau_{twist} = \min(\mu_{twist}F_{n,0},\, \tau_{twist,0})$$

Similar to the sliding and rolling displacement, the angular displacement is rescaled so that it corresponds to the critical value if the twisting torque exceeds this critical value:

$$\xi_{twist} = \frac{1}{k_{twist}}(\mu_{twist}F_{n,0}sgn(\Omega_{twist}) - \gamma_{twist}\Omega_{twist})$$

For *twisting sds*, the coefficients $k_{twist}$, $\gamma_{twist}$ and $\mu_{twist}$ are simply the user input parameters that follow the *twisting sds* keywords in the *pair_coeff* command.

For *twisting_marshall*, the coefficients are expressed in terms of sliding friction coefficients, as discussed in *Marshall* (see equations 32 and 33 of that work):

$$k_{twist} = 0.5k_t a^2$$

$$\eta_{twist} = 0.5\eta_t a^2$$

$$\mu_{twist} = \frac{2}{3}a\mu_t$$

Finally, the twisting torque on each particle is given by:

$$\tau_{twist,i} = \tau_{twist}\mathbf{n}$$

$$\tau_{twist,j} = -\tau_{twist,i}$$

---

If two particles are moving away from each other while in contact, there is a possibility that the particles could experience an effective attractive force due to damping. If the optional *limit_damping* keyword is used, this option will zero out the normal component of the force if there is an effective attractive force. This keyword cannot be used with the JKR or DMT models.

---

The standard velocity-Verlet integration scheme's half-step staggering of position and velocity can introduce inaccuracies in frictional tangential force calculations, resulting in unphysical kinematics in certain systems. These effects are particularly pronounced in polydisperse frictional flows characterized by large-to-small size ratios exceeding three. The *synchronized_verlet* flag implements an alternate Velocity-Verlet integration scheme, as detailed in *Vyas et al*, that synchronizes position and velocity updates for force evaluation. By refining tangential force calculations, the *synchronized_verlet* method ensures physically consistent results without significantly impacting computational cost.

---

The optional *heat* keyword enables heat conduction. The options currently supported are:

1. *none*
2. *radius* : $k_s$

3. *area* : $h_s$

If the *heat* keyword is not specified, the model defaults to *none*. All heat models calculate an additional pairwise quantity accessible by the single() function (described below) which is the heat conducted between the two particles.

For *heat radius*, the heat $Q$ conducted between two particles is given by

$$Q = 2k_s a \Delta T$$

where $\Delta T$ is the difference in the two particles' temperature, $k_s$ is a non-negative numeric value for the conductivity (in units of power/(length*temperature)), and $a$ is the radius of the contact and depends on the normal force model. This is the model proposed by *Vargas and McCarthy*.

For *heat area*, the heat $Q$ conducted between two particles is given by

$$Q = h_s A \Delta T$$

where $\Delta T$ is the difference in the two particles' temperature, $h_s$ is a non-negative numeric value for the heat transfer coefficient (in units of power/(area*temperature)), and $A = \pi a^2$ is the area of the contact and depends on the normal force model.

Note that the option *none* must either be used in all or none of of the *pair_coeff* calls. See *fix heat/flow* and *fix property/atom* for more information on this option.

---

The *granular* pair style can reproduce the behavior of the *pair gran/** styles with the appropriate settings (some very minor differences can be expected due to corrections in displacement history frame-of-reference, and the application of the torque at the center of the contact rather than at each particle). The first example above is equivalent to *pair gran/hooke 1000.0 NULL 50.0 50.0 0.4 1*. The second example is equivalent to *pair gran/hooke/history 1000.0 500.0 50.0 50.0 0.4 1*. The third example is equivalent to *pair gran/hertz/history 1000.0 500.0 50.0 50.0 0.4 1 limit_damping*.

---

LAMMPS automatically sets pairwise cutoff values for *pair_style granular* based on particle radii (and in the case of *jkr* pull-off distances). In the vast majority of situations, this is adequate. However, a cutoff value can optionally be appended to the *pair_style granular* command to specify a global cutoff (i.e. a cutoff for all atom types). Additionally, the optional *cutoff* keyword can be passed to the *pair_coeff* command, followed by a cutoff value. This will set a pairwise cutoff for the atom types in the *pair_coeff* command. These options may be useful in some rare cases where the automatic cutoff determination is not sufficient, e.g. if particle diameters are being modified via the *fix adapt* command. In that case, the global cutoff specified as part of the *pair_style granular* command is applied to all atom types, unless it is overridden for a given atom type combination by the *cutoff* value specified in the *pair coeff* command. If *cutoff* is only specified in the *pair coeff* command and no global cutoff is appended to the *pair_style granular* command, then LAMMPS will use that cutoff for the specified atom type combination, and automatically set pairwise cutoffs for the remaining atom types.

---

### 4.129.4 Mixing, shift, table, tail correction, restart, rRESPA info

The *pair_modify* mix, shift, table, and tail options are not relevant for any granular pair styles.

Mixing of coefficients is carried out using geometric averaging for most quantities, e.g. if friction coefficient for type 1-type 1 interactions is set to $\mu_1$, and friction coefficient for type 2-type 2 interactions is set to $\mu_2$, the friction coefficient for type1-type2 interactions is computed as $\sqrt{\mu_1 \mu_2}$ (unless explicitly specified to a different value by a *pair_coeff 1 2 ...* command). The exception to this is elastic modulus, only applicable to *hertz/material*, *dmt* and *jkr* normal contact models. In that case, the effective elastic modulus is computed as:

$$E_{eff,ij} = \left( \frac{1 - \nu_i^2}{E_i} + \frac{1 - \nu_j^2}{E_j} \right)^{-1}$$

If the *i-j* coefficients $E_{ij}$ and $\nu_{ij}$ are explicitly specified, the effective modulus is computed as:

$$E_{eff,ij} = \left( \frac{1 - \nu_{ij}^2}{E_{ij}} + \frac{1 - \nu_{ij}^2}{E_{ij}} \right)^{-1}$$

or

$$E_{eff,ij} = \frac{E_{ij}}{2(1 - \nu_{ij}^2)}$$

This pair style writes its information to *binary restart files*, so a pair_style command does not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

The single() function of this pair style returns 0.0 for the energy of a pairwise interaction, since energy is not conserved in these dissipative potentials. It also returns only the normal component of the pairwise interaction force. However, the single() function also calculates at least 13 extra pairwise quantities. The first 3 are the components of the tangential force between particles I and J, acting on particle I. The fourth is the magnitude of this tangential force. The next 3 (5-7) are the components of the rolling torque acting on particle I. The next entry (8) is the magnitude of the rolling torque. The next entry (9) is the magnitude of the twisting torque acting about the vector connecting the two particle centers. The next 3 (10-12) are the components of the vector connecting the centers of the two particles (x_I - x_J). If a granular sub-model calculates additional contact information (e.g. the heat sub-models calculate the amount of heat exchanged), these quantities are appended to the end of this list. First, any extra values from the normal sub-model are appended followed by the damping, tangential, rolling, twisting, then heat models. See the descriptions of specific granular sub-models above for information on any extra quantities. If two or more models are defined by pair coefficients, the size of the array is set by the maximum number of extra quantities in a model but the order of quantities is determined by each model's specific set of sub-models. Any unused quantities are zeroed.

These extra quantities can be accessed by the *compute pair/local* command, as *p1*, *p2*, ..., *p12*.

---

## 4.129.5 Restrictions

This pair style is part of the GRANULAR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires that atoms store per-particle radius, torque, and angular velocity (omega) as defined by the *atom_style sphere*.

This pair style requires you to use the *comm_modify vel yes* command so that velocities are stored by ghost atoms.

This pair style will not restart exactly when using the *read_restart* command, though it should provide statistically similar results. This is because the forces it computes depend on atom velocities and the atom velocities have been propagated half a timestep between the force computation and when the restart is written, due to using Velocity Verlet time integration. See the *read_restart* command for more details.

Accumulated values for individual contacts are saved to restart files but are not saved to data files. Therefore, forces may differ significantly when a system is reloaded using the *read_data* command.

## 4.129.6 Related commands

*pair_coeff* *pair gran/**

### 4.129.7 Default

For the *pair_coeff* settings: *damping viscoelastic*, *rolling none*, *twisting none*.

### 4.129.8 References

**(Brilliantov et al, 1996)** Brilliantov, N. V., Spahn, F., Hertzsch, J. M., & Poschel, T. (1996). Model for collisions in granular gases. Physical review E, 53(5), 5382.

**(Tsuji et al, 1992)** Tsuji, Y., Tanaka, T., & Ishida, T. (1992). Lagrangian numerical simulation of plug flow of cohesionless particles in a horizontal pipe. Powder technology, 71(3), 239-250.

**(Johnson et al, 1971)** Johnson, K. L., Kendall, K., & Roberts, A. D. (1971). Surface energy and the contact of elastic solids. Proc. R. Soc. Lond. A, 324(1558), 301-313.

**(Derjaguin et al, 1975)** Derjaguin, B. V., Muller, V. M., & Toporov, Y. P. (1975). Effect of contact deformations on the adhesion of particles. Journal of Colloid and interface science, 53(2), 314-326.

**(Zunker and Kamrin, 2024)** Zunker, W., & Kamrin, K. (2024). A mechanically-derived contact model for adhesive elastic-perfectly plastic particles, Part I: Utilizing the method of dimensionality reduction. Journal of the Mechanics and Physics of Solids, 183, 105492.

**(Zunker and Kamrin, 2024)** Zunker, W., & Kamrin, K. (2024). A mechanically-derived contact model for adhesive elastic-perfectly plastic particles, Part II: Contact under high compaction-modeling a bulk elastic response. Journal of the Mechanics and Physics of Solids, 183, 105493.

**(Zunker et al, 2025)** Zunker, W., Dunatunga, S., Thakur, S., Tang, P., & Kamrin, K. (2025). Experimentally validated DEM for large deformation powder compaction: Mechanically-derived contact model and screening of non-physical contacts. Powder Technology, 120972.

**(Luding, 2008)** Luding, S. (2008). Cohesive, frictional powders: contact models for tension. Granular matter, 10(4), 235.

**(Marshall, 2009)** Marshall, J. S. (2009). Discrete-element modeling of particulate aerosol flows. Journal of Computational Physics, 228(5), 1541-1561.

**(Silbert, 2001)** Silbert, L. E., Ertas, D., Grest, G. S., Halsey, T. C., Levine, D., & Plimpton, S. J. (2001). Granular flow down an inclined plane: Bagnold scaling and rheology. Physical Review E, 64(5), 051302.

**(Kuhn and Bagi, 2005)** Kuhn, M. R., & Bagi, K. (2004). Contact rolling and deformation in granular media. International journal of solids and structures, 41(21), 5793-5820.

**(Wang et al, 2015)** Wang, Y., Alonso-Marroquin, F., & Guo, W. W. (2015). Rolling and sliding in 3-D discrete element models. Particuology, 23, 49-55.

**(Thornton, 1991)** Thornton, C. (1991). Interparticle sliding in the presence of adhesion. J. Phys. D: Appl. Phys. 24 1942

**(Mindlin, 1949)** Mindlin, R. D. (1949). Compliance of elastic bodies in contact. J. Appl. Mech., ASME 16, 259-268.

**(Thornton et al, 2013)** Thornton, C., Cummins, S. J., & Cleary, P. W. (2013). An investigation of the comparative behavior of alternative contact force models during inelastic collisions. Powder Technology, 233, 30-46.

**(Otis R. Walton)** Walton, O.R., Personal Communication

**(Mindlin and Deresiewicz, 1953)** Mindlin, R.D., & Deresiewicz, H (1953). Elastic Spheres in Contact under Varying Oblique Force. J. Appl. Mech., ASME 20, 327-344.

**(Agnolin and Roux 2007)** Agnolin, I. & Roux, J-N. (2007). Internal states of model isotropic granular packings. I. Assembling process, geometry, and contact networks. Phys. Rev. E, 76, 061302.

**(Vargas and McCarthy 2001)** Vargas, W.L. and McCarthy, J.J. (2001). Heat conduction in granular materials. AIChE Journal, 47(5), 1052-1059.

**(Vyas et al, 2025)** Vyas D. R., Ottino J. M., Lueptow R. M., & Umbanhowar P. B. (2025). Improved Velocity-Verlet Algorithm for the Discrete Element Method. Computer Physics Communications, 109524.

# 4.130 pair_style granular/superellipsoid command

## 4.130.1 Syntax

```
pair_style granular/superellipsoid cutoff no_bounding_box curvature_gaussian

Optional settings, see discussion below.
* cutoff = global cutoff value
* no_bounding_box = skip oriented bounding box check
* curvature_gaussian = gaussian curvature coeff approximation for contact patch
```

## 4.130.2 Examples

```
pair_style granular/superellipsoid
pair_coeff * * hooke 1000.0 50.0 tangential linear_history 1000.0 1.0 0.5 damping mass_
↪velocity

pair_style granular/superellipsoid 10.0 curvature_gaussian
pair_coeff 1 1 hertz 1000.0 50.0 tangential linear_history 500.0 1.0 0.4 damping␣
↪viscoelastic
pair_coeff 2 2 hertz 500.0 50.0 tangential linear_history 250.0 1.0 0.1 damping␣
↪viscoelastic
```

## 4.130.3 Description

Added in version 30Mar2026.

The *granular/superellipsoid* style calculates granular contact forces between superellipsoidal particles (see *atom style ellipsoid*). Similar to the *granular pairstyle* which is designed for spherical particles, various normal, damping, and tangential contact models are available (rolling and twisting may be added later). The total computed forces and torques are the sum of various models selected.

All model choices and parameters are entered in the *pair_coeff* command, as described below. Coefficient values are not global, but can be set to different values for different combinations of particle types, as determined by the *pair_coeff* command. If the contact model choice is the same for two particle types, the mixing for the cross-coefficients can be carried out automatically. This is shown in the last example, where model choices are the same for type 1 - type 1 as for type 2 - type2 interactions, but coefficients are different. In this case, the mixed coefficients for type 1 - type 2 interactions can be determined from mixing rules discussed below. For additional flexibility, coefficients as well as model forms can vary between particle types.

---

This pair_style allows granular contact between two superellipsoid particles whose surface is implicitly defined as:

$$f(\mathbf{x}) = \left( \left| \frac{x}{a} \right|^{n_2} + \left| \frac{y}{b} \right|^{n_2} \right)^{n_1/n_2} + \left| \frac{z}{c} \right|^{n_1} - 1 = 0$$

for a point $\mathbf{x} = (x, y, z)$ where the coordinates are given in the reference of the principal directions of inertia of the particle. The half-diameters $a$, $b$, and $c$ correspond to the *shape* property, and the exponents $n_1$ and $n_2$ to the *block* property of the ellipsoid atom. See the doc page for the *set* command for more details.

**Note**

The contact solver strictly requires convex particle shapes to ensure a mathematically unique point of deepest penetration. Therefore, the blockiness parameters must be $n_1 \geq 2.0$ and $n_2 \geq 2.0$. Attempting to simulate concave or "pointy" particles ($n < 2.0$) will result in an error.

**Note**

For particles with high blockiness exponents ($n > 4.0$) involved in edge-to-edge or corner-to-corner contacts, the surface normal vector varies rapidly over small distances. The Newton solver may occasionally fail to converge to the strict gradient alignment tolerance (typically $10^{-10}$). You may see warning messages in the log indicating that the solver returned a sub-optimal solution, but the simulation will proceed using this best-effort contact point.

Contact detection for these aspherical particles uses the so-called "midway" minimization approach from *(Houlsby)*. Considering two particles with shape functions, $F_i$ and $F_j$, the contact point $\mathbf{X}_0$ in the global frame is obtained as:

$$\mathbf{X}_0 = \underset{\mathbf{X}}{\operatorname{argmin}}\, F_i(\mathbf{X}) + F_j(\mathbf{X}), \text{ subject to } F_i(\mathbf{X}) = F_j(\mathbf{X})$$

where the shape function is given by $F_i(\mathbf{X}) = f_i(\mathbf{R}_i^T(\mathbf{X} - \mathbf{X}_i))$ and where $\mathbf{X}_i$ and $\mathbf{R}_i$ are the center of mass and rotation matrix of the particle, respectively. The constrained minimization problem is solved using Lagrange multipliers and Newton's method with a line search as described by *(Podlozhnyuk)*.

**Note**

The shape function $F$ is not a signed distance function and does not have unit gradient $\|\nabla F\| \neq 1$ so that the so-called "midway" point is not actually located at an equal distance from the surface of both particles. For contact between non-identical particles, the contact point tends to be closer to the surface of the smaller and blockier particle.

**Note**

This formulation leads to a 4x4 system of non-linear equations. Tikhonov regularization and step clumping is used to ensure robustness of the direct solver and high convergence rate, even for blocky particles with near flat faces.

The particles overlap if both shape functions are negative at the contact point. The contact normal is obtained as: $\mathbf{n}_{ij} = \nabla F_i(\mathbf{X}_0)/\|\nabla F_i(\mathbf{X}_0)\| = -\nabla F_j(\mathbf{X}_0)/\|\nabla F_j(\mathbf{X}_0)\|$ and the overlap $\delta = \|\mathbf{X}_j^{\text{surf}} - \mathbf{X}_i^{\text{surf}}\|$ is computed as the distance between the points on the particles surfaces that are closest to the contact point in the direction of the contact normal: $F_i(\mathbf{X}_i^{\text{surf}} = \mathbf{X}_0 + \lambda_i \mathbf{n}_{ij}) = 0$ and $F_j(\mathbf{X}_j^{\text{surf}} = \mathbf{X}_0 + \lambda_j \mathbf{n}_{ij}) = 0$. Newton's method is used to solve this equation for the scalars $\lambda_i$ and $\lambda_j$ and find the surface points $\mathbf{X}_i^{\text{surf}}$ and $\mathbf{X}_j^{\text{surf}}$.

**Note**

A modified representation of the particle surface is defined as $G(\mathbf{X}) = (F(\mathbf{X}) + 1)^{1/n_1} - 1$ which is a quasi-radial distance function formulation. This formulation is used to compute the surface points once the "midway" contact point is found. This formulation is also used when the *geometric* keyword is specified in the pair_style command and the following optimization problem is solved instead for the contact point: $\mathbf{X}_0 = \underset{\mathbf{X}}{\operatorname{argmin}} \left(r_i G_i(\mathbf{X}) + r_j G_j(\mathbf{X})\right)$, subject to $r_i G_i(\mathbf{X}) = r_j G_j(\mathbf{X})$, where $r_i$ and $r_j$ are the average radii of the two particles. The geometric formulation thus yields a better approximation of the contact point for particles with different sizes, and it is slightly more robust for particles with high *block* exponents, albeit more computationally expensive.

A hierarchical approach is used to limit the cost of contact detection. First, intersection of the bounding spheres of the two particles of bounding radii $r_i$ and $r_j$ is checked. If the distance between the particles center is more than the sum of the radii $\|\mathbf{X}_j - \mathbf{X}_j\| > r_i + r_j$, the particles do not intersect. Then, if the bounding spheres intersect, intersection of the oriented bounding box is checked. This is done following the equations of *(Eberly)*. This check is always performed, unless the *no_bounding_box* keyword is used. This is advantageous for all particle shapes except for superellipses with aspect ratio close to one and both blockiness indexes close to 2.

**Warning**

The Newton-Raphson minimization used to find the midway contact point can fail to converge if the initial starting guess is too far from the true physical surface. This typically occurs if a user specifies a manual global *cutoff* that is significantly larger than the particles **and** enables the *no_bounding_box* keyword. Under these conditions, the solver attempts to resolve contacts between widely separated particles, which might cause the math to diverge and instantly crashing the simulation. It is strongly recommended to keep bounding box checks enabled if a large cutoff is specified.

---

This section provides an overview of the various normal, tangential, and damping contact models available. For additional context, see the discussion in the *granular pairstyle* doc page which includes all of these options.

The first required keyword for the *pair_coeff* command is the normal contact model. Currently supported options for normal contact models and their required arguments are:

1. *hooke* : $k_n$, $\eta_{n0}$ (or $e$)
2. *hertz* : $k_n$, $\eta_{n0}$ (or $e$)

Here, $k_n$ is spring stiffness (with units that depend on model choice, see below); $\eta_{n0}$ is a damping prefactor (or, in its place a coefficient of restitution $e$, depending on the choice of damping mode, see below).

For the *hooke* model, the normal, elastic component of force acting on particle $i$ due to contact with particle $j$ is given by:

$$\mathbf{F}_{ne,Hooke} = k_n \delta_{ij} \mathbf{n}$$

Where $\delta_{ij}$ is the particle overlap, (note the i-j ordering so that $\mathbf{F}_{ne}$ is positive for repulsion), and $\mathbf{n}$ is the contact normal vector at the contact point. Therefore, for *hooke*, the units of the spring constant $k_n$ are *force/distance*, or equivalently *mass/time*^2.

For the *hertz* model, the normal component of force is given by:

$$\mathbf{F}_{ne,Hertz} = k_n R_{eff}^{1/2} \delta_{ij}^{3/2} \mathbf{n}$$

Here, $R_{eff} = R = \frac{R_i R_j}{R_i + R_j}$ is the effective radius, and $R_i$ is the equivalent radius of the i-th particle at the surface contact point with the j-th particle. This radius is either the inverse of the mean curvature coefficient, $R_i = 2/(\kappa_1 + \kappa_2)$, or the gaussian curvature coefficient $R_i = 1/\sqrt{\kappa_1 \kappa_2}$, where $\kappa_{1,2}$ are the principal curvatures of the particle surface at the contact point. For *hertz*, the units of the spring constant $k_n$ are *force/length*^2, or equivalently *pressure*.

**Note**

To ensure numerical stability and preserve physical realism, the computed contact radius is mathematically capped. For highly blocky particles undergoing flat-on-flat contact, the theoretical curvature approaches zero, which would yield an infinite contact radius and cause a force explosion. To prevent this, the maximum contact radius is capped at the physical bounding radius of the smallest interacting particle. Conversely, for sharp corner contacts where curvature approaches infinity, the calculated radius would drop to zero, eliminating the repulsive force entirely. The

contact radius is therefore lower-bounded by a minimum fraction of the physical radius ($10^{-4}\min(r_i, r_j)$) to prevent particles from unphysically interpenetrating.

In addition, the normal force is augmented by a damping term of the following general form:

$$\mathbf{F}_{n,damp} = -\eta_n \mathbf{v}_{n,rel}$$

Here, $\mathbf{v}_{n,rel} = (\mathbf{v}_j - \mathbf{v}_i) \cdot \mathbf{n}\,\mathbf{n}$ is the component of relative velocity along $\mathbf{n}$.

The optional *damping* keyword to the *pair_coeff* command followed by a keyword determines the model form of the damping factor $\eta_n$, and the interpretation of the $\eta_{n0}$ or $e$ coefficients specified as part of the normal contact model settings. The *damping* keyword and corresponding model form selection may be appended anywhere in the *pair coeff* command. Note that the choice of damping model affects both the normal and tangential damping. The options for the damping model currently supported are:

1. *mass_velocity*
2. *viscoelastic*

If the *damping* keyword is not specified, the *viscoelastic* model is used by default.

For *damping mass_velocity*, the normal damping is given by:

$$\eta_n = \eta_{n0} m_{eff}$$

Here, $\eta_{n0}$ is the damping coefficient specified for the normal contact model, in units of 1/*time* and $m_{eff} = m_i m_j/(m_i+m_j)$ is the effective mass. Use *damping mass_velocity* to reproduce the damping behavior of *pair gran/hooke/**.

The *damping viscoelastic* model is based on the viscoelastic treatment of *(Brilliantov et al)*, where the normal damping is given by:

$$\eta_n = \eta_{n0}\, a m_{eff}$$

Here, $a$ is the contact radius, given by $a = \sqrt{R\delta}$ for all models. For *damping viscoelastic*, $\eta_{n0}$ is in units of 1/(*time***distance*).

The total normal force is computed as the sum of the elastic and damping components:

$$\mathbf{F}_n = \mathbf{F}_{ne} + \mathbf{F}_{n,damp}$$

---

The *pair_coeff* command also requires specification of the tangential contact model. The required keyword *tangential* is expected, followed by the model choice and associated parameters. Currently there is only one supported tangential model with expected parameters as follows:

1. *linear_history* : $k_t$, $x_{\gamma,t}$, $\mu_s$

Here, $x_{\gamma,t}$ is a dimensionless multiplier for the normal damping $\eta_n$ that determines the magnitude of the tangential damping, $\mu_t$ is the tangential (or sliding) friction coefficient, and $k_t$ is the tangential stiffness coefficient.

The tangential damping force $\mathbf{F}_{\text{t,damp}}$ is given by:

$$\mathbf{F}_{\text{t,damp}} = -\eta_t \mathbf{v}_{t,rel}$$

The tangential damping prefactor $\eta_t$ is calculated by scaling the normal damping $\eta_n$ (see above):

$$\eta_t = -x_{\gamma,t}\eta_n$$

The normal damping prefactor $\eta_n$ is determined by the choice of the *damping* keyword, as discussed above. Thus, the *damping* keyword also affects the tangential damping. The parameter $x_{\gamma,t}$ is a scaling coefficient. Several works in the literature use $x_{\gamma,t} = 1$ (*Marshall*, *Tsuji et al*, *Silbert et al*). The relative tangential velocity at the point of contact is given by $\mathbf{v}_{t,rel} = \mathbf{v}_t - (R_i\Omega_i + R_j\Omega_j) \times \mathbf{n}$, where $\mathbf{v}_t = \mathbf{v}_r - \mathbf{v}_r \cdot \mathbf{n}\,\mathbf{n}$, $\mathbf{v}_r = \mathbf{v}_j - \mathbf{v}_i$ . The normal force value $F_{n0}$ used to compute the critical force depends on the form of the contact model. It is given by the magnitude of the normal force:

$$F_{n0} = \|\mathbf{F}_n\|$$

The remaining tangential options all use accumulated tangential displacement (i.e. contact history). The accumulated tangential displacement is discussed in details below in the context of the *linear_history* option. The same treatment of the accumulated displacement will apply to other (future) options as well.

For *tangential linear_history*, the tangential force is given by:

$$\mathbf{F}_t = \min(\mu_t F_{n0}, \| - k_t\xi + \mathbf{F}_{\mathrm{t,damp}}\|)\mathbf{t}$$

Here, $\mathbf{t}$ is the direction of the tangential force given by:

$$\mathbf{t} = \frac{-k_t\xi + \mathbf{F}_{\mathrm{t,damp}}}{\| - k_t\xi + \mathbf{F}_{\mathrm{t,damp}}\|}$$

and, $\xi$ is the tangential displacement accumulated during the entire duration of the contact:

$$\xi = \int_{t0}^{t} \mathbf{v}_{t,rel}(\tau)\mathrm{d}\tau$$

This accumulated tangential displacement must be adjusted to account for changes in the frame of reference of the contacting pair of particles during contact. This occurs due to the overall motion of the contacting particles in a rigid-body-like fashion during the duration of the contact. There are two modes of motion that are relevant: the 'tumbling' rotation of the contacting pair, which changes the orientation of the plane in which tangential displacement occurs; and 'spinning' rotation of the contacting pair about the vector connecting their centers of mass ($\mathbf{n}$). Corrections due to the former mode of motion are made by rotating the accumulated displacement into the plane that is tangential to the contact vector at each step, or equivalently removing any component of the tangential displacement that lies along $\mathbf{n}$, and rescaling to preserve the magnitude. This follows the discussion in *Luding*, see equation 17 and relevant discussion in that work:

$$\xi = \left(\xi' - (\mathbf{n}\cdot\xi')\mathbf{n}\right)\frac{\|\xi'\|}{\|\xi' - (\mathbf{n}\cdot\xi')\mathbf{n}\|}$$

Here, $\xi'$ is the accumulated displacement prior to the current time step and $\xi$ is the corrected displacement. Corrections to the displacement due to the second mode of motion described above (rotations about $\mathbf{n}$) are not currently implemented, but are expected to be minor for most simulations.

Furthermore, when the tangential force exceeds the critical force, the tangential displacement is re-scaled to match the value for the critical force (see *Luding*, equation 20 and related discussion):

$$\xi = -\frac{1}{k_t}\left(\mu_t F_{n0}\mathbf{t} - \mathbf{F}_{t,damp}\right)$$

The tangential force is added to the total normal force (elastic plus damping) to produce the total force on the particle.

Unlike perfect spheres, the surface normal at the contact point of a superellipsoid does not generally pass through the particle's center of mass. Therefore, both the normal and tangential forces act at the contact point to induce a torque on each particle.

Using the exact contact point $\mathbf{X}_0$ determined by the geometric solver, the branch vectors from the particle centers of mass to the contact point are defined as $\mathbf{r}_{ci} = \mathbf{X}_0 - \mathbf{x}_i$ and $\mathbf{r}_{cj} = \mathbf{X}_0 - \mathbf{x}_j$. The resulting torques are calculated as:

$$\tau_i = \mathbf{r}_{ci} \times \mathbf{F}_{tot}$$

$$\tau_j = -\mathbf{r}_{cj} \times \mathbf{F}_{tot}$$

---

If two particles are moving away from each other while in contact, there is a possibility that the particles could experience an effective attractive force due to damping. If the optional *limit_damping* keyword is used, this option will zero out the normal component of the force if there is an effective attractive force.

---

LAMMPS automatically sets pairwise cutoff values for *pair_style granular/superellipsoid* based on particle radii. In the vast majority of situations, this is adequate. However, a cutoff value can optionally be appended to the *pair_style granular/superellipsoid* command to specify a global cutoff (i.e. a cutoff for all atom types). This option may be useful in some rare cases where the automatic cutoff determination is not sufficient.

---

### 4.130.4 Mixing, shift, table, tail correction, restart, rRESPA info

The *pair_modify* mix, shift, table, and tail options are not relevant for granular pair styles.

Mixing of coefficients is carried out using geometric averaging for most quantities, e.g. if friction coefficient for type 1-type 1 interactions is set to $\mu_1$, and friction coefficient for type 2-type 2 interactions is set to $\mu_2$, the friction coefficient for type1-type2 interactions is computed as $\sqrt{\mu_1 \mu_2}$ (unless explicitly specified to a different value by a *pair_coeff 1 2 ...* command).

This pair style writes its information to *binary restart files*, so a pair_style command does not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

The `single()` function of these pair styles returns 0.0 for the energy of a pairwise interaction, since energy is not conserved in these dissipative potentials. It also returns only the normal component of the pairwise interaction force.

---

### 4.130.5 Restrictions

The *atom_style* must be set to *ellipsoid superellipsoid* to enable superellipsoid particles' shape parameters (3 lengths and two blockiness parameters), see *atom_style* for more details.

This pair style require Newton's third law be set to *off* for pair interactions.

There are currently no versions of *fix wall/gran* or *fix wall/gran/region* that are compatible with superellipsoid particles.

This pair style is part of the ASPHERE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires that atoms store per-particle bounding radius, shapes, blockiness, inertia, torque, and angular momentum (omega) as defined by the *atom_style ellipsoid superellipsoid*.

This pair style requires you to use the *comm_modify vel yes* command so that velocities are stored by ghost atoms.

This pair style will not restart exactly when using the *read_restart* command, though it should provide statistically similar results. This is because the forces it computes depend on atom velocities and the atom velocities have been propagated half a timestep between the force computation and when the restart is written, due to using Velocity Verlet time integration. See the *read_restart* command for more details.

Accumulated values for individual contacts are saved to restart files but are not saved to data files. Therefore, forces may differ significantly when a system is reloaded using the *read_data* command.

### 4.130.6 Related commands

*pair_coeff* *pair granular*

### 4.130.7 Default

For the *pair_coeff* settings: *damping viscoelastic*

### 4.130.8 References

**(Brilliantov et al, 1996)** Brilliantov, N. V., Spahn, F., Hertzsch, J. M., & Poschel, T. (1996). Model for collisions in granular gases. Physical review E, 53(5), 5382.

**(Tsuji et al, 1992)** Tsuji, Y., Tanaka, T., & Ishida, T. (1992). Lagrangian numerical simulation of plug flow of cohesionless particles in a horizontal pipe. Powder technology, 71(3), 239-250.

**(Luding, 2008)** Luding, S. (2008). Cohesive, frictional powders: contact models for tension. Granular matter, 10(4), 235.

**(Marshall, 2009)** Marshall, J. S. (2009). Discrete-element modeling of particulate aerosol flows. Journal of Computational Physics, 228(5), 1541-1561.

**(Silbert, 2001)** Silbert, L. E., Ertas, D., Grest, G. S., Halsey, T. C., Levine, D., & Plimpton, S. J. (2001). Granular flow down an inclined plane: Bagnold scaling and rheology. Physical Review E, 64(5), 051302.

**(Thornton, 1991)** Thornton, C. (1991). Interparticle sliding in the presence of adhesion. J. Phys. D: Appl. Phys. 24 1942

**(Thornton et al, 2013)** Thornton, C., Cummins, S. J., & Cleary, P. W. (2013). An investigation of the comparative behavior of alternative contact force models during inelastic collisions. Powder Technology, 233, 30-46.

**(Otis R. Walton)** Walton, O.R., Personal Communication

**(Podlozhnyuk)** Podlozhnyuk, Pirker, Kloss, Comp. Part. Mech., 4:101-118 (2017).

**(Houlsby)** Houlsby, Computers and Geotechnics, 36, 953-959 (2009).

**(Eberly)** Eberly, Geometric Tools: Dynamic Collision Detection Using Oriented Bounding Boxes (2008).

## 4.131 pair_style lj/gromacs command

Accelerator Variants: *lj/gromacs/gpu*, *lj/gromacs/kk*, *lj/gromacs/omp*

## 4.132 pair_style lj/gromacs/coul/gromacs command

Accelerator Variants: *lj/gromacs/coul/gromacs/kk*, *lj/gromacs/coul/gromacs/omp*

### 4.132.1 Syntax

```
pair_style style args
```

- style = *lj/gromacs* or *lj/gromacs/coul/gromacs*
- args = list of arguments for a particular style

```
lj/gromacs args = inner outer
  inner, outer = global switching cutoffs for Lennard Jones
lj/gromacs/coul/gromacs args = inner outer (inner2) (outer2)
```

```
  inner, outer = global switching cutoffs for Lennard Jones (and Coulombic if only 2␣
↪args)
  inner2, outer2 = global switching cutoffs for Coulombic (optional)
```

### 4.132.2 Examples

```
pair_style lj/gromacs 9.0 12.0
pair_coeff * * 100.0 2.0
pair_coeff 2 2 100.0 2.0 8.0 10.0

pair_style lj/gromacs/coul/gromacs 9.0 12.0
pair_style lj/gromacs/coul/gromacs 8.0 10.0 7.0 9.0
pair_coeff * * 100.0 2.0
```

### 4.132.3 Description

The *lj/gromacs* styles compute shifted LJ and Coulombic interactions with an additional switching function S(r) that ramps the energy and force smoothly to zero between an inner and outer cutoff. It is a commonly used potential in the GROMACS MD code and for the coarse-grained models of *(Marrink)*.

$$
\begin{aligned}
E_{LJ} &= 4\varepsilon \left[ \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] + S_{LJ}(r) \qquad r < r_c \\
E_C &= \frac{C q_i q_j}{\varepsilon r} + S_C(r) \qquad r < r_c \\
S(r) &= C \qquad r < r_1 \\
S(r) &= \frac{A}{3}(r - r_1)^3 + \frac{B}{4}(r - r_1)^4 + C \qquad r_1 < r < r_c \\
A &= (-3E'(r_c) + (r_c - r_1)E''(r_c))/(r_c - r_1)^2 \\
B &= (2E'(r_c) - (r_c - r_1)E''(r_c))/(r_c - r_1)^3 \\
C &= -E(r_c) + \frac{1}{2}(r_c - r_1)E'(r_c) - \frac{1}{12}(r_c - r_1)^2 E''(r_c)
\end{aligned}
$$

$r_1$ is the inner cutoff; $r_c$ is the outer cutoff. The coefficients A, B, and C are computed by LAMMPS to perform the shifting and smoothing. The function S(r) is actually applied once to each term of the LJ formula and once to the Coulombic formula, so there are 2 or 3 sets of A,B,C coefficients depending on which pair_style is used. The boundary conditions applied to the smoothing function are as follows: $S'(r_1) = S''(r_1) = 0, S(r_c) = -E(r_c), S'(r_c) = -E'(r_c)$, and $S''(r_c) = -E''(r_c)$, where E(r) is the corresponding term in the LJ or Coulombic potential energy function. Single and double primes denote first and second derivatives with respect to r, respectively.

The inner and outer cutoff for the LJ and Coulombic terms can be the same or different depending on whether 2 or 4 arguments are used in the pair_style command. The inner LJ cutoff must be > 0, but the inner Coulombic cutoff can be >= 0.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- $\varepsilon$ (energy units)
- $\sigma$ (distance units)
- inner (distance units)
- outer (distance units)

Note that sigma is defined in the LJ formula as the zero-crossing distance for the potential, not as the energy minimum at $2^{1/6}\sigma$.

The last 2 coefficients are optional inner and outer cutoffs for style *lj/gromacs*. If not specified, the global *inner* and *outer* values are used.

The last 2 coefficients cannot be used with style *lj/gromacs/coul/gromacs* because this force field does not allow varying cutoffs for individual atom pairs; all pairs use the global cutoff(s) specified in the pair_style command.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.132.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon and sigma coefficients and cutoff distance for all of the lj/cut pair styles can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

None of the GROMACS pair styles support the *pair_modify* shift option, since the Lennard-Jones portion of the pair interaction is already smoothed to 0.0 at the cutoff.

The *pair_modify* table option is not relevant for this pair style.

None of the GROMACS pair styles support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure, since there are no corrections for a potential that goes to 0.0 at the cutoff.

All of the GROMACS pair styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

All of the GROMACS pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

---

### 4.132.5 Restrictions

This pair style is part of the EXTRA-PAIR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.132.6 Related commands

*pair_coeff*

### 4.132.7 Default

none

---

**(Marrink)** Marrink, de Vries, Mark, J Phys Chem B, 108, 750-760 (2004).

## 4.133 pair_style gw command

Accelerator Variants: *gw/omp*

## 4.134 pair_style gw/zbl command

Accelerator Variants: *gw/zbl/omp*

### 4.134.1 Syntax

```
pair_style style
```

- style = *gw* or *gw/zbl*

### 4.134.2 Examples

```
pair_style gw
pair_coeff * * SiC.gw Si C C

pair_style gw/zbl
pair_coeff * * SiC.gw.zbl C Si
```

### 4.134.3 Description

The *gw* style computes a 3-body *Gao-Weber* potential; similarly *gw/zbl* combines this potential with a modified repulsive ZBL core function in a similar fashion as implemented in the *tersoff/zbl* pair style.

Unfortunately the author of this contributed code has not been able to submit a suitable documentation explaining the details of the potentials. The LAMMPS developers thus have finally decided to release the code anyway with only the technical explanations. For details of the model and the parameters, please refer to the linked publication.

Only a single pair_coeff command is used with the *gw* and *gw/zbl* styles which specifies a Gao-Weber potential file with parameters for all needed elements. These are mapped to LAMMPS atom types by specifying N additional arguments after the filename in the pair_coeff command, where N is the number of LAMMPS atom types:

- filename
- N element names = mapping of GW elements to atom types

See the *pair_coeff* page for alternate ways to specify the path for the potential file.

As an example, imagine a file SiC.gw has Gao-Weber values for Si and C. If your LAMMPS simulation has 4 atoms types and you want the first 3 to be Si, and the fourth to be C, you would use the following pair_coeff command:

```
pair_coeff * * SiC.gw Si Si Si C
```

The first 2 arguments must be * * so as to span all LAMMPS atom types. The first three Si arguments map LAMMPS atom types 1,2,3 to the Si element in the GW file. The final C argument maps LAMMPS atom type 4 to the C element in the GW file. If a mapping value is specified as NULL, the mapping is not performed. This can be used when a *gw* potential is used as part of the *hybrid* pair style. The NULL values are placeholders for atom types that will be used with other potentials.

Gao-Weber files in the *potentials* directory of the LAMMPS distribution have a ".gw" suffix. Gao-Weber with ZBL files have a ".gz.zbl" suffix. The structure of the potential files is similar to other many-body potentials supported by LAMMPS. You have to refer to the comments in the files and the literature to learn more details.

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.134.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, where types I and J correspond to two different element types, mixing is performed by LAMMPS as described above from values in the potential file.

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.134.5 Restrictions

This pair style is part of the MANYBODY package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires the *newton* setting to be "on" for pair interactions.

The Gao-Weber potential files provided with LAMMPS (see the potentials directory) are parameterized for metal *units*. You can use the GW potential with any LAMMPS units, but you would need to create your own GW potential file with coefficients listed in the appropriate units if your simulation does not use "metal" units.

### 4.134.6 Related commands

*pair_coeff*

### 4.134.7 Default

none

---

**(Gao)** Gao and Weber, Nuclear Instruments and Methods in Physics Research B 191 (2012) 504.

## 4.135 pair_style harmonic/cut command

Accelerator Variants: *harmonic/cut/kk*, *harmonic/cut/omp*

### 4.135.1 Syntax

```
pair_style style
```

- style = *harmonic/cut*

### 4.135.2 Examples

```
pair_style harmonic/cut
pair_coeff * * 0.2 2.0
pair_coeff 1 1 0.5 2.5
```

### 4.135.3 Description

Added in version 17Feb2022.

Style *harmonic/cut* computes pairwise repulsive-only harmonic interactions with the formula

$$E = k(r_c - r)^2 \qquad r < r_c$$

where $r_c$ is the cutoff. Note that the usual 1/2 factor is included in $k$.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $k$ (energy/distance^2 units)
- $r_c$ (distance units)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.135.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the $k$ and $r_c$ coefficients can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

Since the potential is zero at and beyond the cutoff parameter by construction, there is no need to support the *pair_modify* shift or tail options for the energy and pressure of the pair interaction.

These pair styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

These pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

---

### 4.135.5 Restrictions

The *harmonic/cut* pair style is only enabled if LAMMPS was built with the EXTRA-PAIR package. See the *Build package* page for more info.

### 4.135.6 Related commands

*pair_coeff*

### 4.135.7 Default

none

## 4.136 pair_style hbond/dreiding/lj command

Accelerator Variants: *hbond/dreiding/lj/omp*

## 4.137 pair_style hbond/dreiding/lj/angleoffset command

Accelerator Variants: *hbond/dreiding/lj/angleoffset/omp*

## 4.138 pair_style hbond/dreiding/morse command

Accelerator Variants: *hbond/dreiding/morse/omp*

## 4.139 pair_style hbond/dreiding/morse/angleoffset command

Accelerator Variants: *hbond/dreiding/morse/angleoffset/omp*

### 4.139.1 Syntax

```
pair_style style N inner_distance_cutoff outer_distance_cutoff angle_cutoff equilibrium_
↪angle
```

- style = *hbond/dreiding/lj* or *hbond/dreiding/morse* or *hbond/dreiding/lj/angleoffset* or *hbond/dreiding/morse/angleoffset*
- N = power of cosine of angle theta (integer)
- inner_distance_cutoff = global inner cutoff for Donor-Acceptor interactions (distance units)
- outer_distance_cutoff = global cutoff for Donor-Acceptor interactions (distance units)
- angle_cutoff = global angle cutoff for Acceptor-Hydrogen-Donor interactions (degrees)
- (with style angleoffset) equilibrium_angle = global equilibrium angle for Acceptor-Hydrogen-Donor interactions (degrees)

### 4.139.2 Examples

```
pair_style hybrid/overlay lj/cut 10.0 hbond/dreiding/lj 4 9.0 11.0 90.0
pair_coeff 1 2 hbond/dreiding/lj 3 i 9.5 2.75 4 9.0 11.0 90.0
```

(continues on next page)

(continued from previous page)

```
pair_style hybrid/overlay lj/cut 10.0 hbond/dreiding/morse 2 9.0 11.0 90.0
pair_coeff 1 2 hbond/dreiding/morse 3 i 3.88 1.7241379 2.9 2 9.0 11.0 90.0

labelmap atom 1 C 2 O 3 H
pair_coeff C O hbond/dreiding/morse H i 3.88 1.7241379 2.9 2 9.0 11.0 90.0

pair_style hybrid/overlay lj/cut 10.0 hbond/dreiding/lj 4 9.0 11.0 90 170.0
pair_coeff 1 2 hbond/dreiding/lj 3 i 9.5 2.75 4 9.0 11.0 90.0
```

### 4.139.3 Description

The *hbond/dreiding* styles compute the Acceptor-Hydrogen-Donor (AHD) 3-body hydrogen bond interaction for the *DREIDING* force field, given by:

$$
\begin{aligned}
E =& \left[LJ(r)|Morse(r)\right] && r < r_{\text{in}} \\
=& S(r) * \left[LJ(r)|Morse(r)\right] && r_{\text{in}} < r < r_{\text{out}} \\
=& 0 && r > r_{\text{out}}
\end{aligned}
$$

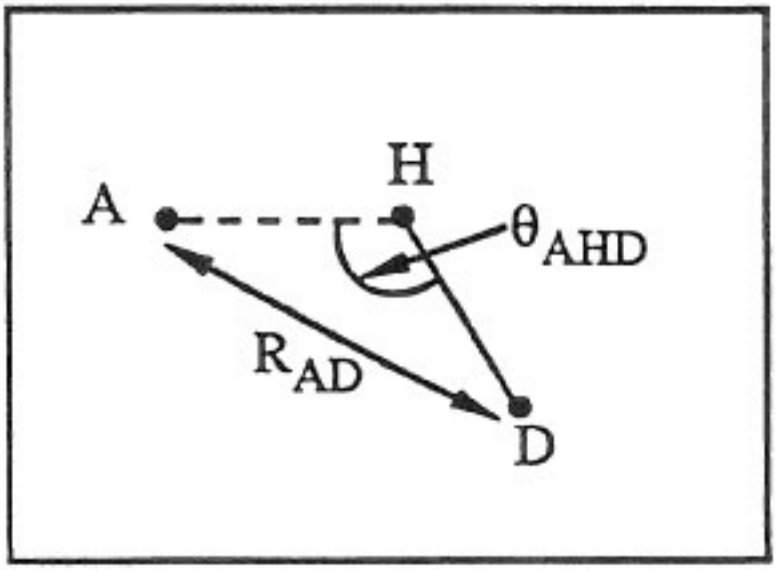
$$
LJ(r) = AR^{-12} - BR^{-10} cos^n\theta = \varepsilon \left\{ 5 \left[\frac{\sigma}{r}\right]^{12} - 6 \left[\frac{\sigma}{r}\right]^{10} \right\} cos^n\theta
$$

$$
Morse(r) = D_0 \left\{\chi^2 - 2\chi\right\} cos^n\theta = D_0 \left\{e^{-2\alpha(r-r_0)} - 2e^{-\alpha(r-r_0)}\right\} cos^n\theta
$$

$$
S(r) = \frac{\left[r_{\text{out}}^2 - r^2\right]^2 \left[r_{\text{out}}^2 + 2r^2 - 3r_{\text{in}}^2\right]}{\left[r_{\text{out}}^2 - {r_{\text{in}}}^2\right]^3}
$$

where $r_{\text{in}}$ is the inner spline distance cutoff, $r_{\text{out}}$ is the outer distance cutoff, $\theta_c$ is the angle cutoff, and $n$ is the power of the cosine of the angle $\theta$.

Here, $r$ is the radial distance between the donor (D) and acceptor (A) atoms and $\theta$ is the bond angle between the acceptor, the hydrogen (H) and the donor atoms:




These 3-body interactions can be defined for pairs of acceptor and donor atoms, based on atom types. For each donor/acceptor atom pair, the third atom in the interaction is a hydrogen permanently bonded to the donor atom, e.g. in a bond list read in from a data file via the *read_data* command. The atom types of possible hydrogen atoms for each donor/acceptor type pair are specified by the *pair_coeff* command (see below).

Style *hbond/dreiding/lj* is the original DREIDING potential of *(Mayo)*. It uses a LJ 12/10 functional for the Donor-Acceptor interactions. To match the results in the original paper, use n = 4.

Style *hbond/dreiding/morse* is an improved version using a Morse potential for the Donor-Acceptor interactions. *(Liu)* showed that the Morse form gives improved results for Dendrimer simulations, when n = 2.

Added in version 4Feb2025.

The style variants *hbond/dreiding/lj/angleoffset* and *hbond/dreiding/lj/angleoffset* take the equilibrium angle of the AHD as input, allowing it to reach 180 degrees. This variant option was added to account for cases (especially in some coarse-grained models) in which the equilibrium state of the bonds may equal the minimum energy state.

See the *Howto bioFF* page for more information on the DREIDING force field.

**Note**

Because the Dreiding hydrogen bond potential is only one portion of an overall force field which typically includes other pairwise interactions, it is common to use it as a sub-style in a *pair_style hybrid/overlay* command, where another pair style provides the repulsive core interaction between pairs of atoms, e.g. a 1/r^12 Lennard-Jones repulsion.

**Note**

When using the hbond/dreiding pair styles with *pair_style hybrid/overlay*, you should explicitly define pair interactions between the donor atom and acceptor atoms, (as well as between these atoms and ALL other atoms in your system). Whenever *pair_style hybrid/overlay* is used, ordinary mixing rules are not applied to atoms like the donor and acceptor atoms because they are typically referenced in multiple pair styles. Neglecting to do this can cause difficult-to-detect physics problems.

**Note**

In the original Dreiding force field paper 1-4 non-bonded interactions ARE allowed. If this is desired for your model, use the special_bonds command (e.g. "special_bonds lj 0.0 0.0 1.0") to turn these interactions on.

**Note**

For the *angleoffset* variants, the referenced angle offset is the supplementary angle of the equilibrium angle parameter. It means if the equilibrium angle is 166.6 degrees, the calculated angle offset is 13.4 degrees.

---

The following coefficients must be defined for pairs of eligible donor/acceptor types via the *pair_coeff* command as in the examples above.

**Note**

Unlike other pair styles and their associated *pair_coeff* commands, you do not need to specify pair_coeff settings for all possible I,J type pairs. Only I,J type pairs for atoms which act as joint donors/acceptors need to be specified; all other type pairs are assumed to be inactive.

**Note**

A *pair_coeff* command can be specified multiple times for the same donor/acceptor type pair. This enables multiple hydrogen types to be assigned to the same donor/acceptor type pair. For other pair_styles, if the pair_coeff command is re-used for the same I,J type pair, the settings for that type pair are overwritten. For the hydrogen bond potentials this is not the case; the settings are cumulative. This means the only way to turn off a previous setting, is to re-use the pair_style command and start over.

For the *hbond/dreiding/lj* style the list of coefficients is as follows:

- K = hydrogen atom type = 1 to Ntypes, or type label
- donor flag = *i* or *j*
- $\varepsilon$ (energy units)
- $\sigma$ (distance units)
- $n$ = exponent in formula above
- distance cutoff $r_{\text{in}}$ (distance units)
- distance cutoff $r_{\text{out}}$ (distance units)
- angle cutoff (degrees)

For the *hbond/dreiding/morse* style the list of coefficients is as follows:

- K = hydrogen atom type = 1 to Ntypes, or type label
- donor flag = *i* or *j*
- $D_0$ (energy units)
- $\alpha$ (1/distance units)
- $r_0$ (distance units)
- $n$ = exponent in formula above
- distance cutoff $r_{\text{in}}$ (distance units)
- distance cutoff $r_{out}$ (distance units)
- angle cutoff (degrees)

For both the *hbond/dreiding/lj/angleoffset* and *hbond/dreiding/morse/angleoffset* styles an additional parameter is added: * equilibrium angle (degrees)

For all styles, a single hydrogen atom type K can be specified, or a wild-card asterisk can be used in place of or in conjunction with the K arguments to select multiple types as hydrogen atoms. This takes the form "*" or "*n" or "n*" or "m*n". See the *pair_coeff* command page for details.

If the donor flag is *i*, then the atom of type I in the pair_coeff command is treated as the donor, and J is the acceptor. If the donor flag is *j*, then the atom of type J in the pair_coeff command is treated as the donor and I is the donor. This option is required because the *pair_coeff* command requires that I <= J.

$\varepsilon$ and $\sigma$ are settings for the hydrogen bond potential based on a Lennard-Jones functional form. Note that sigma is defined as the zero-crossing distance for the potential, not as the energy minimum at $2^{1/6}\sigma$.

$D_0$ and $\alpha$ and $r_0$ are settings for the hydrogen bond potential based on a Morse functional form.

The last 3 coefficients for both styles are optional. If not specified, the global n, distance cutoff, and angle cutoff specified in the pair_style command are used. If you wish to only override the second or third optional parameter, you must also specify the preceding optional parameters.

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 4.139.4 Mixing, shift, table, tail correction, restart, rRESPA info

These pair styles do not support mixing. You must explicitly identify each donor/acceptor type pair.

These styles do not support the *pair_modify* shift option for the energy of the interactions.

The *pair_modify* table option is not relevant for these pair styles.

These pair styles do not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

These pair styles do not write their information to *binary restart files*, so pair_style and pair_coeff commands need to be re-specified in an input script that reads a restart file.

These pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

These pair styles tally a count of how many hydrogen bonding interactions they calculate each timestep and the hbond energy. These quantities can be accessed via the *compute pair* command as a vector of values of length 2.

To print these quantities to the log file (with a descriptive column heading) the following commands could be included in an input script:

```
compute hb all pair hbond/dreiding/lj
variable n_hbond equal c_hb[1] #number hbonds
variable E_hbond equal c_hb[2] #hbond energy
thermo_style custom step temp epair v_E_hbond
```

### 4.139.5 Restrictions

The base pair styles can only be used if LAMMPS was built with the MOLECULE package. The *angleoffset* variant also requires the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 4.139.6 Related commands

*pair_coeff*

### 4.139.7 Default

none

**(Mayo)** Mayo, Olfason, Goddard III, J Phys Chem, 94, 8897-8909 (1990).

**(Liu)** Liu, Bryantsev, Diallo, Goddard III, J. Am. Chem. Soc 131 (8) 2798 (2009)

# 4.140 pair_style hdnnp command

## 4.140.1 Syntax

```
pair_style hdnnp cutoff keyword value ...
```

- cutoff = short-range cutoff of HDNNP (maximum symmetry function cutoff radius)
- zero or more keyword/value pairs may be appended
- keyword = *dir* or *showew* or *showewsum* or *maxew* or *resetew* or *cflength* or *cfenergy*
- value depends on the preceding keyword:

```
dir value = directory
    directory = Path to HDNNP configuration files
showew value = yes or no
showewsum value = summary
          summary = Write EW summary every this many timesteps (0 turns summary off)
maxew value = threshold
      threshold = Maximum number of EWs allowed
resetew value = yes or no
cflength value = length
         length = Length unit conversion factor
cfenergy value = energy
         energy = Energy unit conversion factor
```

## 4.140.2 Examples

```
pair_style hdnnp 6.35 showew yes showewsum 100 maxew 1000 resetew yes cflength 1.
↪8897261328 cfenergy 0.0367493254
pair_coeff * * H O

pair_style hdnnp 6.01 dir "./" showewsum 10000
pair_coeff * * S Cu NULL Cu
```

## 4.140.3 Description

This pair style adds an interaction based on the high-dimensional neural network potential (HDNNP) method as presented in *(Behler and Parrinello 2007)*. HDNNPs are machine learning potentials which require careful training of neural networks prior to application in MD simulations. The pair style uses an interface to the *n2p2* library *(Singraber, Behler and Dellago 2019)* which is available on Github here. Please see the *n2p2* documentation for further details. *n2p2* (and hence this pair style) is compatible with neural network potentials trained with its own tools *(Singraber et al 2019)* and with RuNNer.

Only a single *pair_coeff* command with two asterisk wild-cards is used with this pair style. Its additional arguments define the mapping of LAMMPS atom types to n2p2 elements.

```
pair_coeff * * H O
```

In the above example LAMMPS types 1 and 2 are mapped to the elements "H" and "O" in n2p2, respectively. Multiple types may map to the same element, or some types may not be mapped at all. For example, if the LAMMPS simulation has four atom types, the command

```
pair_coeff * * H H O NULL
```

maps atom types 1 and 2 to the element "H", type 3 to "O" and type 4 is not mapped (indicated by NULL). Atoms mapped to NULL are ignored by the HDNNP calculation, i.e. they do not contribute in any way to the evaluation of HDNNP energies and forces. This may be useful in a setup with *hybrid pair styles*.

---

The mandatory pair style argument *cutoff* must match the short-range cutoff radius of the HDNNP. This corresponds to the maximum cutoff radius of all symmetry functions (the atomic environment descriptors of HDNNPs) used.

**Note**

The cutoff must be given in LAMMPS length units, even if the neural network potential has been trained using a different unit system (see remarks about the *cflength* and *cfenergy* keywords below for details).

The numeric value may be slightly larger than the actual maximum symmetry function cutoff radius (to account for rounding errors when converting units), but must not be smaller.

Use the *dir* keyword to specify the directory containing the HDNNP configuration files. The directory must contain `input.nn` with neural network and symmetry function setup, `scaling.data` with symmetry function scaling data and `weights.???.data` with weight parameters for each element.

The keyword *showew* can be used to turn on/off the display of extrapolation warnings (EWs) which are issued whenever a symmetry function value is out of bounds defined by minimum/maximum values in `scaling.data`. An extrapolation warning may look like this:

```
### NNP EXTRAPOLATION WARNING ### STRUCTURE:      0 ATOM:       119 ELEMENT: Cu SYMFUNC:␣
↪  32 TYPE:  3 VALUE:  2.166E-02 MIN:  2.003E-05 MAX:  1.756E-02
```

stating that the value 2.166E-02 of symmetry function 32 of type 3 (Narrow Angular symmetry function), element Cu (see the log file for a symmetry function listing) was out of bounds (maximum in `scaling.data` is 1.756E-02) for atom 119. Here, the atom index refers to the LAMMPS tag (global index) and the structure index is used to print out the MPI rank the atom belongs to.

**Note**

The *showew* keyword should only be set to *yes* for debugging purposes. Extrapolation warnings may add lots of overhead as they are communicated each timestep. Also, if the simulation is run in a region where the HDNNP was not correctly trained, lots of extrapolation warnings may clog log files and the console. In a production run use *showewsum* instead.

The keyword *showewsum* can be used to get an overview of extrapolation warnings occurring during an MD simulation. The argument specifies the interval at which extrapolation warning summaries are displayed and logged. An EW summary may look like this:

```
### NNP EW SUMMARY ### TS:        100 EW         11 EWPERSTEP  1.100E-01
```

Here, at timestep 100 the occurrence of 11 extrapolation warnings since the last summary is reported, which corresponds to an EW rate of 0.11 per timestep. Setting *showewsum* to 0 deactivates the EW summaries.

A maximum number of allowed extrapolation warnings can be specified with the *maxew* keyword. If the number of EWs exceeds the *maxew* argument the simulation is stopped. Note however that this is merely an approximate threshold since the check is only performed at the end of each timestep and each MPI process counts individually to minimize communication overhead.

The keyword *resetew* alters the behavior of the above mentioned *maxew* threshold. If *resetew* is set to *yes* the threshold is applied on a per-timestep basis and the internal EW counters are reset at the beginning of each timestep. With *resetew* set to *no* the counters accumulate EWs along the whole trajectory.

If the training of a neural network potential has been performed with different physical units for length and energy than those set in LAMMPS, it is still possible to use the potential when the unit conversion factors are provided via the *cflength* and *cfenergy* keywords. If for example, the HDNNP was parameterized with Bohr and Hartree training data and symmetry function parameters (i.e. distances and energies in "input.nn" are given in Bohr and Hartree) but LAMMPS is set to use *metal* units (Angstrom and eV) the correct conversion factors are:

```
cflength 1.8897261328

cfenergy 0.0367493254
```

Thus, arguments of *cflength* and *cfenergy* are the multiplicative factors required to convert lengths and energies given in LAMMPS units to respective quantities in native HDNNP units (1 Angstrom = 1.8897261328 Bohr, 1 eV = 0.0367493254 Hartree).

---

### 4.140.4 Mixing, shift, table, tail correction, restart, rRESPA info

This style does not support mixing. The *pair_coeff* command should only be invoked with asterisk wild cards (see above).

This style does not support the *pair_modify* shift, table, and tail options.

This style does not write information to *binary restart files*. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.140.5 Restrictions

This pair style is part of the ML-HDNNP package. It is only enabled if LAMMPS was built with that package. See the *Build package* doc page for more info.

Please report bugs and feature requests to the n2p2 GitHub issue page.

#### Related commands

*pair_coeff*, *pair_hybrid*, *units*

#### Default

The default options are *dir* = "hdnnp/", *showew* = yes, *showewsum* = 0, *maxew* = 0, *resetew* = no, *cflength* = 1.0, *cfenergy* = 1.0.

---

**(Behler and Parrinello 2007)** Behler, J.; Parrinello, M. Phys. Rev. Lett. 2007, 98 (14), 146401.

**(Singraber, Behler and Dellago 2019)** Singraber, A.; Behler, J.; Dellago, C. J., Chem. Theory Comput. 2019, 15 (3), 1827-1840

**(Singraber et al 2019)** Singraber, A.; Morawietz, T.; Behler, J.; Dellago, C., J. Chem. Theory Comput. 2019, 15 (5), 3075-3092.

## 4.141 pair_style hybrid command

Accelerator Variants: *hybrid/kk*, *hybrid/omp*

## 4.142 pair_style hybrid/molecular command

Accelerator Variants: *hybrid/molecular/omp*

## 4.143 pair_style hybrid/overlay command

Accelerator Variants: *hybrid/overlay/omp*, *hybrid/overlay/kk*

## 4.144 pair_style hybrid/scaled command

Accelerator Variants: *hybrid/scaled/kk*, *hybrid/scaled/omp*

### 4.144.1 Syntax

```
pair_style hybrid style1 args style2 args ...
pair_style hybrid/molecular factor1 style1 args factor2 style 2 args
pair_style hybrid/overlay style1 args style2 args ...
pair_style hybrid/scaled factor1 style1 args factor2 style 2 args ...
```

- style1,style2 = list of one or more pair styles and their arguments
- factor1,factor2 = scale factors for pair styles, may be a variable

### 4.144.2 Examples

```
pair_style hybrid lj/cut/coul/cut 10.0 eam lj/cut 5.0
pair_coeff 1*2 1*2 eam niu3
pair_coeff 3 3 lj/cut/coul/cut 1.0 1.0
pair_coeff 1*2 3 lj/cut 0.5 1.2

pair_style hybrid/overlay lj/cut 2.5 coul/long 2.0
pair_coeff * * lj/cut 1.0 1.0
pair_coeff * * coul/long

pair_style hybrid/scaled 0.5 tersoff 0.5 sw
pair_coeff * * tersoff Si.tersoff Si
pair_coeff * * sw Si.sw Si

pair_style hybrid/molecular lj/cut 2.5 lj/cut 2.5
pair_coeff * * lj/cut 1 1.0 1.0
pair_coeff * * lj/cut 2 1.5 1.0

variable one equal ramp(1.0,0.0)
variable two equal 1.0-v_one
pair_style hybrid/scaled v_one lj/cut 2.5 v_two morse 2.5
pair_coeff 1 1 lj/cut 1.0 1.0 2.5
pair_coeff 1 1 morse 1.0 1.0 1.0 2.5
```

(continues on next page)

(continued from previous page)

```
variable peratom1 atom 1/(1+exp(-$k*vx^2)
variable peratom2 atom 1-v_peratom1
pair_style hybrid/scaled v_peratom1 lj/cut 2.5 v_peratom2 morse 2.5
pair_coeff 1 1 lj/cut 1.0 1.0 2.5
pair_coeff 1 1 morse 1.0 1.0 1.0 2.5
```

### 4.144.3 Description

The *hybrid*, *hybrid/overlay*, *hybrid/molecular*, and *hybrid/scaled* styles enable the use of multiple pair styles in one simulation. With the *hybrid* style, exactly one pair style is assigned to each pair of atom types. With the *hybrid/overlay* and *hybrid/scaled* styles, one or more pair styles can be assigned to each pair of atom types. With the *hybrid/molecular* style, pair styles are assigned to either intra- or inter-molecular interactions.

The assignment of pair styles to type pairs is made via the *pair_coeff* command. The major difference between the *hybrid/overlay* and *hybrid/scaled* styles is that the *hybrid/scaled* adds a scale factor for each sub-style contribution to forces, energies and stresses. Because of the added complexity, the *hybrid/scaled* style has more overhead and thus may be slower than *hybrid/overlay*.

The *hybrid/molecular* pair style accepts *only* two sub-styles: the first is assigned to intra-molecular interactions (i.e. both atoms have the same molecule ID), the second to inter-molecular interactions (i.e. interacting atoms have different molecule IDs).

**When NOT to use a hybrid pair style**

Using pair style *hybrid* can be very tempting to use if you need a **many-body potential** supporting a mix of elements for which you cannot find a potential file that covers *all* of them. Regardless of how this is set up, there will be *errors*. The major use case where the error is *small*, is when the many-body sub-styles are used on different objects (for example a slab and a liquid, a metal and a nano-machining work piece). In that case the *mixed* terms **should** be provided by a pair-wise additive potential (like Lennard-Jones or Morse) to avoid unexpected behavior and reduce errors. LAMMPS cannot easily check for this condition and thus will accept good and bad choices alike.

Outside of this, we *strongly* recommend *against* using pair style hybrid with many-body potentials for the following reasons:

1. When trying to combine EAM or MEAM potentials, there is a *large* error in the embedding term, since it is computed separately for each sub-style only.
2. When trying to combine many-body potentials like Stillinger-Weber, Tersoff, AIREBO, Vashishta, or similar, you have to understand that the potential of a sub-style cannot be applied in a pair-wise fashion but will need to be applied to multiples of atoms (e.g. a Tersoff potential of elements A and B includes the interactions A-A, B-B, A-B, A-A-A, A-A-B, A-B-B, A-B-A, B-A-A, B-A-B, B-B-A, B-B-B; AIREBO also considers all quadruples of atom elements).
3. When one of the sub-styles uses charge-equilibration (= QEq; like in ReaxFF or COMB) you have inconsistent QEq behavior because either you try to apply QEq to *all* atoms but then you are missing the QEq parameters for the non-QEq pair style (and it would be inconsistent to apply QEq for pair styles that are not parameterized for QEq) or else you would have either no charges or fixed charges interacting with the QEq which also leads to inconsistent behavior between two sub-styles. When attempting to use multiple ReaxFF instances to combine different potential files, you might be able to work around the QEq limitations, but point 2. still applies.

We understand that it is frustrating to not be able to run simulations due to lack of available potential files, but that does not justify combining potentials in a broken way via pair style hybrid. This is not what the hybrid pair styles are designed for.

Here are two examples of hybrid simulations. The *hybrid* style could be used for a simulation of a metal droplet on a LJ surface. The metal atoms interact with each other via an *eam* potential, the surface atoms interact with each other via a *lj/cut* potential, and the metal/surface interaction is also computed via a *lj/cut* potential. The *hybrid/overlay* style could be used as in the second example above, where multiple potentials are superimposed in an additive fashion to compute the interaction between atoms. In this example, using *lj/cut* and *coul/long* together gives the same result as if the *lj/cut/coul/long* potential were used by itself. In this case, it would be more efficient to use the single combined potential, but in general any combination of pair potentials can be used together in to produce an interaction that is not encoded in any single pair_style file, e.g. adding Coulombic forces between granular particles. Another limitation of using the *hybrid/overlay* variant, that it does not generate *lj/cut* parameters for mixed atom types from a mixing rule due to restrictions discussed below.

If the *hybrid/scaled* style is used instead of *hybrid/overlay*, contributions from sub-styles are weighted by their scale factors, which may be fractional or even negative. Furthermore the scale factor for each sub-style may a constant, an *equal* style variable, or an *atom* style variable. Variable scale factors may change during the simulation. Different sub-styles may use different scale factor styles. In the case of a sub-style scale factor that is an *atom* style variable, the force contribution to each atom from that sub-style is weighted by the value of the variable for that atom, while the contribution from that sub-style to the global potential energy is zero. All other contributions to the per-atom energy, per-atom virial, and global virial (if not obtained from forces) from that sub-style are zero. This enables switching smoothly between two different pair styles or two different parameter sets during a run in a similar fashion as could be done with *fix adapt* or *fix alchemy*. All pair styles that will be used are listed as "sub-styles" following the *hybrid* or *hybrid/overlay* keyword, in any order. In case of the *hybrid/scaled* pair style, each sub-style is prefixed with a scale factor. The scale factor is either a floating point number or an *equal* or *atom* style (or equivalent) variable. Each sub-style's name is followed by its usual arguments, as illustrated in the examples above. See the doc pages of the individual pair styles for a listing and explanation of the appropriate arguments for them.

Note that an individual pair style can be used multiple times as a sub-style. For efficiency reasons this should only be done if your model requires it. E.g. if you have different regions of Si and C atoms and wish to use a Tersoff potential for pure Si for one set of atoms, and a Tersoff potential for pure C for the other set (presumably with some third potential for Si-C interactions), then the sub-style *tersoff* could be listed twice. But if you just want to use a Lennard-Jones or other pairwise potential for several different atom type pairs in your model, then you should just list the sub-style once and use the pair_coeff command to assign parameters for the different type pairs.

**Note**

There is one exception to this option to list an individual pair style multiple times: GPU-enabled pair styles in the GPU package. This is because the GPU package currently assumes that only one instance of a pair style is being used.

In the pair_coeff commands, the name of a pair style must be added after the I,J type specification, with the remaining coefficients being those appropriate to that style. If the pair style is used multiple times in the pair_style command, then an additional numeric argument must also be specified which is a number from 1 to M where M is the number of times the sub-style was listed in the pair style command. The extra number indicates which instance of the sub-style these coefficients apply to.

For example, consider a simulation with 3 atom types: types 1 and 2 are Ni atoms, type 3 are LJ atoms with charges. The following commands would set up a hybrid simulation:

```
pair_style hybrid eam/alloy lj/cut/coul/cut 10.0 lj/cut 8.0
pair_coeff * * eam/alloy nialhjea Ni Ni NULL
pair_coeff 3 3 lj/cut/coul/cut 1.0 1.0
pair_coeff 1*2 3 lj/cut 0.8 1.3
```

As an example of using the same pair style multiple times, consider a simulation with 2 atom types. Type 1 is Si, type 2 is C. The following commands would model the Si atoms with Tersoff, the C atoms with Tersoff, and the cross-interactions with Lennard-Jones:

```
pair_style hybrid lj/cut 2.5 tersoff tersoff
pair_coeff * * tersoff 1 Si.tersoff Si NULL
pair_coeff * * tersoff 2 C.tersoff NULL C
pair_coeff 1 2 lj/cut 1.0 1.5
```

It is not recommended to read pair coefficients for a hybrid style from a "Pair Coeffs" or "PairIJ Coeffs" section of a data file via the *read_data* command, since those sections expect a fixed number of lines, either one line per atom type or one line pair pair of atom types, respectively. When reading from a data file, the lines of the "Pair Coeffs" and "PairIJ Coeffs" are changed in the same way as the *pair_coeff* command, i.e. the name of the pair style to which the parameters apply must follow the atom type (or atom types), e.g.

```
Pair Coeffs

1 lj/cut/coul/cut 1.0 1.0
...

PairIJ Coeffs

1 1 lj/cut/coul/cut 1.0 1.0
...
```

Note that the pair_coeff command for some potentials such as *pair_style eam/alloy* includes a mapping specification of elements to all atom types, which in the hybrid case, can include atom types not assigned to the *eam/alloy* potential. The NULL keyword is used by many such potentials (eam/alloy, Tersoff, AIREBO, etc), to denote an atom type that will be assigned to a different sub-style.

For the *hybrid* style, each atom type pair I,J is assigned to exactly one sub-style. Just as with a simulation using a single pair style, if you specify the same atom type pair in a second pair_coeff command, the previous assignment will be overwritten.

For the *hybrid/overlay* and *hybrid/scaled* styles, each atom type pair I,J can be assigned to one or more sub-styles. If you specify the same atom type pair in a second pair_coeff command with a new sub-style, then the second sub-style is added to the list of potentials that will be calculated for two interacting atoms of those types. If you specify the same atom type pair in a second pair_coeff command with a sub-style that has already been defined for that pair of atoms, then the new pair coefficients simply override the previous ones, as in the normal usage of the pair_coeff command. E.g. these two sets of commands are the same:

```
pair_style lj/cut 2.5
pair_coeff * * 1.0 1.0
pair_coeff 2 2 1.5 0.8

pair_style hybrid/overlay lj/cut 2.5
pair_coeff * * lj/cut 1.0 1.0
pair_coeff 2 2 lj/cut 1.5 0.8
```

Coefficients must be defined for each pair of atoms types via the *pair_coeff* command as described above, or in the "Pair Coeffs" or "PairIJ Coeffs" section of the data file read by the *read_data* command, or by mixing as described below.

For all of the *hybrid*, *hybrid/overlay*, and *hybrid/scaled* styles, every atom type pair I,J (where I <= J) must be assigned to at least one sub-style via the *pair_coeff* command as in the examples above, or in the data file read by the *read_data*, or by mixing as described below. Also all sub-styles must be used at least once in a *pair_coeff* command.

**Warning**

With hybrid pair styles the use of mixing to generate pair coefficients is significantly limited compared to the individual pair styles. LAMMPS **never** performs mixing of parameters from different sub-styles, **even** if they use the same type of coefficients, e.g. contain a Lennard-Jones potential variant. Those parameters must be provided explicitly. Also for *hybrid/overlay* and *hybrid/scaled* mixing is **only** performed for pairs of atom types for which only a single pair style is assigned.

Thus it is strongly recommended to provide all mixed terms explicitly. For non-hybrid styles those could be generated and written out using the *write_coeff command* and then edited as needed to comply with the requirements for hybrid styles as explained above.

If you want there to be no interactions between a particular pair of atom types, you have 3 choices. You can assign the pair of atom types to some sub-style and use the *neigh_modify exclude type* command. You can assign it to some sub-style and set the coefficients so that there is effectively no interaction (e.g. epsilon = 0.0 in a LJ potential). Or, for *hybrid*, *hybrid/overlay*, or *hybrid/scaled* simulations, you can use this form of the pair_coeff command in your input script or the "PairIJ Coeffs" section of your data file:

```
pair_coeff  2 3  none
```

or this form in the "Pair Coeffs" section of the data file:

```
3  none
```

If an assignment to *none* is made in a simulation with the *hybrid/overlay* or *hybrid/scaled* pair style, it wipes out all previous assignments of that pair of atom types to sub-styles.

Note that you may need to use an *atom_style* hybrid command in your input script, if atoms in the simulation will need attributes from several atom styles, due to using multiple pair styles with different requirements.

---

Different force fields (e.g. CHARMM vs. AMBER) may have different rules for applying exclusions or weights that change the strength of pairwise non-bonded interactions between pairs of atoms that are also 1-2, 1-3, and 1-4 neighbors in the molecular bond topology. This is normally a global setting defined the *special_bonds* command. However, different weights can be assigned to different hybrid sub-styles via the *pair_modify special* command. This allows multiple force fields to be used in a model of a hybrid system, however, there is no consistent approach to determine parameters automatically for the interactions **between** atoms of the two force fields, thus this approach this is only recommended when particles described by the different force fields do not mix.

Here is an example for combining CHARMM and AMBER: The global *amber* setting sets the 1-4 interactions to non-zero scaling factors and then overrides them with 0.0 only for CHARMM:

```
special_bonds amber
pair_style hybrid lj/charmm/coul/long 8.0 10.0 lj/cut/coul/long 10.0
pair_modify pair lj/charmm/coul/long special lj/coul 0.0 0.0 0.0
```

This input achieves the same effect:

```
special_bonds lj/coul 0.0 0.0 0.1
pair_style hybrid lj/charmm/coul/long 8.0 10.0 lj/cut/coul/long 10.0
pair_modify pair lj/cut/coul/long special lj 0.0 0.0 0.5
pair_modify pair lj/cut/coul/long special coul 0.0 0.0 0.83333333
pair_modify pair lj/charmm/coul/long special lj/coul 0.0 0.0 0.0
```

Here is an example for combining Tersoff with OPLS/AA based on a data file that defines bonds for all atoms where - for the Tersoff part of the system - the force constants for the bonded interactions have been set to 0. Note the global settings are effectively *lj/coul 0.0 0.0 0.5* as required for OPLS/AA:

```
special_bonds lj/coul 1e-20 1e-20 0.5
pair_style hybrid tersoff lj/cut/coul/long 12.0
pair_modify pair tersoff special lj/coul 1.0 1.0 1.0
```

For use with the various *compute */tally* computes, the *pair_modify compute/tally* command can be used to selectively turn off processing of the compute tally styles, for example, if those pair styles (e.g. many-body styles) do not support this feature.

See the *pair_modify* page for details on the specific syntax, requirements and restrictions.

---

The potential energy contribution to the overall system due to an individual sub-style can be accessed and output via the *compute pair* command. Note that in the case of pair style *hybrid/scaled* this is the **unscaled** potential energy of the selected sub-style.

---

Even though the command name "pair_style" would suggest that these are pair-wise interactions, several of the potentials defined via the pair_style command in LAMMPS are really many-body potentials, such as Tersoff, AIREBO, MEAM, ReaxFF, etc. The way to think about using these potentials in a hybrid setting is as follows.

A subset of atom types is assigned to the many-body potential with a single *pair_coeff* command, using "* *" to include all types and the NULL keywords described above to exclude specific types not assigned to that potential. If types 1,3,4 were assigned in that way (but not type 2), this means that all many-body interactions between all atoms of types 1,3,4 will be computed by that potential. Pair_style hybrid allows interactions between type pairs 2-2, 1-2, 2-3, 2-4 to be specified for computation by other pair styles. You could even add a second interaction for 1-1 to be computed by another pair style, assuming pair_style *hybrid/overlay* is used.

But you should not, as a general rule, attempt to exclude the many-body interactions for some subset of the type pairs within the set of 1,3,4 interactions, e.g. exclude 1-1 or 1-3 interactions. That is not conceptually well-defined for many-body interactions, since the potential will typically calculate energies and foces for small groups of atoms, e.g. 3 or 4 atoms, using the neighbor lists of the atoms to find the additional atoms in the group.

However, you can still use the pair_coeff none setting or the *neigh_modify exclude* command to exclude certain type pairs from the neighbor list that will be passed to a many-body sub-style. This will alter the calculations made by a many-body potential beyond the specific pairs, since it builds its list of 3-body, 4-body, etc interactions from the pair lists. You will need to think **carefully** as to whether excluding such pairs produces a physically meaningful result for your model.

For example, imagine you have two atom types in your model, type 1 for atoms in one surface, and type 2 for atoms in the other, and you wish to use a Tersoff potential to compute interactions within each surface, but not between the surfaces. Then either of these two command sequences would implement that model:

```
pair_style hybrid tersoff
pair_coeff * * tersoff SiC.tersoff C C
pair_coeff 1 2 none

pair_style tersoff
pair_coeff * * SiC.tersoff C C
neigh_modify exclude type 1 2
```

Either way, only neighbor lists with 1-1 or 2-2 interactions would be passed to the Tersoff potential, which means it would compute no 3-body interactions containing both type 1 and 2 atoms.

Here is another example to use 2 many-body potentials together in an overlapping manner using *hybrid/overlay*. Imagine you have CNT (C atoms) on a Si surface. You want to use Tersoff for Si/Si and Si/C interactions, and AIREBO for C/C interactions. Si atoms are type 1; C atoms are type 2. Something like this will work:

```
pair_style hybrid/overlay tersoff airebo 3.0
pair_coeff * * tersoff SiC.tersoff.custom Si C
pair_coeff * * airebo CH.airebo NULL C
```

Note that to prevent the Tersoff potential from computing C/C interactions, you would need to **modify** the SiC.tersoff potential file to turn off C/C interaction, i.e. by setting the appropriate coefficients to 0.0.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

**Note**

Since the *hybrid*, *hybrid/overlay*, *hybrid/scaled* styles delegate computation to the individual sub-styles, the suffix versions of the *hybrid* and *hybrid/overlay* styles are used to propagate the corresponding suffix to all sub-styles, if those versions exist. Otherwise the non-accelerated version will be used. The individual accelerated sub-styles are part of the GPU, KOKKOS, INTEL, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

---

### 4.144.4 Mixing, shift, table, tail correction, restart, rRESPA info

Any pair potential settings made via the *pair_modify* command are passed along to all sub-styles of the hybrid potential.

For atom type pairs I,J and I != J, if the sub-style assigned to I,I and J,J is the same, and if the sub-style allows for mixing, then the coefficients for I,J can be mixed. This means you do not have to specify a pair_coeff command for I,J since the I,J type pair will be assigned automatically to the sub-style defined for both I,I and J,J and its coefficients generated by the mixing rule used by that sub-style. For the *hybrid/overlay* and *hybrid/scaled* style, there is an additional requirement that both the I,I and J,J pairs are assigned to a single sub-style. If this requirement is not met, no I,J coeffs will be generated, even if the sub-styles support mixing, and I,J pair coefficients must be explicitly defined.

See the *pair_modify* command for details of mixing rules. See the See the page for the sub-style to see if allows for mixing.

The hybrid pair styles supports the *pair_modify* shift, table, and tail options for an I,J pair interaction, if the associated sub-style supports it.

For the hybrid pair styles, the list of sub-styles and their respective settings are written to *binary restart files*, so a *pair_style* command does not need to specified in an input script that reads a restart file. However, the coefficient information is not stored in the restart file. The same is true for *data files*. Thus, pair_coeff commands need to be re-specified in the restart input script. For pair style *hybrid/scaled* also the names of any variables used as scale factors are restored, but not the variables themselves, so those may need to be redefined when continuing from a restart.

These pair styles support the use of the *inner*, *middle*, and *outer* keywords of the *run_style respa* command, if their sub-styles do.

### 4.144.5 Restrictions

When using a long-range Coulombic solver (via the *kspace_style* command) with a hybrid pair_style, one or more sub-styles will be of the "long" variety, e.g. *lj/cut/coul/long* or *buck/coul/long*. You must ensure that the short-range Coulombic cutoff used by each of these long pair styles is the same or else LAMMPS will generate an error.

Pair style *hybrid/molecular* is not compatible with manybody potentials.

When using pair styles from the GPU package they must not be listed multiple times. LAMMPS will detect this and abort.

### 4.144.6 Related commands

*pair_coeff*

### 4.144.7 Default

none

## 4.145 pair_style ilp/graphene/hbn command

Accelerator Variants: *ilp/graphene/hbn/opt*

### 4.145.1 Syntax

```
pair_style [hybrid/overlay ...] ilp/graphene/hbn cutoff tap_flag
```

- cutoff = global cutoff (distance units)
- tap_flag = 0/1 to turn off/on the taper function

### 4.145.2 Examples

```
pair_style  hybrid/overlay ilp/graphene/hbn 16.0 1
pair_coeff  * * ilp/graphene/hbn  BNCH.ILP B N C

pair_style  hybrid/overlay rebo tersoff ilp/graphene/hbn 16.0 coul/shield 16.0
pair_coeff  * * rebo              CH.rebo     NULL NULL C
pair_coeff  * * tersoff           BNC.tersoff B    N    NULL
pair_coeff  * * ilp/graphene/hbn  BNCH.ILP    B    N    C
pair_coeff  1 1 coul/shield 0.70
pair_coeff  1 2 coul/shield 0.695
pair_coeff  2 2 coul/shield 0.69
```

### 4.145.3 Description

The *ilp/graphene/hbn* style computes the registry-dependent interlayer potential (ILP) potential as described in *(Leven1)*, *(Leven2)* and *(Maaravi)*. The normals are calculated in the way as described in *(Kolmogorov)*.

$$
\begin{aligned}
E =& \frac{1}{2}\sum_i \sum_{j \neq i} V_{ij} \\
V_{ij} =& \mathrm{Tap}(r_{ij}) \left\{ e^{-\alpha(r_{ij}/\beta - 1)} \left[\varepsilon + f(\rho_{ij}) + f(\rho_{ji})\right] - \frac{1}{1 + e^{-d\left[(r_{ij}/(s_R \cdot r^{eff})) - 1\right]}} \cdot \frac{C_6}{r_{ij}^6} \right\} \\
\rho_{ij}^2 =& r_{ij}^2 - (\mathbf{r}_{ij} \cdot \mathbf{n}_i)^2 \\
\rho_{ji}^2 =& r_{ij}^2 - (\mathbf{r}_{ij} \cdot \mathbf{n}_j)^2 \\
f(\rho) =& C e^{-(\rho/\delta)^2} \\
\mathrm{Tap}(r_{ij}) =& 20 \left(\frac{r_{ij}}{R_{cut}}\right)^7 - 70 \left(\frac{r_{ij}}{R_{cut}}\right)^6 + 84 \left(\frac{r_{ij}}{R_{cut}}\right)^5 - 35 \left(\frac{r_{ij}}{R_{cut}}\right)^4 + 1
\end{aligned}
$$

Where $\mathrm{Tap}(r_{ij})$ is the taper function which provides a continuous cutoff (up to third derivative) for interatomic separations larger than $r_c$ *(Maaravi)*. The definitions of each parameter in the above equation can be found in *(Leven1)* and *(Maaravi)*.

It is important to include all the pairs to build the neighbor list for calculating the normals.

**Note**

This potential (ILP) is intended for interlayer interactions between two different layers of graphene, hexagonal boron nitride (h-BN) and their hetero-junction. To perform a realistic simulation, this potential must be used in combination with intralayer potential, such as *AIREBO* or *Tersoff* potential. To keep the intralayer properties unaffected, the interlayer interaction within the same layers should be avoided. Hence, each atom has to have a layer identifier such that atoms residing on the same layer interact via the appropriate intralayer potential and atoms residing on different layers interact via the ILP. Here, the molecule id is chosen as the layer identifier, thus a data file with the "full" atom style is required to use this potential.

The parameter file (e.g. BNCH.ILP), is intended for use with *metal* *units*, with energies in meV. Two additional parameters, *S*, and *rcut* are included in the parameter file. *S* is designed to facilitate scaling of energies. *rcut* is designed to build the neighbor list for calculating the normals for each atom pair.

**Note**

The parameters presented in the parameter file (e.g. BNCH.ILP), are fitted with taper function by setting the cutoff equal to 16.0 Angstrom. Using different cutoff or taper function should be careful. The parameters for atoms pairs between Boron and Nitrogen are fitted with a screened Coulomb interaction *coul/shield*. Therefore, to simulated the properties of h-BN correctly, this potential must be used in combination with the pair style *coul/shield*.

**Note**

Four new sets of parameters of ILP for 2D layered Materials with bilayer and bulk configurations are presented in *(Ouyang1)* and *(Ouyang2)*, respectively. These parameters provide a good description in both short- and long-range interaction regimes. While the old ILP parameters published in *(Leven2)* and *(Maaravi)* are only suitable for long-range interaction regime. This feature is essential for simulations in high pressure regime (i.e., the interlayer distance is smaller than the equilibrium distance). The benchmark tests and comparison of these parameters can be found in *(Ouyang1)* and *(Ouyang2)*.

This potential must be used in combination with hybrid/overlay. Other interactions can be set to zero using pair_style *none*.

This pair style tallies a breakdown of the total interlayer potential energy into sub-categories, which can be accessed via the *compute pair* command as a vector of values of length 2. The 2 values correspond to the following sub-categories:

1. *E_vdW* = vdW (attractive) energy
2. *E_Rep* = Repulsive energy

To print these quantities to the log file (with descriptive column headings) the following commands could be included in an input script:

```
compute 0 all pair ilp/graphene/hbn
variable Evdw  equal c_0[1]
variable Erep  equal c_0[2]
thermo_style custom step temp epair v_Erep v_Evdw
```

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.145.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support the pair_modify mix, shift, table, and tail options.

This pair style does not write their information to binary restart files, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

### 4.145.5 Restrictions

This pair style is part of the INTERLAYER package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires the newton setting to be *on* for pair interactions.

The BNCH.ILP potential file provided with LAMMPS (see the potentials directory) are parameterized for *metal* units. You can use this potential with any LAMMPS units, but you would need to create your own custom BNCH.ILP potential file with coefficients listed in the appropriate units, if your simulation does not use *metal* units.

### 4.145.6 Related commands

*pair_coeff*, *pair_none*, *pair_style hybrid/overlay*, *pair_style drip*, *pair_style ilp_tmd*, *pair_style saip_metal*, *pair_style pair_kolmogorov_crespi_z*, *pair_style pair_kolmogorov_crespi_full*, *pair_style pair_lebedeva_z*, *pair_style pair_coul_shield*.

### 4.145.7 Default

tap_flag = 1

---

**(Ouyang1)** W. Ouyang, D. Mandelli, M. Urbakh and O. Hod, Nano Lett. 18, 6009-6016 (2018).

**(Ouyang2)** W. Ouyang et al., J. Chem. Theory Comput. 16(1), 666-676 (2020).

**(Leven1)** I. Leven, I. Azuri, L. Kronik and O. Hod, J. Chem. Phys. 140, 104106 (2014).

**(Leven2)** I. Leven et al, J. Chem.Theory Comput. 12, 2896-905 (2016).

**(Maaravi)** T. Maaravi et al, J. Phys. Chem. C 121, 22826-22835 (2017).

**(Kolmogorov)** A. N. Kolmogorov, V. H. Crespi, Phys. Rev. B 71, 235415 (2005).

## 4.146 pair_style ilp/tmd command

Accelerator Variants: *ilp/tmd/opt*

### 4.146.1 Syntax

```
pair_style [hybrid/overlay ...] ilp/tmd cutoff tap_flag
```

- cutoff = global cutoff (distance units)
- tap_flag = 0/1 to turn off/on the taper function

### 4.146.2 Examples

```
pair_style  hybrid/overlay ilp/tmd 16.0 1
pair_coeff  * * ilp/tmd  TMD.ILP Mo S S

pair_style  hybrid/overlay sw/mod sw/mod ilp/tmd 16.0
pair_coeff  * * sw/mod 1  tmd.sw.mod Mo S S NULL NULL NULL
pair_coeff  * * sw/mod 2  tmd.sw.mod NULL NULL NULL W Se Se
pair_coeff  * * ilp/tmd   TMD.ILP   Mo S S W Se Se
```

### 4.146.3 Description

Added in version 17Feb2022.

The *ilp/tmd* style computes the registry-dependent interlayer potential (ILP) potential for heterointerfaces of transition metal dichalcogenides (TMD)/TMD, TMD/graphene, and TMD / hexagonal boron nitride (h-BN), as described in

*(Ouyang7)*, *(Jiang4)* and *(Jiang5)*.

$$
\begin{aligned}
E =& \frac{1}{2}\sum_i \sum_{j \neq i} V_{ij} \\
V_{ij} =& \mathrm{Tap}(r_{ij}) \left\{ e^{-\alpha(r_{ij}/\beta - 1)} \left[\varepsilon + f(\rho_{ij}) + f(\rho_{ji})\right] - \frac{1}{1 + e^{-d\left[(r_{ij}/(s_R \cdot r^{eff})) - 1\right]}} \cdot \frac{C_6}{r_{ij}^6} \right\} \\
\rho_{ij}^2 =& r_{ij}^2 - (\mathbf{r}_{ij} \cdot \mathbf{n}_i)^2 \\
\rho_{ji}^2 =& r_{ij}^2 - (\mathbf{r}_{ij} \cdot \mathbf{n}_j)^2 \\
f(\rho) =& C e^{-(\rho/\delta)^2} \\
\mathrm{Tap}(r_{ij}) =& 20\left(\frac{r_{ij}}{R_{cut}}\right)^7 - 70\left(\frac{r_{ij}}{R_{cut}}\right)^6 + 84\left(\frac{r_{ij}}{R_{cut}}\right)^5 - 35\left(\frac{r_{ij}}{R_{cut}}\right)^4 + 1
\end{aligned}
$$

Where $\mathrm{Tap}(r_{ij})$ is the taper function which provides a continuous cutoff (up to third derivative) for interatomic separations larger than $r_c$ *pair_style ilp_graphene_hbn*.

It is important to include all the pairs to build the neighbor list for calculating the normals.

**Note**

Since each MX2 (M = Mo, W and X = S, Se Te) layer contains two sub-layers of X atoms and one sub-layer of M atoms, the definition of the normal vectors used for graphene and h-BN is no longer valid for TMDs. In *(Ouyang7)*, a new definition is proposed, where for each atom *i*, its six nearest neighboring atoms belonging to the same sub-layer are chosen to define the normal vector *mathbf{n}_i*.

The parameter file (e.g. TMD.ILP), is intended for use with *metal* *units*, with energies in meV. Two additional parameters, *S*, and *rcut* are included in the parameter file. *S* is designed to facilitate scaling of energies. *rcut* is designed to build the neighbor list for calculating the normals for each atom pair.

**Note**

The parameters presented in the parameter file (e.g. TMD.ILP), are fitted with taper function by setting the cutoff equal to 16.0 Angstrom. Using different cutoff or taper function should be careful. These parameters provide a good description in both short- and long-range interaction regimes. This feature is essential for simulations in high pressure regime (i.e., the interlayer distance is smaller than the equilibrium distance). The benchmark tests and comparison of these parameters can be found in *(Ouyang7)*.

This potential must be used in combination with hybrid/overlay. Other interactions can be set to zero using pair_style *none*.

This pair style tallies a breakdown of the total interlayer potential energy into sub-categories, which can be accessed via the *compute pair* command as a vector of values of length 2. The 2 values correspond to the following sub-categories:

1. *E_vdW* = vdW (attractive) energy
2. *E_Rep* = Repulsive energy

To print these quantities to the log file (with descriptive column headings) the following commands could be included in an input script:

```
compute 0 all pair ilp/tmd
variable Evdw  equal c_0[1]
```

(continues on next page)

(continued from previous page)

```
variable Erep  equal c_0[2]
thermo_style custom step temp epair v_Erep v_Evdw
```

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.146.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support the pair_modify mix, shift, table, and tail options.

This pair style does not write their information to binary restart files, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

### 4.146.5 Restrictions

This pair style is part of the INTERLAYER package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires the newton setting to be *on* for pair interactions.

The TMD.ILP potential file provided with LAMMPS (see the potentials directory) are parameterized for *metal* units. You can use this potential with any LAMMPS units, but you would need to create your own custom TMD.ILP potential file with coefficients listed in the appropriate units, if your simulation does not use *metal* units.

### 4.146.6 Related commands

*pair_coeff*, *pair_none*, *pair_style hybrid/overlay*, *pair_style drip*, *pair_style saip_metal*, *pair_style ilp_graphene_hbn*, *pair_style pair_kolmogorov_crespi_z*, *pair_style pair_kolmogorov_crespi_full*, *pair_style pair_lebedeva_z*, *pair_style pair_coul_shield*.

### 4.146.7 Default

tap_flag = 1

---

**(Ouyang7)** W. Ouyang, et al., J. Chem. Theory Comput. 17, 7237 (2021).

**(Jiang4)** W. Jiang, … , W. Ouyang, et al., J. Phys. Chem. A, 127, 46, 9820-9830 (2023).

**(Jiang5)** W. Jiang, …, W. Ouyang, J. Phys. Chem. C, 129, 2, 1417-1427 (2024).

# 4.147 pair_style kim command

## 4.147.1 Syntax

```
pair_style kim model
```

model = name of a KIM model (the KIM ID for models archived in OpenKIM)

## 4.147.2 Examples

```
pair_style kim SW_StillingerWeber_1985_Si__MO_405512056662_005
pair_coeff * * Si
```

## 4.147.3 Description

This pair style is a wrapper on the Open Knowledgebase of Interatomic Models (OpenKIM) repository of interatomic potentials to enable their use in LAMMPS scripts.

The preferred interface for using interatomic models archived in OpenKIM is the *kim command* interface. That interface supports both "KIM Portable Models" (PMs) that conform to the KIM API Portable Model Interface (PMI) and can be used by any simulation code that conforms to the KIM API/PMI, and "KIM Simulator Models" (SMs) that are natively implemented within a single simulation code (like LAMMPS) and can only be used with it. The *pair_style kim* command is limited to KIM PMs. It is used by the *kim command* interface as needed.

**Note**

Since *pair_style kim* is called by *kim interactions* as needed, it is not recommended to be directly used in input scripts.

---

The argument *model* is the name of the KIM PM. For potentials archived in OpenKIM this is the extended KIM ID (see *kim command* for details). LAMMPS can invoke any KIM PM, however there can be incompatibilities (for example due to unit matching issues). In the event of an incompatibility, the code will terminate with an error message. Check both the LAMMPS and KIM log files for details.

Only a single *pair_coeff* command is used with the *kim* style, which specifies the mapping of LAMMPS atom types to the species supported by the KIM PM. This is done by specifying *N* additional arguments after the * * in the *pair_coeff* command, where *N* is the number of LAMMPS atom types:

- N element names = mapping of KIM elements to atom types

For example, consider a KIM PM that supports Si and C species. If the LAMMPS simulation has four atom types, where the first three are Si, and the fourth is C, the following *pair_coeff* command would be used:

```
pair_coeff * * Si Si Si C
```

The first two arguments must be * * so as to span all LAMMPS atom types. The first three Si arguments map LAMMPS atom types 1, 2, and 3 to Si as defined within KIM PM. The final C argument maps LAMMPS atom type 4 to C.

---

In addition to the usual LAMMPS error messages, the KIM library itself may generate errors, which should be printed to the screen. In this case it is also useful to check the *kim.log* file for additional error information. The file *kim.log* should be generated in the same directory where LAMMPS is running.

To download, build, and install the KIM library on your system, see the *lib/kim/README* file. Once you have done this and built LAMMPS with the KIM package installed you can run the example input scripts in *examples/kim*.

---

### 4.147.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support the *pair_modify* mix, shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since KIM stores the potential parameters. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.147.5 Restrictions

This pair style is part of the KIM package. See details on restrictions in *kim command*.

This current version of pair_style kim is compatible with the kim-api package version 2.0.0 and higher.

### 4.147.6 Related commands

*pair_coeff* , *kim command*

### 4.147.7 Default

none

## 4.148 pair_style kolmogorov/crespi/full command

### 4.148.1 Syntax

```
pair_style hybrid/overlay kolmogorov/crespi/full cutoff tap_flag
```

- cutoff = global cutoff (distance units)
- tap_flag = 0/1 to turn off/on the taper function

### 4.148.2 Examples

```
pair_style hybrid/overlay kolmogorov/crespi/full 20.0 0
pair_coeff * * none
pair_coeff * * kolmogorov/crespi/full  CH.KC   C C

pair_style hybrid/overlay rebo kolmogorov/crespi/full 16.0 1
pair_coeff * * rebo                    CH.rebo      C H
pair_coeff * * kolmogorov/crespi/full  CH_taper.KC  C H
```

### 4.148.3 Description

The *kolmogorov/crespi/full* style computes the Kolmogorov-Crespi (KC) interaction potential as described in *(Kolmogorov)*. No simplification is made,

$$
\begin{aligned}
E =& \frac{1}{2}\sum_i \sum_{j \neq i} V_{ij} \\
V_{ij} =& e^{-\lambda(r_{ij}-z_0)}\left[C + f(\rho_{ij}) + f(\rho_{ji})\right] - A\left(\frac{r_{ij}}{z_0}\right)^{-6} \\
\rho_{ij}^2 =& r_{ij}^2 - (\mathbf{r}_{ij}\cdot\mathbf{n}_i)^2 \\
\rho_{ji}^2 =& r_{ij}^2 - (\mathbf{r}_{ij}\cdot\mathbf{n}_j)^2 \\
f(\rho) =& e^{-(\rho/\delta)^2}\sum_{n=0}^{2} C_{2n}(\rho/\delta)^{2n}
\end{aligned}
$$

It is important to have a sufficiently large cutoff to ensure smooth forces and to include all the pairs to build the neighbor list for calculating the normals. Energies are shifted so that they go continuously to zero at the cutoff assuming that the exponential part of $V_{ij}$ (first term) decays sufficiently fast. This shift is achieved by the last term in the equation for $V_{ij}$ above. This is essential only when the tapper function is turned off. The formula of taper function can be found in pair style *ilp/graphene/hbn*.

**Note**

This potential (ILP) is intended for interlayer interactions between two different layers of graphene. To perform a realistic simulation, this potential must be used in combination with intralayer potential, such as *AIREBO* or *Tersoff* potential. To keep the intralayer properties unaffected, the interlayer interaction within the same layers should be avoided. Hence, each atom has to have a layer identifier such that atoms residing on the same layer interact via the appropriate intralayer potential and atoms residing on different layers interact via the ILP. Here, the molecule id is chosen as the layer identifier, thus a data file with the "full" atom style is required to use this potential.

The parameter file (e.g. CH.KC), is intended for use with *metal* *units*, with energies in meV. Two additional parameters, *S*, and *rcut* are included in the parameter file. *S* is designed to facilitate scaling of energies. *rcut* is designed to build the neighbor list for calculating the normals for each atom pair.

**Note**

Two new sets of parameters of KC potential for hydrocarbons, CH.KC (without the taper function) and CH_taper.KC (with the taper function) are presented in *(Ouyang1)*. The energy for the KC potential with the taper function goes continuously to zero at the cutoff. The parameters in both CH.KC and CH_taper.KC provide a good description in both short- and long-range interaction regimes. While the original parameters (CC.KC) published in *(Kolmogorov)* are only suitable for long-range interaction regime. This feature is essential for simulations in high pressure regime (i.e., the interlayer distance is smaller than the equilibrium distance). The benchmark tests and comparison of these parameters can be found in *(Ouyang1)* and *(Ouyang2)*.

This potential must be used in combination with hybrid/overlay. Other interactions can be set to zero using pair_style *none*.

This pair style tallies a breakdown of the total interlayer potential energy into sub-categories, which can be accessed via the *compute pair* command as a vector of values of length 2. The 2 values correspond to the following sub-categories:

1. *E_vdW* = vdW (attractive) energy
2. *E_Rep* = Repulsive energy

To print these quantities to the log file (with descriptive column headings) the following commands could be included in an input script:

```
compute 0 all pair kolmogorov/crespi/full
variable Evdw  equal c_0[1]
variable Erep  equal c_0[2]
thermo_style custom step temp epair v_Erep v_Evdw
```

---

### 4.148.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support the pair_modify mix, shift, table, and tail options.

This pair style does not write their information to binary restart files, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

### 4.148.5 Restrictions

This pair style is part of the INTERLAYER package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires the newton setting to be *on* for pair interactions.

The CH.KC potential file provided with LAMMPS (see the potentials folder) is parameterized for metal units. You can use this pair style with any LAMMPS units, but you would need to create your own custom CH.KC potential file with all coefficients converted to the appropriate units.

### 4.148.6 Related commands

*pair_coeff*, *pair_none*, *pair_style hybrid/overlay*, *pair_style drip*, *pair_style pair_lebedeva_z*, *pair_style kolmogorov/crespi/z*, *pair_style ilp/graphene/hbn*.

### 4.148.7 Default

tap_flag = 0

---

**(Kolmogorov)** A. N. Kolmogorov, V. H. Crespi, Phys. Rev. B 71, 235415 (2005)

**(Ouyang1)** W. Ouyang, D. Mandelli, M. Urbakh and O. Hod, Nano Lett. 18, 6009-6016 (2018).

**(Ouyang2)** W. Ouyang et al., J. Chem. Theory Comput. 16(1), 666-676 (2020).

## 4.149 pair_style kolmogorov/crespi/z command

### 4.149.1 Syntax

```
pair_style [hybrid/overlay ...] kolmogorov/crespi/z cutoff
```

### 4.149.2 Examples

```
pair_style hybrid/overlay kolmogorov/crespi/z 20.0
pair_coeff * * none
pair_coeff 1 2 kolmogorov/crespi/z  CC.KC   C C

pair_style hybrid/overlay rebo kolmogorov/crespi/z 14.0
pair_coeff * * rebo                 CH.rebo    C C
pair_coeff 1 2 kolmogorov/crespi/z  CC.KC      C C
```

### 4.149.3 Description

The *kolmogorov/crespi/z* style computes the Kolmogorov-Crespi interaction potential as described in *(Kolmogorov)*. An important simplification is made, which is to take all normals along the z-axis.

$$
\begin{aligned}
E =& \frac{1}{2}\sum_i \sum_{j \neq i} V_{ij} \\
V_{ij} =& e^{-\lambda(r_{ij}-z_0)}\left[C + f(\rho_{ij}) + f(\rho_{ji})\right] - A\left(\frac{r_{ij}}{z_0}\right)^{-6} + A\left(\frac{\text{cutoff}}{z_0}\right)^{-6} \\
\rho_{ij}^2 =& \rho_{ji}^2 = x_{ij}^2 + y_{ij}^2 \qquad\qquad (\mathbf{n}_i \equiv \hat{\mathbf{z}}) \\
f(\rho) =& e^{-(\rho/\delta)^2} \sum_{n=0}^{2} C_{2n} \left(\rho/\delta\right)^{2n}
\end{aligned}
$$

It is important to have a sufficiently large cutoff to ensure smooth forces. Energies are shifted so that they go continuously to zero at the cutoff assuming that the exponential part of $V_{ij}$ (first term) decays sufficiently fast. This shift is achieved by the last term in the equation for $V_{ij}$ above.

This potential is intended for interactions between two layers of graphene. Therefore, to avoid interaction between layers in multi-layered materials, each layer should have a separate atom type and interactions should only be computed between atom types of neighboring layers.

The parameter file (e.g. CC.KC), is intended for use with metal *units*, with energies in meV. An additional parameter, $S$, is available to facilitate scaling of energies in accordance with *(vanWijk)*.

This potential must be used in combination with hybrid/overlay. Other interactions can be set to zero using pair_style *none*.

### 4.149.4 Restrictions

This fix is part of the INTERLAYER package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.149.5 Related commands

*pair_coeff*, *pair_none*, *pair_style hybrid/overlay*, *pair_style drip*, *pair_style ilp/graphene/hbn*. *pair_style kolmogorov/crespi/full*, *pair_style lebedeva/z*

### 4.149.6 Default

none

---

**(Kolmogorov)** A. N. Kolmogorov, V. H. Crespi, Phys. Rev. B 71, 235415 (2005)

**(vanWijk)** M. M. van Wijk, A. Schuring, M. I. Katsnelson, and A. Fasolino, Physical Review Letters, 113, 135504 (2014)

# 4.150 pair_style lambda/input/apip command

# 4.151 pair_style lambda/input/csp/apip command

## 4.151.1 Syntax

```
pair_style lambda/input/apip cutoff
```

- lambda/input/apip = style name of this pair style
- cutoff = global cutoff (distance units)

# 4.152 pair_style lambda/input/csp/apip command

## 4.152.1 Syntax

```
pair_style lambda/input/csp/apip lattice keyword args
```

- lambda/input/csp/apip = style name of this pair style
- lattice = *fcc* or *bcc* or integer

  ```
  fcc = use 12 nearest neighbors to calculate the CSP like in a perfect fcc lattice
  bcc = use 8 nearest neighbors to calculate the CSP like in a perfect bcc lattice
  integer = use N nearest neighbors to calculate the CSP
  ```

- zero or more keyword/args pairs may be appended
- keyword = *cutoff* or *N_buffer*

  ```
  cutoff args = cutoff
    cutoff = distance in which neighboring atoms are considered (> 0)
  N_buffer args = N_buffer
    N_buffer = number of additional neighbors, which are included in the j-j+N/2␣
  ↪calculation
  ```

## 4.152.2 Examples

```
pair_style lambda/input/csp/apip fcc
pair_style lambda/input/csp/apip fcc cutoff 5.0
pair_style lambda/input/csp/apip bcc cutoff 5.0 N_buffer 2
pair_style lambda/input/csp/apip 14
```

## 4.152.3 Description

This pair_styles calculates $\lambda_i^{\text{input}}(t)$, which is required for *fix lambda/apip*.

The pair_style lambda_input sets $\lambda_i^{\text{input}}(t) = 0$.

The pair_style lambda_input/csp calculates $\lambda_i^{\text{input}}(t) = \text{CSP}_i(t)$. The centro-symmetry parameter (CSP) *(Kelchner)* is described in *compute centro/atom*.

The lattice argument is described in *compute centro/atom* and determines the number of neighboring atoms that are used to compute the CSP. The *N_buffer* argument allows to include more neighboring atoms in the calculation of the contributions from the pair j,j+N/2 to the CSP as discussed in *(Immel)*.

The computation of $\lambda_i^{\text{input}}(t)$ is done by this pair_style instead of by *fix lambda/apip*, as this computation takes time and this pair_style can be included in the load-balancing via *fix atom_weight/apip*.

A code example for the calculation of the switching parameter for an adaptive- precision potential is given in the following: The adaptive-precision potential is created by combining *pair_style eam/fs/apip* and *pair_style pace/precise/apip*. The input, from which the switching parameter is calculated, is provided by this pair_style. The switching parameter is calculated by *fix lambda/apip*, whereas the spatial transition zone of the switching parameter is calculated by *pair_style lambda/zone/apip*.

```
pair_style hybrid/overlay eam/fs/apip pace/precise/apip lambda/input/csp/apip fcc cutoff␣
↪5.0 lambda/zone/apip 12.0
pair_coeff * * eam/fs/apip Cu.eam.fs Cu
pair_coeff * * pace/precise/apip Cu_precise.yace Cu
pair_coeff * * lambda/input/csp/apip
pair_coeff * * lambda/zone/apip
fix 2 all lambda/apip 3.0 3.5 time_averaged_zone 4.0 12.0 110 110 min_delta_lambda 0.01
```

---

### 4.152.4 Mixing, shift, table, tail correction, restart, rRESPA info

The cutoff distance for this pair style can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style writes no information to *binary restart files*, so pair_style and pair_coeff commands need to be specified in an input script that reads a restart file.

This pair style does not support the use of the *inner*, *middle*, and *outer* keywords of the *run_style respa* command.

---

### 4.152.5 Restrictions

This fix is part of the APIP package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.152.6 Related commands

*compute centro/atom*, *fix lambda/apip*, *fix lambda_thermostat/apip*, *pair_style lambda/zone/apip*, *pair_style eam/apip*, *pair_style pace/apip*, *fix atom_weight/apip*

### 4.152.7 Default

N_buffer=0, cutoff=5.0

---

**(Kelchner)** Kelchner, Plimpton, Hamilton, Phys Rev B, 58, 11085 (1998).

**(Immel)** Immel, Drautz and Sutmann, J Chem Phys, 162, 114119 (2025)

## 4.153 pair_style lambda/zone/apip command

### 4.153.1 Syntax

```
pair_style lambda/zone/apip cutoff
```

- lambda/zone/apip = style name of this pair style
- cutoff = global cutoff (distance units)

### 4.153.2 Examples

```
pair_style lambda/zone/apip 12.0
```

### 4.153.3 Description

This pair_style calculates $\lambda_{\mathrm{min},i}$, which is required for *fix lambda/apip*. The meaning of $\lambda_{\mathrm{min},i}$ is documented in *fix lambda/apip*, as this pair_style is for use with *fix lambda/apip* only.

This pair_style requires only the global cutoff as argument. The remaining quantities, that are required to calculate $\lambda_{\mathrm{min},i}$ are extracted from *fix lambda/apip* and, thus, do not need to be passed to this pair_style as arguments.

**Warning**

The cutoff given as argument to this pair style is only relevant for the neighbor list creation. The radii, which define $r_{\lambda,\mathrm{hi}}$ and $r_{\lambda,\mathrm{lo}}$ are defined by *fix lambda/apip*.

The computation of $\lambda_{\mathrm{min},i}$ is done by this pair_style instead of by *fix lambda/apip*, as this computation takes time and this pair_style can be included in the load-balancing via *fix atom_weight/apip*.

A code example for the calculation of the switching parameter for an adaptive-precision interatomic potential (APIP) is given in the following: The adaptive-precision potential is created by combining *pair_style eam/fs/apip* and *pair_style pace/precise/apip*. The input, from which the switching parameter is calculated, is provided by *pair lambda/input/csp/apip*. The switching parameter is calculated by *fix lambda/apip*, whereas the spatial transition zone of the switching parameter is calculated by this pair style.

```
pair_style hybrid/overlay eam/fs/apip pace/precise/apip lambda/input/csp/apip fcc cutoff␣
↪5.0 lambda/zone/apip 12.0
pair_coeff * * eam/fs/apip Cu.eam.fs Cu
pair_coeff * * pace/precise/apip Cu_precise.yace Cu
pair_coeff * * lambda/input/csp/apip
pair_coeff * * lambda/zone/apip
fix 2 all lambda/apip 3.0 3.5 time_averaged_zone 4.0 12.0 110 110 min_delta_lambda 0.01
```

---

### 4.153.4 Mixing, shift, table, tail correction, restart, rRESPA info

The cutoff distance for this pair style can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style writes no information to *binary restart files*, so pair_style and pair_coeff commands need to be specified in an input script that reads a restart file.

This pair style does not support the use of the *inner*, *middle*, and *outer* keywords of the *run_style respa* command.

---

### 4.153.5 Restrictions

This fix is part of the APIP package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.153.6 Related commands

*fix lambda/apip*, *fix atom_weight/apip pair_style lambda/input/apip*, *pair_style eam/apip*, *pair_style pace/apip*, *fix lambda_thermostat/apip*,

### 4.153.7 Default

none

## 4.154 pair_style lcbop command

Accelerator Variants: *lcbop/omp*

### 4.154.1 Syntax

```
pair_style lcbop
```

### 4.154.2 Examples

```
pair_style lcbop
pair_coeff * * ../potentials/C.lcbop C
```

### 4.154.3 Description

The *lcbop* pair style computes the long-range bond-order potential for carbon (LCBOP) of *(Los and Fasolino)*. See section II in that paper for the analytic equations associated with the potential.

Only a single pair_coeff command is used with the *lcbop* style which specifies an LCBOP potential file with parameters for specific elements. These are mapped to LAMMPS atom types by specifying N additional arguments after the filename in the pair_coeff command, where N is the number of LAMMPS atom types:

- filename
- N element names = mapping of LCBOP elements to atom types

See the *pair_coeff* page for alternate ways to specify the path for the potential file.

As an example, if your LAMMPS simulation has 4 atom types and you want the first 3 to be C you would use the following pair_coeff command:

```
pair_coeff * * C.lcbop C C C NULL
```

The first 2 arguments must be * * so as to span all LAMMPS atom types. The first C argument maps LAMMPS atom type 1 to the C element in the LCBOP file. If a mapping value is specified as NULL, the mapping is not performed. This can be used when a *lcbop* potential is used as part of the *hybrid* pair style. The NULL values are placeholders for atom types that will be used with other potentials.

The parameters/coefficients for the LCBOP potential as applied to C are listed in the C.lcbop file to agree with the original *(Los and Fasolino)* paper. Thus the parameters are specific to this potential and the way it was fit, so modifying the file should be done carefully.

### 4.154.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support the *pair_modify* mix, shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.154.5 Restrictions

This pair style is part of the MANYBODY package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair potential requires the *newton* setting to be "on" for pair interactions.

The `C.lcbop` potential file provided with LAMMPS (see the potentials directory) is parameterized for *metal units*. You can use the LCBOP potential with any LAMMPS units, but you would need to create your own LCBOP potential file with coefficients listed in the appropriate units if your simulation does not use "metal" units.

### 4.154.6 Related commands

*pair_airebo*, *pair_coeff*

### 4.154.7 Default

none

---

**(Los and Fasolino)** J. H. Los and A. Fasolino, Phys. Rev. B 68, 024107 (2003).

## 4.155 pair_style ldd command

### 4.155.1 Syntax

```
pair_style ldd
pair_coeff * * file.ldd S1 S2 ...
```

- ldd = name of this pair style (takes no arguments)
- file.ldd = name of the local-density potential file
- S1, S2, ... = mapping of N LAMMPS atom types to species names from the file (N = # of atom types)

### 4.155.2 Examples

```
# two atom types mapped to species A and B
pair_style ldd
pair_coeff * * LDINDSET1.ldd A B

# combine local-density interactions with other pair styles
pair_style hybrid/overlay ldd lj/cut 14.0
pair_coeff 1 1 lj/cut 1.0 1.0
pair_coeff * * ldd ld_input_file.ldd A
```

A potential file entry has the form (one ordered species pair per line):

```
Si Sj indicator wtype r0 rc self yes/no potential ptype args [gradient gtype args]␣
↪[ignore]
```

### 4.155.3 Description

Added in version 4Jul2026.

Style *ldd* implements the local density potential as first described by Pagonabarraga and Frenkel *(Pagonabarraga)* and additionally the square gradient of local densities first introduced by *(DeLyser)*. The pair_style *ldd* is compatible with a variety of molecular and atomic topologies (see the *Howto ldd* page for details) and offers a variety of options for how to define the local density.

Following the manybody potential convention (as for *pair_style sw* or *tersoff*), all interactions are read from a potential file with a single `pair_coeff * *` command. The file defines a set of *species*; the arguments after the file name map each LAMMPS atom type to one of these species (one species name per atom type, in order). Several atom types may map to the same species. The set and number of species are inferred from the file. The cutoff of each interaction is the `rc` of its indicator function; there is no separate global cutoff.

Each line of the potential file specifies the interaction for one *ordered* species pair `Si Sj` (the local density of species `Sj` around a central atom of species `Si`), so the `Si Sj` and `Sj Si` interactions can differ. All $N_{\text{species}}^2$ ordered pairs must be listed (use the *ignore* keyword for pairs with no interaction). Comment lines (starting with #) and the customary `# DATE: ... UNITS: ... CONTRIBUTOR: ...` header are allowed.

### 4.155.4 Theory

Here, for notational simplicity, we outline the theory for style *ldd* potentials with just one species of particle. The *Howto ldd* page explains the more general $n_{\text{species}}$ case.

Consider an indicator function of pair distance, $w(r)$, where $w(0) = 1$, $w(r_c) = 0$, and $w(r)$ is continuous/differentiable for $0 \le r \le r_c$. We define $[w]$ as the spatial integral of $w(r)$ and the normalized indicator function as $\bar{w}(r) = w(r)/[w]$. The local density around a given particle is then defined as $\rho_I = \sum_J \bar{w}(r_{IJ})$, where the sum over $J$ can either include or exclude $I$, depending on whether the argument following the *self* keyword is yes or no. Practically, the only difference is a horizontal shift in the argument of the potential by $1/[w]$. The local density potential and corresponding pair forces are given by:

$$U_{LD}(\mathbf{R}) = \sum_I U_\rho(\rho_I)$$

$$F_{IJ;\rho}(\mathbf{R}) = (F_\rho(\rho_I) + F_\rho(\rho_J))\bar{w}'(r_{IJ})\mathbf{e}_{IJ}$$

where $F_\rho(x) = -dU_\rho(x)/dx$.

The optional *gradient* keyword implements a potential that is a function of the square gradient of the local density as described in *(DeLyser)*. This keyword adds an extra square gradient (SG) term to the overall potential:

$$U_{SG}(\mathbf{R}) = \sum_I U_\nabla(\rho_I)\|\nabla_I \rho_I\|^2$$

The *self* argument indicates whether the particle $I = J$ term is included in the local densities and gradients calculated. Note that if site $I$ is of species $\alpha$, then it cannot contribute to the local density of $\beta$ sites around $I$ when $\beta \neq \alpha$. Thus in that case the self term is automatically excluded and a warning is generated.

The *ignore* keyword is used in simulations where only some of the species pairs have local density potentials acting between them. Once a pair is marked *ignore*, no local density or gradient is computed for it.

The *noforce* potential is an alternative way to turn an interaction off: it computes the local densities and gradients (so they can be inspected, see below) without applying any force.

### 4.155.5 Indicator functions

The (required) *indicator* keyword defines the form for $w(r)$ and takes three arguments, `wtype r0 rc`: the indicator style, the inner range $r_0$, and the outer cutoff $r_c$. In every case $\theta(x)$ is the Heaviside function (1 for $x \geq 0$, 0 otherwise).

*lucy* employs the Lucy function popularized from smoothed particle hydrodynamics *(Lucy)* (requires $r_0 = 0$):

$$
\begin{aligned}
w(r) &= (1 - r/r_c)^3(1 + 3r/r_c) \\
[w] &= 16\pi r_c^3/105 \\
\bar{w}(r) &= 105(1 - r/r_c)^3(1 + 3r/r_c)\theta(r_c - r)/(16\pi r_c^3)
\end{aligned}
$$

*dpd* employs a simple quadratic indicator function (requires $r_0 = 0$):

$$
\begin{aligned}
w(r) &= (1 - r/r_c)^2 \\
[w] &= 2\pi r_c^3/15 \\
\bar{w}(r) &= 15(1 - r/r_c)^2\theta(r_c - r)/(2\pi r_c^3)
\end{aligned}
$$

*sphere* employs a smoothed Heaviside indicator function. If you consider a large sphere of radius $X$ and a small sphere of radius $x$, with $r_0 = X - x$ and $r_c = X + x$, then $w(r)$ is the fraction of the small sphere contained in the large sphere when their centers are separated by $r$:

$$
w(r) = \begin{cases} 1 & r \leq r_0 \\ c_3 r^3 + c_1 r + c_0 + c_{-1}/r & r_0 \leq r \leq r_c \\ 0 & r \geq r_c \end{cases}
$$

*shell* employs a smoothed Heaviside indicator function originally parameterized in *(Sanyal)*. It is one for $r \leq r_0$, zero for $r \geq r_c$, and smoothly interpolates in between with a continuous first derivative (but a discontinuous second derivative at $r_0$ and $r_c$):

$$
w(r) = \begin{cases} 1 & r \leq r_0 \\ c_0 + c_2 r^2 + c_4 r^4 + c_6 r^6 & r_0 \leq r \leq r_c \\ 0 & r \geq r_c \end{cases}
$$

*smooth* is similar to *shell* but has a continuous second derivative:

$$
w(r) = \begin{cases} 1 & r \leq r_0 \\ \sum_{i=0}^{5} c_i r^i & r_0 \leq r \leq r_c \\ 0 & r \geq r_c \end{cases}
$$

### 4.155.6 Potential and gradient functions

The (required) *potential* keyword defines the form for $U_\rho$, and the (optional) *gradient* keyword defines the form for $U_\nabla$. Both take a style name followed by style-specific arguments. Below, $X$ is a placeholder for either $\rho$ or $\nabla$, and $F_X = -dU_X/d\rho$.

*noforce* (potential only) computes the densities/gradients without applying a force: $U_\rho(\rho) = 0$, $F_\rho(\rho) = 0$.

*constant* takes one argument $c$. A constant *potential* does not change the dynamics, but a constant *gradient* does:

$$U_X(\rho) = c, \qquad F_X(\rho) = 0.$$

*linear* takes two arguments, the slope $m$ and intercept $b$:

$$U_X(\rho) = m\rho + b, \qquad F_X(\rho) = -m.$$

*quadratic* takes three arguments $a$, $b$, $c$:

$$U_X(\rho) = a\rho^2 + b\rho + c, \qquad F_X(\rho) = -2a\rho - b.$$

*mdpd* takes one argument $B$ and applies the quadratic local density potential used in many MDPD studies (e.g. *(Warren)* or *(Ghoufi)*). Combined with the *dpd* indicator and *pair_style dpd* it reproduces the traditional many-body DPD conservative force:

$$U_X(\rho) = B\pi r_c^4 \rho^2/30, \qquad F_X(\rho) = -B\pi r_c^4 \rho/15.$$

*table/lin*, *table/spline*, *table/gradlin*, *table/gradspline* each take one argument, the name of a tabulated potential file. The `table/lin` and `table/spline` forms interpolate $U_X$ from the table linearly or with a cubic spline; the `table/gradlin` and `table/gradspline` forms are the corresponding interpolating styles used after the *gradient* keyword. Each line of a table file holds three numbers, $\rho$, $U(\rho)$, and $-dU/d\rho$, on a uniform $\rho$ grid. Example tables are provided under `examples/PACKAGES/bocs`.

### 4.155.7 Accessing the local-density data

The per-atom local densities, density gradients, and energies are recomputed every step but are not stored in data or restart files. They can be made available to the rest of LAMMPS with the *fix pair* command, which exposes the following per-atom fields (one column per species, in the order of the type map; *grad_density* has three columns per species):

- *local_density* – local density of each species around an atom
- *grad_density* – gradient of the local density (x, y, z per species)
- *energy* – $U_\rho$ contribution from each species
- *grad_energy* – $U_\nabla$ contribution from each species
- *total_energy* – sum of all local-density energies on an atom (scalar)

For example, `fix f all pair 1 ldd local_density 0 total_energy 0` creates the per-atom array `f_f` whose first columns are the per-species local densities followed by the total energy; these can then be output with *dump custom* or read by other per-atom computes and fixes.

### 4.155.8 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support automatic mixing.

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write its information to *binary restart files*. Therefore, you must re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

### 4.155.9 Restrictions

This pair style is part of the BOCS package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires `newton on` for pair interactions.

The `pair_style ldd` command must be issued *after* the simulation box has been created (e.g. with *read_data*, *read_restart*, or *create_box*), because it sizes its per-type species map and inter-processor communication buffers from the number of atom types. Since `ldd` is a manybody-style potential that does not read per-type `Pair Coeffs` from a data file, there is no need to define it earlier. When continuing from a binary restart, re-specify `pair_style ldd` and its `pair_coeff` settings after the *read_restart* command.

The *indicator*, *self*, and *potential* keywords are mandatory for each species pair unless the *ignore* keyword is provided; the *gradient* keyword is optional. Every ordered species pair must appear exactly once in the potential file.

Indicator styles with a non-zero $r_0$ (`sphere`, `shell`, `smooth`) are non-zero inside $r_0$; *lucy* and *dpd* require $r_0 = 0$.

### 4.155.10 Related commands

*pair_coeff*, *fix pair*, *Howto ldd*

---

**(Pagonabarraga)** I. Pagonabarraga, D. Frenkel. "Dissipative particle dynamics for interacting systems.", J. Chem. Phys., 115, 5015 (2001).

**(DeLyser)** M.R. DeLyser, W.G. Noid. "Coarse-grained models for local density gradients." J. Chem. Phys., 156, 034106 (2021).

**(Lucy)** L. B. Lucy, "A numerical approach to the testing of the fission hypothesis.", Astronomical Journal, 82, 1013-1024 (1977).

**(Sanyal)** T. Sanyal, M. S. Shell, "Coarse-grained models using local-density potentials optimized with the relative entropy: Application to implicit solvation.", J. Chem. Phys., 145, 034109 (2016).

**(Warren)** P. B. Warren, "Vapor-liquid coexistence in many-body dissipative particle dynamics.", Phys. Rev. E, 68, 066702 (2003).

**(Ghoufi)** A. Ghoufi, P. Malfreyt, "Mesoscale modeling of the water liquid-vapor interface: A surface tension calculation.", Phys. Rev. E, 83, 051601 (2011).

## 4.156 pair_style lebedeva/z command

### 4.156.1 Syntax

```
pair_style [hybrid/overlay ...] lebedeva/z cutoff
```

### 4.156.2 Examples

```
pair_style hybrid/overlay lebedeva/z 20.0
pair_coeff * * none
pair_coeff 1 2 lebedeva/z  CC.Lebedeva   C C

pair_style hybrid/overlay rebo lebedeva/z 14.0
pair_coeff * * rebo        CH.rebo       C C
pair_coeff 1 2 lebedeva/z  CC.Lebedeva   C C
```

### 4.156.3 Description

The *lebedeva/z* pair style computes the Lebedeva interaction potential as described in *(Lebedeva1)* and *(Lebedeva2)*. An important simplification is made, which is to take all normals along the z-axis.

The Lebedeva potential is intended for the description of the interlayer interaction between graphene layers. To perform a realistic simulation, this potential must be used in combination with an intralayer potential such as *AIREBO* or *Tersoff* facilitated by using pair style *hybrid/overlay*. To keep the intralayer properties unaffected, the interlayer interaction within the same layers should be avoided. This can be achieved by assigning different atom types to atoms of different layers (e.g. 1 and 2 in the examples above).

Other interactions can be set to zero using pair_style *none*.

$$
\begin{aligned}
E =& \frac{1}{2} \sum_i \sum_{j \neq i} V_{ij} \\
V_{ij} =& Be^{-\alpha(r_{ij} - z_0)} \\
& + C(1 + D_1\rho_{ij}^2 + D_2\rho_{ij}^4) e^{-\lambda_1 \rho_{ij}^2} e^{-\lambda_2 (z_{ij}^2 - z_0^2)} \\
& - A \left( \frac{z_0}{r_i j} \right)^6 + A \left( \frac{z_0}{r_c} \right)^6 \\
\rho_{ij}^2 =& x_{ij}^2 + y_{ij}^2 \qquad (\mathbf{n_i} \equiv \hat{\mathbf{z}})
\end{aligned}
$$

It is important to have a sufficiently large cutoff to ensure smooth forces. Energies are shifted so that they go continuously to zero at the cutoff assuming that the exponential part of $V_{ij}$ (first term) decays sufficiently fast. This shift is achieved by the last term in the equation for $V_{ij}$ above.

The provided parameter file (CC.Lebedeva) contains two sets of parameters.

- The first set (element name "C") is suitable for normal conditions and is taken from *(Popov1)*
- The second set (element name "C1") is suitable for high-pressure conditions and is taken from *(Koziol1)*

Both sets contain an additional parameter, $S$, that can be used to facilitate scaling of energies and is set to 1.0 by default.

### 4.156.4 Restrictions

This pair style is part of the INTERLAYER package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.156.5 Related commands

*pair_coeff*, *pair_style none*, *pair_style hybrid/overlay*, *pair_style drip*, *pair_style ilp/graphene/hbd*, *pair_style kolmogorov/crespi/z*, *pair_style kolmogorov/crespi/full*.

### 4.156.6 Default

none

---

**(Lebedeva1)** I. V. Lebedeva, A. A. Knizhnik, A. M. Popov, Y. E. Lozovik, B. V. Potapkin, Phys. Rev. B, 84, 245437 (2011)

**(Lebedeva2)** I. V. Lebedeva, A. A. Knizhnik, A. M. Popov, Y. E. Lozovik, B. V. Potapkin, Physica E: 44, 949-954 (2012)

**(Popov1)** A.M. Popov, I. V. Lebedeva, A. A. Knizhnik, Y. E. Lozovik and B. V. Potapkin, Chem. Phys. Lett. 536, 82-86 (2012).

**(Koziol1)** Z. Koziol, G. Gawlik and J. Jagielski, Chinese Phys. B 28, 096101 (2019).

## 4.157 pair_style lepton command

Accelerator Variants: *lepton/omp*

## 4.158 pair_style lepton/coul command

Accelerator Variants: *lepton/coul/omp*

## 4.159 pair_style lepton/sphere command

Accelerator Variants: *lepton/sphere/omp*

### 4.159.1 Syntax

```
pair_style style args
```

- style = *lepton* or *lepton/coul* or *lepton/sphere*
- args = list of arguments for a particular style

```
lepton args = cutoff
  cutoff = global cutoff for the interactions (distance units)
lepton/coul args = cutoff keyword
  cutoff = global cutoff for the interactions (distance units)
  zero or more keywords may be appended
  keyword = ewald or pppm or msm or dispersion or tip4p
lepton/sphere args = cutoff
  cutoff = global cutoff for the interactions (distance units)
```

### 4.159.2 Examples

```
pair_style lepton 2.5

pair_coeff  * *  "k*((r-r0)^2*step(r0-r)); k=200; r0=1.5" 2.0
pair_coeff  1 2  "4.0*eps*((sig/r)^12 - (sig/r)^6);eps=1.0;sig=1.0" 1.12246204830937
pair_coeff  2 2  "eps*(2.0*(sig/r)^9 - 3.0*(sig/r)^6);eps=1.0;sig=1.0"
pair_coeff  1 3  "zbl(13,6,r)"
pair_coeff  3 3  "(1.0-switch)*zbl(6,6,r)-switch*4.0*eps*((sig/r)^6);switch=0.5*(tanh(10.
↪0*(r-sig))+1.0);eps=0.05;sig=3.20723"

pair_style lepton/coul 2.5
pair_coeff 1 1 "qi*qj/r" 4.0
pair_coeff 1 2 "lj+coul; lj=4.0*eps*((sig/r)^12 - (sig/r)^6); eps=1.0; sig=1.0;␣
↪coul=qi*qj/r"

pair_style lepton/coul 2.5 pppm
kspace_style pppm 1.0e-4
pair_coeff 1 1 "qi*qj/r*erfc(alpha*r); alpha=1.067"

pair_style lepton/sphere 2.5
pair_coeff 1 * "k*((r-r0)^2*step(r0-r)); k=200; r0=radi+radj"
pair_coeff 2 2 "4.0*eps*((sig/r)^12 - (sig/r)^6); eps=1.0; sig=2.0*sqrt(radi*radj)"
```

### 4.159.3 Description

Added in version 8Feb2023: added pair styles *lepton* and *lepton/coul*

Changed in version 15Jun2023: added pair style *lepton/sphere*

Pair styles *lepton*, *lepton/coul*, *lepton/sphere* compute pairwise interactions between particles which depend on the distance and have a cutoff. The potential function must be provided as an expression string using "r" as the distance variable. With pair style *lepton/coul* one may additionally reference the charges of the two atoms of the pair with "qi" and "qj", respectively. With pair style *lepton/sphere* one may instead reference the radii of the two atoms of the pair with "radi" and "radj", respectively; this is half of the diameter that can be set in *data files* or the *set command*.

Note that further constants in the expressions can be defined in the same string as additional expressions separated by semicolons as shown in the examples above.

The expression *"200.0*(r-1.5)^2"* represents a harmonic potential around the pairwise distance $r_0$ of 1.5 distance units and a force constant $K$ of 200.0 energy units:

$$U_{ij} = K(r - r_0)^2$$

The expression *"qi*qj/r"* represents a regular Coulombic potential with cutoff:

$$U_{ij} = \frac{Cq_iq_j}{\varepsilon r} \qquad r < r_c$$

The expression *"200.0*(r-(radi+radj)^2"* represents a harmonic potential that has the equilibrium distance chosen so that the radii of the two atoms touch:

$$U_{ij} = K(r - (r_i + r_j))^2$$

The Lepton library, that the *lepton* pair style interfaces with, evaluates this expression string at run time to compute the pairwise energy. It also creates an analytical representation of the first derivative of this expression with respect to "r" and then uses that to compute the force between the pairs of particles within the given cutoff.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- Lepton expression (energy units)
- cutoff (distance units)

The Lepton expression must be either enclosed in quotes or must not contain any whitespace so that LAMMPS recognizes it as a single keyword. More on valid Lepton expressions below. The last coefficient is optional; it allows to set the cutoff for a pair of atom types to a different value than the global cutoff.

For pair style *lepton* only the "lj" values of the *special_bonds* settings apply in case the interacting pair is also connected with a bond. The potential energy will *only* be added to the "evdwl" property.

For pair style *lepton/coul* only the "coul" values of the *special_bonds* settings apply in case the interacting pair is also connected with a bond. The potential energy will *only* be added to the "ecoul" property.

For pair style *lepton/sphere* only the "lj" values of the *special_bonds* settings apply in case the interacting pair is also connected with a bond. The potential energy will *only* be added to the "evdwl" property.

In addition to the functions listed below, both pair styles support in addition a custom "zbl(zi,zj,r)" function which computes the Ziegler-Biersack-Littmark (ZBL) screened nuclear repulsion for describing high-energy collisions between atoms. For details of the function please see the documentation for *pair style zbl*. The arguments of the function are the atomic numbers of atom i (zi), atom j (zj) and the distance r. Please see the examples above.

### 4.159.4 Lepton expression syntax and features

Lepton supports the following operators in expressions:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| + | Add | - | Subtract | * | Multiply | / | Divide | ^ | Power |

The following mathematical functions are available:

| | | | |
|---|---|---|---|
| sqrt(x) | Square root | exp(x) | Exponential |
| log(x) | Natural logarithm | sin(x) | Sine (angle in radians) |
| cos(x) | Cosine (angle in radians) | sec(x) | Secant (angle in radians) |
| csc(x) | Cosecant (angle in radians) | tan(x) | Tangent (angle in radians) |
| cot(x) | Cotangent (angle in radians) | asin(x) | Inverse sine (in radians) |
| acos(x) | Inverse cosine (in radians) | atan(x) | Inverse tangent (in radians) |
| sinh(x) | Hyperbolic sine | cosh(x) | Hyperbolic cosine |
| tanh(x) | Hyperbolic tangent | erf(x) | Error function |
| erfc(x) | Complementary Error function | abs(x) | Absolute value |
| min(x,y) | Minimum of two values | max(x,y) | Maximum of two values |
| delta(x) | delta(x) is 1 for $x = 0$, otherwise 0 | step(x) | step(x) is 0 for $x < 0$, otherwise 1 |

Numbers may be given in either decimal or exponential form. All of the following are valid numbers: *5*, *-3.1*, *1e6*, and *3.12e-2*.

As an extension to the standard Lepton syntax, it is also possible to use LAMMPS *variables* in the format "v_name". Before evaluating the expression, "v_name" will be replaced with the value of the variable "name". This is compatible with all kinds of scalar variables, but not with vectors, arrays, local, or per-atom variables. If necessary, a custom scalar variable needs to be defined that can access the desired (single) item from a non-scalar variable. As an example, the following lines will instruct LAMMPS to ramp the force constant for a harmonic bond from 100.0 to 200.0 during the next run:

```
variable fconst equal ramp(100.0, 200)
bond_style lepton
bond_coeff 1 1.5 "v_fconst * (r^2)"
```

An expression may be followed by definitions for intermediate values that appear in the expression. A semicolon ";" is used as a delimiter between value definitions. For example, the expression:

```
a^2+a*b+b^2; a=a1+a2; b=b1+b2
```

is exactly equivalent to

```
(a1+a2)^2+(a1+a2)*(b1+b2)+(b1+b2)^2
```

The definition of an intermediate value may itself involve other intermediate values. Whitespace and quotation characters ('' and "") are ignored. All uses of a value must appear *before* that value's definition. For efficiency reasons, the expression string is parsed, optimized, and then stored in an internal, pre-parsed representation for evaluation.

Evaluating a Lepton expression is typically between 2.5 and 5 times slower than the corresponding compiled and optimized C++ code. If additional speed or GPU acceleration (via GPU or KOKKOS) is required, the interaction can be represented as a table. Suitable table files can be created either internally using the *pair_write* or *bond_write* command or through the Python scripts in the *tools/tabulate* folder.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages*

page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.159.5 Mixing, shift, table, tail correction, restart, rRESPA info

Pair styles *lepton*, *lepton/coul*, and *lepton/sphere* do not support mixing. Thus, expressions for *all* I,J pairs must be specified explicitly.

Only pair style *lepton* supports the *pair_modify shift* option for shifting the potential energy of the pair interaction so that it is 0 at the cutoff, pair styles *lepton/coul* and *lepton/sphere* do *not*.

The *pair_modify table* options are not relevant for the these pair styles.

These pair styles do not support the *pair_modify tail* option for adding long-range tail corrections to energy and pressure.

These pair styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

These pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

---

### 4.159.6 Restrictions

These pair styles are part of the LEPTON package and only enabled if LAMMPS was built with this package. See the *Build package* page for more info.

Pair style *lepton/coul* requires that atom atoms have a charge property, e.g. via *atom_style charge*.

Pair style *lepton/sphere* requires that atom atoms have a radius property, e.g. via *atom_style sphere*.

### 4.159.7 Related commands

*pair_coeff* , *pair_style python*, *pair_style table*, *pair_write*

### 4.159.8 Default

none

## 4.160 pair_style line/lj command

Accelerator Variants: *line/lj/omp*

### 4.160.1 Syntax

```
pair_style line/lj cutoff
```

cutoff = global cutoff for interactions (distance units)

### 4.160.2 Examples

```
pair_style line/lj 3.0
pair_coeff * * 1.0 1.0 1.0 0.8 1.12
pair_coeff 1 2 1.0 2.0 1.0 1.5 1.12 5.0
pair_coeff 1 2 1.0 0.0 1.0 1.0 2.5
```

### 4.160.3 Description

Style *line/lj* treats particles which are line segments as a set of small spherical particles that tile the line segment length as explained below. Interactions between two line segments, each with N1 and N2 spherical particles, are calculated as the pairwise sum of N1*N2 Lennard-Jones interactions. Interactions between a line segment with N spherical particles and a point particle are treated as the pairwise sum of N Lennard-Jones interactions. See the *pair_style lj/cut* page for the definition of Lennard-Jones interactions.

The set of non-overlapping spherical sub-particles that represent a line segment are generated in the following manner. Their size is a function of the line segment length and the specified sub-particle size for that particle type. If a line segment has a length L and is of type I, then the number of spheres N that represent the segment is calculated as N = L/sizeI, rounded up to an integer value. Thus if L is not evenly divisible by sizeI, N is incremented to include one extra sphere. The centers of the spheres are spaced equally along the line segment. Imagine N+1 equally-space points, which include the 2 end points of the segment. The sphere centers are halfway between each pair of points.

The LJ interaction between 2 spheres on different line segments (or a sphere on a line segment and a point particles) is computed with sub-particle $\varepsilon$, $\sigma$, and *cutoff* values that are set by the pair_coeff command, as described below. If the distance between the 2 spheres is greater than the sub-particle cutoff, there is no interaction. This means that some pairs of sub-particles on 2 line segments may interact, but others may not.

For purposes of creating the neighbor list for pairs of interacting line segments or lines/point particles, a regular particle-particle cutoff is used, as defined by the *cutoff* setting above in the pair_style command or overridden with an optional argument in the pair_coeff command for a type pair as discussed below. The distance between the centers of 2 line segments, or the center of a line segment and a point particle, must be less than this distance (plus the neighbor skin; see the *neighbor* command), for the pair of particles to be included in the neighbor list.

**Note**

This means that a too-short value for the *cutoff* setting can exclude a pair of particles from the neighbor list even if pairs of their sub-particle spheres would interact, based on the sub-particle cutoff specified in the pair_coeff command. E.g. sub-particles at the ends of the line segments that are close to each other. Which may not be what you want, since it means the ends of 2 line segments could pass through each other. It is up to you to specify a *cutoff* setting that is consistent with the length of the line segments you are using and the sub-particle cutoff settings.

For style *line/lj*, the following coefficients must be defined for each pair of atom types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- sizeI (distance units)
- sizeJ (distance units)
- $\varepsilon$ (energy units)
- $\sigma$ (distance units)
- subcutoff (distance units)
- cutoff (distance units)

The *sizeI* and *sizeJ* coefficients are the sub-particle sizes for line particles of type I and type J. They are used to define the N sub-particles per segment as described above. These coefficients are actually stored on a per-type basis. Thus

if there are multiple pair_coeff commands that involve type I, as either the first or second atom type, you should use consistent values for sizeI or sizeJ in all of them. If you do not do this, the last value specified for sizeI will apply to all segments of type I. If typeI or typeJ refers to point particles, the corresponding sizeI or sizeJ is ignored; it can be set to 0.0.

The $\varepsilon$, $\sigma$, and *subcutoff* coefficients are used to compute an LJ interactions between a pair of sub-particles on 2 line segments (of type I and J), or between a sub particle/point particle pair. As discussed above, the *subcutoff* and *cutoff* params are different. The latter is only used for building the neighbor list when the distance between centers of two line segments or one segment and a point particle is calculated.

The *cutoff* coefficient is optional. If not specified, the global cutoff is used.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.160.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, coefficients must be specified. No default mixing rules are used.

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write its information to *binary restart files*.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.160.5 Restrictions

This style is part of the ASPHERE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Defining particles to be line segments so they participate in line/line or line/particle interactions requires the use the *atom_style line* command.

### 4.160.6 Related commands

*pair_coeff*, *pair_style tri/lj*

### 4.160.7 Default

none

# 4.161 pair_style list command

## 4.161.1 Syntax

```
pair_style list listfile cutoff keyword
```

- listfile = name of file with list of pairwise interactions
- cutoff = global cutoff (distance units)
- keyword = optional flag *nocheck* or *check* (default is *check*)

## 4.161.2 Examples

```
pair_style list restraints.txt 200.0
pair_coeff * *

pair_style hybrid/overlay lj/cut 1.1225 list pair_list.txt 300.0
pair_coeff * * lj/cut 1.0 1.0
pair_coeff 3* 3* list
```

## 4.161.3 Description

Style *list* computes interactions between explicitly listed pairs of atoms with the option to select functional form and parameters for each individual pair. Because the parameters are set in the list file, the pair_coeff command has no parameters (but still needs to be provided). The *check* and *nocheck* keywords enable/disable tests that checks whether all listed pairs of atom IDs were present and the interactions computed. If *nocheck* is set and either atom ID is not present, the interaction is skipped.

This pair style can be thought of as a hybrid between bonded, non-bonded, and restraint interactions. It will typically be used as an additional interaction within the *hybrid/overlay* pair style. It currently supports three interaction styles: a 12-6 Lennard-Jones, a Morse and a harmonic potential.

The format of the list file is as follows:

- one line per pair of atoms
- empty lines will be ignored
- comment text starts with a '#' character
- line syntax: *ID1 ID2 style coeffs cutoff*

```
ID1 = atom ID of first atom
ID2 = atom ID of second atom
style = style of interaction
coeffs = list of coeffs
cutoff = cutoff for interaction (optional)
```

The cutoff parameter is optional for all but the *quartic* interactions. If it is not specified, the global cutoff is used.

Here is an example file:

```
# this is a comment

15 259 lj126     1.0 1.0      50.0
15 603 morse    10.0 1.2 2.0  10.0 # and another comment
```

(continues on next page)

(continued from previous page)

```
18 470 harmonic 50.0 1.2       5.0
19 332 quartic  10.0 5.0 -1.2 1.2
```

The style *lj126* computes pairwise interactions with the formula

$$E = 4\varepsilon \left[ \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \qquad r < r_c$$

and the coefficients:

- $\varepsilon$ (energy units)
- $\sigma$ (distance units)

The style *morse* computes pairwise interactions with the formula

$$E = D_0 \left[1 - e^{-\alpha(r-r_0)}\right]^2 \qquad r < r_c$$

and the coefficients:

- $D_0$ (energy units)
- $\alpha$ (1/distance units)
- $r_0$ (distance units)

The style *harmonic* computes pairwise interactions with the formula

$$E = K(r - r_0)^2 \qquad r < r_c$$

and the coefficients:

- $K$ (energy units)
- $r_0$ (distance units)

Note that the usual 1/2 factor is included in $K$.

The style *quartic* computes pairwise interactions with the formula

$$E = K(r - r_0)^2(r - r_0 - b_1)(r - r_0 - b_2) \qquad r < r_c$$

and the coefficients:

- $K$ (energy units)
- $r_0$ (distance units)
- $b_1$ (distance units)
- $b_2$ (distance units)

### 4.161.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support mixing since all parameters are explicit for each pair.

The *pair_modify* shift option is supported by this pair style.

The *pair_modify* table and tail options are not relevant for this pair style.

This pair style does not write its information to *binary restart files*, so pair_style and pair_coeff commands need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.161.5 Restrictions

This pair style does not use a neighbor list and instead identifies atoms by their IDs. This has two consequences: 1) The cutoff has to be chosen sufficiently large, so that the second atom of a pair has to be a ghost atom on the same node on which the first atom is local; otherwise the interaction will be skipped. You can use the *check* option to detect, if interactions are missing. 2) Unlike other pair styles in LAMMPS, an atom I will not interact with multiple images of atom J (assuming the images are within the cutoff distance), but only with the closest image.

This style is part of the MISC package. It is only enabled if LAMMPS is build with that package. See the *Build package* page on for more info.

### 4.161.6 Related commands

*pair_coeff*, *pair_style hybrid/overlay*, *pair_style lj/cut*, *bond_style morse*, *bond_style harmonic bond_style quartic*

### 4.161.7 Default

none

## 4.162 pair_style lj/cut command

Accelerator Variants: *lj/cut/gpu*, *lj/cut/intel*, *lj/cut/kk*, *lj/cut/opt*, *lj/cut/omp*

### 4.162.1 Syntax

```
pair_style style args
```

- style = *lj/cut*
- args = list of arguments for a particular style

```
lj/cut args = cutoff
  cutoff = global cutoff for Lennard Jones interactions (distance units)
```

### 4.162.2 Examples

```
pair_style lj/cut 2.5
pair_coeff * * 1 1
pair_coeff 1 1 1 1.1 2.8
```

### 4.162.3 Description

The *lj/cut* styles compute the standard 12/6 Lennard-Jones potential, given by

$$E = 4\varepsilon \left[ \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \qquad r < r_c$$

$r_c$ is the cutoff.

See the *lj/cut/coul* styles to add a Coulombic pairwise interaction and the *lj/cut/tip4p* styles to add the TIP4P water model.

### 4.162.4 Coefficients

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- $\varepsilon$ (energy units)
- $\sigma$ (distance units)
- LJ cutoff (distance units)

The last coefficient is optional. If not specified, the global LJ cutoff specified in the pair_style command is used.

Note that $\sigma$ is defined in the LJ formula as the zero-crossing distance for the potential, *not* as the energy minimum at $r_0 = 2^{\frac{1}{6}}\sigma$. The *same* potential function becomes:

$$E = \varepsilon \left[ \left( \frac{r_0}{r} \right)^{12} - 2 \left( \frac{r_0}{r} \right)^{6} \right] \qquad r < r_c$$

When using the minimum as reference width. In the literature both formulations are used, but the describe the same potential, only the $\sigma$ value must be computed by $\sigma = r_0/2^{\frac{1}{6}}$ for use with LAMMPS, if this latter formulation is used.

---

A version of these styles with a soft core, *lj/cut/soft*, suitable for use in free energy calculations, is part of the FEP package and is documented with the *pair_style */soft* styles.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.162.5 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon and sigma coefficients and cutoff distance for all of the lj/cut pair styles can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

All of the *lj/cut* pair styles support the *pair_modify* shift option for the energy of the Lennard-Jones portion of the pair interaction.

All of the *lj/cut* pair styles support the *pair_modify* tail option for adding a long-range tail correction to the energy and pressure for the Lennard-Jones portion of the pair interaction.

All of the *lj/cut* pair styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

The *lj/cut* pair styles support the use of the *inner*, *middle*, and *outer* keywords of the *run_style respa* command, meaning the pairwise forces can be partitioned by distance at different levels of the rRESPA hierarchy. The other styles only support the *pair* keyword of run_style respa. See the *run_style* command for details.

---

### 4.162.6 Related commands

- *pair_coeff*
- *pair_style lj/cut/coul/cut*
- *pair_style lj/cut/coul/debye*
- *pair_style lj/cut/coul/dsf*
- *pair_style lj/cut/coul/long*
- *pair_style lj/cut/coul/msm*
- *pair_style lj/cut/coul/wolf*
- *pair_style lj/cut/tip4p/cut*
- *pair_style lj/cut/tip4p/long*

### 4.162.7 Default

none

## 4.163 pair_style lj96/cut command

Accelerator Variants: *lj96/cut/gpu*, *lj96/cut/kk*, *lj96/cut/omp*

### 4.163.1 Syntax

```
pair_style lj96/cut cutoff
```

- cutoff = global cutoff for lj96/cut interactions (distance units)

### 4.163.2 Examples

```
pair_style lj96/cut 2.5
pair_coeff * * 1.0 1.0 4.0
pair_coeff 1 1 1.0 1.0
```

### 4.163.3 Description

The *lj96/cut* style compute a 9/6 Lennard-Jones potential, instead of the standard 12/6 potential, given by

$$E = 4\varepsilon \left[ \left(\frac{\sigma}{r}\right)^9 - \left(\frac{\sigma}{r}\right)^6 \right] \qquad r < r_c$$

$r_c$ is the cutoff.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- $\varepsilon$ (energy units)
- $\sigma$ (distance units)
- cutoff (distance units)

The last coefficient is optional. If not specified, the global LJ cutoff specified in the pair_style command is used.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.163.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon and sigma coefficients and cutoff distance for all of the lj/cut pair styles can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

This pair style supports the *pair_modify* shift option for the energy of the pair interaction.

The *pair_modify* table option is not relevant for this pair style.

This pair style supports the *pair_modify* tail option for adding a long-range tail correction to the energy and pressure of the pair interaction.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style supports the use of the *inner*, *middle*, and *outer* keywords of the *run_style respa* command, meaning the pairwise forces can be partitioned by distance at different levels of the rRESPA hierarchy. See the *run_style* command for details.

---

### 4.163.5 Restrictions

This pair style is part of the EXTRA-PAIR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.163.6 Related commands

*pair_coeff*

### 4.163.7 Default

none

## 4.164 pair_style lj/cubic command

Accelerator Variants: *lj/cubic/gpu*, *lj/cubic/kk*, *lj/cubic/omp*

### 4.164.1 Syntax

```
pair_style lj/cubic
```

### 4.164.2 Examples

```
pair_style lj/cubic
pair_coeff * * 1.0 0.8908987
```

### 4.164.3 Description

The *lj/cubic* style computes a truncated LJ interaction potential whose energy and force are continuous everywhere. Inside the inflection point the interaction is identical to the standard 12/6 *Lennard-Jones* potential. The LJ function outside the inflection point is replaced with a cubic function of distance. The energy, force, and second derivative are continuous at the inflection point. The cubic coefficient A3 is chosen so that both energy and force go to zero at the cutoff distance. Outside the cutoff distance the energy and force are zero.

$$\begin{aligned} E &= u_{LJ}(r) \qquad r \le r_s \\ &= u_{LJ}(r_s) + (r - r_s)u'_{LJ}(r_s) - \frac{1}{6}A_3(r - r_s)^3 \qquad r_s < r \le r_c \\ &= 0 \qquad r > r_c \end{aligned}$$

The location of the inflection point $r_s$ is defined by the LJ diameter, $r_s/\sigma = (26/7)^{1/6}$. The cutoff distance is defined by $r_c/r_s = 67/48$ or $r_c/\sigma = 1.737...$ The analytic expression for the cubic coefficient $A_3 r_{min}^3/\varepsilon = 27.93...$ is given in the paper by Holian and Ravelo *(Holian)*.

This potential is commonly used to study the shock mechanics of FCC solids, as in Ravelo et al. *(Ravelo)*.

The following coefficients must be defined for each pair of atom types via the *pair_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- $\varepsilon$ (energy units)
- $\sigma$ (distance units)

Note that $\sigma$ is defined in the LJ formula as the zero-crossing distance for the potential, not as the energy minimum, which is located at $r_{min} = 2^{\frac{1}{6}}\sigma$. In the above example, $\sigma = 0.8908987$, so $r_{min} = 1.0$.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.164.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon and sigma coefficients and cutoff distance for all of the lj/cut pair styles can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

The lj/cubic pair style does not support the *pair_modify* shift option, since pair interaction is already smoothed to 0.0 at the cutoff.

The *pair_modify* table option is not relevant for this pair style.

The lj/cubic pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure, since there are no corrections for a potential that goes to 0.0 at the cutoff.

The lj/cubic pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

The lj/cubic pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.164.5 Restrictions

This pair style is part of the EXTRA-PAIR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.164.6 Related commands

*pair_coeff*

### 4.164.7 Default

none

---

**(Holian)** Holian and Ravelo, Phys Rev B, 51, 11275 (1995).

**(Ravelo)** Ravelo, Holian, Germann and Lomdahl, Phys Rev B, 70, 014103 (2004).

## 4.165 pair_style lj/cut/coul/cut command

Accelerator Variants: *lj/cut/coul/cut/gpu*, *lj/cut/coul/cut/kk*, *lj/cut/coul/cut/omp*

## 4.166 pair_style lj/cut/coul/debye command

Accelerator Variants: *lj/cut/coul/debye/gpu*, *lj/cut/coul/debye/kk*, *lj/cut/coul/debye/omp*

## 4.167 pair_style lj/cut/coul/dsf command

Accelerator Variants: *lj/cut/coul/dsf/gpu*, *lj/cut/coul/dsf/kk*, *lj/cut/coul/dsf/omp*

## 4.168 pair_style lj/cut/coul/long command

Accelerator Variants: *lj/cut/coul/long/gpu*, *lj/cut/coul/long/kk*, *lj/cut/coul/long/intel*, *lj/cut/coul/long/opt*, *lj/cut/coul/long/omp*

## 4.169 pair_style lj/cut/coul/msm command

Accelerator Variants: *lj/cut/coul/msm/gpu*, *lj/cut/coul/msm/omp*

## 4.170 pair_style lj/cut/coul/wolf command

Accelerator Variants: *lj/cut/coul/wolf/kk*, *lj/cut/coul/wolf/omp*

### 4.170.1 Syntax

```
pair_style style args
```

- style = *lj/cut/coul/cut* or *lj/cut/coul/debye* or *lj/cut/coul/dsf* or *lj/cut/coul/esp* or *lj/cut/coul/long* *lj/cut/coul/msm* or *lj/cut/coul/wolf*
- args = list of arguments for a particular style

```
lj/cut/coul/cut args = cutoff (cutoff2)
  cutoff = global cutoff for LJ (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
lj/cut/coul/debye args = kappa cutoff (cutoff2)
  kappa = inverse of the Debye length (inverse distance units)
  cutoff = global cutoff for LJ (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
lj/cut/coul/dsf args = alpha cutoff (cutoff2)
  alpha = damping parameter (inverse distance units)
  cutoff = global cutoff for LJ (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (distance units)
```

Added in version 4Jul2026: *lj/cut/coul/esp* args = cutoff (cutoff2) cutoff = global cutoff for LJ (and Coulombic if only 1 arg) (distance units) cutoff2 = global cutoff for Coulombic (optional) (distance units) *lj/cut/coul/long* args = cutoff (cutoff2) cutoff = global cutoff for LJ (and Coulombic if only 1 arg) (distance units) cutoff2 = global cutoff for Coulombic (optional) (distance units) *lj/cut/coul/msm* args = cutoff (cutoff2) cutoff = global cutoff for LJ (and Coulombic if only 1 arg) (distance units) cutoff2 = global cutoff for Coulombic (optional) (distance units) *lj/cut/coul/wolf* args = alpha cutoff (cutoff2) alpha = damping parameter (inverse distance units) cutoff = global cutoff for LJ (and Coulombic if only 2 arg) (distance units) cutoff2 = global cutoff for Coulombic (optional) (distance units)

### 4.170.2 Examples

```
pair_style lj/cut/coul/cut 10.0
pair_style lj/cut/coul/cut 10.0 8.0
pair_coeff * * 100.0 3.0
pair_coeff 1 1 100.0 3.5 9.0
pair_coeff 1 1 100.0 3.5 9.0 9.0

pair_style lj/cut/coul/debye 1.5 3.0
pair_style lj/cut/coul/debye 1.5 2.5 5.0
pair_coeff * * 1.0 1.0
pair_coeff 1 1 1.0 1.5 2.5
pair_coeff 1 1 1.0 1.5 2.5 5.0

pair_style lj/cut/coul/dsf 0.05 2.5 10.0
pair_coeff * * 1.0 1.0
pair_coeff 1 1 1.0 1.0 2.5
```

(continues on next page)

(continued from previous page)

```
pair_style lj/cut/coul/esp 10.0
pair_style lj/cut/coul/esp 10.0 8.0
pair_coeff * * 100.0 3.0
pair_coeff 1 1 100.0 3.5 9.0

pair_style lj/cut/coul/long 10.0
pair_style lj/cut/coul/long 10.0 8.0
pair_coeff * * 100.0 3.0
pair_coeff 1 1 100.0 3.5 9.0

pair_style lj/cut/coul/msm 10.0
pair_style lj/cut/coul/msm 10.0 8.0
pair_coeff * * 100.0 3.0
pair_coeff 1 1 100.0 3.5 9.0

pair_style lj/cut/coul/wolf 0.2 5. 10.0
pair_coeff * * 1.0 1.0
pair_coeff 1 1 1.0 1.0 2.5
```

### 4.170.3 Description

The *lj/cut/coul* styles compute the standard 12/6 Lennard-Jones potential, given by

$$E = 4\varepsilon \left[ \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \qquad r < r_c$$

$r_c$ is the cutoff.

Style *lj/cut/coul/cut* adds a Coulombic pairwise interaction given by

$$E = \frac{C q_i q_j}{\varepsilon r} \qquad r < r_c$$

where $C$ is an energy-conversion constant, $q_i$ and $q_j$ are the charges on the two atoms, and $\varepsilon$ is the dielectric constant which can be set by the *dielectric* command. If one cutoff is specified in the pair_style command, it is used for both the LJ and Coulombic terms. If two cutoffs are specified, they are used as cutoffs for the LJ and Coulombic terms respectively.

Style *lj/cut/coul/debye* adds an additional exp() damping factor to the Coulombic term, given by

$$E = \frac{C q_i q_j}{\varepsilon r} \exp(-\kappa r) \qquad r < r_c$$

where $\kappa$ is the inverse of the Debye length. This potential is another way to mimic the screening effect of a polar solvent.

Style *lj/cut/coul/dsf* computes the Coulombic term via the damped shifted force model described in *Fennell*, given by:

$$E = q_i q_j \left[ \frac{\mathrm{erfc}(\alpha r)}{r} - \frac{\mathrm{erfc}(\alpha r_c)}{r_c} + \left( \frac{\mathrm{erfc}(\alpha r_c)}{r_c^2} + \frac{2\alpha}{\sqrt{\pi}} \frac{\exp(-\alpha^2 r_c^2)}{r_c} \right) (r - r_c) \right] \qquad r < r_c$$

where $\alpha$ is the damping parameter and erfc() is the complementary error-function. This potential is essentially a short-range, spherically-truncated, charge-neutralized, shifted, pairwise *1/r* summation. The potential is based on Wolf summation, proposed as an alternative to Ewald summation for condensed phase systems where charge screening causes electrostatic interactions to become effectively short-ranged. In order for the electrostatic sum to be absolutely

convergent, charge neutralization within the cutoff radius is enforced by shifting the potential through placement of image charges on the cutoff sphere. Convergence can often be improved by setting $\alpha$ to a small non-zero value.

Styles *lj/cut/coul/esp*, *lj/cut/coul/long* and *lj/cut/coul/msm* compute the same Coulombic interactions as style *lj/cut/coul/cut* except that an additional damping factor is applied to the Coulombic term so it can be used in conjunction with the *kspace_style* command and its *ewald* or *pppm* option. The Coulombic cutoff specified for this style means that pairwise interactions within this distance are computed directly; interactions outside that distance are computed in reciprocal space.

Style *lj/cut/coul/wolf* adds a Coulombic pairwise interaction via the Wolf summation method, described in *Wolf*, given by:

$$E_i = \frac{1}{2}\sum_{j \neq i} \frac{q_i q_j \mathrm{erfc}(\alpha r_{ij})}{r_{ij}} + \frac{1}{2}\sum_{j \neq i} \frac{q_i q_j \mathrm{erf}(\alpha r_{ij})}{r_{ij}} \qquad r < r_c$$

where $\alpha$ is the damping parameter, and erfc() is the complementary error-function terms. This potential is essentially a short-range, spherically-truncated, charge-neutralized, shifted, pairwise *1/r* summation. With a manipulation of adding and subtracting a self term (for i = j) to the first and second term on the right-hand-side, respectively, and a small enough $\alpha$ damping parameter, the second term shrinks and the potential becomes a rapidly-converging real-space summation. With a long enough cutoff and small enough $\alpha$ parameter, the energy and forces calculated by the Wolf summation method approach those of the Ewald sum. So it is a means of getting effective long-range interactions with a short-range potential.

### 4.170.4 Coefficients

For all of the *lj/cut/coul* pair styles, the following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- $\varepsilon$ (energy units)
- $\sigma$ (distance units)
- cutoff1 (distance units)
- cutoff2 (distance units)

Note that $\sigma$ is defined in the LJ formula as the zero-crossing distance for the potential, not as the energy minimum at $2^{\frac{1}{6}}\sigma$.

The latter 2 coefficients are optional. If not specified, the global LJ and Coulombic cutoffs specified in the pair_style command are used. If only one cutoff is specified, it is used as the cutoff for both LJ and Coulombic interactions for this type pair. If both coefficients are specified, they are used as the LJ and Coulombic cutoffs for this type pair.

For *lj/cut/coul/esp*, *lj/cut/coul/long* and *lj/cut/coul/msm* only the LJ cutoff can be specified since a Coulombic cutoff cannot be specified for an individual I,J type pair. All type pairs use the same global Coulombic cutoff specified in the pair_style command.

---

A version of these styles with a soft core, *lj/cut/coul/soft*and *lj/cut/coul/long/soft*, suitable for use in free energy calculations, is part of the FEP package and is documented with the *pair_style */soft* styles.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.170.5 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon and sigma coefficients and cutoff distance for all of the lj/cut pair styles can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

All of the *lj/cut* pair styles support the *pair_modify* shift option for the energy of the Lennard-Jones portion of the pair interaction.

The *lj/cut/coul/esp* and *lj/cut/coul/long* pair styles support the *pair_modify* table option since they can tabulate the short-range portion of the long-range Coulombic interaction.

All of the *lj/cut* pair styles support the *pair_modify* tail option for adding a long-range tail correction to the energy and pressure for the Lennard-Jones portion of the pair interaction.

All of the *lj/cut* pair styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

The *lj/cut/coul/esp* and *lj/cut/coul/long* pair styles support the use of the *inner*, *middle*, and *outer* keywords of the *run_style respa* command, meaning the pairwise forces can be partitioned by distance at different levels of the rRESPA hierarchy. The other styles only support the *pair* keyword of run_style respa. See the *run_style* command for details.

---

### 4.170.6 Restrictions

The *lj/cut/coul/esp*, *lj/cut/coul/long* and *lj/cut/coul/msm* styles are part of the KSPACE package.

The *lj/cut/coul/debye*, *lj/cut/coul/dsf*, and *lj/cut/coul/wolf* styles are part of the EXTRA-PAIR package.

These styles are only enabled if LAMMPS was built with those respective packages. See the *Build package* page for more info.

### 4.170.7 Related commands

*pair_coeff*

### 4.170.8 Default

none

---

**(Wolf)** D. Wolf, P. Keblinski, S. R. Phillpot, J. Eggebrecht, J Chem Phys, 110, 8254 (1999).

**(Fennell)** C. J. Fennell, J. D. Gezelter, J Chem Phys, 124, 234104 (2006).

## 4.171 pair_style lj/cut/sphere command

Accelerator Variants: *lj/cut/sphere/kk*, *lj/cut/sphere/omp*

### 4.171.1 Syntax

```
pair_style style args
```

- style = *lj/cut/sphere*
- args = list of arguments for a particular style

```
lj/cut/sphere args = cutoff ratio
  cutoff = global cutoff ratio for Lennard Jones interactions (unitless)
```

### 4.171.2 Examples

```
pair_style lj/cut/sphere 2.5
pair_coeff * * 1.0
pair_coeff 1 1 1.1 2.8
```

### 4.171.3 Description

Added in version 15Jun2023.

The *lj/cut/sphere* style compute the standard 12/6 Lennard-Jones potential, given by

$$E = 4\varepsilon \left[ \left( \frac{\sigma_{ij}}{r} \right)^{12} - \left( \frac{\sigma_{ij}}{r} \right)^{6} \right] \qquad r < r_c * \sigma_{ij}$$

$r_c$ is the cutoff ratio.

This is the same potential function used by the *lj/cut* pair style, but the $\sigma_{ij}$ parameter is not set as a per-type parameter via the *pair_coeff command*. Instead it is calculated individually for each pair using the per-atom diameter attribute of *atom_style sphere* for the two atoms as $\sigma_i$ and $\sigma_j$; $\sigma_{ij}$ is then computed by the mixing rule for pair coefficients as set by the *pair_modify mix* command (defaults to geometric mixing). The cutoff is not specified as a distance, but as ratio that is internally multiplied by $\sigma_{ij}$ to obtain the actual cutoff for each pair of atoms.

Note that $\sigma_{ij}$ is defined in the LJ formula above as the zero-crossing distance for the potential, *not* as the energy minimum which is at $2^{\frac{1}{6}}\sigma_{ij}$.

**Notes on cutoffs, neighbor lists, and efficiency**

If your system is mildly polydisperse, meaning the ratio of the diameter of the largest particle to the smallest is less than 2.0, then the neighbor lists built by the code should be reasonably efficient. Which means they will not contain too many particle pairs that do not interact. However, if your system is highly polydisperse (ratio > 2.0), the neighbor list build and force computations may be inefficient. There are two ways to try and speed up the simulations.

The first is to assign atoms to different atom types so that atoms of each type are similar in size. E.g. if particle diameters range from 1 to 5 use 4 atom types, ensuring atoms of type 1 have diameters from 1.0-2.0, type 2 from 2.0-3.0, etc. This will reduce the number of non-interacting pairs in the neighbor lists and thus reduce the time spent on computing pairwise interactions.

The second is to use the *neighbor multi* command which enabled a different algorithm for building neighbor lists. This will also require that you assign multiple atom types according to diameters, but will in addition use a more efficient size-dependent strategy to construct the neighbor lists and thus reduce the time spent on building neighbor lists.

Here are example input script commands using both ideas for a highly polydisperse system:

```
units           lj
atom_style      sphere
lattice         fcc 0.8442
region          box block 0 10 0 10 0 10
create_box      2 box
create_atoms    1 box

# create atoms with random diameters from bimodal distribution
variable switch atom random(0.0,1.0,345634)
variable diam atom (v_switch<0.75)*normal(0.4,0.075,325)+(v_switch>=0.7)*normal(1.2,0.
↪2,453)
set group all diameter v_diam

# assign type 2 to atoms with diameter > 0.6
variable large atom (2.0*radius)>0.6
group large variable large
set group large type 2

pair_style      lj/cut/sphere 2.5
pair_coeff      * * 1.0

neighbor 0.3 multi
```

Using multiple atom types speeds up the calculation for this example by more than a factor of 2, and using the multi-style neighbor list build causes an additional speedup of about 20 percent.

### 4.171.4 Coefficients

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- $\varepsilon$ (energy units)
- LJ cutoff ratio (unitless) (optional)

The last coefficient is optional. If not specified, the global LJ cutoff ratio specified in the *pair_style command* is used.

If a repulsive only LJ interaction is desired, the coefficient for the cutoff ratio should be set to the minimum of the LJ potential using `$(2.0^(1.0/6.0))`

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.171.5 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon coefficients and cutoff ratio for the *lj/cut/sphere* pair style can be mixed. The default mixing style is *geometric*. See the *pair_modify command* for details.

The *lj/cut/sphere* pair style supports the *pair_modify shift* option for the energy of the Lennard-Jones portion of the pair interaction.

The *lj/cut/sphere* pair style does *not* support the *pair_modify* tail option for adding a long-range tail corrections to the energy and pressure.

The *lj/cut/sphere* pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does *not* support the *inner*, *middle*, *outer* keywords.

---

### 4.171.6 Restrictions

The *lj/cut/sphere* pair style is only enabled if LAMMPS was built with the EXTRA-PAIR package. See the *Build package* page for more info.

The *lj/cut/sphere* pair style does not support the *sixthpower* mixing rule.

---

### 4.171.7 Related commands

- *pair_coeff*
- *pair_style lj/cut*
- *pair_style lj/expnd/sphere*

### 4.171.8 Default

none

## 4.172 pair_style lj/cut/tip4p/cut command

Accelerator Variants: *lj/cut/tip4p/cut/omp*

## 4.173 pair_style lj/cut/tip4p/long command

Accelerator Variants: *lj/cut/tip4p/long/gpu*, *lj/cut/tip4p/long/omp*, *lj/cut/tip4p/long/opt*

### 4.173.1 Syntax

```
pair_style style args
```

- style = *lj/cut/tip4p/cut* or *lj/cut/tip4p/long*
- args = list of arguments for a particular style

```
lj/cut/tip4p/cut args = otype htype btype atype qdist cutoff (cutoff2)
  otype,htype = atom types (numeric or type label) for TIP4P O and H
  btype,atype = bond and angle types (numeric or type label) for TIP4P waters
  qdist = distance from O atom to massless charge (distance units)
  cutoff = global cutoff for LJ (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
lj/cut/tip4p/long args = otype htype btype atype qdist cutoff (cutoff2)
  otype,htype = atom types (numeric or type label) for TIP4P O and H
  btype,atype = bond and angle types (numeric or type label) for TIP4P waters
  qdist = distance from O atom to massless charge (distance units)
  cutoff = global cutoff for LJ (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
```

### 4.173.2 Examples

```
pair_style lj/cut/tip4p/cut 1 2 7 8 0.15 12.0
pair_style lj/cut/tip4p/cut 1 2 7 8 0.15 12.0 10.0
pair_coeff * * 100.0 3.0
pair_coeff 1 1 100.0 3.5 9.0

pair_style lj/cut/tip4p/long 1 2 7 8 0.15 12.0
pair_style lj/cut/tip4p/long 1 2 7 8 0.15 12.0 10.0
pair_coeff * * 100.0 3.0
pair_coeff 1 1 100.0 3.5 9.0

pair_style lj/cut/tip4p/long OW HW HW-OW HW-OW-HW 0.15 12.0
labelmap atom 1 OW 2 HW
labelmap bond 1 HW-OW
labelmap angle 1 HW-OW-HW
pair_coeff * * 100.0 3.0
pair_coeff OW OW 100.0 3.5 9.0
```

### 4.173.3 Description

The *lj/cut/tip4p* styles implement the TIP4P water model of *(Jorgensen)* and similar models, which introduce a massless site M located a short distance away from the oxygen atom along the bisector of the HOH angle. The atomic types of the oxygen and hydrogen atoms, the bond and angle types for OH and HOH interactions, and the distance to the massless charge site are specified as pair_style arguments and are used to identify the TIP4P-like molecules and determine the position of the M site from the positions of the hydrogen and oxygen atoms of the water molecules. The M site location is used for all Coulomb interactions instead of the oxygen atom location, also with all other atom types, while the location of the oxygen atom is used for the Lennard-Jones interactions. Style *lj/cut/tip4p/cut* uses a cutoff for Coulomb interactions; style *lj/cut/tip4p/long* is for use with a long-range Coulombic solver (Ewald or PPPM).

**Note**

For each TIP4P water molecule in your system, the atom IDs for the O and 2 H atoms must be consecutive, with the O atom first. This is to enable LAMMPS to "find" the 2 H atoms associated with each O atom. For example, if the atom ID of an O atom in a TIP4P water molecule is 500, then its 2 H atoms must have IDs 501 and 502.

**Note**

If using type labels, the type labels must be defined before calling the *pair_coeff* command.

See the *Howto tip4p* page for more information on how to use the TIP4P pair styles and lists of parameters to set. Note that the neighbor list cutoff for Coulomb interactions is effectively extended by a distance 2*qdist when using the TIP4P pair style, to account for the offset distance of the fictitious charges on O atoms in water molecules. Thus it is typically best in an efficiency sense to use a LJ cutoff >= Coulombic cutoff + 2*qdist, to shrink the size of the neighbor list. This leads to slightly larger cost for the long-range calculation, so you can test the trade-off for your model.

The *lj/cut/tip4p* styles compute the standard 12/6 Lennard-Jones potential, given by

$$E = 4\varepsilon \left[ \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \qquad r < r_c$$

$r_c$ is the cutoff.

They add Coulombic pairwise interactions given by

$$E = \frac{Cq_iq_j}{\varepsilon r} \qquad r < r_c$$

where $C$ is an energy-conversion constant, $q_i$ and $q_j$ are the charges on the two atoms, and $\varepsilon$ is the dielectric constant which can be set by the *dielectric* command. If one cutoff is specified in the pair_style command, it is used for both the LJ and Coulombic terms. If two cutoffs are specified, they are used as cutoffs for the LJ and Coulombic terms respectively.

Style *lj/cut/tip4p/long* compute the same Coulombic interactions as style *lj/cut/tip4p/cut* except that an additional damping factor is applied to the Coulombic term so it can be used in conjunction with the *kspace_style* command and its *ewald* or *pppm* option. The Coulombic cutoff specified for this style means that pairwise interactions within this distance are computed directly; interactions outside that distance are computed in reciprocal space.

### 4.173.4 Coefficients

For all of the *lj/cut* pair styles, the following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- $\varepsilon$ (energy units)
- $\sigma$ (distance units)
- LJ cutoff (distance units)

Note that $\sigma$ is defined in the LJ formula as the zero-crossing distance for the potential, not as the energy minimum at $2^{\frac{1}{6}}\sigma$.

The last coefficient is optional. If not specified, the global LJ cutoff specified in the pair_style command is used.

For *lj/cut/tip4p/cut* and *lj/cut/tip4p/long* only the LJ cutoff can be specified since a Coulombic cutoff cannot be specified for an individual I,J type pair. All type pairs use the same global Coulombic cutoff specified in the pair_style command.

**Warning**

Because of how these pair styles implement the coulomb interactions by implicitly defining a fourth site for the negative charge of the TIP4P and similar water models, special care must be taken when using these pair styles with other computations that also use charges. Unless they are specially set up to also handle the implicit definition of the 4th site, results are likely incorrect. Example: *compute dipole/chunk*. For the same reason, when using one

of these pair styles with *pair_style hybrid*, **all** coulomb interactions should be handled by a single sub-style with TIP4P support. All other instances and styles will "see" the M point charges at the position of the Oxygen atom and thus compute incorrect forces and energies. LAMMPS will print a warning when it detects one of these issues.

---

A version of these styles with a soft core, *lj/cut/tip4p/long/soft*, suitable for use in free energy calculations, is part of the FEP package and is documented with the *pair_style */soft* styles.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.173.5 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon and sigma coefficients and cutoff distance for all of the lj/cut pair styles can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

All of the *lj/cut* pair styles support the *pair_modify* shift option for the energy of the Lennard-Jones portion of the pair interaction.

The *lj/cut/coul/long* and *lj/cut/tip4p/long* pair styles support the *pair_modify* table option since they can tabulate the short-range portion of the long-range Coulombic interaction.

All of the *lj/cut* pair styles support the *pair_modify* tail option for adding a long-range tail correction to the energy and pressure for the Lennard-Jones portion of the pair interaction.

All of the *lj/cut* pair styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

The *lj/cut* and *lj/cut/coul/long* pair styles support the use of the *inner*, *middle*, and *outer* keywords of the *run_style respa* command, meaning the pairwise forces can be partitioned by distance at different levels of the rRESPA hierarchy. The other styles only support the *pair* keyword of run_style respa. See the *run_style* command for details.

---

### 4.173.6 Restrictions

The *lj/cut/tip4p/long* styles are part of the KSPACE package. The *lj/cut/tip4p/cut* style is part of the MOLECULE package. These styles are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

### 4.173.7 Related commands

*pair_coeff*

### 4.173.8 Default

none

---

**(Jorgensen)** Jorgensen, Chandrasekhar, Madura, Impey, Klein, J Chem Phys, 79, 926 (1983).

## 4.174 pair_style lj/expand command

Accelerator Variants: *lj/expand/gpu*, *lj/expand/kk*, *lj/expand/omp*

## 4.175 pair_style lj/expand/coul/long command

Accelerator Variants: *lj/expand/coul/long/kk*, *lj/expand/coul/long/gpu*, *lj/expand/coul/long/omp*

### 4.175.1 Syntax

```
pair_style lj/expand cutoff
```

- cutoff = global cutoff for lj/expand interactions (distance units)

### 4.175.2 Examples

```
pair_style lj/expand 2.5
pair_coeff * * 1.0 1.0 0.5
pair_coeff 1 1 1.0 1.0 -0.2 2.0

pair_style lj/expand/coul/long 2.5
pair_style lj/expand/coul/long 2.5 4.0
pair_coeff * * 1.0 1.0 0.5
pair_coeff 1 1 1.0 1.0 -0.2 3.0
```

### 4.175.3 Description

Style *lj/expand* computes a LJ interaction with a distance shifted by delta which can be useful when particles are of different sizes, since it is different that using different sigma values in a standard LJ formula:

$$E = 4\varepsilon \left[ \left( \frac{\sigma}{r - \Delta} \right)^{12} - \left( \frac{\sigma}{r - \Delta} \right)^{6} \right] \qquad r < r_c + \Delta$$

$r_c$ is the cutoff which does not include the $\Delta$ distance. I.e. the actual force cutoff is the sum of $r_c + \Delta$.

For all of the *lj/expand* pair styles, the following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- $\varepsilon$ (energy units)
- $\sigma$ (distance units)
- $\Delta$ (distance units)
- cutoff (distance units)

The Δ values can be positive or negative. The last coefficient is optional. If not specified, the global LJ cutoff is used.

For *lj/expand/coul/long* only the LJ cutoff can be specified since a Coulombic cutoff cannot be specified for an individual I,J type pair. All type pairs use the same global Coulombic cutoff specified in the pair_style command.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.175.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon, sigma, and shift coefficients and cutoff distance for this pair style can be mixed. Shift is always mixed via an *arithmetic* rule. The other coefficients are mixed according to the pair_modify mix value. The default mix value is *geometric*. See the "pair_modify" command for details.

This pair style supports the *pair_modify* shift option for the energy of the pair interaction.

The *pair_modify* table option is not relevant for this pair style.

This pair style supports the *pair_modify* tail option for adding a long-range tail correction to the energy and pressure of the pair interaction.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.175.5 Restrictions

none

### 4.175.6 Related commands

*pair_coeff*

### 4.175.7 Default

none

## 4.176 pair_style lj/expand/sphere command

Accelerator Variants: *lj/expand/sphere/omp*

### 4.176.1 Syntax

```
pair_style style args
```

- style = *lj/expand/sphere*
- args = list of arguments for a particular style

```
lj/expand/sphere args = cutoff
  cutoff = global cutoff for Lennard Jones interactions (distance units)
```

### 4.176.2 Examples

```
pair_style lj/expand/sphere 2.5
pair_coeff * * 1.0 1.0
pair_coeff 1 1 1.1 0.4 2.8
```

### 4.176.3 Description

Added in version 15Jun2023.

The *lj/expand/sphere* style compute a 12/6 Lennard-Jones potential with a distance shifted by $\Delta = \frac{1}{2}(d_i+d_j)$, the average diameter of both atoms. This can be used to model particles of different sizes but same interactions, which is different from using different sigma values as in *pair style lj/cut/sphere*.

$$E = 4\varepsilon \left[ \left( \frac{\sigma}{r-\Delta} \right)^{12} - \left( \frac{\sigma}{r-\Delta} \right)^{6} \right] \qquad r < r_c + \Delta$$

$r_c$ is the cutoff which does not include the distance $\Delta$. I.e. the actual force cutoff is the sum $r_c + \Delta$.

This is the same potential function used by the *lj/expand* pair style, but the $\Delta$ parameter is not set as a per-type parameter via the *pair_coeff command*. Instead it is calculated individually for each pair using the per-atom diameter attribute of *atom_style sphere* for the two atoms as the average diameter, $\Delta = \frac{1}{2}(d_i + d_j)$

Note that $\sigma$ is defined in the LJ formula above as the zero-crossing distance for the potential, *not* as the energy minimum which is at $2^{\frac{1}{6}}\sigma$.

**Notes on cutoffs, neighbor lists, and efficiency**

If your system is mildly polydisperse, meaning the ratio of the diameter of the largest particle to the smallest is less than 2.0, then the neighbor lists built by the code should be reasonably efficient. Which means they will not contain too many particle pairs that do not interact. However, if your system is highly polydisperse (ratio > 2.0), the neighbor list build and force computations may be inefficient. There are two ways to try and speed up the simulations.

The first is to assign atoms to different atom types so that atoms of each type are similar in size. E.g. if particle diameters range from 1 to 5 use 4 atom types, ensuring atoms of type 1 have diameters from 1.0-2.0, type 2 from 2.0-3.0, etc. This will reduce the number of non-interacting pairs in the neighbor lists and thus reduce the time spent on computing pairwise interactions.

The second is to use the *neighbor multi* command which enabled a different algorithm for building neighbor lists. This will also require that you assign multiple atom types according to diameters, but will in addition use a more efficient size-dependent strategy to construct the neighbor lists and thus reduce the time spent on building neighbor lists.

Here are example input script commands using the first option for a highly polydisperse system:

```
units           lj
atom_style      sphere
lattice         fcc 0.8442
region          box block 0 10 0 10 0 10
create_box      2 box
create_atoms    1 box

# create atoms with random diameters from bimodal distribution
variable switch atom random(0.0,1.0,345634)
variable diam atom (v_switch<0.75)*normal(0.2,0.04,325)+(v_switch>=0.7)*normal(0.6,0.
↪2,453)
set group all diameter v_diam

# assign type 2 to atoms with diameter > 0.35
variable large atom (2.0*radius)>0.35
group large variable large
set group large type 2

pair_style      lj/expand/sphere 2.0
pair_coeff      * * 1.0 0.5

neighbor 0.3 bin
```

Using multiple atom types speeds up the calculation for this example by more than 30 percent, but using the multi-style neighbor list does not provide a speedup.

### 4.176.4 Coefficients

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- $\varepsilon$ (energy units)
- $\sigma$ (distance units)
- LJ cutoff (distance units) (optional)

The last coefficient is optional. If not specified, the global LJ cutoff specified in the *pair_style command* is used.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.176.5 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon, sigma, and cutoff coefficients for the *lj/expand/sphere* pair style can be mixed. The default mixing style is *geometric*. See the *pair_modify command* for details.

The *lj/expand/sphere* pair style supports the *pair_modify shift* option for the energy of the Lennard-Jones portion of the pair interaction.

The *lj/expand/sphere* pair style does *not* support the *pair_modify* tail option for adding a long-range tail corrections to the energy and pressure.

The *lj/expand/sphere* pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does *not* support the *inner*, *middle*, *outer* keywords.

---

### 4.176.6 Restrictions

The *lj/expand/sphere* pair style is only enabled if LAMMPS was built with the EXTRA-PAIR package. See the *Build package* page for more info.

---

### 4.176.7 Related commands

- *pair_coeff*
- *pair_style lj/cut*
- *pair_style lj/cut/sphere*

### 4.176.8 Default

none

## 4.177 pair_style lj/long/coul/long command

Accelerator Variants: *lj/long/coul/long/intel*, *lj/long/coul/long/omp*, *lj/long/coul/long/opt*

## 4.178 pair_style lj/long/tip4p/long command

Accelerator Variants: *lj/long/tip4p/long/omp*

### 4.178.1 Syntax

```
pair_style style args
```

- style = *lj/long/coul/long* or *lj/long/tip4p/long*
- args = list of arguments for a particular style

```
lj/long/coul/long args = flag_lj flag_coul cutoff (cutoff2)
  flag_lj = long or cut or off
    long = use Kspace long-range summation for dispersion 1/r^6 term
    cut = use a cutoff on dispersion 1/r^6 term
```

```
    off = omit dispersion 1/r^6 term entirely
  flag_coul = long or off
    long = use Kspace long-range summation for Coulombic 1/r term
    off = omit Coulombic term
  cutoff = global cutoff for LJ (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
lj/long/tip4p/long args = flag_lj flag_coul otype htype btype atype qdist cutoff␣
↪(cutoff2)
  flag_lj = long or cut
    long = use Kspace long-range summation for dispersion 1/r^6 term
    cut = use a cutoff
  flag_coul = long or off
    long = use Kspace long-range summation for Coulombic 1/r term
    off = omit Coulombic term
  otype,htype = atom types (numeric or type label) for TIP4P O and H
  btype,atype = bond and angle types (numeric or type label) for TIP4P waters
  qdist = distance from O atom to massless charge (distance units)
  cutoff = global cutoff for LJ (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
```

### 4.178.2 Examples

```
pair_style lj/long/coul/long cut off 2.5
pair_style lj/long/coul/long cut long 2.5 4.0
pair_style lj/long/coul/long long long 2.5 4.0
pair_coeff * * 1 1
pair_coeff 1 1 1 3 4

pair_style lj/long/tip4p/long long long 1 2 7 8 0.15 12.0
pair_style lj/long/tip4p/long long long 1 2 7 8 0.15 12.0 10.0
pair_coeff * * 100.0 3.0
pair_coeff 1 1 100.0 3.5 9.0

pair_style lj/long/tip4p/long long long OW HW HW-OW HW-OW-HW 0.15 12.0
labelmap atom 1 OW 2 HW
labelmap bond 1 HW-OW
labelmap angle 1 HW-OW-HW
pair_coeff * * 100.0 3.0
pair_coeff OW OW 100.0 3.5 9.0
```

### 4.178.3 Description

Style *lj/long/coul/long* computes the standard 12/6 Lennard-Jones potential:

$$E = 4\varepsilon \left[ \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \qquad r < r_c$$

with $\varepsilon$ and $\sigma$ being the usual Lennard-Jones potential parameters, plus the Coulomb potential, given by:

$$E = \frac{Cq_iq_j}{\varepsilon r} \qquad r < r_c$$

where C is an energy-conversion constant, $q_i$ and $q_j$ are the charges on the two atoms, $\varepsilon$ is the dielectric constant which can be set by the *dielectric* command, and $r_c$ is the cutoff. If one cutoff is specified in the pair_style command, it is used for both the LJ and Coulombic terms. If two cutoffs are specified, they are used as cutoffs for the LJ and Coulombic terms respectively.

The purpose of this pair style is to capture long-range interactions resulting from both attractive 1/r^6 Lennard-Jones and Coulombic 1/r interactions. This is done by use of the *flag_lj* and *flag_coul* settings. The *In 't Veld* paper has more details on when it is appropriate to include long-range 1/r^6 interactions, using this potential.

Style *lj/long/tip4p/long* implements the TIP4P water model of *(Jorgensen)*, which introduces a massless site located a short distance away from the oxygen atom along the bisector of the HOH angle. The atomic types of the oxygen and hydrogen atoms, the bond and angle types for OH and HOH interactions, and the distance to the massless charge site are specified as pair_style arguments.

**Note**

For each TIP4P water molecule in your system, the atom IDs for the O and 2 H atoms must be consecutive, with the O atom first. This is to enable LAMMPS to "find" the 2 H atoms associated with each O atom. For example, if the atom ID of an O atom in a TIP4P water molecule is 500, then its 2 H atoms must have IDs 501 and 502.

**Note**

If using type labels, the type labels must be defined before calling the *pair_coeff* command.

See the *Howto tip4p* page for more information on how to use the TIP4P pair style. Note that the neighbor list cutoff for Coulomb interactions is effectively extended by a distance 2*qdist when using the TIP4P pair style, to account for the offset distance of the fictitious charges on O atoms in water molecules. Thus it is typically best in an efficiency sense to use a LJ cutoff >= Coulombic cutoff + 2*qdist, to shrink the size of the neighbor list. This leads to slightly larger cost for the long-range calculation, so you can test the trade-off for your model.

If *flag_lj* is set to *long*, no cutoff is used on the LJ 1/r^6 dispersion term. The long-range portion can be calculated by using the *kspace_style ewald/disp or pppm/disp* commands. The specified LJ cutoff then determines which portion of the LJ interactions are computed directly by the pair potential versus which part is computed in reciprocal space via the Kspace style. If *flag_lj* is set to *cut*, the LJ interactions are simply cutoff, as with *pair_style lj/cut*.

If *flag_coul* is set to *long*, no cutoff is used on the Coulombic interactions. The long-range portion can calculated by using any of several *kspace_style* command options such as *pppm* or *ewald*. Note that if *flag_lj* is also set to long, then the *ewald/disp* or *pppm/disp* Kspace style needs to be used to perform the long-range calculations for both the LJ and Coulombic interactions. If *flag_coul* is set to *off*, Coulombic interactions are not computed.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- $\varepsilon$ (energy units)
- $\sigma$ (distance units)
- cutoff1 (distance units)
- cutoff2 (distance units)

Note that sigma is defined in the LJ formula as the zero-crossing distance for the potential, not as the energy minimum at 2^(1/6) sigma.

The latter 2 coefficients are optional. If not specified, the global LJ and Coulombic cutoffs specified in the pair_style command are used. If only one cutoff is specified, it is used as the cutoff for both LJ and Coulombic interactions for this type pair. If both coefficients are specified, they are used as the LJ and Coulombic cutoffs for this type pair.

Note that if you are using *flag_lj* set to *long*, you cannot specify a LJ cutoff for an atom type pair, since only one global LJ cutoff is allowed. Similarly, if you are using *flag_coul* set to *long*, you cannot specify a Coulombic cutoff for an atom type pair, since only one global Coulombic cutoff is allowed.

For *lj/long/tip4p/long* only the LJ cutoff can be specified since a Coulombic cutoff cannot be specified for an individual I,J type pair. All type pairs use the same global Coulombic cutoff specified in the pair_style command.

---

A version of these styles with a soft core, *lj/cut/soft*, suitable for use in free energy calculations, is part of the FEP package and is documented with the *pair_style */soft* styles. The version with soft core is only available if LAMMPS was built with that package. See the *Build package* page for more info.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.178.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon and sigma coefficients and cutoff distance for all of the lj/long pair styles can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

These pair styles support the *pair_modify* shift option for the energy of the Lennard-Jones portion of the pair interaction, assuming *flag_lj* is *cut*.

These pair styles support the *pair_modify* table and table/disp options since they can tabulate the short-range portion of the long-range Coulombic and dispersion interactions.

These pair styles do not support the *pair_modify* tail option for adding a long-range tail correction to the Lennard-Jones portion of the energy and pressure.

These pair styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

The pair lj/long/coul/long styles support the use of the *inner*, *middle*, and *outer* keywords of the *run_style respa* command, meaning the pairwise forces can be partitioned by distance at different levels of the rRESPA hierarchy. See the *run_style* command for details.

---

### 4.178.5 Restrictions

These styles are part of the KSPACE package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.178.6 Related commands

*pair_coeff*

### 4.178.7 Default

none

---

**(In 't Veld)** In 't Veld, Ismail, Grest, J Chem Phys, 127, 144711 (2007).

**(Jorgensen)** Jorgensen, Chandrasekhar, Madura, Impey, Klein, J Chem Phys, 79, 926 (1983).

## 4.179 pair_style lj/pirani command

Accelerator Variants: *lj/pirani/omp*, *lj/pirani/kk*

### 4.179.1 Syntax

```
pair_style lj/pirani cutoff
```

- lj/pirani = name of the pair style
- cutoff = global cutoff (distance units)

### 4.179.2 Examples

```
pair_style lj/pirani 10.0
pair_coeff 1 1 4.0 7.0 6.0 3.5 0.0045
```

### 4.179.3 Description

Added in version 12Jun2025.

Pair style *lj/pirani* computes pairwise interactions from an Improved Lennard-Jones (ILJ) potential according to *(Pirani)*. The ILJ force field is adequate to model both equilibrium and non-equilibrium properties of matter, in gaseous and condensed phases, and at gas-surface interfaces. In particular, its use improves the description of elementary process dynamics where the traditional Lennard-Jones (LJ) formulation is usually applied.

$$x = r/R_m$$
$$n_x = \alpha * x^2 + \beta$$
$$\gamma \equiv m$$

$$V(x) = \varepsilon \cdot \left( \frac{\gamma}{n_x - \gamma} \left( \frac{1}{x} \right)^{n_x} - \frac{n_x}{n_x - \gamma} \left( \frac{1}{x} \right)^{\gamma} \right) \qquad r < r_c$$

$r_c$ is the cutoff.

An additional parameter, $\alpha$, has been introduced in order to be able to recover the traditional Lennard-Jones 12-6 with a specific choice of parameters. With $R_m \equiv r_0 = \sigma \cdot 2^{1/6}$, $\alpha = 0$, $\beta = 12$ and $\gamma = 6$ it is straightforward to prove that LJ 12-6 is obtained. Also, it can be verified that using $\alpha = 4$, $\beta = 8$ and $\gamma = 6$, at the equilibrium distance, the first and second derivatives of ILJ match those of LJ 12-6. The parameter $R_m$ corresponds to the equilibrium distance and $\varepsilon$ to the well depth.

This potential provides some advantages with respect to the standard LJ potential, as explained in *(Pirani)*: it provides a more realistic description of the long range behavior and an attenuation of the hardness of the repulsive wall.

This force field can be used for neutral-neutral ($\gamma = 6$), ion-neutral ($\gamma = 4$) or ion-ion systems ($\gamma = 1$). Notice that this implementation does not include explicit electrostatic interactions. If these are desired, this pair style should be used

along with a Coulomb pair style like *pair styles coul/cut or coul/long* by using *pair style hybrid/overlay* and a suitable *kspace style*, if needed.

As discussed in *(Pirani)*, analysis of a variety of systems showed that $\alpha = 4$ generally works very well. In some special cases (e.g. those involving very small multiple charged ions) this factor may take a slightly different value. The parameter $\beta$ codifies the hardness (polarizability) of the interacting partners, and for neutral-neutral systems it usually ranges from 6 to 11. Moreover, the modulation of $\beta$ can model additional interaction effects, such as charge transfer in the perturbative limit, and can mitigate the effect of some uncertainty in the data used to build up the potential function.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $\alpha$ (dimensionless)
- $\beta$ (dimensionless)
- $\gamma$ (dimensionless)
- $R_m$ (distance units)
- $\varepsilon$ (energy units)
- cutoff (distance units)

The last coefficient is optional. If not specified, the global cutoff is used.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.179.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

This pair style supports the *pair_modify* shift option for the energy of the pair interaction.

The *pair_modify* table options are not relevant for this pair style.

This pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style supports the use of the *inner*, *middle*, and *outer* keywords of the *run_style respa* command, meaning the pairwise forces can be partitioned by distance at different levels of the rRESPA hierarchy. See the *run_style* command for details.

---

### 4.179.5 Restrictions

This pair style is only enabled if LAMMPS was built with the EXTRA-PAIR package. See the *Build package* page for more info.

### 4.179.6 Related commands

- *pair_coeff*
- *pair_style lj/cut*

### 4.179.7 Default

none

---

**(Pirani)** F. Pirani, S. Brizi, L. Roncaratti, P. Casavecchia, D. Cappelletti and F. Vecchiocattivi, Phys. Chem. Chem. Phys., 2008, 10, 5489-5503.

## 4.180 pair_style lj/relres command

Accelerator Variants: *lj/relres/omp*

### 4.180.1 Syntax

```
pair_style lj/relres Rsi Rso Rci Rco
```

- Rsi = inner switching cutoff between the fine-grained and coarse-grained potentials (distance units)
- Rso = outer switching cutoff between the fine-grained and coarse-grained potentials (distance units)
- Rci = inner cutoff beyond which the force smoothing for all interactions is applied (distance units)
- Rco = outer cutoff for all interactions (distance units)

### 4.180.2 Examples

```
pair_style lj/relres 4.0 5.0 8.0 10.0
pair_coeff 1 1 0.5 1.0 1.5 1.1
pair_coeff 2 2 0.5 1.0 0.0 0.0 3.0 3.5 6.0 7.0
```

### 4.180.3 Description

Pair style *lj/relres* computes a LJ interaction using the Relative Resolution (RelRes) framework which applies a fine-grained (FG) potential between near neighbors and a coarse-grained (CG) potential between far neighbors *(Chaimovich1)*. This approach can improve the computational efficiency by almost an order of magnitude, while maintaining the correct static and dynamic behavior of a reference system *(Chaimovich2)*.

$$E = \begin{cases} 4\varepsilon^{FG}\left[\left(\frac{\sigma^{FG}}{r}\right)^{12} - \left(\frac{\sigma^{FG}}{r}\right)^{6}\right] - \Gamma_{si}, & \text{if} \quad r < r_{si}, \\ \sum_{m=0}^{4} \gamma_{sm}\,(r - r_{si})^{m} - \Gamma_{so}, & \text{if} \quad r_{si} \le r < r_{so}, \\ 4\varepsilon^{CG}\left[\left(\frac{\sigma^{CG}}{r}\right)^{12} - \left(\frac{\sigma^{CG}}{r}\right)^{6}\right] - \Gamma_{c}, & \text{if} \quad r_{so} \le r < r_{ci}, \\ \sum_{m=0}^{4} \gamma_{cm}\,(r - r_{ci})^{m} - \Gamma_{c}, & \text{if} \quad r_{ci} \le r < r_{co}, \\ 0, & \text{if} \quad r \ge r_{co}. \end{cases}$$

The FG parameters of the LJ potential ($\varepsilon^{FG}$ and $\sigma^{FG}$) are applied up to the inner switching cutoff, $r_{si}$, while the CG parameters of the LJ potential ($\varepsilon^{CG}$ and $\sigma^{CG}$) are applied beyond the outer switching cutoff, $r_{so}$. Between $r_{si}$ and $r_{so}$ a polynomial smoothing function is applied so that the force and its derivative are continuous between the FG and CG potentials. An analogous smoothing function is applied between the inner and outer cutoffs ($r_{ci}$ and $r_{co}$). The offsets $\Gamma_{si}$, $\Gamma_{so}$ and $\Gamma_c$ ensure the continuity of the energy over the entire domain. The corresponding polynomial coefficients $\gamma_{sm}$ and $\gamma_{cm}$, as well as the offsets are automatically computed by LAMMPS.

**Note**

Energy and force resulting from this methodology can be plotted via the *pair_write* command.

The following coefficients must be defined for each pair of atom types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as will be described below:

- $\varepsilon^{FG}$ (energy units)
- $\sigma^{FG}$ (distance units)
- $\varepsilon^{CG}$ (energy units)
- $\sigma^{CG}$ (distance units)

Additional parameters can be defined to specify different $r_{si}$, $r_{so}$, $r_{ci}$, $r_{co}$ for a particular set of atom types:

- $r_{si}$ (distance units)
- $r_{so}$ (distance units)
- $r_{ci}$ (distance units)
- $r_{co}$ (distance units)

These parameters are optional, and they are used to override the global cutoffs as defined in the pair_style command. If not specified, the global values for $r_{si}$, $r_{so}$, $r_{ci}$, and $r_{co}$ are used. If this override option is employed, all four arguments must be specified.

---

Here are some guidelines for using the pair_style *lj/relres* command.

In general, RelRes focuses on the speedup of pairwise interactions between all LJ sites. Importantly, it works with any settings and flags (e.g., *special_bonds* settings and *newton* flags) that can be used in a molecular simulation with the conventional LJ potential. In particular, all intramolecular topology with its energetics (i.e., bonds, angles, etc.) remains unaltered.

At the most basic level in the RelRes framework, all sites are mapped into clusters. Each cluster is just a collection of sites bonded together (the bonds themselves are not part of the cluster). In general, a molecule may be comprised of several clusters, and preferably, no two sites in a cluster are separated by more than two bonds. There are two categories of sites in RelRes: "hybrid" sites embody both FG and CG models, while "ordinary" sites embody just FG characteristics with no CG features. A given cluster has a single hybrid site (typically its central site) and several ordinary sites (typically its peripheral sites). Notice that while clusters are necessary for the RelRes parameterization (discussed below), they are not actually defined in LAMMPS. Besides, the total number of sites in the cluster are called the "mapping ratio", and this substantially impacts the computational efficiency of RelRes: For a mapping ratio of 3, the efficiency factor is around 4, and for a mapping ratio of 5, the efficiency factor is around 5 *(Chaimovich2)*.

The flexibility of LAMMPS allows placing any values for the LJ parameters in the input script. However, here are the optimal recommendations for the RelRes parameters, which yield the correct structural and thermal behavior in a system of interest *(Chaimovich1)*. One must first assign a complete set of parameters for the FG interactions that are applicable to all atom types. Regarding the parameters for the CG interactions, the rules rely on the site category (if it

is a hybrid or an ordinary site). For atom types of ordinary sites, $\varepsilon^{CG}$ must be set to 0 (zero) while the specific value of $\sigma^{CG}$ is irrelevant. For atom types of hybrid sites, the CG parameters should be generally calculated using the following equations:

$$\sigma_I^{CG} = \frac{\left(\left(\sum_{\alpha \in A} \sqrt{\varepsilon_\alpha^{FG} \left(\sigma_\alpha^{FG}\right)^{12}}\right)^{1/2}\right.}{\left(\left(\sum_{\alpha \in A} \sqrt{\varepsilon_\alpha^{FG} \left(\sigma_\alpha^{FG}\right)^{6}}\right)^{1/3}\right.} \quad \text{and} \quad \varepsilon_I^{CG} = \frac{\left(\left(\sum_{\alpha \in A} \sqrt{\varepsilon_\alpha^{FG} \left(\sigma_\alpha^{FG}\right)^{6}}\right)^{4}\right.}{\left(\left(\sum_{\alpha \in A} \sqrt{\varepsilon_\alpha^{FG} \left(\sigma_\alpha^{FG}\right)^{12}}\right)^{2}\right.}$$

where $I$ is an atom type of a hybrid site of a particular cluster $A$, and corresponding with this cluster, the summation proceeds over all of its sites $\alpha$. These equations are derived from the monopole term in the underlying Taylor series, and they are indeed relevant only if geometric mixing is applicable for the FG model; if this is not the case, Ref. *(Chaimovich2)* discusses the alternative formula, and in such a situation, the pair_coeff command should be explicitly used for all combinations of atom types $I$ ! = $J$.

The switching distance (the midpoint between inner and outer switching cutoffs) is another crucial factor in RelRes: decreasing it improves the computational efficiency, yet if it is too small, the molecular simulations may not capture the system behavior correctly. As a rule of thumb, the switching distance should be approximately $\sim 1.5\sigma$ *(Chaimovich1)*; recommendations can be found in Ref. *(Chaimovich2)*. Regarding the switching smoothing zone, $\sim 0.1\sigma$ is recommended; if desired, smoothing can be eliminated by setting the inner switching cutoff, $r_{si}$, equal to the outer switching cutoff, $r_{so}$ (the same is true for the other cutoffs $r_{ci}$ and $r_{co}$).

---

As an example, imagine that in your system, a molecule is comprised just of one cluster such that one atom type (#1) is associated with its hybrid site, and another atom type (#2) is associated with its ordinary sites (in total, there are 2 atom types). If geometric mixing is applicable, the following commands should be used:

```
pair_style lj/relres Rsi Rso Rci Rco
pair_coeff 1 1 epsilon_FG1 sigma_FG1 epsilon_CG1 sigma_CG1
pair_coeff 2 2 epsilon_FG2 sigma_FG2 0.0         0.0
pair_modify shift yes
```

In a more complex situation, there may be two distinct clusters in a system (these two clusters may be on same molecule or on different molecules), each with its own switching cutoffs. If there are still two atom types in each cluster as in the earlier example, the commands should be:

```
pair_style lj/relres Rsi Rso Rci Rco
pair_coeff 1 1 epsilon_FG1 sigma_FG1 epsilon_CG1 sigma_CG1 Rsi1 Rso1 Rci Rco
pair_coeff 2 2 epsilon_FG2 sigma_FG2 0.0         0.0       Rsi1 Rso1 Rci Rco
pair_coeff 3 3 epsilon_FG3 sigma_FG3 epsilon_CG3 sigma_CG3
pair_coeff 4 4 epsilon_FG4 sigma_FG4 0.0         0.0
pair_modify shift yes
```

In this example, the switching cutoffs for the first cluster (atom types 1 and 2) is defined explicitly in the pair_coeff command which overrides the global values, while the second cluster (atom types 3 and 4) uses the global definition from the pair_style command. The emphasis here is that the atom types that belong to a specific cluster should have the same switching/cutoff arguments.

In the case that geometric mixing is not applicable, for simulating the system from the previous example, we recommend using the following commands:

```
pair_style lj/relres Rsi Rso Rci Rco
pair_coeff 1 1 epsilon_FG1  sigma_FG1  epsilon_CG1  sigma_CG1  Rsi1  Rso1  Rci Rco
pair_coeff 1 2 epsilon_FG12 sigma_FG12 0.0          0.0        Rsi1  Rso1  Rci Rco
pair_coeff 1 3 epsilon_FG13 sigma_FG13 epsilon_CG13 sigma_CG13 Rsi13 Rso13 Rci Rco
```

(continues on next page)

(continued from previous page)

```
pair_coeff 1 4 epsilon_FG14 sigma_FG14 0.0          0.0        Rsi13 Rso13 Rci Rco
pair_coeff 2 2 epsilon_FG2  sigma_FG2  0.0          0.0        Rsi1  Rso1  Rci Rco
pair_coeff 2 3 epsilon_FG23 sigma_FG23 0.0          0.0        Rsi13 Rso13 Rci Rco
pair_coeff 2 4 epsilon_FG24 sigma_FG24 0.0          0.0        Rsi13 Rso13 Rci Rco
pair_coeff 3 3 epsilon_FG3  sigma_FG3  epsilon_CG3  sigma_CG3
pair_coeff 3 4 epsilon_FG34 sigma_FG34 0.0          0.0
pair_coeff 4 4 epsilon_FG4  sigma_FG4  0.0          0.0
pair_modify shift yes
```

Notice that the CG parameters are mixed only for interactions between atom types associated with hybrid sites, and that the cutoffs are mixed on the cluster basis.

More examples can be found in the *examples/relres* folder.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.180.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs $I$, $J$ with $I$ != $J$, the $\varepsilon^{FG}$, $\sigma^{FG}$, $\varepsilon^{CG}$, $\sigma^{CG}$, $r_{si}$, $r_{so}$, $r_{ci}$, and $r_{co}$ parameters for this pair style can be mixed, if not defined explicitly. All parameters are mixed according to the pair_modify mix option. The default mix value is *geometric*, and it is recommended to use with this *lj/relres* style. See the "pair_modify" command for details.

This pair style supports the *pair_modify* shift option for the energy of the pair interaction. It is recommended to set this option to *yes*. Otherwise, the offset $\Gamma_c$ is set to zero. Constants $\Gamma_{si}$ and $\Gamma_{so}$ are not impacted by this option.

The *pair_modify* table option is not relevant for this pair style.

This pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure, since the energy of the pair interaction is smoothed to 0.0 at the cutoff.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.180.5 Restrictions

This pair style is part of the EXTRA-PAIR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.180.6 Related commands

*pair_coeff*

### 4.180.7 Default

none

---

**(Chaimovich1)** A. Chaimovich, C. Peter and K. Kremer, J. Chem. Phys. 143, 243107 (2015).

**(Chaimovich2)** M. Chaimovich and A. Chaimovich, J. Chem. Theory Comput. 17, 1045-1059 (2021).

## 4.181 pair_style lj/smooth command

Accelerator Variants: *lj/smooth/gpu*, *lj/smooth/kk*, *lj/smooth/omp*

### 4.181.1 Syntax

```
pair_style lj/smooth Rin Rc
```

- Rin = inner cutoff beyond which force smoothing will be applied (distance units)
- Rc = outer cutoff for lj/smooth interactions (distance units)

### 4.181.2 Examples

```
pair_style lj/smooth 8.0 10.0
pair_coeff * * 10.0 1.5
pair_coeff 1 1 20.0 1.3 7.0 9.0
```

### 4.181.3 Description

Style *lj/smooth* computes a LJ interaction with a force smoothing applied between the inner and outer cutoff.

$$E = 4\varepsilon \left[ \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \qquad r < r_{in}$$

$$F = C_1 + C_2(r - r_{in}) + C_3(r - r_{in})^2 + C_4(r - r_{in})^3 \qquad r_{in} < r < r_c$$

The polynomial coefficients C1, C2, C3, C4 are computed by LAMMPS to cause the force to vary smoothly from the inner cutoff $r_{in}$ to the outer cutoff $r_c$.

At the inner cutoff the force and its first derivative will match the non-smoothed LJ formula. At the outer cutoff the force and its first derivative will be 0.0. The inner cutoff cannot be 0.0.

Explicit expressions for the coefficients C1, C2, C3, C4, as well as the energy discontinuity at the cutoff can be found here *(Leoni_1)* and here *(Leoni_2)*

**Note**

this force smoothing causes the energy to be discontinuous both in its values and first derivative. This can lead to poor energy conservation and may require the use of a thermostat. The energy value discontinuity can be eliminated by shifting the potential energy to be zero at the outer cutoff using the pair_modify shift option. With or without shifting, you can plot the resulting energy and force via the *pair_write* command to see the effect.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- $\varepsilon$ (energy units)
- $\sigma$ (distance units)
- $r_{in}$ (distance units)
- $r_c$ (distance units)

The last 2 coefficients are optional inner and outer cutoffs. If not specified, the global values for $r_{in}$ and $r_c$ are used.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.181.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon, sigma, Rin coefficients and the cutoff distance for this pair style can be mixed. Rin is a cutoff value and is mixed like the cutoff. The other coefficients are mixed according to the pair_modify mix option. The default mix value is *geometric*. See the "pair_modify" command for details.

This pair style supports the *pair_modify* shift option for the energy of the pair interaction.

The *pair_modify* table option is not relevant for this pair style.

This pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure, since the energy of the pair interaction is smoothed to 0.0 at the cutoff.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.181.5 Restrictions

This pair style is part of the EXTRA-PAIR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.181.6 Related commands

*pair_coeff*, *pair lj/smooth/linear*

### 4.181.7 Default

none

---

**(Leoni_1)** F. Leoni et al., Phys Rev Lett, 134, 128201 (2025).

**(Leoni_2)** F. Leoni et al., Phys Rev Lett, 134, Supplementary Material (2025).

## 4.182 pair_style lj/smooth/linear command

Accelerator Variants: *lj/smooth/linear/omp*

### 4.182.1 Syntax

```
pair_style lj/smooth/linear cutoff
```

- cutoff = global cutoff for Lennard-Jones interactions (distance units)

### 4.182.2 Examples

```
pair_style lj/smooth/linear 2.5
pair_coeff * * 1.0 1.0
pair_coeff 1 1 0.3 3.0 9.0
```

### 4.182.3 Description

Style *lj/smooth/linear* computes a truncated and force-shifted LJ interaction (aka Shifted Force Lennard-Jones) that combines the standard 12/6 Lennard-Jones function and subtracts a linear term based on the cutoff distance, so that both, the potential and the force, go continuously to zero at the cutoff $r_c$ *(Toxvaerd)*:

$$\phi(r) = 4\varepsilon \left[ \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]$$

$$E(r) = \phi(r) - \phi(r_c) - (r - r_c) \left. \frac{d\phi}{dr} \right|_{r=r_c} \qquad r < r_c$$

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- $\varepsilon$ (energy units)
- $\sigma$ (distance units)
- cutoff (distance units)

The last coefficient is optional. If not specified, the global LJ cutoff specified in the pair_style command is used.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.182.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon and sigma coefficients and cutoff distance can be mixed. The default mix value is geometric. See the "pair_modify" command for details.

This pair style does not support the *pair_modify* shift option for the energy of the pair interaction, since it goes to 0.0 at the cutoff by construction.

The *pair_modify* table option is not relevant for this pair style.

This pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure, since the energy of the pair interaction is smoothed to 0.0 at the cutoff.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.182.5 Restrictions

This pair style is part of the EXTRA-PAIR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.182.6 Related commands

*pair_coeff* , *pair lj/smooth*

### 4.182.7 Default

none

---

**(Toxvaerd)** Toxvaerd, Dyre, J Chem Phys, 134, 081102 (2011).

## 4.183 pair_style lj/switch3/coulgauss/long command

Accelerator Variants: *lj/switch3/coulgauss/long/kk*, *lj/switch3/coulgauss/long/omp*

## 4.184 pair_style mm3/switch3/coulgauss/long command

Accelerator Variants: *mm3/switch3/coulgauss/long/kk*, *mm3/switch3/coulgauss/long/omp*

### 4.184.1 Syntax

```
pair_style style args
```

- style = *lj/switch3/coulgauss/long* or *mm3/switch3/coulgauss/long*
- args = list of arguments for a particular style

```
lj/switch3/coulgauss/long args = cutoff (cutoff2) width
  cutoff  = global cutoff for LJ (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
  width   = width parameter of the smoothing function (distance units)

mm3/switch3/coulgauss/long args = cutoff (cutoff2) width
  cutoff  = global cutoff for MM3 (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
  width   = width parameter of the smoothing function (distance units)
```

### 4.184.2 Examples

```
pair_style lj/switch3/coulgauss/long    12.0 3.0
pair_coeff 1  0.2 2.5 1.2

pair_style lj/switch3/coulgauss/long   12.0 10.0 3.0
pair_coeff 1  0.2 2.5 1.2

pair_style mm3/switch3/coulgauss/long    12.0 3.0
pair_coeff 1  0.2 2.5 1.2

pair_style mm3/switch3/coulgauss/long   12.0 10.0 3.0
pair_coeff 1  0.2 2.5 1.2
```

### 4.184.3 Description

The *lj/switch3/coulgauss* style evaluates the LJ vdW potential

$$E = 4\varepsilon \left[ \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]$$

The *mm3/switch3/coulgauss/long* style evaluates the MM3 vdW potential *(Allinger)*

$$E = \varepsilon_{ij} \left[ -2.25 \left(\frac{r_{v,ij}}{r_{ij}}\right)^{6} + 1.84(10)^5 \exp\left[-12.0 r_{ij}/r_{v,ij}\right] \right] S_3(r_{ij})$$

$$r_{v,ij} = r_{v,i} + r_{v,j}$$

$$\varepsilon_{ij} = \sqrt{\varepsilon_i \varepsilon_j}$$

Both potentials go smoothly to zero at the cutoff r_c as defined by the switching function

$$S_3(r) = \begin{cases} 1 & \text{if} \quad r < r_c - w \\ 3x^2 - 2x^3 & \text{if} \quad r < r_c \quad \text{with} \quad x = \frac{r_c - r}{w} \\ 0 & \text{if} \quad r >= r_c \end{cases}$$

where w is the width defined in the arguments. This potential is combined with Coulomb interaction between Gaussian charge densities:

$$E = \frac{q_i q_j \mathrm{erf}\left(r/\sqrt{\gamma_1^2 + \gamma_2^2}\right)}{\varepsilon r_{ij}}$$

where $q_i$ and $q_j$ are the charges on the two atoms, $\varepsilon$ is the dielectric constant which can be set by the *dielectric* command, $\gamma_i$ and $\gamma_j$ are the widths of the Gaussian charge distribution and erf() is the error-function. This style has to be used in conjunction with the *kspace_style* command

If one cutoff is specified it is used for both the vdW and Coulomb terms. If two cutoffs are specified, the first is used as the cutoff for the vdW terms, and the second is the cutoff for the Coulombic term.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $\varepsilon$ (energy)
- $\sigma$ (distance)
- $\gamma$ (distance)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.184.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon and sigma coefficients and cutoff distance for all of the lj/long pair styles can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

Shifting the potential energy is not necessary because the switching function ensures that the potential is zero at the cut-off.

These pair styles support the *pair_modify* table and options since they can tabulate the short-range portion of the long-range Coulombic interactions.

Thes pair styles do not support the *pair_modify* tail option for adding a long-range tail correction to the Lennard-Jones portion of the energy and pressure.

These pair styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

These pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

### 4.184.5 Restrictions

These styles are part of the YAFF package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.184.6 Related commands

*pair_coeff*

### 4.184.7 Default

none

## 4.185 pair_style local/density command

Accelerator Variants: *local/density/omp*

### 4.185.1 Syntax

```
pair_style style arg
```

- style = *local/density*
- arg = name of file containing tabulated values of local density and the potential

### 4.185.2 Examples

```
pair_style local/density benzene_water.localdensity.table

pair_style hybrid/overlay table spline 500 local/density
pair_coeff * * local/density  benzene_water.localdensity.table
```

### 4.185.3 Description

The local density (LD) potential is a mean-field manybody potential, and, in some way, a generalization of embedded atom models (EAM). The name "local density potential" arises from the fact that it assigns an energy to an atom depending on the number of neighboring atoms of a given type around it within a predefined spherical volume (i.e., within the cutoff). The bottom-up coarse-graining (CG) literature suggests that such potentials can be widely useful in capturing effective multibody forces in a computationally efficient manner and thus improve the quality of CG models of implicit solvation *(Sanyal1)* and phase-segregation in liquid mixtures *(Sanyal2)*, and provide guidelines to determine the extent of manybody correlations present in a CG model *(Rosenberger)*. The LD potential in LAMMPS is primarily intended to be used as a corrective potential over traditional pair potentials in bottom-up CG models via *hybrid/overlay pair style* with other explicit pair interaction terms (e.g., tabulated, Lennard-Jones, Morse etc.). Because the LD potential is not a pair potential per se, it is implemented simply as a single auxiliary file with all specifications that will be read upon initialization.

**Note**

Thus when used as the only interaction in the system, there is no corresponding pair_coeff command and when used with other pair styles using the hybrid/overlay option, the corresponding pair_coeff command must be supplied * * as placeholders for the atom types.

---

**System with a single CG atom type:**

A system of a single atom type (e.g., LJ argon) with a single local density (LD) potential would have an energy given by:

$$U_{LD} = \sum_i F(\rho_i)$$

where $\rho_i$ is the LD at atom $i$ and $F(\rho)$ is similar in spirit to the embedding function used in EAM potentials. The LD at atom $i$ is given by the sum

$$\rho_i = \sum_{j \neq i} \varphi(r_{ij})$$

where $\varphi$ is an indicator function that is one at r=0 and zero beyond a cutoff distance R2. The choice of the functional form of $\varphi$ is somewhat arbitrary, but the following piecewise cubic function has proven sufficiently general: *(Sanyal1)*, *(Sanyal2)* *(Rosenberger)*

$$\varphi(r) = \begin{cases} 1 & r \leq R_1 \\ c_0 + c_2 r^2 + c_4 r^4 + c_6 r^6 & r \in (R_1, R_2) \\ 0 & r \geq R_2 \end{cases}$$

The constants $c$ are chosen so that the indicator function smoothly interpolates between 1 and 0 between the distances R1 and R2, which are called the inner and outer cutoffs, respectively. Thus phi satisfies phi(R1) = 1, phi(R2) = dphi/dr @ (r=R1) = dphi/dr @ (r=R2) = 0. The embedding function F(rho) may or may not have a closed-form expression. To maintain generality, it is practically represented with a spline-interpolated table over a predetermined range of rho. Outside of that range it simply adopts zero values at the endpoints.

It can be shown that the total force between two atoms due to the LD potential takes the form of a pair force, which motivates its designation as a LAMMPS pair style. Please see *(Sanyal1)* for details of the derivation.

---

**Systems with arbitrary numbers of atom types:**

The potential is easily generalized to systems involving multiple atom types:

$$U_{LD} = \sum_i a_\alpha F(\rho_i)$$

with the LD expressed as

$$\rho_i = \sum_{j \neq i} b_\beta \varphi(r_{ij})$$

where $\alpha$ gives the type of atom $i$, $\beta$ the type of atom $j$, and the coefficients $a$ and $b$ filter for atom types as specified by the user. $a$ is called the central atom filter as it determines to which atoms the potential applies; $a_\alpha = 1$ if the LD potential applies to atom type $\alpha$ else zero. On the other hand, $b$ is called the neighbor atom filter because it specifies which atom types to use in the calculation of the LD; $b_\beta = 1$ if atom type $\beta$ contributes to the LD and zero otherwise.

**Note**

Note that the potentials need not be symmetric with respect to atom types, which is the reason for two distinct sets of coefficients $a$ and $b$. An atom type may contribute to the LD but not the potential, or to the potential but not the LD. Such decisions are made by the user and should (ideally) be motivated on physical grounds for the problem at hand.

---

**General form for implementation in LAMMPS:**

Of course, a system with many atom types may have many different possible LD potentials, each with their own atom type filters, cutoffs, and embedding functions. The most general form of this potential as implemented in the pair_style local/density is:

$$U_{LD} = \sum_k U_{LD}^{(k)} = \sum_i \left[ \sum_k a_\alpha^{(k)} F^{(k)} \left( \rho_i^{(k)} \right) \right]$$

where, $k$ is an index that spans the (arbitrary) number of applied LD potentials N_LD. Each LD is calculated as before with:

$$\rho_i^{(k)} = \sum_j b_\beta^{(k)} \varphi^{(k)}(r_{ij})$$

The superscript on the indicator function phi simply indicates that it is associated with specific values of the cutoff distances R1(k) and R2(k). In summary, there may be N_LD distinct LD potentials. With each potential type (k), one must specify:

- the inner and outer cutoffs as R1 and R2
- the central type filter a(k), where k = 1,2,...N_LD
- the neighbor type filter b(k), where k = 1,2,...N_LD
- the LD potential function F(k)(rho), typically as a table that is later spline-interpolated

---

**Tabulated input file format:**

```
Line 1:             comment or blank (ignored)
Line 2:             comment or blank (ignored)
Line 3:             N_LD N_rho (# of LD potentials and # of tabulated values, single␣
 ↪space separated)
Line 4:             blank (ignored)
Line 5:             R1(k) R2(k) (lower and upper cutoffs, single space separated)
Line 6:             central-types (central atom types, single space separated)
Line 7:             neighbor-types (neighbor atom types single space separated)
Line 8:             rho_min rho_max drho (min, max and diff. in tabulated rho values,␣
 ↪single space separated)
Line 9:             F(k)(rho_min + 0.drho)
Line 10:            F(k)(rho_min + 1.drho)
Line 11:            F(k)(rho_min + 2.drho)
...
Line 9+N_rho:       F(k)(rho_min + N_rho . drho)
Line 10+N_rho:      blank (ignored)

Block 2

Block 3

Block N_LD
```

Lines 5 to 9+N_rho constitute the first block. Thus the input file is separated (by blank lines) into N_LD blocks each representing a separate LD potential and each specifying its own upper and lower cutoffs, central and neighbor atoms, and potential. In general, blank lines anywhere are ignored.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.185.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support automatic mixing. For atom type pairs $\alpha$, $\beta$ and $\alpha$ != $\beta$, even if LD potentials of type ($\alpha$, $\alpha$) and ($\beta$, $\beta$) are provided, you will need to explicitly provide LD potential types ($\alpha$, $\beta$) and ($\beta$, $\alpha$) if need be (Here, the notation ($\alpha$, $\beta$) means that $\alpha$ is the central atom to which the LD potential is applied and $\beta$ is the neighbor atom which contributes to the LD potential on $\alpha$).

This pair style does not support the *pair_modify* shift, table, and tail options.

The local/density pair style does not write its information to *binary restart files*, since it is stored in tabulated potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

---

### 4.185.5 Restrictions

The local/density pair style is a part of the MANYBODY package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.185.6 Related commands

*pair_coeff*

### 4.185.7 Default

none

---

**(Sanyal1)** Sanyal and Shell, Journal of Chemical Physics, 2016, 145 (3), 034109.

**(Sanyal2)** Sanyal and Shell, Journal of Physical Chemistry B, 122 (21), 5678-5693.

**(Rosenberger)** Rosenberger, Sanyal, Shell and van der Vegt, Journal of Chemical Physics, 2019, 151 (4), 044111.

## 4.186 pair_style lubricate command

Accelerator Variants: *lubricate/omp*

## 4.187 pair_style lubricate/poly command

Accelerator Variants: *lubricate/poly/omp*

### 4.187.1 Syntax

```
pair_style style mu flaglog flagfld cutinner cutoff flagHI flagVF
```

- style = *lubricate* or *lubricate/poly*
- mu = dynamic viscosity (dynamic viscosity units)

- flaglog = 0/1 to exclude/include log terms in the lubrication approximation
- flagfld = 0/1 to exclude/include Fast Lubrication Dynamics (FLD) effects
- cutinner = inner cutoff distance (distance units)
- cutoff = outer cutoff for interactions (distance units)
- flagHI (optional) = 0/1 to exclude/include 1/r hydrodynamic interactions
- flagVF (optional) = 0/1 to exclude/include volume fraction corrections in the long-range isotropic terms

### 4.187.2 Examples

(all assume radius = 1)

```
pair_style lubricate 1.5 1 1 2.01 2.5
pair_coeff 1 1 2.05 2.8
pair_coeff * *

pair_style lubricate 1.5 1 1 2.01 2.5
pair_coeff * *
variable mu equal ramp(1,2)
fix 1 all adapt 1 pair lubricate mu * * v_mu
```

### 4.187.3 Description

Styles *lubricate* and *lubricate/poly* compute hydrodynamic interactions between mono-disperse finite-size spherical particles in a pairwise fashion. The interactions have 2 components. The first is Ball-Melrose lubrication terms via the formulas in *(Ball and Melrose)*

$$W = -a_{sq}|(v_1 - v_2) \bullet \mathbf{nn}|^2 - a_{sh}|(\omega_1 + \omega_2) \bullet (\mathbf{I} - \mathbf{nn}) - 2\Omega_N|^2 - a_{pu}|(\omega_1 - \omega_2) \bullet (\mathbf{I} - \mathbf{nn})|^2 - a_{tw}|(\omega_1 - \omega_2) \bullet \mathbf{nn}|^2 \qquad r < r_c$$

$$\Omega_N = \mathbf{n} \times (v_1 - v_2)/r$$

which represents the dissipation W between two nearby particles due to their relative velocities in the presence of a background solvent with viscosity *mu*. Note that this is dynamic viscosity which has units of mass/distance/time, not kinematic viscosity.

The Asq (squeeze) term is the strongest and is included if *flagHI* is set to 1 (default). It scales as 1/gap where gap is the separation between the surfaces of the 2 particles. The Ash (shear) and Apu (pump) terms are only included if *flaglog* is set to 1. They are the next strongest interactions, and the only other singular interaction, and scale as log(gap). Note that *flaglog* = 1 and *flagHI* = 0 is invalid, and will result in a warning message, after which *flagHI* will be set to 1. The Atw (twist) term is currently not included. It is typically a very small contribution to the lubrication forces.

The *flagHI* and *flagVF* settings are optional. Neither should be used, or both must be defined.

*Cutinner* sets the minimum center-to-center separation that will be used in calculations irrespective of the actual separation. *Cutoff* is the maximum center-to-center separation at which an interaction is computed. Using a *cutoff* less than 3 radii is recommended if *flaglog* is set to 1.

The other component is due to the Fast Lubrication Dynamics (FLD) approximation, described in *(Kumar)*, which can be represented by the following equation

$$F^H = -R_{FU}(U - U^\infty) + R_{FE}E^\infty$$

where U represents the velocities and angular velocities of the particles, $U^\infty$ represents the velocity and the angular velocity of the undisturbed fluid, and $E^\infty$ represents the rate of strain tensor of the undisturbed fluid with viscosity

*mu*. Again, note that this is dynamic viscosity which has units of mass/distance/time, not kinematic viscosity. Volume fraction corrections to R_FU are included as long as *flagVF* is set to 1 (default).

**Note**

When using the FLD terms, these pair styles are designed to be used with explicit time integration and a correspondingly small timestep. Thus either *fix nve/sphere* or *fix nve/asphere* should be used for time integration. To perform implicit FLD, see the *pair_style lubricateU* command.

Style *lubricate* requires monodisperse spherical particles; style *lubricate/poly* allows for polydisperse spherical particles.

Changed in version 4Jul2026.

The near-field resistance functions used by *lubricate/poly* for particles of unequal radii were corrected so that the pairwise lubrication force obeys Newton's third law (is equal and opposite). Previously the log-order terms were evaluated using a per-particle reference length, which made the force on particle *i* differ from minus the force on particle *j* for strongly polydisperse pairs. Results for monodisperse systems are unchanged.

The viscosity *mu* can be varied in a time-dependent manner over the course of a simulation, in which case in which case the pair_style setting for *mu* will be overridden. See the *fix adapt* command for details.

If the suspension is sheared via the *fix deform* command then the pair style uses the shear rate to adjust the hydrodynamic interactions accordingly. Volume changes due to fix deform are accounted for when computing the volume fraction corrections to R_FU.

When computing the volume fraction corrections to R_FU, the presence of walls (whether moving or stationary) will affect the volume fraction available to colloidal particles. This is currently accounted for with the following types of walls: *wall/lj93*, *wall/lj126*, *wall/colloid*, and *wall/harmonic*. For these wall styles, the correct volume fraction will be used when walls do not coincide with the box boundary, as well as when walls move and thereby cause a change in the volume fraction. Other wall styles may still work, but they will result in the volume fraction being computed based on the box boundaries. Several wall styles are not compatible with these pair styles and using them will result in an error.

Since lubrication forces are dissipative, it is usually desirable to thermostat the system at a constant temperature. If Brownian motion (at a constant temperature) is desired, it can be set using the *pair_style brownian* command. These pair styles and the brownian style should use consistent parameters for *mu*, *flaglog*, *flagfld*, *cutinner*, *cutoff*, *flagHI* and *flagVF*.

---

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- cutinner (distance units)
- cutoff (distance units)

The two coefficients are optional. If neither is specified, the two cutoffs specified in the pair_style command are used. Otherwise both must be specified.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.187.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the two cutoff distances for this pair style can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

This pair style does not support the *pair_modify* shift option for the energy of the pair interaction.

The *pair_modify* table option is not relevant for this pair style.

This pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.187.5 Restrictions

These styles are part of the COLLOID package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Only spherical monodisperse particles are allowed for pair_style lubricate.

Only spherical particles are allowed for pair_style lubricate/poly.

These pair styles will not restart exactly when using the *read_restart* command, though they should provide statistically similar results. This is because the forces they compute depend on atom velocities. See the *read_restart* command for more details.

### 4.187.6 Related commands

*pair_coeff*, *pair_style lubricateU*

### 4.187.7 Default

The default settings for the optional args are flagHI = 1 and flagVF = 1.

---

**(Ball)** Ball and Melrose, Physica A, 247, 444-472 (1997).

**(Kumar)** Kumar and Higdon, Phys Rev E, 82, 051401 (2010). See also his thesis for more details: A. Kumar, "Microscale Dynamics in Suspensions of Non-spherical Particles", Thesis, University of Illinois Urbana-Champaign, (2010). (https://www.ideals.illinois.edu/handle/2142/16032)

# 4.188 pair_style lubricateU command

# 4.189 pair_style lubricateU/poly command

## 4.189.1 Syntax

```
pair_style style mu flaglog cutinner cutoff gdot flagHI flagVF
```

- style = *lubricateU* or *lubricateU/poly*
- mu = dynamic viscosity (dynamic viscosity units)
- flaglog = 0/1 to exclude/include log terms in the lubrication approximation
- cutinner = inner cut off distance (distance units)
- cutoff = outer cutoff for interactions (distance units)
- gdot = shear rate (1/time units)
- flagHI (optional) = 0/1 to exclude/include 1/r hydrodynamic interactions
- flagVF (optional) = 0/1 to exclude/include volume fraction corrections in the long-range isotropic terms

## 4.189.2 Examples

(all assume radius = 1)

```
pair_style lubricateU 1.5 1 2.01 2.5 0.01 1 1
pair_coeff 1 1 2.05 2.8
pair_coeff * *
```

## 4.189.3 Description

Styles *lubricateU* and *lubricateU/poly* compute velocities and angular velocities for finite-size spherical particles such that the hydrodynamic interaction balances the force and torque due to all other types of interactions.

The interactions have 2 components. The first is Ball-Melrose lubrication terms via the formulas in *(Ball and Melrose)*

$$W = -a_{sq}|(v_1 - v_2) \bullet \mathbf{nn}|^2 - a_{sh}|(\omega_1 + \omega_2) \bullet (\mathbf{I} - \mathbf{nn}) - 2\Omega_N|^2 - a_{pu}|(\omega_1 - \omega_2) \bullet (\mathbf{I} - \mathbf{nn})|^2 - a_{tw}|(\omega_1 - \omega_2) \bullet \mathbf{nn}|^2 \qquad r < r_c$$

$$\Omega_N = \mathbf{n} \times (v_1 - v_2)/r$$

which represents the dissipation W between two nearby particles due to their relative velocities in the presence of a background solvent with viscosity *mu*. Note that this is dynamic viscosity which has units of mass/distance/time, not kinematic viscosity.

The Asq (squeeze) term is the strongest and is included as long as *flagHI* is set to 1 (default). It scales as 1/gap where gap is the separation between the surfaces of the 2 particles. The Ash (shear) and Apu (pump) terms are only included if *flaglog* is set to 1. They are the next strongest interactions, and the only other singular interaction, and scale as log(gap). Note that *flaglog* = 1 and *flagHI* = 0 is invalid, and will result in a warning message, after which *flagHI* will be set to 1. The Atw (twist) term is currently not included. It is typically a very small contribution to the lubrication forces.

The *flagHI* and *flagVF* settings are optional. Neither should be used, or both must be defined.

*Cutinner* sets the minimum center-to-center separation that will be used in calculations irrespective of the actual separation. *Cutoff* is the maximum center-to-center separation at which an interaction is computed. Using a *cutoff* less than 3 radii is recommended if *flaglog* is set to 1.

The other component is due to the Fast Lubrication Dynamics (FLD) approximation, described in *(Kumar)*. The equation being solved to balance the forces and torques is

$$-R_{FU}(U - U^{\infty}) = -R_{FE}E^{\infty} - F^{rest}$$

where U represents the velocities and angular velocities of the particles, $U^{\infty}$ represents the velocities and the angular velocities of the undisturbed fluid, and $E^{\infty}$ represents the rate of strain tensor of the undisturbed fluid flow with viscosity *mu*. Again, note that this is dynamic viscosity which has units of mass/distance/time, not kinematic viscosity. Volume fraction corrections to R_FU are included if *flagVF* is set to 1 (default).

F*rest* represents the forces and torques due to all other types of interactions, e.g. Brownian, electrostatic etc. Note that this algorithm neglects the inertial terms, thereby removing the restriction of resolving the small interial time scale, which may not be of interest for colloidal particles. These pair styles solve for the velocity such that the hydrodynamic force balances all other types of forces, thereby resulting in a net zero force (zero inertia limit). When defining these pair styles, they must be defined last so that when these styles are invoked all other types of forces have already been computed. For the same reason, they won't work if additional non-pair styles are defined (such as bond or Kspace forces) as they are calculated in LAMMPS after the pairwise interactions have been computed.

**Note**

When using these styles, the pair styles are designed to be used with implicit time integration and a correspondingly larger timestep. Thus either *fix nve/noforce* should be used for spherical particles defined via *atom_style sphere* or *fix nve/asphere/noforce* should be used for spherical particles defined via *atom_style ellipsoid*. This is because the velocity and angular momentum of each particle is set by the pair style, and should not be reset by the time integration fix.

Style *lubricateU* requires monodisperse spherical particles; style *lubricateU/poly* allows for polydisperse spherical particles.

Changed in version 4Jul2026.

The near-field resistance functions used by *lubricateU/poly* for particles of unequal radii were corrected for consistency with *pair_style lubricate/poly* so that the squeeze and shear resistances are symmetric under exchange of the two particles. Results for monodisperse systems are unchanged.

If the suspension is sheared via the *fix deform* command then the pair style uses the shear rate to adjust the hydrodynamic interactions accordingly. Volume changes due to fix deform are accounted for when computing the volume fraction corrections to R_FU.

When computing the volume fraction corrections to R_FU, the presence of walls (whether moving or stationary) will affect the volume fraction available to colloidal particles. This is currently accounted for with the following types of walls: *wall/lj93*, *wall/lj126*, *wall/colloid*, and *wall/harmonic*. For these wall styles, the correct volume fraction will be used when walls do not coincide with the box boundary, as well as when walls move and thereby cause a change in the volume fraction. To use these wall styles with pair_style *lubricateU* or *lubricateU/poly*, the *fld yes* option must be specified in the fix wall command. Other wall styles may still work, but they will result in the volume fraction being computed based on the box boundaries. Several wall styles are not compatible with these pair styles and using them will result in an error.

Since lubrication forces are dissipative, it is usually desirable to thermostat the system at a constant temperature. If Brownian motion (at a constant temperature) is desired, it can be set using the *pair_style brownian* command. These pair styles and the brownian style should use consistent parameters for *mu*, *flaglog*, *flagfld*, *cutinner*, *cutoff*, *flagHI* and *flagVF*.

---

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- cutinner (distance units)
- cutoff (distance units)

The two coefficients are optional. If neither is specified, the two cutoffs specified in the pair_style command are used. Otherwise both must be specified.

---

### 4.189.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the two cutoff distances for this pair style can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

These pair styles do not support the *pair_modify* shift option for the energy of the pair interaction.

The *pair_modify* table option is not relevant for these pair styles.

These pair styles do not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

These pair styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

These pair styles can only be used via the *pair* keyword of the *run_style respa* command. They does not support the *inner*, *middle*, *outer* keywords.

---

### 4.189.5 Restrictions

These styles are part of the COLLOID package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Currently, these pair styles assume that all other types of forces/torques on the particles have been already been computed when it is invoked. This requires this style to be defined as the last of the pair styles, and that no fixes apply additional constraint forces. One exception is the *fix wall/colloid* commands, which has an "fld" option to apply their wall forces correctly.

Only spherical monodisperse particles are allowed for pair_style lubricateU.

Only spherical particles are allowed for pair_style lubricateU/poly.

For sheared suspensions, it is assumed that the shearing is done in the xy plane, with x being the velocity direction and y being the velocity-gradient direction. In this case, one must use *fix deform* with the same rate of shear (erate).

These pair styles are only compatible with the following wall fixes: doc:*fix wall/lj93, fix wall/lj126, fix wall/lj1043, fix wall/colloid, fix wall/harmonic, fix wall/lepton, fix wall/morse, fix wall/table <fix_wall>*.

### 4.189.6 Related commands

*pair_coeff* , *pair_style lubricate*

### 4.189.7 Default

The default settings for the optional args are flagHI = 1 and flagVF = 1.

---

**(Ball)** Ball and Melrose, Physica A, 247, 444-472 (1997).

**(Kumar)** Kumar and Higdon, Phys Rev E, 82, 051401 (2010).

# 4.190 pair_style mbx command

## 4.190.1 Syntax

```
pair_style mbx cutoff
```

- cutoff = real-space cutoff for MBX in Angstroms

## 4.190.2 Examples

```
pair_style      mbx 9.0
pair_coeff      * * 1 h2o 1 2 2 json mbx.json
compute         mbx all pair mbx

# For a system involving ch4 (atom types C=1, H=2) and
# water (atom types O=3, H=4)
pair_style      mbx 9.0
pair_coeff      * * 2 ch4 1 2 2 2 2 h2o 3 4 4 json mbx.json
compute         mbx all pair mbx

# For a system involving water (atom types O=12, H=13) in a hybrid simulation
pair_style      hybrid/overlay mbx 9.0 lj/cut 9.0 coul/exclude 9.0
pair_coeff      * * mbx 2 dp1 1*11 h2o 12 13 13 json mbx.json
pair_coeff      1*11 1*11 coul/exclude
compute         mbx all pair mbx

# For a system involving water (atom types O=12, H=13) in a hybrid simulation
# with special_bonds and coul/exclude to exclude 1-2, 1-3, and 1-4 electrostatics
# for the charmm framework
special_bonds   charmm
pair_style      hybrid/overlay mbx 9.0 lj/cut 9.0 coul/exclude 9.0
pair_coeff      * * mbx 2 dp1 1*11 h2o 12 13 13 json mbx.json
pair_coeff      1*11 1*11 coul/exclude
compute         mbx all pair mbx
```

See `examples/PACKAGES/mbx` for full examples of how to use MBX in LAMMPS.

## 4.190.3 Description

Added in version 11Feb2026.

The MBX (Many-Body eXpansion) software is a C++ library that provides access to many-body energy (MB-nrg) potential energy functions, such as the MB-pol water model. Developed over the past decade, these potential energy functions integrate physics-based and machine-learned many-body terms trained on electronic structure data calculated at the "gold standard" coupled-cluster level of theory. *(Gupta)*

This pair_style instructs LAMMPS to call the MBX library in order to simulate MB-nrg models such as MB-pol. The MBX library source code is available at https://github.com/paesanilab/MBX. MBX is heavily OpenMP parallelized (OMP), and the OMP_NUM_THREADS environment variable should be properly set to the number of threads desired. A detailed discussion of the code structure can be found in the manuscript *(Riera)*, while a detailed description of the performance scaling can be found in the manuscript *(Gupta)*.

The *cutoff* argument specifies the real-space cutoff for MBX in Angstroms. This real-space cutoff is used for the dispersion interactions of the MB-nrg monomers, as well as for the electrostatics of the **entire** system. For periodic systems, a safe value for the real-space cutoff is **9.0 Angstroms**, and all classical interactions beyond this cutoff will

then be handled via particle-mesh Ewald (PME) within MBX. For non-periodic systems, the cutoff can be set to a large value, such as 100.0 Angstroms, to ensure that all interactions are captured in the real-space.

**Warning**

MBX must currently be used with *processors* mapping style *xyz*. If you do not, MBX will throw the error:

```
[MBX] Inconsistent proc mapping: 'processors * * * map xyz' required for PME solver
```

For hybrid simulations involving MB-nrg and non-MB-nrg molecules in the same simulation, one can use *pair_style hybrid/overlay* to combine the MB-nrg molecules with other pair styles, such as *lj/cut*. This has been used to simulate MB-pol water within host frameworks such as metal-organic frameworks (MOFs) and carbon nanotubes (CNTs). If using MBX in a hybrid simulation involving *special_bonds*, (such as when using the CHARMM, Amber, OPLS, or ClayFF force fields etc.), please see the warning below for more details about using *special_bonds* with MBX *dp1*. See `examples/PACKAGES/mbx` for a complete hybrid example.

If you have questions not answered by this documentation, please reference the MBX website mbxsimulations.com or reach out to the MBX team at https://groups.google.com/g/mbx-users

### 4.190.4 Pair coeff syntax

MBX is many-body method, and only a single pair_coeff command is needed to specify the mapping of LAMMPS atom IDs to MBX monomers. The syntax is as follows:

```
pair_coeff * * num_mon_types mon_name atom_mapping <mon_name2> <atom_mapping2> ... json␣
↪mbx.json print/settings
```

- num_mon_types = number of monomer types in the system
- mon_name = name of the monomer type (e.g. h2o, ch4, etc)
- atom mapping = list of LAMMPS atom IDs that correspond to the atoms in the monomer
- *json* arg = specifies the name of the MBX json configuration file, such as mbx.json
- print/settings = optionally print MBX settings to logfile

The *num_mon_types* argument specifies the number of different MB-nrg monomer types in the system.

For each monomer type, the *mon_name* argument specifies the name of the monomer, such as *h2o* for water or *ch4* for methane. The *atom mapping* argument specifies then the mapping of LAMMPS atom IDs to the atoms in the monomer, such as *1 2 2* for water (O=1, H=2). For hybrid simulations, the *dp1* (drude particle) monomer should be used to represent the non-MB-nrg molecules. *dp1* is a special monomer in MBX in that its *atom_mapping* can be a range of LAMMPS atom IDs, such as *1*11* to represent atom types 1 through 11. For a complete list of available monomers in MBX, please see the MBX documentation.

**Warning**

When using MBX, **all electrostatics are handled internally by MBX.** This is important since MB-nrg models such as MB-pol are polarizable models that may also use geometrically dependent charges, such as the Partridge and Schwenke charges used in MB-pol water.

Therefore, one should **never** use a coulombic pair style in LAMMPS such as *coul/cut* or *coul/long* when also using MBX. This would result in double counting of electrostatic interactions.

When performing a hybrid simulation using dp1, note that many frameworks (Amber, CHARMM, OPLS, ClayFF etc.) require the usage of special_bonds to exclude some bonded coulomb interactions (1-2, 1-3, and/or 1-4). Since MBX is handling all electrostatics, this should therefore be accounted for using the *coul/exclude* command.

```
# For a system involving water (atom types O=12, H=13) in a hybrid simulation
# with special_bonds and coul/exclude to exclude 1-2, 1-3, and 1-4 electrostatics
# for the charmm framework
processors      * * * map xyz
special_bonds   charmm
pair_style      hybrid/overlay mbx 9.0 lj/cut 9.0 coul/exclude 9.0
pair_coeff      * * mbx 2 dp1 1*11 h2o 12 13 13 json mbx.json
pair_coeff      1*11 1*11 coul/exclude
compute         mbx all pair mbx
```

The *json* argument specifies the name of the MBX JSON configuration file to use, such as *mbx.json*. If this file is not provided, the pair style will attempt to use a default configuration. See the mbx.json documentation for more details on how to create this file.

The *print/settings* argument optionally will print the MBX settings to the LAMMPS logfile at the start of the simulation. This is optionally used for debugging and ensuring that the settings are being correctly applied.

### 4.190.5 Restrictions

This pair_style is part of the MBX package. A pair style is only enabled if LAMMPS was built with its corresponding package. See the *Build package* page for more info.

MBX requires the FFTW3 library to be installed. This is needed as part of the internal PME solver used for long-range electrostatics.

All electrostatic interactions are calculated internally in MBX. Therefore one should never calculate coulombic interactions in LAMMPS such as using *coul/cut* or *coul/long* when also using MBX. See the warning above for more details.

MBX currently only supports *processors* mapping style *xyz*.

MBX is primarily tested to work with *units real* and *atom_style full*. If you encounter issues with other unit or atom styles, please contact the MBX developers.

### 4.190.6 Related commands

*pair hybrid/overlay*, *pair coul/exclude*

---

**(Riera)** M. Riera, C. Knight, E. Bull-Vulpe, X. Zhu, H. Agnew, D. Smith, A. Simmonett, F. Paesani, J. Chem. Phys. 159, 054802 (2023)

**(Gupta)** S. Gupta, E. Bull-Vulpe, H. Agnew, S. Iyer, X. Zhu, R. Zhou, C. Knight, F. Paesani, J. Chem. Theory Comput. 21, 1938 (2025)

## 4.191 pair_style lj/mdf command

Accelerator Variants: *lj/mdf/kk*, *lj/mdf/omp*

## 4.192 pair_style buck/mdf command

Accelerator Variants: *buck/mdf/kk*, *buck/mdf/omp*

## 4.193 pair_style lennard/mdf command

Accelerator Variants: *lennard/mdf/omp*, *lennard/mdf/kk*

### 4.193.1 Syntax

```
pair_style style args
```

- style = *lj/mdf* or *buck/mdf* or *lennard/mdf*
- args = list of arguments for a particular style

```
lj/mdf args = cutoff1 cutoff2
  cutoff1 = inner cutoff for the start of the tapering function
  cutoff1 = out cutoff for the end of the tapering function
buck/mdf args = cutoff1 cutoff2
  cutoff1 = inner cutoff for the start of the tapering function
  cutoff1 = out cutoff for the end of the tapering function
lennard/mdf args = cutoff1 cutoff2
  cutoff1 = inner cutoff for the start of the tapering function
  cutoff1 = out cutoff for the end of the tapering function
```

### 4.193.2 Examples

```
pair_style lj/mdf 2.5 3.0
pair_coeff * * 1.0 1.0
pair_coeff 1 1 1.1 2.8 3.0 3.2

pair_style buck/mdf 2.5 3.0
pair_coeff * * 100.0 1.5 200.0
pair_coeff * * 100.0 1.5 200.0 3.0 3.5

pair_style lennard/mdf 2.5 3.0
pair_coeff * * 1.0 1.0
pair_coeff 1 1 1021760.3664 2120.317338 3.0 3.2
```

### 4.193.3 Description

The *lj/mdf*, *buck/mdf* and *lennard/mdf* compute the standard 12-6 Lennard-Jones and Buckingham potential with the addition of a taper function that ramps the energy and force smoothly to zero between an inner and outer cutoff.

$$E_{smooth}(r) = E(r) * f(r)$$

The tapering, *f(r)*, is done by using the Mei, Davenport, Fernando function *(Mei)*.

$$\begin{aligned} f(r) &= 1.0 && \text{for} \quad r < r_m \\ f(r) &= (1-x)^3 * (1+3x+6x^2) && \text{for} \quad r_m < r < r_{cut} \\ f(r) &= 0.0 && \text{for} \quad r >= r_{cut} \end{aligned}$$

where

$$x = \frac{(r - r_m)}{(r_{cut} - r_m)}$$

Here $r_m$ is the inner cutoff radius and $r_{cut}$ is the outer cutoff radius.

---

For the *lj/mdf* pair_style, the potential energy, *E(r)*, is the standard 12-6 Lennard-Jones written in the epsilon/sigma form:

$$E(r) = 4\varepsilon \left[ \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]$$

Either the first two or all of the following coefficients must be defined for each pair of atoms types via the pair_coeff command as in the examples above, or in the data file read by the *read_data*. The two cutoffs default to the global values and $\varepsilon$ and $\sigma$ can also be determined by mixing as described below:

- $\varepsilon$ (energy units)
- $\sigma$ (distance units)
- $r_m$ (distance units)
- $r_{cut}$ (distance units)

---

For the *buck/mdf* pair_style, the potential energy, *E(r)*, is the standard Buckingham potential with three required coefficients. The two cutoffs can be omitted and default to the corresponding global values:

$$E(r) = Ae^{(-r/\rho)} - \frac{C}{r^6}$$

- $A$ (energy units)
- $\rho$ (distance units)
- $C$ (energy-distance^6 units)
- $r_m$ (distance units)
- $r_{cut}$ (distance units)

---

For the *lennard/mdf* pair_style, the potential energy, *E(r)*, is the standard 12-6 Lennard-Jones written in the A/B form:

$$E(r) = \frac{A}{r^{12}} - \frac{B}{r^6}$$

The following coefficients must be defined for each pair of atoms types via the pair_coeff command as in the examples above, or in the data file read by the read_data commands, or by mixing as described below. The two cutoffs default to their global values and must be either both given or both left out:

- $A$ (energy-distance^12 units)
- $B$ (energy-distance^6 units)
- $r_m$ (distance units)
- $r_{cut}$ (distance units)

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 4.193.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the $\varepsilon$ and $\sigma$ coefficients and cutoff distances for the lj/mdf pair style can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details. The other two pair styles buck/mdf and lennard/mdf do not support mixing, so all I,J pairs of coefficients must be specified explicitly.

None of the lj/mdf, buck/mdf, or lennard/mdf pair styles supports the *pair_modify* shift option or long-range tail corrections to pressure and energy.

These styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

These styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

### 4.193.5 Restrictions

These pair styles can only be used if LAMMPS was built with the EXTRA-PAIR package. See the *Build package* doc page for more info.

### 4.193.6 Related commands

*pair_coeff*

### 4.193.7 Default

none

**(Mei)** Mei, Davenport, Fernando, Phys Rev B, 43 4653 (1991)

## 4.194 pair_style meam command

Accelerator Variants: *meam/kk*

## 4.195 pair_style meam/ms command

Accelerator Variants: *meam/ms/kk*

### 4.195.1 Syntax

```
pair_style style
```

- style = *meam* or *meam/ms*

### 4.195.2 Examples

```
pair_style meam
pair_coeff * * ../potentials/library.meam Si ../potentials/si.meam Si
pair_coeff * * ../potentials/library.meam Ni Al NULL Ni Al Ni Ni

pair_style meam/ms
pair_coeff * * ../potentials/library.msmeam H Ga ../potentials/HGa.meam H Ga
```

### 4.195.3 Description

**Note**

The behavior of the MEAM potential for alloy systems has changed as of November 2010; see description below of the mixture_ref_t parameter

Pair style *meam* computes non-bonded interactions for a variety of materials using the modified embedded-atom method (MEAM) *(Baskes)*. Conceptually, it is an extension to the original *EAM method* which adds angular forces. It is thus suitable for modeling metals and alloys with fcc, bcc, hcp and diamond cubic structures, as well as materials with covalent interactions like silicon and carbon.

The *meam* pair style is a translation of the original Fortran version to C++. It is functionally equivalent but more efficient and has additional features. The Fortran version of the *meam* pair style has been removed from LAMMPS after the 12 December 2018 release.

Pair style *meam/ms* uses the multi-state MEAM (MS-MEAM) method according to *(Baskes2)*, which is an extension to MEAM. This pair style is mostly equivalent to *meam* and differs only where noted in the documentation below.

In the MEAM formulation, the total energy E of a system of atoms is given by:

$$E = \sum_i \left\{ F_i(\bar{\rho}_i) + \frac{1}{2} \sum_{i \neq j} \phi_{ij}(r_{ij}) \right\}$$

where $F$ is the embedding energy which is a function of the atomic electron density $\rho$, and $\phi$ is a pair potential interaction. The pair interaction is summed over all neighbors J of atom I within the cutoff distance. As with EAM, the multi-body nature of the MEAM potential is a result of the embedding energy term. Details of the computation of the embedding and pair energies, as implemented in LAMMPS, are given in *(Gullet)* and references therein.

The various parameters in the MEAM formulas are listed in two files which are specified by the *pair_coeff* command. These are ASCII text files in a format consistent with other MD codes that implement MEAM potentials, such as the serial DYNAMO code and Warp. Several MEAM potential files with parameters for different materials are included in the "potentials" directory of the LAMMPS distribution with a ".meam" suffix. All of these are parameterized in terms of LAMMPS *metal units*.

Note that unlike for other potentials, cutoffs for MEAM potentials are not set in the pair_style or pair_coeff command; they are specified in the MEAM potential files themselves.

Only a single pair_coeff command is used with the *meam* style which specifies two MEAM files and the element(s) to extract information for. The MEAM elements are mapped to LAMMPS atom types by specifying N additional arguments after the second filename in the pair_coeff command, where N is the number of LAMMPS atom types:

- MEAM library file
- Element1, Element2, ...
- MEAM parameter file
- N element names = mapping of MEAM elements to atom types

See the *pair_coeff* page for alternate ways to specify the path for the potential files.

As an example, the `potentials/library.meam` file has generic MEAM settings for a variety of elements. The `potentials/SiC.meam` file has specific parameter settings for a Si and C alloy system. If your LAMMPS simulation has 4 atoms types and you want the first 3 to be Si, and the fourth to be C, you would use the following pair_coeff command:

```
pair_coeff * * library.meam Si C sic.meam Si Si Si C
```

The first 2 arguments must be * * so as to span all LAMMPS atom types. The first filename is the element library file. The list of elements following it extracts lines from the library file and assigns numeric indices to these elements. The second filename is the alloy parameter file, which refers to elements using the numeric indices assigned before. The arguments after the parameter file map LAMMPS atom types to elements, i.e. LAMMPS atom types 1,2,3 to the MEAM Si element. The final C argument maps LAMMPS atom type 4 to the MEAM C element.

If the second filename is specified as NULL, no parameter file is read, which simply means the generic parameters in the library file are used. Use of the NULL specification for the parameter file is discouraged for systems with more than a single element type (e.g. alloys), since the parameter file is expected to set element interaction terms that are not captured by the information in the library file.

If a mapping value is specified as NULL, the mapping is not performed. This can be used when a *meam* potential is used as part of the *hybrid* pair style. The NULL values are placeholders for atom types that will be used with other potentials.

**Note**

If the second filename is NULL, the element names between the two filenames can appear in any order, e.g. "Si C" or "C Si" in the example above. However, if the second filename is **not** NULL (as in the example above), it contains settings that are indexed **by numbers** for the elements that precede it. Thus you need to ensure that you list the elements between the filenames in an order consistent with how the values in the second filename are indexed. See details below on the syntax for settings in the second file.

The MEAM library file provided with LAMMPS has the name `potentials/library.meam`. It is the "meamf" file used by other MD codes. Aside from blank and comment lines (starting with # which can appear anywhere), it is formatted as a series of entries, each of which has 19 parameters and can span multiple lines:

elt, lat, z, ielement, atwt, alpha, b0, b1, b2, b3, alat, esub, asub, t0, t1, t2, t3, rozero, ibar

The *elt* and *lat* parameters are text strings, such as *elt* = Si or Cu and *lat* = dia or fcc. Because the library file is used by Fortran MD codes, these strings may be enclosed in single quotes, but this is not required. The other numeric parameters match values in the formulas above. The value of the *elt* string is what is used in the pair_coeff command to identify which settings from the library file you wish to read in. There can be multiple entries in the library file with the same *elt* value; LAMMPS reads the first matching entry it finds and ignores the rest.

Other parameters in the MEAM library file correspond to single-element potential parameters:

```
lat      = lattice structure of reference configuration
z        = number of nearest neighbors in the reference structure
ielement = atomic number
atwt     = atomic weight
alat     = lattice constant of reference structure
```

```
esub     = energy per atom (eV) in the reference structure at equilibrium
asub     = "A" parameter for MEAM (see e.g. (Baskes))
```

The *alpha*, *b0*, *b1*, *b2*, *b3*, *t0*, *t1*, *t2*, *t3* parameters correspond to the standard MEAM parameters in the literature *(Baskes)* (the b parameters are the standard beta parameters). Note that only parameters normalized to $t0 = 1.0$ are supported. The *rozero* parameter is an element-dependent density scaling that weights the reference background density (see e.g. equation 4.5 in *(Gullet)*) and is typically 1.0 for single-element systems. The *ibar* parameter selects the form of the function G(Gamma) used to compute the electron density; options are

```
 0 => G = sqrt(1+Gamma)
 1 => G = exp(Gamma/2)
 2 => not implemented
 3 => G = 2/(1+exp(-Gamma))
 4 => G = sqrt(1+Gamma)
-5 => G = +-sqrt(abs(1+Gamma))
```

If used, the MEAM parameter file contains settings that override or complement the library file settings. Examples of such parameter files are in the potentials directory with a ".meam" suffix. Their format is the same as is read by other Fortran MD codes. Aside from blank and comment lines (start with # which can appear anywhere), each line has one of the following forms. Each line can also have a trailing comment (starting with #) which is ignored.

```
keyword = value
keyword(I) = value
keyword(I,J) = value
keyword(I,J,K) = value
```

The indices I, J, K correspond to the elements selected from the MEAM library file numbered in the order of how those elements were selected starting from 1. Thus for the example given before

```
pair_coeff * * library.meam Si C sic.meam Si Si Si C
```

an index of 1 would refer to Si and an index of 2 to C.

The recognized keywords for the parameter file are as follows:

```
rc          = cutoff radius for cutoff function; default = 4.0
delr        = length of smoothing distance for cutoff function; default = 0.1
rho0(I)     = relative density for element I (overwrites value
              read from meamf file)
Ec(I,J)     = cohesive energy of reference structure for I-J mixture
delta(I,J)  = heat of formation for I-J alloy; if Ec_IJ is input as
              zero, then LAMMPS sets Ec_IJ = (Ec_II + Ec_JJ)/2 - delta_IJ
alpha(I,J)  = alpha parameter for pair potential between I and J (can
              be computed from bulk modulus of reference structure)
re(I,J)     = equilibrium distance between I and J in the reference
              structure
Cmax(I,J,K) = Cmax screening parameter when I-J pair is screened
              by K (I<=J); default = 2.8
Cmin(I,J,K) = Cmin screening parameter when I-J pair is screened
              by K (I<=J); default = 2.0
lattce(I,J) = lattice structure of I-J reference structure:
                fcc = face centered cubic
                bcc = body centered cubic
                hcp = hexagonal close-packed
                dim = dimer
                dia = diamond (interlaced fcc for alloy)
```

```
                dia3= diamond structure with primary 1NN and secondary 3NN interaction
                b1  = rock salt (NaCl structure)
                c11 = MoSi2 structure
                l12 = Cu3Au structure (lower case L, followed by 12)
                b2  = CsCl structure (interpenetrating simple cubic)
                ch4 = methane-like structure, only for binary system
                lin = linear structure (180 degree angle)
                zig = zigzag structure with a uniform angle
                tri = H2O-like structure that has an angle
                sc  = simple cubic
nn2(I,J)    = turn on second-nearest neighbor MEAM formulation for
              I-J pair (see for example (Lee)).
                0 = second-nearest neighbor formulation off
                1 = second-nearest neighbor formulation on
                default = 0
attrac(I,J) = additional cubic attraction term in Rose energy I-J pair potential
                default = 0
repuls(I,J) = additional cubic repulsive term in Rose energy I-J pair potential
                default = 0
zbl(I,J)    = blend the MEAM I-J pair potential with the ZBL potential for small
              atom separations (ZBL)
                default = 1
theta(I,J)  = angle between three atoms in line, zigzag, and trimer reference structures␣
↪in degrees
                default = 180
gsmooth_factor  = factor determining the length of the G-function smoothing
                  region; only significant for ibar=0 or ibar=4.
                      99.0 = short smoothing region, sharp step
                      0.5  = long smoothing region, smooth step
                      default = 99.0
augt1           = integer flag for whether to augment t1 parameter by
                  3/5*t3 to account for old vs. new meam formulations;
                    0 = don't augment t1
                    1 = augment t1
                    default = 1
ialloy          = integer flag to use alternative averaging rule for t parameters,
                  for comparison with the DYNAMO MEAM code
                    0 = standard averaging (matches ialloy=0 in DYNAMO)
                    1 = alternative averaging (matches ialloy=1 in DYNAMO)
                    2 = no averaging of t (use single-element values)
                    default = 0
mixture_ref_t   = integer flag to use mixture average of t to compute the background
                  reference density for alloys, instead of the single-element values
                  (see description and warning elsewhere in this doc page)
                    0 = do not use mixture averaging for t in the reference density
                    1 = use mixture averaging for t in the reference density
                    default = 0
erose_form      = integer value to select the form of the Rose energy function
                  (see description below).
                    default = 0
emb_lin_neg     = integer value to select embedding function for negative densities
                    0 = F(rho)=0
                    1 = F(rho) = -asub*esub*rho (linear in rho, matches DYNAMO)
                    default = 0
```

```
bkgd_dyn        = integer value to select background density formula
                    0 = rho_bkgd = rho_ref_meam(a) (as in the reference structure)
                    1 = rho_bkgd = rho0_meam(a)*Z_meam(a) (matches DYNAMO)
                    default = 0
```

*Rc*, *delr*, *re* are in distance units (Angstroms in the case of metal units). *Ec* and *delta* are in energy units (eV in the case of metal units).

Each keyword represents a quantity which is either a scalar, vector, 2d array, or 3d array and must be specified with the correct corresponding array syntax. The indices I,J,K each run from 1 to N where N is the number of MEAM elements being used.

Thus these lines

```
rho0(2) = 2.25
alpha(1,2) = 4.37
```

set *rho0* for the second element to the value 2.25 and set *alpha* for the alloy interaction between elements 1 and 2 to 4.37.

The *augt1* parameter is related to modifications in the MEAM formulation of the partial electron density function. In recent literature, an extra term is included in the expression for the third-order density in order to make the densities orthogonal (see for example *(Wang)*, equation 3d); this term is included in the MEAM implementation in LAMMPS. However, in earlier published work this term was not included when deriving parameters, including most of those provided in the `library.meam` file included with LAMMPS, and to account for this difference the parameter *t1* must be augmented by 3/5**t3*. If *augt1* = 1, the default, this augmentation is done automatically. When parameter values are fit using the modified density function, as in more recent literature, augt1 should be set to 0.

The *mixture_ref_t* parameter is available to match results with those of previous versions of LAMMPS (before January 2011). Newer versions of LAMMPS, by default, use the single-element values of the *t* parameters to compute the background reference density. This is the proper way to compute these parameters. Earlier versions of LAMMPS used an alloy mixture averaged value of *t* to compute the background reference density. Setting *mixture_ref_t* = 1 gives the old behavior. WARNING: using *mixture_ref_t* = 1 will give results that are demonstrably incorrect for second-neighbor MEAM, and non-standard for first-neighbor MEAM; this option is included only for matching with previous versions of LAMMPS and should be avoided if possible.

The parameters *attrac* and *repuls*, along with the integer selection parameter *erose_form*, can be used to modify the Rose energy function used to compute the pair potential. This function gives the energy of the reference state as a function of interatomic spacing. The form of this function is:

```
astar = alpha * (r/re - 1.d0)
if erose_form = 0: erose = -Ec*(1+astar+a3*(astar**3)/(r/re))*exp(-astar)
if erose_form = 1: erose = -Ec*(1+astar+(-attrac+repuls/r)*(astar**3))*exp(-astar)
if erose_form = 2: erose = -Ec*(1 +astar + a3*(astar**3))*exp(-astar)
a3 = repuls, astar < 0
a3 = attrac, astar >= 0
```

Most published MEAM parameter sets use the default values *attrac* = *repulse* = 0. Setting *repuls* = *attrac* = *delta* corresponds to the form used in several recent published MEAM parameter sets, such as *(Valone)*

Then using *meam/ms* pair style the multi-state MEAM (MS-MEAM) method is activated. This requires 6 extra parameters in the MEAM library file, resulting in 25 parameters ordered that are ordered like this:

elt, lat, z, ielement, atwt, alpha, b0, b1, b2, b3, b1m, b2m, b3m, alat, esub, asub, t0, t1, t2, t3, t1m, t2m, t3m, rozero, ibar

The 6 extra MS-MEAM parameters are *b1m, b2m, b3m, t1m, t2m, t3m*. In the LAMMPS `potentials` folder, compatible files have an ".msmeam" extension.

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

**Note**

The default form of the *erose* expression in LAMMPS was corrected in March 2009. The current version is correct, but may show different behavior compared with earlier versions of LAMMPS with the attrac and/or repuls parameters are non-zero. To obtain the previous default form, use *erose_form* = 1 (this form does not seem to appear in the literature). An alternative form (see e.g. *(Lee2)*) is available using *erose_form* = 2.

---

### 4.195.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, where types I and J correspond to two different element types, mixing is performed by LAMMPS with user-specifiable parameters as described above.

This pair style does not support the *pair_modify* *shift*, *table*, and *tail* options.

This pair style does not write its information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.195.5 Restrictions

The *meam* and *meam/ms* pair styles are provided in the MEAM package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The maximum number of elements that can be read from the MEAM library file is determined at compile time. The default is 8. If you need support for more elements, you have to change the the constant 'MAXELT' at the beginning of the file `src/MEAM/meam.h` and update/recompile LAMMPS. There is no limit on the number of atoms types.

### 4.195.6 Related commands

*pair_coeff*, *pair_style eam*, *pair_style meam/spline*

### 4.195.7 Default

none

---

**(Baskes)** Baskes, Phys Rev B, 46, 2727-2742 (1992).

**(Baskes2)** Baskes, Phys Rev B, 75, 094113 (2007).

**(Gullet)** Gullet, Wagner, Slepoy, SANDIA Report 2003-8782 (2003). DOI:10.2172/918395 This report may be accessed on-line via this link.

**(Lee)** Lee, Baskes, Phys. Rev. B, 62, 8564-8567 (2000).

**(Lee2)** Lee, Baskes, Kim, Cho. Phys. Rev. B, 64, 184102 (2001).

**(Valone)** Valone, Baskes, Martin, Phys. Rev. B, 73, 214209 (2006).

**(Wang)** Wang, Van Hove, Ross, Baskes, J. Chem. Phys., 121, 5410 (2004).

**(ZBL)** J.F. Ziegler, J.P. Biersack, U. Littmark, "Stopping and Ranges of Ions in Matter", Vol 1, 1985, Pergamon Press.

## 4.196 pair_style meam/spline command

Accelerator Variants: *meam/spline/omp*

### 4.196.1 Syntax

```
pair_style meam/spline
```

### 4.196.2 Examples

```
pair_style meam/spline
pair_coeff * * Ti.meam.spline Ti
pair_coeff * * Ti.meam.spline Ti O
```

### 4.196.3 Description

The *meam/spline* style computes pairwise interactions for metals using a variant of modified embedded-atom method (MEAM) potentials *(Lenosky)*. For a single species ("old-style") MEAM, the total energy E is given by

$$E = \sum_{i<j} \phi(r_{ij}) + \sum_i U(n_i)$$
$$n_i = \sum_j \rho(r_{ij}) + \sum_{\substack{j<k,\\ j,k\neq i}} f(r_{ij}) f(r_{ik}) g[\cos(\theta_{jik})]$$

where $\rho_i$ is the density at atom I, $\theta_{jik}$ is the angle between atoms J, I, and K centered on atom I. The five functions $\phi, U, \rho, f$, and $g$ are represented by cubic splines.

The *meam/spline* style also supports a new style multicomponent modified embedded-atom method (MEAM) potential *(Zhang)*, where the total energy E is given by

$$E = \sum_{i<j} \phi_{ij}(r_{ij}) + \sum_i U_i(n_i)$$
$$n_i = \sum_{j\neq i} \rho_j(r_{ij}) + \sum_{\substack{j<k,\\ j,k\neq i}} f_j(r_{ij}) f_k(r_{ik}) g_{jk}[\cos(\theta_{jik})]$$

where the five functions $\phi, U, \rho, f$, and $g$ depend on the chemistry of the atoms in the interaction. In particular, if there are N different chemistries, there are N different $U$, $\rho$, and $f$ functions, while there are N(N+1)/2 different $\phi$ and $g$ functions. The new style multicomponent MEAM potential files are indicated by the second line in the file starts with "meam/spline" followed by the number of elements and the name of each element.

The cutoffs and the coefficients for these spline functions are listed in a parameter file which is specified by the *pair_coeff* command. Parameter files for different elements are included in the "potentials" directory of the LAMMPS distribution and have a ".meam.spline" file suffix. All of these files are parameterized in terms of LAMMPS *metal units*.

Note that unlike for other potentials, cutoffs for spline-based MEAM potentials are not set in the pair_style or pair_coeff command; they are specified in the potential files themselves.

Unlike the EAM pair style, which retrieves the atomic mass from the potential file, the spline-based MEAM potentials do not include mass information; thus you need to use the *mass* command to specify it.

Only a single pair_coeff command is used with the *meam/spline* style which specifies a potential file with parameters for all needed elements. These are mapped to LAMMPS atom types by specifying N additional arguments after the filename in the pair_coeff command, where N is the number of LAMMPS atom types:

- filename
- N element names = mapping of spline-based MEAM elements to atom types

See the *pair_coeff* page for alternate ways to specify the path for the potential file.

As an example, imagine the Ti.meam.spline file has values for Ti (old style). In that case your LAMMPS simulation may only have one atom type which has to be mapped to the Ti element as follows:

```
pair_coeff * * Ti.meam.spline Ti
```

The first 2 arguments must be * * and there may be only one element following or NULL. Systems where there would be multiple atom types assigned to the same element are **not** supported by this pair style due to limitations in its implementation. If a mapping value is specified as NULL, the mapping is not performed. This can be used when a *meam/spline* potential is used as part of the *hybrid* pair style. The NULL values are placeholders for atom types that will be used with other potentials.

An example with a two component spline (new style) is TiO.meam.spline, where the command

```
pair_coeff * * TiO.meam.spline Ti O
```

will map the first atom type to Ti and the second atom type to O. Note in this case that the species names need to match exactly with the names of the elements in the TiO.meam.spline file; otherwise an error will be raised. This behavior is different than the old style MEAM files.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.196.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support the *pair_modify* shift, table, and tail options.

The *meam/spline* pair style does not write its information to *binary restart files*, since it is stored in an external potential parameter file. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

The *meam/spline* pair style can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

---

### 4.196.5 Restrictions

This pair style requires the *newton* setting to be "on" for pair interactions.

This pair style does not support mapping multiple atom types to the same element.

This pair style is only enabled if LAMMPS was built with the MANYBODY package. See the *Build package* page for more info.

### 4.196.6 Related commands

*pair_coeff*, *pair_style meam*

### 4.196.7 Default

none

---

**(Lenosky)** Lenosky, Sadigh, Alonso, Bulatov, de la Rubia, Kim, Voter, Kress, Modelling Simulation Materials Science Engineering, 8, 825 (2000).

**(Zhang)** Zhang and Trinkle, Computational Materials Science, 124, 204-210 (2016).

## 4.197 pair_style meam/sw/spline command

### 4.197.1 Syntax

```
pair_style meam/sw/spline
```

### 4.197.2 Examples

```
pair_style meam/sw/spline
pair_coeff * * Ti.meam.sw.spline Ti
pair_coeff * * Ti.meam.sw.spline Ti Ti Ti
```

### 4.197.3 Description

The *meam/sw/spline* style computes pairwise interactions for metals using a variant of modified embedded-atom method (MEAM) potentials *(Lenosky)* with an additional Stillinger-Weber (SW) term *(Stillinger)* in the energy. This

form of the potential was first proposed by Nicklas, Fellinger, and Park *(Nicklas)*. We refer to it as MEAM+SW. The total energy E is given by

$$
\begin{aligned}
E &= E_{MEAM} + E_{SW} \\
E_{MEAM} &= \sum_{IJ} \phi(r_{IJ}) + \sum_{I} U(\rho_I) \\
E_{SW} &= \sum_{I} \sum_{JK} F(r_{IJ})\, F(r_{IK})\, G(\cos(\theta_{JIK})) \\
\rho_I &= \sum_{J} \rho(r_{IJ}) + \sum_{JK} f(r_{IJ})\, f(r_{IK})\, g(\cos(\theta_{JIK}))
\end{aligned}
$$

where $\rho_I$ is the density at atom I, $\theta_{JIK}$ is the angle between atoms J, I, and K centered on atom I. The seven functions $\phi, F, G, U, \rho, f$, and $g$ are represented by cubic splines.

The cutoffs and the coefficients for these spline functions are listed in a parameter file which is specified by the *pair_coeff* command. Parameter files for different elements are included in the "potentials" directory of the LAMMPS distribution and have a ".meam.sw.spline" file suffix. All of these files are parameterized in terms of LAMMPS *metal units*.

Note that unlike for other potentials, cutoffs for spline-based MEAM+SW potentials are not set in the pair_style or pair_coeff command; they are specified in the potential files themselves.

Unlike the EAM pair style, which retrieves the atomic mass from the potential file, the spline-based MEAM+SW potentials do not include mass information; thus you need to use the *mass* command to specify it.

Only a single pair_coeff command is used with the meam/sw/spline style which specifies a potential file with parameters for all needed elements. These are mapped to LAMMPS atom types by specifying N additional arguments after the filename in the pair_coeff command, where N is the number of LAMMPS atom types:

- filename
- N element names = mapping of spline-based MEAM+SW elements to atom types

See the *pair_coeff* page for alternate ways to specify the path for the potential file.

As an example, imagine the Ti.meam.sw.spline file has values for Ti. If your LAMMPS simulation has 3 atoms types and they are all to be treated with this potential, you would use the following pair_coeff command:

```
pair_coeff * * Ti.meam.sw.spline Ti Ti Ti
```

The first 2 arguments must be * * so as to span all LAMMPS atom types. The three Ti arguments map LAMMPS atom types 1,2,3 to the Ti element in the potential file. If a mapping value is specified as NULL, the mapping is not performed. This can be used when a *meam/sw/spline* potential is used as part of the hybrid pair style. The NULL values are placeholders for atom types that will be used with other potentials.

**Note**

The *meam/sw/spline* style currently supports only single-element MEAM+SW potentials. It may be extended for alloy systems in the future.

Example input scripts that use this pair style are provided in the examples/PACKAGES/meam_sw_spline directory.

### 4.197.4 Mixing, shift, table, tail correction, restart, rRESPA info

The pair style does not support multiple element types or mixing. It has been designed for pure elements only.

This pair style does not support the *pair_modify* shift, table, and tail options.

The *meam/sw/spline* pair style does not write its information to *binary restart files*, since it is stored in an external potential parameter file. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

The *meam/sw/spline* pair style can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

---

### 4.197.5 Restrictions

This pair style requires the *newton* setting to be "on" for pair interactions.

This pair style is only enabled if LAMMPS was built with the MANYBODY package. See the *Build package* page for more info.

### 4.197.6 Related commands

*pair_coeff*, *pair_style meam*, *pair_style meam/spline*

### 4.197.7 Default

none

---

**(Lenosky)** Lenosky, Sadigh, Alonso, Bulatov, de la Rubia, Kim, Voter, Kress, Modell. Simul. Mater. Sci. Eng. 8, 825 (2000).

**(Stillinger)** Stillinger, Weber, Phys. Rev. B 31, 5262 (1985).

**(Nicklas)** The spline-based MEAM+SW format was first devised and used to develop potentials for bcc transition metals by Jeremy Nicklas, Michael Fellinger, and Hyoungki Park at The Ohio State University.

## 4.198 pair_style mesocnt command

## 4.199 pair_style mesocnt/viscous command

### 4.199.1 Syntax

```
pair_style style neigh_cutoff mode neigh_mode
```

- style = *mesocnt* or *mesocnt/viscous*
- neigh_cutoff = neighbor list cutoff (distance units)
- mode = *chain* or *segment* (optional)
- neigh_mode = *id* or *topology* (optional)

### 4.199.2 Examples

```
pair_style mesocnt 30.0
pair_coeff * * C_10_10.mesocnt 2

pair_style mesocnt/viscous 60.0 chain topology
pair_coeff * * C_10_10.mesocnt 0.001 20.0 0.2 2 4
```

### 4.199.3 Description

Style *mesocnt* implements a mesoscopic potential for the interaction of carbon nanotubes (CNTs), or other quasi-1D objects such as other kinds of nanotubes or nanowires. In this potential, CNTs are modelled as chains of cylindrical segments in which each infinitesimal surface element interacts with all other CNT surface elements with the Lennard-Jones (LJ) term adopted from the *airebo* style. The interaction energy is then computed by integrating over the surfaces of all interacting CNTs.

In LAMMPS, cylindrical segments are represented by bonds. Each segment is defined by its two end points ("nodes") which correspond to atoms in LAMMPS. For the exact functional form of the potential and implementation details, the reader is referred to the original papers *(Volkov1)* and *(Volkov2)*.

Changed in version 15Sep2022.

The potential supports two modes, *segment* and *chain*. By default, *chain* mode is enabled. In *segment* mode, interactions are pair-wise between all neighboring segments based on a segment-segment approach (keyword *segment* in pair_style command). In *chain* mode, interactions are calculated between each segment and infinitely or semi-infinitely long CNTs as described in *(Volkov1)*. Chains of segments are converted to these (semi-)infinite CNTs bases on an approximate chain approach outlined in *(Volkov2)*. Hence, interactions are calculated on a segment-chain basis (keyword *chain* in the pair_style command). Using *chain* mode allows to simplify the computation of the interactions significantly and reduces the computational times to the same order of magnitude as for regular bead spring models where beads interact with the standard *pair_lj/cut* potential. However, this method is only valid when the curvature of the CNTs in the system is small. When CNTs are buckled (see *angle_mesocnt*), local curvature can be very high and the pair_style automatically switches to *segment* mode for interactions involving buckled CNTs.

The potential further implements two different neighbor list construction modes. Mode *id* uses atom and mol IDs to construct neighbor lists while *topology* modes uses only the bond topology of the system. While *id* mode requires bonded atoms to have consecutive LAMMPS atom IDs and atoms in different CNTs to have different LAMMPS molecule IDs, *topology* mode has no such requirement. Using *id* mode is faster and is enabled by default.

**Note**

Neighbor *id* mode requires all CNTs in the system to have distinct LAMMPS molecule IDs and bonded atoms to have consecutive LAMMPS atom IDs. If this is not possible (e.g. in simulations of CNT rings), *topology* mode needs to be enabled in the pair_style command.

Added in version 15Sep2022.

In addition to the LJ interactions described above, style *mesocnt/viscous* explicitly models friction between neighboring segments. Friction forces are a function of the relative velocity between a segment and its neighboring approximate chain (even in *segment* mode) and only act along the axes of the interacting segment and chain. In this potential, friction forces acting per unit length of a nanotube segment are modelled as a shifted logistic function:

$$F^{\mathrm{FRICTION}}(v)/L = \frac{F^{\max}}{1+\exp(-k(v-v_0))} - \frac{F^{\max}}{1+\exp(kv_0)}$$

In the pair_style command, the modes described above can be toggled using the *segment* or *chain* keywords. The neighbor list cutoff defines the cutoff within which atoms are included in the neighbor list for constructing neighboring CNT chains. This is different from the potential cutoff, which is directly calculated from parameters specified in the potential file. We recommend using a neighbor list cutoff of at least 3 times the maximum segment length used in the simulation to ensure proper neighbor chain construction.

**Note**

CNT ends are treated differently by all *mesocnt* styles. Atoms on CNT ends need to be assigned different LAMMPS atom types than atoms not on CNT ends.

Style *mesocnt* requires tabulated data provided in a single ASCII text file, as well as a list of integers corresponding to all LAMMPS atom types representing CNT ends:

- filename
- *N* CNT end atom types

For example, if your LAMMPS simulation of (10, 10) nanotubes has 4 atom types where atom types 1 and 3 are assigned to 'inner' nodes and atom types 2 and 4 are assigned to CNT end nodes, the pair_coeff command would be:

```
pair_coeff * * C_10_10.mesocnt 2 4
```

Likewise, style *mesocnt/viscous* also requires the same information as style *mesocnt*, with the addition of 3 parameters for the viscous friction forces as listed above:

- filename
- $F^{\max}$
- $k$
- $v_0$
- *N* CNT end atom types

Using the same example system as with style *mesocnt* with the addition of friction, the pair_coeff command is:

```
pair_coeff * * C_10_10.mesocnt 0.03 20.0 0.20 2 4
```

Potential files for CNTs can be readily generated using the freely available code provided on

```
https://github.com/phankl/cntpot
```

Using the same approach, it should also be possible to generate potential files for other 1D systems mentioned above.

**Note**

Because of their size, *mesocnt* style potential files are not bundled with LAMMPS. When compiling LAMMPS from source code, the file `C_10_10.mesocnt` should be downloaded separately from https://download.lammps.org/potentials/C_10_10.mesocnt

The first line of the potential file provides a time stamp and general information. The second line lists four integers giving the number of data points provided in the subsequent four data tables. The third line lists four floating point numbers: the CNT radius R, the LJ parameter sigma and two numerical parameters delta1 and delta2. These four parameters are given in Angstroms. This is followed by four data tables each separated by a single empty line. The first two tables have two columns and list the parameters uInfParallel and Gamma respectively. The last two tables

have three columns giving data on a quadratic array and list the parameters Phi and uSemiParallel respectively. uInfParallel and uSemiParallel are given in eV/Angstrom, Phi is given in eV and Gamma is unitless.

If a simulation produces many warnings about segment-chain interactions falling outside the interpolation range, we recommend generating a potential file with lower values of delta1 and delta2.

---

### 4.199.4 Mixing, shift, table, tail correction, restart, rRESPA info

These pair styles does not support mixing.

These pair styles does not support the *pair_modify* shift, table, and tail options.

These pair styles do not write their information to *binary restart files*, since it is stored in tabulated potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

These pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

---

### 4.199.5 Restrictions

These styles are part of the MESONT package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

These pair styles require the *newton* setting to be "on" for pair interactions.

These pair styles require all 3 *special_bonds lj* settings to be non-zero for proper neighbor list construction.

Pair style *mesocnt/viscous* requires you to use the *comm_modify vel yes* command so that velocities are stored by ghost atoms.

### 4.199.6 Related commands

*pair_coeff* , *bond_style mesocnt*, *angle_style mesocnt*

### 4.199.7 Default

mode = chain, neigh_mode = id

---

**(Volkov1)** Volkov and Zhigilei, J Phys Chem C, 114, 5513 (2010).

**(Volkov2)** Volkov, Simov and Zhigilei, APS Meeting Abstracts, Q31.013 (2008).

## 4.200 pair_style edpd command

Accelerator Variants: *edpd/gpu*

## 4.201 pair_style mdpd command

Accelerator Variants: *mdpd/gpu*

# 4.202 pair_style mdpd/rhosum command

# 4.203 pair_style tdpd command

## 4.203.1 Syntax

```
pair_style style args
```

- style = *edpd* or *mdpd* or *mdpd/rhosum* or *tdpd*
- args = list of arguments for a particular style

```
edpd args = cutoff seed
  cutoff = global cutoff for eDPD interactions (distance units)
  seed = random # seed (integer) (if <= 0, eDPD will use current time as the seed)
mdpd args = T cutoff seed
  T = temperature (temperature units)
  cutoff = global cutoff for mDPD interactions (distance units)
  seed = random # seed (integer) (if <= 0, mDPD will use current time as the seed)
mdpd/rhosum args =
tdpd args = T cutoff seed
  T = temperature (temperature units)
  cutoff = global cutoff for tDPD interactions (distance units)
  seed = random # seed (integer) (if <= 0, tDPD will use current time as the seed)
```

## 4.203.2 Examples

```
pair_style edpd 1.58 9872598
pair_coeff * * 18.75 4.5 0.41 1.58 1.42E-5 2.0 1.58
pair_coeff 1 1 18.75 4.5 0.41 1.58 1.42E-5 2.0 1.58 power 10.54 -3.66 3.44 -4.10
pair_coeff 1 1 18.75 4.5 0.41 1.58 1.42E-5 2.0 1.58 power 10.54 -3.66 3.44 -4.10 kappa -
↪0.44 -3.21 5.04 0.00

pair_style hybrid/overlay mdpd/rhosum mdpd 1.0 1.0 65689
pair_coeff 1 1 mdpd/rhosum  0.75
pair_coeff 1 1 mdpd -40.0 25.0 18.0 1.0 0.75

pair_style tdpd 1.0 1.58 935662
pair_coeff * * 18.75 4.5 0.41 1.58 1.58 1.0 1.0E-5 2.0
pair_coeff 1 1 18.75 4.5 0.41 1.58 1.58 1.0 1.0E-5 2.0 3.0 1.0E-5 2.0
```

## 4.203.3 Description

The *edpd* style computes the pairwise interactions and heat fluxes for eDPD particles following the formulations in *(Li2014_JCP)* and *Li2015_CC*. The time evolution of an eDPD particle is governed by the conservation of momentum and energy given by

$$\frac{\mathrm{d}^2\mathbf{r}_i}{\mathrm{d}t^2} = \frac{\mathrm{d}\mathbf{v}_i}{\mathrm{d}t} = \mathbf{F}_i = \sum_{i \neq j} (\mathbf{F}_{ij}^C + \mathbf{F}_{ij}^D + \mathbf{F}_{ij}^R)$$

$$C_v \frac{\mathrm{d}T_i}{\mathrm{d}t} = q_i = \sum_{i \neq j} (q_{ij}^C + q_{ij}^V + q_{ij}^R),$$

where the three components of $F_i$ including the conservative force $F_{ij}^C$, dissipative force $F_{ij}^D$ and random force $F_{ij}^R$ are expressed as

$$\begin{aligned}
\mathbf{F}_{ij}^C &= \alpha_{ij}\omega_C(r_{ij})\mathbf{e}_{ij} \\
\mathbf{F}_{ij}^D &= -\gamma\omega_D(r_{ij})(\mathbf{e}_{ij}\cdot\mathbf{v}_{ij})\mathbf{e}_{ij} \\
\mathbf{F}_{ij}^R &= \sigma\omega_R(r_{ij})\xi_{ij}\Delta t^{-1/2}\mathbf{e}_{ij} \\
\omega_C(r) &= 1 - r/r_c \\
\alpha_{ij} &= A\cdot k_B(T_i+T_j)/2 \\
\omega_D(r) &= \omega_R^2(r) = (1-r/r_c)^s \\
\sigma_{ij}^2 &= 4\gamma k_B T_i T_j/(T_i+T_j)
\end{aligned}$$

in which the exponent of the weighting function $s$ can be defined as a temperature-dependent variable. The heat flux between particles accounting for the collisional heat flux $q^C$, viscous heat flux $q^V$, and random heat flux $q^R$ are given by

$$\begin{aligned}
q_i^C &= \sum_{j\neq i} k_{ij}\omega_{CT}(r_{ij})\left(\frac{1}{T_i}-\frac{1}{T_j}\right) \\
q_i^V &= \frac{1}{2C_v}\sum_{j\neq i}\left\{\omega_D(r_{ij})\left[\gamma_{ij}\left(\mathbf{e}_{ij}\cdot\mathbf{v}_{ij}\right)^2 - \frac{\left(\sigma_{ij}\right)^2}{m}\right] - \sigma_{ij}\omega_R(r_{ij})\left(\mathbf{e}_{ij}\cdot\mathbf{v}_{ij}\right)\xi_{ij}\right\} \\
q_i^R &= \sum_{j\neq i}\beta_{ij}\omega_{RT}(r_{ij})dt^{-1/2}\xi_{ij}^e \\
\omega_{CT}(r) &= \omega_{RT}^2(r) = \left(1-r/r_{ct}\right)^{s_T} \\
k_{ij} &= C_v^2\kappa(T_i+T_j)^2/4k_B \\
\beta_{ij}^2 &= 2k_B k_{ij}
\end{aligned}$$

where the mesoscopic heat friction $\kappa$ is given by

$$\kappa = \frac{315k_B\upsilon}{2\pi\rho C_v r_{ct}^5}\frac{1}{Pr},$$

with $\upsilon$ being the kinematic viscosity. For more details, see Eq.(15) in *(Li2014_JCP)*.

The following coefficients must be defined in eDPD system for each pair of atom types via the *pair_coeff* command as in the examples above.

- A (force units)
- $\gamma$ (force/velocity units)
- power_f (positive real)
- cutoff (distance units)
- kappa (thermal conductivity units)
- power_T (positive real)
- cutoff_T (distance units)
- optional keyword = power or kappa

The keyword *power* or *kappa* is optional. Both "power" and "kappa" require 4 parameters $c_1, c_2, c_3, c_4$ showing the temperature dependence of the exponent $s(T) = \text{power}_f(1+c_1(T-1)+c_2(T-1)^2+c_3(T-1)^3+c_4(T-1)^4)$ and of the mesoscopic heat friction $s_T(T) = \kappa(1+c_1(T-1)+c_2(T-1)^2+c_3(T-1)^3+c_4(T-1)^4)$. If the keyword *power* or *kappa* is not specified, the eDPD system will use constant power_f and $\kappa$, which is independent to temperature changes.

The *mdpd/rhosum* style computes the local particle mass density $\rho$ for mDPD particles by kernel function interpolation.

The following coefficients must be defined for each pair of atom types via the *pair_coeff* command as in the examples above.

- cutoff (distance units)

---

The *mdpd* style computes the many-body interactions between mDPD particles following the formulations in *(Li2013_POF)*. The dissipative and random forces are in the form same as the classical DPD, but the conservative force is local density dependent, which are given by

$$\begin{aligned}\mathbf{F}_{ij}^{C} &= Aw_c(r_{ij})\mathbf{e}_{ij} + B(\rho_i + \rho_j)w_d(r_{ij})\mathbf{e}_{ij}\\ \mathbf{F}_{ij}^{D} &= -\gamma\omega_D(r_{ij})(\mathbf{e}_{ij}\cdot\mathbf{v}_{ij})\mathbf{e}_{ij}\\ \mathbf{F}_{ij}^{R} &= \sigma\omega_R(r_{ij})\xi_{ij}\Delta t^{-1/2}\mathbf{e}_{ij}\end{aligned}$$

where the first term in $F_C$ with a negative coefficient $A < 0$ stands for an attractive force within an interaction range $r_c$, and the second term with $B > 0$ is the density-dependent repulsive force within an interaction range $r_d$.

The following coefficients must be defined for each pair of atom types via the *pair_coeff* command as in the examples above.

- A (force units)
- B (force units)
- $\gamma$ (force/velocity units)
- cutoff_c (distance units)
- cutoff_d (distance units)

---

The *tdpd* style computes the pairwise interactions and chemical concentration fluxes for tDPD particles following the formulations in *(Li2015_JCP)*. The time evolution of a tDPD particle is governed by the conservation of momentum and concentration given by

$$\begin{aligned}\frac{\mathrm{d}^2\mathbf{r}_i}{\mathrm{d}t^2} = \frac{\mathrm{d}\mathbf{v}_i}{\mathrm{d}t} &= \mathbf{F}_i = \sum_{i\neq j}(\mathbf{F}_{ij}^{C} + \mathbf{F}_{ij}^{D} + \mathbf{F}_{ij}^{R})\\ \frac{\mathrm{d}C_i}{\mathrm{d}t} &= Q_i = \sum_{i\neq j}(Q_{ij}^{D} + Q_{ij}^{R}) + Q_i^{S}\end{aligned}$$

where the three components of $F_i$ including the conservative force $F_{ij}^C$, dissipative force $F_{ij}^C$ and random force $F_{ij}^C$ are expressed as

$$\begin{aligned}\mathbf{F}_{ij}^{C} &= A\omega_C(r_{ij})\mathbf{e}_{ij}\\ \mathbf{F}_{ij}^{D} &= -\gamma\omega_D(r_{ij})(\mathbf{e}_{ij}\cdot\mathbf{v}_{ij})\mathbf{e}_{ij}\\ \mathbf{F}_{ij}^{R} &= \sigma\omega_R(r_{ij})\xi_{ij}\Delta t^{-1/2}\mathbf{e}_{ij}\\ \omega_C(r) &= 1 - r/r_c\\ \omega_D(r) &= \omega_R^2(r) = (1 - r/r_c)^{\mathrm{power_f}}\\ \sigma^2 &= 2\gamma k_B T\end{aligned}$$

The concentration flux between two tDPD particles includes the Fickian flux $Q_{ij}^D$ and random flux $Q_{ij}^R$, which are given by

$$
\begin{aligned}
Q_{ij}^D &= -\kappa_{ij} w_{DC}(r_{ij}) \left(C_i - C_j\right) \\
Q_{ij}^R &= \varepsilon_{ij} \left(C_i + C_j\right) w_{RC}(r_{ij}) \xi_{ij} \\
w_{DC}(r_{ij}) &= w_{RC}^2(r_{ij}) = (1 - r/r_{cc})^{\text{power}_{cc}} \\
\varepsilon_{ij}^2 &= m_s^2 \kappa_{ij} \rho
\end{aligned}
$$

where the parameters kappa and epsilon determine the strength of the Fickian and random fluxes. $m_s$ is the mass of a single solute molecule. In general, $m_s$ is much smaller than the mass of a tDPD particle $m$. For more details, see *(Li2015_JCP)*.

The following coefficients must be defined for each pair of atom types via the *pair_coeff* command as in the examples above.

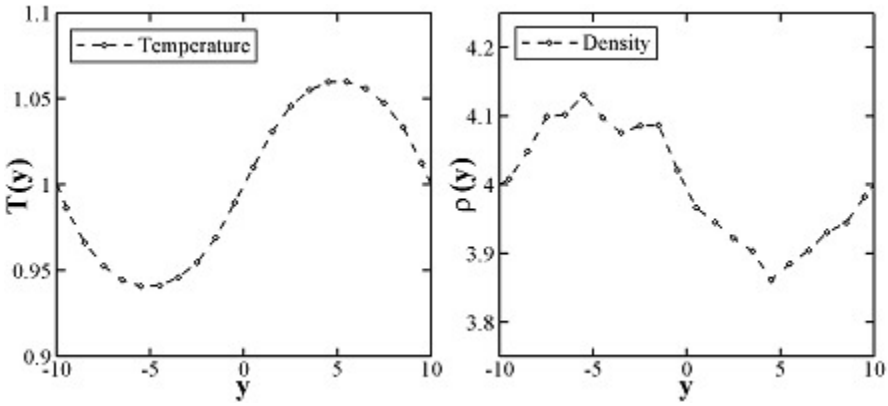
- A (force units)
- $\gamma$ (force/velocity units)
- power_f (positive real)
- cutoff (distance units)
- cutoff_CC (distance units)
- $\kappa_i$ (diffusivity units)
- $\varepsilon_i$ (diffusivity units)
- power_cc_i (positive real)

The last 3 values must be repeated Nspecies times, so that values for each of the Nspecies chemical species are specified, as indicated by the "I" suffix. In the first pair_coeff example above for pair_style tdpd, Nspecies = 1. In the second example, Nspecies = 2, so 3 additional coeffs are specified (for species 2).

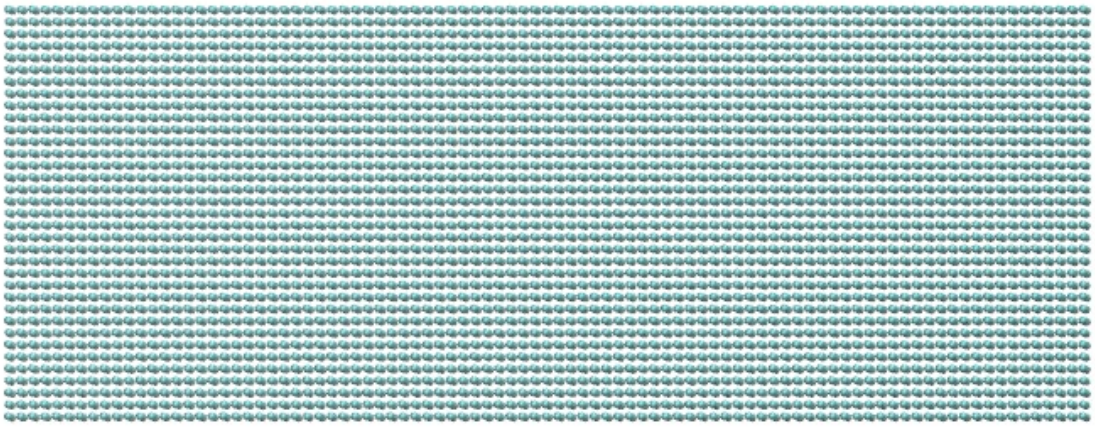
---

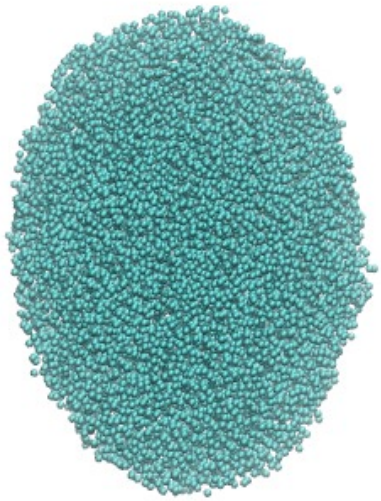
## 4.203.4 Example scripts

There are example scripts for using all these pair styles in examples/PACKAGES/mesodpd. The example for an eDPD simulation models heat conduction with source terms analog of periodic Poiseuille flow problem. The setup follows Fig.12 in *(Li2014_JCP)*. The output of the short eDPD simulation (about 2 minutes on a single core) gives a temperature and density profiles as



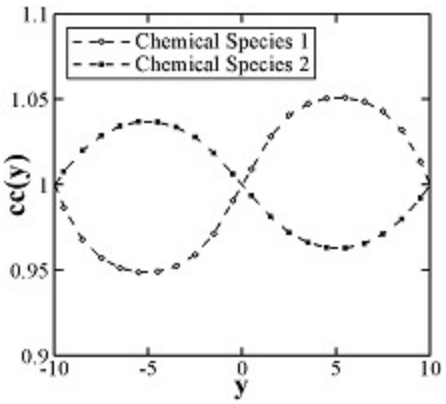

The example for a mDPD simulation models the oscillations of a liquid droplet started from a liquid film. The mDPD parameters are adopted from *(Li2013_POF)*. The short mDPD run (about 2 minutes on a single core) generates a particle trajectory which can be visualized as follows.





The first image is the initial state of the simulation. If you click it a GIF movie should play in your browser. The second image is the final state of the simulation.

The example for a tDPD simulation computes the effective diffusion coefficient of a tDPD system using a method analogous to the periodic Poiseuille flow. The tDPD system is specified with two chemical species, and the setup follows Fig.1 in *(Li2015_JCP)*. The output of the short tDPD simulation (about one and a half minutes on a single core) gives the concentration profiles of the two chemical species as




Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 4.203.5 Mixing, shift, table, tail correction, restart, rRESPA info

The styles *edpd*, *mdpd*, *mdpd/rhosum* and *tdpd* do not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

The styles *edpd*, *mdpd*, *mdpd/rhosum* and *tdpd* do not support the *pair_modify* shift, table, and tail options.

The styles *edpd*, *mdpd*, *mdpd/rhosum* and *tdpd* do not write information to *binary restart files*. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

### 4.203.6 Restrictions

The pair styles *edpd*, *mdpd*, *mdpd/rhosum* and *tdpd* are part of the DPD-MESO package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.203.7 Related commands

*pair_coeff*, *fix mvv/dpd*, *fix mvv/edpd*, *fix mvv/tdpd*, *fix edpd/source*, *fix tdpd/source*, *compute edpd/temp/atom*, *compute tdpd/cc/atom*

### 4.203.8 Default

none

**(Li2014_JCP)** Li, Tang, Lei, Caswell, Karniadakis, J Comput Phys, 265: 113-127 (2014). DOI: 10.1016/j.jcp.2014.02.003.

**(Li2015_CC)** Li, Tang, Li, Karniadakis, Chem Commun, 51: 11038-11040 (2015). DOI: 10.1039/C5CC01684C.

**(Li2013_POF)** Li, Hu, Wang, Ma, Zhou, Phys Fluids, 25: 072103 (2013). DOI: 10.1063/1.4812366.

**(Li2015_JCP)** Li, Yazdani, Tartakovsky, Karniadakis, J Chem Phys, 143: 014101 (2015). DOI: 10.1063/1.4923254.

## 4.204 pair_style mgpt command

### 4.204.1 Syntax

```
pair_style mgpt
```

### 4.204.2 Examples

```
pair_style mgpt
pair_coeff * * Ta6.8x.mgpt.parmin Ta6.8x.mgpt.potin Omega
cp ~/lammps/potentials/Ta6.8x.mgpt.parmin parmin
cp ~/lammps/potentials/Ta6.8x.mgpt.potin potin
pair_coeff * * parmin potin Omega volpress yes nbody 1234 precision double
pair_coeff * * parmin potin Omega volpress yes nbody 12
```

### 4.204.3 Description

Within DFT quantum mechanics, generalized pseudopotential theory (GPT) (*Moriarty1*) provides a first-principles approach to multi-ion interatomic potentials in d-band transition metals, with a volume-dependent, real-space total-energy functional for the N-ion elemental bulk material in the form

$$E_{\rm tot}(\mathbf{R}_1 \ldots \mathbf{R}_N) = NE_{\rm vol}(\Omega) + \frac{1}{2}\sum_{i,j}{}' v_2(ij;\Omega) + \frac{1}{6}\sum_{i,j,k}{}' v_3(ijk;\Omega) + \frac{1}{24}\sum_{i,j,k,l}{}' v_4(ijkl;\Omega)$$

where the prime on each summation sign indicates the exclusion of all self-interaction terms from the summation. The leading volume term E_vol as well as the two-ion central-force pair potential v_2 and the three- and four-ion angular-force potentials, v_3 and v_4, depend explicitly on the atomic volume Omega, but are structure independent and transferable to all bulk ion configurations, either ordered or disordered, and with of without the presence of point and line defects. The simplified model GPT or MGPT (*Moriarty2*, *Moriarty3*), which retains the form of E_tot and permits more efficient large-scale atomistic simulations, derives from the GPT through a series of systematic approximations applied to E_vol and the potentials v_n that are valid for mid-period transition metals with nearly half-filled d bands.

Both analytic (*Moriarty2*) and matrix (*Moriarty3*) representations of MGPT have been developed. In the more general matrix representation, which can also be applied to f-band actinide metals and permits both canonical and non-canonical d/f bands, the multi-ion potentials are evaluated on the fly during a simulation through d- or f-state matrix multiplication, and the forces that move the ions are determined analytically. Fast matrix-MGPT algorithms have been developed independently by Glosli (*Glosli*, *Moriarty3*) and by Oppelstrup (*Oppelstrup*)

The *mgpt* pair style calculates forces, energies, and the total energy per atom, E_tot/N, using the Oppelstrup matrix-MGPT algorithm. Input potential and control data are entered through the *pair_coeff* command. Each material treated requires input parmin and potin potential files, as shown in the above examples, as well as specification by the user of the initial atomic volume Omega through pair_coeff. At the beginning of a time step in any simulation, the total volume of the simulation cell V should always be equal to Omega*N, where N is the number of metal ions present, taking into account the presence of any vacancies and/or interstitials in the case of a solid. In a constant-volume simulation, which is the normal mode of operation for the *mgpt* pair style, Omega, V and N all remain constant throughout the simulation and thus are equal to their initial values. In a constant-stress simulation, the cell volume V will change (slowly) as the simulation proceeds. After each time step, the atomic volume should be updated by the code as Omega = V/N. In addition, the volume term E_vol and the potentials v_2, v_3 and v_4 have to be removed at the end of the time step, and

then respecified at the new value of Omega. In all simulations, Omega must remain within the defined volume range for E_vol and the potentials for the given material.

The default option volpress yes in the *pair_coeff* command includes all volume derivatives of E_tot required to calculate the stress tensor and pressure correctly. The option volpress no disregards the pressure contribution resulting from the volume term E_vol, and can be used for testing and analysis purposes. The additional optional variable nbody controls the specific terms in E_tot that are calculated. The default option and the normal option for mid-period transition and actinide metals is nbody 1234 for which all four terms in E_tot are retained. The option nbody 12, for example, retains only the volume term and the two-ion pair potential term and can be used for GPT series-end transition metals that can be well described without v_3 and v_4. The nbody option can also be used to test or analyze the contribution of any of the four terms in E_tot to a given calculated property.

The *mgpt* pair style makes extensive use of matrix algebra and includes optimized kernels the Intel/AMD (x86) architecture. When compiled with the appropriate compiler and compiler switches (e.g. `-msse3`) these optimized routines are used automatically. The user will be informed in the output files of the matrix kernels in use. To further improve speed, on x86 the option *precision single* can be added to the *pair_coeff* command, which improves speed (up to a factor of two) at the cost of doing matrix calculations with 7 digit precision instead of the default 15 digits. For consistency, the default option can be specified explicitly by the option *precision double*.

All remaining potential and control data are contained with the parmin and potin files, including cutoffs, atomic mass, and other basic MGPT variables. Specific MGPT potential data for the transition metals tantalum (Ta4 and Ta6.8x potentials), molybdenum (Mo5.2 potentials), and vanadium (V6.1 potentials) are contained in the LAMMPS potentials directory. The stored files are, respectively, Ta4.mgpt.parmin, Ta4.mgpt.potin, Ta6.8x.mgpt.parmin, Ta6.8x.mgpt.potin, Mo5.2.mgpt.parmin, Mo5.2.mgpt.potin, V6.1.mgpt.parmin, and V6.1.mgpt.potin . Useful corresponding informational "README" files on the Ta4, Ta6.8x, Mo5.2 and V6.1 potentials are also included in the potentials directory. These latter files indicate the volume mesh and range for each potential and give appropriate references for the potentials. It is expected that MGPT potentials for additional materials will be added over time.

Useful example MGPT scripts are given in the examples/PACKAGES/mgpt directory. These scripts show the necessary steps to perform constant-volume calculations and simulations. It is strongly recommended that the user work through and understand these examples before proceeding to more complex simulations.

**Note**

For good performance on x86 architecture machines with the GNU compilers, LAMMPS should be built with the compiler flags "-O3 -msse3 -funroll-loops" when including this pair style.

Deprecated since version 4Jul2026: Support for (obsolete) IBM BlueGene machines has been removed.

### 4.204.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support the *pair_modify* mix, shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in potential files. Thus, you needs to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.204.5 Restrictions

This pair style is part of the MGPT package and is only enabled if LAMMPS is built with that package. See the *Build package* page for more info.

The MGPT potentials require the *newton* setting to be "on" for pair style interactions.

The stored parmin and potin potential files provided with LAMMPS in the "potentials" directory are written in Rydberg atomic units, with energies in Rydbergs and distances in Bohr radii. The *mgpt* pair style converts Rydbergs to Hartrees to make the potential files compatible with LAMMPS electron *units*.

The form of E_tot used in the *mgpt* pair style is only appropriate for elemental bulk solids and liquids. This includes solids with point and extended defects such as vacancies, interstitials, grain boundaries and dislocations. Alloys and free surfaces, however, require significant modifications, which are not included in the *mgpt* pair style. Likewise, the *hybrid* pair style is not allowed, where MGPT would be used for some atoms but not for others.

Electron-thermal effects are not included in the standard MGPT potentials provided in the "potentials" directory, where the potentials have been constructed at zero electron temperature. Physically, electron-thermal effects may be important in 3d (e.g., V) and 4d (e.g., Mo) transition metals at high temperatures near melt and above. It is expected that temperature-dependent MGPT potentials for such cases will be added over time.

### 4.204.6 Related commands

*pair_coeff*

### 4.204.7 Default

The options defaults for the *pair_coeff* command are volpress yes, nbody 1234, and precision double.

---

**(Moriarty1)** Moriarty, Physical Review B, 38, 3199 (1988).

**(Moriarty2)** Moriarty, Physical Review B, 42, 1609 (1990). Moriarty, Physical Review B 49, 12431 (1994).

**(Moriarty3)** Moriarty, Benedict, Glosli, Hood, Orlikowski, Patel, Soderlind, Streitz, Tang, and Yang, Journal of Materials Research, 21, 563 (2006).

**(Glosli)** Glosli, unpublished, 2005. Streitz, Glosli, Patel, Chan, Yates, de Supinski, Sexton and Gunnels, Journal of Physics: Conference Series, 46, 254 (2006).

**(Oppelstrup)** Oppelstrup, unpublished, 2015. Oppelstrup and Moriarty, to be published.

## 4.205 pair_style mie/cut command

Accelerator Variants: *mie/cut/gpu*, *mie/cut/kk*, *mie/cut/omp*

### 4.205.1 Syntax

```
pair_style mie/cut cutoff
```

- cutoff = global cutoff for mie/cut interactions (distance units)

### 4.205.2 Examples

```
pair_style mie/cut 10.0
pair_coeff 1 1 0.72 3.40 23.00 6.66
pair_coeff 2 2 0.30 3.55 12.65 6.00
pair_coeff 1 2 0.46 3.32 16.90 6.31
```

### 4.205.3 Description

The *mie/cut* style computes the Mie potential, given by

$$E = C\varepsilon \left[ \left(\frac{\sigma}{r}\right)^{\gamma_{rep}} - \left(\frac{\sigma}{r}\right)^{\gamma_{att}} \right] \qquad r < r_c$$

$r_c$ is the cutoff and C is a function that depends on the repulsive and attractive exponents, given by:

$$C = \left(\frac{\gamma_{rep}}{\gamma_{rep} - \gamma_{att}}\right) \left(\frac{\gamma_{rep}}{\gamma_{att}}\right)^{\left(\frac{\gamma_{att}}{\gamma_{rep} - \gamma_{att}}\right)}$$

Note that for 12/6 exponents, C is equal to 4 and the formula is the same as the standard Lennard-Jones potential.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- epsilon (energy units)
- sigma (distance units)
- gammaR
- gammaA
- cutoff (distance units)

The last coefficient is optional. If not specified, the global cutoff specified in the pair_style command is used.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.205.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon and sigma coefficients and cutoff distance for all of the mie/cut pair styles can be mixed. If not explicitly defined, both the repulsive and attractive gamma exponents for different atoms will be calculated following the same mixing rule defined for distances. The default mix value is *geometric*. See the "pair_modify" command for details.

This pair style supports the *pair_modify* shift option for the energy of the pair interaction.

This pair style supports the *pair_modify* tail option for adding a long-range tail correction to the energy and pressure of the pair interaction.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style supports the use of the *inner*, *middle*, and *outer* keywords of the *run_style respa* command, meaning the pairwise forces can be partitioned by distance at different levels of the rRESPA hierarchy. See the *run_style* command for details.

### 4.205.5 Restrictions

This pair style is part of the EXTRA-PAIR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.205.6 Related commands

*pair_coeff*

### 4.205.7 Default

none

**(Mie)** G. Mie, Ann Phys, 316, 657 (1903).

**(Avendano)** C. Avendano, T. Lafitte, A. Galindo, C. S. Adjiman, G. Jackson, E. Muller, J Phys Chem B, 115, 11154 (2011).

## 4.206 pair_style mliap command

Accelerator Variants: *mliap/kk*

### 4.206.1 Syntax

```
pair_style mliap ... keyword values ...
```

- one or two keyword/value pairs must be appended
- keyword = *model* or *descriptor* or *unified*

```
model values = style filename
  style = linear or quadratic or nn or mliappy
  filename = name of file containing model definitions
descriptor values = style filename
  style = sna or so3 or ace
  filename = name of file containing descriptor definitions
unified values = filename ghostneigh_flag
  filename = name of file containing serialized unified Python object
  ghostneigh_flag = 0/1 to turn off/on inclusion of ghost neighbors in neighbors␣
↪list
```

### 4.206.2 Examples

```
pair_style mliap model linear InP.mliap.model descriptor sna InP.mliap.descriptor
pair_style mliap model quadratic W.mliap.model descriptor sna W.mliap.descriptor
pair_style mliap model nn Si.nn.mliap.model descriptor so3 Si.nn.mliap.descriptor
pair_style mliap model mliappy ACE_NN_Pytorch.pt descriptor ace ccs_single_element.yace
pair_style mliap unified mliap_unified_lj_Ar.pkl 0
pair_coeff * * In P
```

### 4.206.3 Description

Pair style *mliap* provides a general interface to families of machine-learning interatomic potentials. It allows separate definitions of the interatomic potential functional form (*model*) and the geometric quantities that characterize the atomic positions (*descriptor*).

By defining *model* and *descriptor* separately, it is possible to use many different models with a given descriptor, or many different descriptors with a given model. The pair style currently supports *sna*, *so3* and *ace* descriptor styles, but it is straightforward to add new descriptor styles. By using the *unified* keyword, it is possible to define a Python model that combines functionalities of both *model* and *descriptor*.

The SNAP descriptor style *sna* is the same as that used by *pair_style snap*, including the linear, quadratic, and chem variants. The available models are *linear*, *quadratic*, *nn*, and *mliappy*. The *mliappy* style can be used to couple python models, e.g. PyTorch neural network energy models, and requires building LAMMPS with the PYTHON package (see below). In order to train a model, it is useful to know the gradient or derivative of energy, force, and stress w.r.t. model parameters. This information can be accessed using the related *compute mliap* command.

Added in version 2Jun2022.

The descriptor style *so3* is a descriptor that is derived from the the smooth SO(3) power spectrum with the explicit inclusion of a radial basis *(Bartok)* and *(Zagaceta)*. The available models are *linear* and *nn*.

Added in version 17Apr2024.

The descriptor style *ace* is a class of highly general atomic descriptors, atomic cluster expansion descriptors (ACE) from *(Drautz)*, that include a radial basis, an angular basis, and bases for other variables (such as chemical species) if relevant. In descriptor style *ace*, the *ace* descriptors may be defined up to an arbitrary body order. This descriptor style is the same as that used in *pair_style pace* and *compute pace*. The available models with *ace* in ML-IAP are *linear* and *mliappy*. The *ace* descriptors and models require building LAMMPS with the ML-PACE package (see below). The *mliappy* model style may be used with *ace* descriptors, but it requires that LAMMPS is also built with the PYTHON package. As with other model styles, the *mliappy* model style can be used to couple arbitrary python models that use the *ace* descriptors such as Pytorch NNs. Note that *ALL* mliap model styles with *ace* descriptors require that descriptors and hyperparameters are supplied in a *.yace* or *.ace* file, similar to *compute pace*.

The pair_style *mliap* command must be followed by two keywords *model* and *descriptor* in either order, or the one keyword *unified*. A single *pair_coeff* command is also required. The first 2 arguments must be * * so as to span all LAMMPS atom types. This is followed by a list of N arguments that specify the mapping of MLIAP element names to LAMMPS atom types, where N is the number of LAMMPS atom types.

The *model* keyword is followed by the model style. This is followed by a single argument specifying the model filename containing the parameters for a set of elements. The model filename usually ends in the *.mliap.model* extension. It may contain parameters for many elements. The only requirement is that it contain at least those element names appearing in the *pair_coeff* command.

The top of the model file can contain any number of blank and comment lines (start with #), but follows a strict format after that. The first non-blank non-comment line must contain two integers:

- nelems = Number of elements
- nparams = Number of parameters

When the *model* keyword is *linear* or *quadratic*, this is followed by one block for each of the *nelem* elements. Each block consists of *nparams* parameters, one per line. Note that this format is similar, but not identical to that used for the *pair_style snap* coefficient file. Specifically, the line containing the element weight and radius is omitted, since these are handled by the *descriptor*.

When the *model* keyword is *nn* (neural networks), the model file can contain blank and comment lines (start with #) anywhere. The second non-blank non-comment line must contain the string NET, followed by two integers:

- ndescriptors = Number of descriptors
- nlayers = Number of layers (including the hidden layers and the output layer)

and followed by a sequence of a string and an integer for each layer:

- Activation function (linear, sigmoid, tanh or relu)
- nnodes = Number of nodes

This is followed by one block for each of the *nelem* elements. Each block consists of *scale0* minimum value, *scale1* (maximum - minimum) value, in order to normalize the descriptors, followed by *nparams* parameters, including *bias* and *weights* of the model, starting with the first node of the first layer and so on, with a maximum of 30 values per line.

The detail of *nn* module implementation can be found at *(Yanxon)*.

**Notes on mliappy models**

When the *model* keyword is *mliappy*, if the filename ends in '.pt', or '.pth', it will be loaded using pytorch; otherwise, it will be loaded as a pickle file. To load a model from memory (i.e. an existing python object), specify the filename as "LATER", and then call *lammps.mliap.load_model(model)* from python before using the pair style. When using LAMMPS via the library mode, you will need to call *lammps.mliappy.activate_mliappy(lmp)* on the active LAMMPS object before the pair style is defined. This call locates and loads the mliap-specific python module that is built into LAMMPS.

The *descriptor* keyword is followed by a descriptor style, and additional arguments. Currently three descriptor styles are available: *sna*, *so3*, and *ace*.

- *sna* indicates the bispectrum component descriptors used by the Spectral Neighbor Analysis Potential (SNAP) potentials of *pair_style snap*. A single additional argument specifies the descriptor filename containing the parameters and setting used by the SNAP descriptor. The descriptor filename usually ends in the *.mliap.descriptor* extension.
- *so3* indicated the power spectrum component descriptors. A single additional argument specifies the descriptor filename containing the parameters and setting.
- *ace* indicates the atomic cluster expansion (ACE) descriptors. A single additional argument specifies the filename containing parameters, settings, and definitions of the ace descriptors (through tabulated basis function indices and corresponding generalized Clebsch-Gordan coefficients) in the ctilde file format, e.g. in the potential file format with **.ace* or **.yace* extensions from *pair_style pace*. Note that unlike the potential file, the Clebsch-Gordan coefficients in the descriptor file supplied should *NOT* be multiplied by linear or square root embedding terms.

The SNAP descriptor file closely follows the format of the *pair_style snap* parameter file. The file can contain blank and comment lines (start with #) anywhere. Each non-blank non-comment line must contain one keyword/value pair. The required keywords are *rcutfac* and *twojmax*. There are many optional keywords that are described on the *pair_style snap* doc page. In addition, the SNAP descriptor file must contain the *nelems*, *elems*, *radelems*, and *welems* keywords. The *nelems* keyword specifies the number of elements provided in the other three keywords. The *elems* keyword is followed by a list of *nelems* element names that must include the element names appearing in the *pair_coeff* command, but can contain other names too. Similarly, the *radelems* and *welems* keywords are followed by lists of *nelems* numbers giving the element radius and element weight of each element. Obviously, the order in which the elements are listed must be consistent for all three keywords.

The SO3 descriptor file is similar to the SNAP descriptor except that it contains a few more arguments (e.g., *nmax* and *alpha*). The preparation of SO3 descriptor and model files can be done with the PyXtal_FF package.

The ACE descriptor file differs from the SNAP and SO3 files. It more closely resembles the potential file format for linear or square-root embedding ACE potentials used in the *pair_style pace*. As noted above, the key difference is that the Clebsch-Gordan coefficients in the descriptor file with *mliap descriptor ace* are *NOT* multiplied by linear or square root embedding terms. In other words,the model is separated from the descriptor definitions and hyperparameters. In *pair_style pace*, they are combined. The ACE descriptor files required by *mliap* are generated automatically in FitSNAP during linear, pytorch, etc. ACE model fitting. Additional tools are provided there to prepare *ace* descriptor files and

hyperparameters before model fitting. The *ace* descriptor files can also be extracted from ACE model fits in python-ace.. It is important to note that order of the types listed in *pair_coeff* must match the order of the elements/types listed in the ACE descriptor file for all *mliap* styles when using *ace* descriptors.

See the *pair_coeff* page for alternate ways to specify the path for these *model* and *descriptor* files.

**Note**

To significantly reduce SO3 descriptor/force calculation time, some properties are pre-computed and reused during the calculation. These can consume a significant amount of RAM for simulations of larger systems since their size depends on the total number of neighbors per MPI process.

Added in version 3Nov2022.

The *unified* keyword is followed by an argument specifying the filename containing the serialized unified Python object and the "ghostneigh" toggle (0/1) to disable/enable the construction of neighbors lists including neighbors of ghost atoms. If the filename ends in '.pt', or '.pth', it will be loaded using pytorch; otherwise, it will be loaded as a pickle file. If ghostneigh is enabled, it is recommended to set *comm_modify* cutoff manually, such as in the following example.

```
variable ninteractions equal 2
variable cutdist equal 7.5
variable skin equal 1.0
variable commcut equal (${ninteractions}*${cutdist})+${skin}
neighbor ${skin} bin
comm_modify cutoff ${commcut}
```

**Note**

To load a model from memory (i.e. an existing python object), call *lammps.mliap.load_unified(unified)* from python, and then specify the filename as "EXISTS". When using LAMMPS via the library mode, you will need to call *lammps.mliappy.activate_mliappy(lmp)* on the active LAMMPS object before the pair style is defined. This call locates and loads the mliap-specific python module that is built into LAMMPS.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.206.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, where types I and J correspond to two different element types, mixing is performed by LAMMPS with user-specifiable parameters as described above. You never need to specify a pair_coeff command with I != J arguments for this style.

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.206.5 Restrictions

This pair style is part of the ML-IAP package. It is only enabled if LAMMPS was built with that package. In addition, building LAMMPS with the ML-IAP package requires building LAMMPS with the ML-SNAP package. The *mliappy* model requires building LAMMPS with the PYTHON package. The *ace* descriptor requires building LAMMPS with the ML-PACE package. See the *Build package* page for more info. Note that *pair_mliap/kk* acceleration will *not* invoke the *kk* accelerated variants of SNAP or ACE descriptors.

### 4.206.6 Related commands

*pair_style snap*, *compute mliap*

### 4.206.7 Default

none

---

**(Bartok2013)** Bartok, Kondor, Csanyi, Phys Rev B, 87, 184115 (2013).

**(Zagaceta2020)** Zagaceta, Yanxon, Zhu, J Appl Phys, 128, 045113 (2020).

**(Yanxon2020)** Yanxon, Zagaceta, Tang, Matteson, Zhu, Mach. Learn.: Sci. Technol. 2, 027001 (2020).

## 4.207 pair_style momb command

Accelerator Variants: *momb/omp*, *momb/kk*

### 4.207.1 Syntax

```
pair_style momb cutoff s6 d
```

- cutoff = global cutoff (distance units)
- s6 = global scaling factor of the exchange-correlation functional used (unitless)
- d = damping scaling factor of Grimme's method (unitless)

### 4.207.2 Examples

```
pair_style momb 12.0 0.75 20.0
pair_style hybrid/overlay eam/fs lj/charmm/coul/long 10.0 12.0 momb 12.0 0.75 20.0 morse␣
↪5.5

pair_coeff 1 2 momb 0.0 1.0 1.0 10.2847 2.361
```

### 4.207.3 Description

Style *momb* computes pairwise van der Waals (vdW) and short-range interactions using the Morse potential and *(Grimme)* method implemented in the Many-Body Metal-Organic (MOMB) force field described comprehensively in *(Fichthorn)* and *(Zhou)*. Grimme's method is widely used to correct for dispersion in density functional theory calculations.

$$E = D_0[\exp^{-2\alpha(r-r_0)} - 2\exp^{-\alpha(r-r_0)}] - s_6\frac{C_6}{r^6}f_{damp}(r,R_r)$$

$$f_{damp}(r,R_r) = \frac{1}{1+\exp^{-d(r/R_r-1)}}$$

For the *momb* pair style, the following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* as described below:

- $D_0$ (energy units)
- $\alpha$ (1/distance units)
- $r_0$ (distance units)
- $C_6$ (energy*distance^6 units)
- $R_r$ (distance units, typically sum of atomic vdW radii)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.207.4 Restrictions

This style is part of the EXTRA-PAIR package. It is only enabled if LAMMPS is built with that package. See the *Build package* page on for more info.

### 4.207.5 Related commands

*pair_coeff* , *pair_style morse*

### 4.207.6 Default

none

---

**(Grimme)** Grimme, J Comput Chem, 27(15), 1787-1799 (2006).

**(Fichthorn)** Fichthorn, Balankura, Qi, CrystEngComm, 18(29), 5410-5417 (2016).

**(Zhou)** Zhou, Saidi, Fichthorn, J Phys Chem C, 118(6), 3366-3374 (2014).

# 4.208 pair_style morse command

Accelerator Variants: *morse/gpu*, *morse/omp*, *morse/opt*, *morse/kk*

# 4.209 pair_style morse/smooth/linear command

Accelerator Variants: *morse/smooth/linear/omp*, *morse/smooth/linear/kk*

## 4.209.1 Syntax

```
pair_style style args
```

- style = *morse* or *morse/smooth/linear* or *morse/soft*
- args = list of arguments for a particular style

```
morse args = cutoff
  cutoff = global cutoff for Morse interactions (distance units)
morse/smooth/linear args = cutoff
  cutoff = global cutoff for Morse interactions (distance units)
```

## 4.209.2 Examples

```
pair_style morse 2.5
pair_style morse/smooth/linear 2.5
pair_coeff * * 100.0 2.0 1.5
pair_coeff 1 1 100.0 2.0 1.5 3.0
```

## 4.209.3 Description

Style *morse* computes pairwise interactions with the formula

$$E = D_0 \left[ e^{-2\alpha(r-r_0)} - 2e^{-\alpha(r-r_0)} \right] \qquad r < r_c$$

$r_c$ is the cutoff.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $D_0$ (energy units)
- $\alpha$ (1/distance units)
- $r_0$ (distance units)
- cutoff (distance units)

The last coefficient is optional. If not specified, the global morse cutoff is used.

---

The *morse/smooth/linear* variant is similar to the lj/smooth/linear variant in that it adds to the potential a shift and a linear term so that both, potential energy and force, go to zero at the cut-off:

$$\phi(r) = D_0 \left[ e^{-2\alpha(r-r_0)} - 2e^{-\alpha(r-r_0)} \right] \qquad r < r_c$$

$$E(r) = \phi(r) - \phi(r_c) - (r - r_c) \left. \frac{d\phi}{dr} \right|_{r=r_c} \qquad r < r_c$$

The syntax of the pair_style and pair_coeff commands are the same for the *morse* and *morse/smooth/linear* styles.

---

A version of the *morse* style with a soft core, *morse/soft*, suitable for use in free energy calculations, is part of the FEP package and is documented with the *pair_style */soft* styles. The version with soft core is only available if LAMMPS was built with that package. See the *Build package* page for more info.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.209.4 Mixing, shift, table, tail correction, restart, rRESPA info

None of these pair styles support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

All of these pair styles support the *pair_modify* shift option for the energy of the pair interaction.

The *pair_modify* table options are not relevant for the Morse pair styles.

None of these pair styles support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

All of these pair styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

These pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

---

### 4.209.5 Restrictions

The *morse/smooth/linear* pair style is only enabled if LAMMPS was built with the EXTRA-PAIR package. See the *Build package* page for more info.

### 4.209.6 Related commands

*pair_coeff*, *pair_style */soft*

### 4.209.7 Default

none

# 4.210 pair_style multi/lucy command

## 4.210.1 Syntax

```
pair_style multi/lucy style N keyword ...
```

- style = *lookup* or *linear* = method of interpolation
- N = use N values in *lookup*, *linear* tables

## 4.210.2 Examples

```
pair_style multi/lucy linear 1000
pair_coeff * * multibody.table ENTRY1 7.0
```

## 4.210.3 Description

Style *multi/lucy* computes a density-dependent force following from the many-body form described in *(Moore)* and *(Warren)* as

$$F_i^{DD}(\rho_i, \rho_j, r_{ij}) = \frac{1}{2}\omega_{DD}\left(r_{ij}\right)\left[A\left(\rho_i\right) + A\left(\rho_j\right)\right]e_{ij}$$

which consists of a density-dependent function, $A(\rho)$, and a radial-dependent weight function, $\omega_{DD}(r_{ij})$. The radial-dependent weight function, $\omega_{DD}(r_{ij})$, is taken as the Lucy function:

$$\omega_{DD}\left(r_{ij}\right) = \left(1 + \frac{3r_{ij}}{r_{cut}}\right)\left(1 + \frac{r_{ij}}{r_{cut}}\right)^3$$

The density-dependent energy for a given particle is given by:

$$u_i^{DD}\left(\rho_i\right) = \frac{\pi r_{cut}^4}{84}\int_{\rho_0}^{\rho_i} A\left(\rho'\right)d\rho'$$

See the supporting information of *(Brennan)* or the publication by *(Moore)* for more details on the functional form.

An interpolation table is used to evaluate the density-dependent energy ($\int A(\rho')d\rho'$) and force ($A(\rho')$). Note that the prefactor to the energy is computed after the interpolation, thus the $\int A(\rho')d\rho'$ will have units of energy / length^4.

The interpolation table is created as a pre-computation by fitting cubic splines to the file values and interpolating the density-dependent energy and force at each of $N$ densities. During a simulation, the tables are used to interpolate the density-dependent energy and force as needed for each pair of particles separated by a distance $R$. The interpolation is done in one of 2 styles: *lookup* and *linear*.

For the *lookup* style, the density is used to find the nearest table entry, which is the density-dependent energy and force.

For the *linear* style, the density is used to find the 2 surrounding table values from which the density-dependent energy and force are computed by linear interpolation.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above.

- filename
- keyword
- cutoff (distance units)

The filename specifies a file containing the tabulated density-dependent energy and force. The keyword specifies a section of the file. The cutoff is an optional coefficient. If not specified, the outer cutoff in the table itself (see below) will be used to build an interpolation table that extend to the largest tabulated distance. If specified, only file values up to the cutoff are used to create the interpolation table. The format of this file is described below.

---

The format of a tabulated file is a series of one or more sections, defined as follows (without the parenthesized comments):

```
# Density-dependent function (one or more comment or blank lines)

DD-FUNCTION                (keyword is first text on line)
N 500 R 1.0 10.0           (N, R, RSQ parameters)
                           (blank)
1 1.0 25.5 102.34          (index, density, energy/r^4, force)
2 1.02 23.4 98.5
...
500 10.0 0.001 0.003
```

A section begins with a non-blank line whose first character is not a "#"; blank lines or lines starting with "#" can be used as comments between sections. The first line begins with a keyword which identifies the section. The line can contain additional text, but the initial text must match the argument specified in the pair_coeff command. The next line lists (in any order) one or more parameters for the table. Each parameter is a keyword followed by one or more numeric values.

The parameter "N" is required and its value is the number of table entries that follow. Note that this may be different than the *N* specified in the *pair_style multi/lucy* command. Let Ntable = *N* in the pair_style command, and Nfile = "N" in the tabulated file. What LAMMPS does is a preliminary interpolation by creating splines using the Nfile tabulated values as nodal points. It uses these to interpolate the density-dependent energy and force at Ntable different points. The resulting tables of length Ntable are then used as described above, when computing the density-dependent energy and force. This means that if you want the interpolation tables of length Ntable to match exactly what is in the tabulated file (with effectively no preliminary interpolation), you should set Ntable = Nfile, and use the "RSQ" parameter. This is because the internal table abscissa is always RSQ (separation distance squared), for efficient lookup.

All other parameters are optional. If "R" or "RSQ" does not appear, then the distances in each line of the table are used as-is to perform spline interpolation. In this case, the table values can be spaced in *density* uniformly or however you wish to position table values in regions of large gradients.

If used, the parameters "R" or "RSQ" are followed by 2 values *rlo* and *rhi*. If specified, the density associated with each density-dependent energy and force value is computed from these 2 values (at high accuracy), rather than using the (low-accuracy) value listed in each line of the table. The density values in the table file are ignored in this case. For "R", distances uniformly spaced between *rlo* and *rhi* are computed; for "RSQ", squared distances uniformly spaced between *rlo*rlo* and *rhi*rhi* are computed.

**Note**

If you use "R" or "RSQ", the tabulated distance values in the file are effectively ignored, and replaced by new values as described in the previous paragraph. If the density value in the table is not very close to the new value (i.e. round-off difference), then you will be assigning density-dependent energy and force values to a different density, which is probably not what you want. LAMMPS will warn if this is occurring.

Following a blank line, the next N lines list the tabulated values. On each line, the first value is the index from 1 to N, the second value is r (in density units), the third value is the density-dependent function value (in energy units / length^4), and the fourth is the force (in force units). The density values must increase from one line to the next.

Note that one file can contain many sections, each with a tabulated potential. LAMMPS reads the file section by section until it finds one that matches the specified keyword.

---

### 4.210.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

The *pair_modify* shift, table, and tail options are not relevant for this pair style.

This pair style writes the settings for the "pair_style multi/lucy" command to *binary restart files*, so a pair_style command does not need to specified in an input script that reads a restart file. However, the coefficient information is not stored in the restart file, since it is tabulated in the potential files. Thus, pair_coeff commands do need to be specified in the restart input script.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.210.5 Restrictions

This command is part of the DPD-REACT package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.210.6 Related commands

*pair_coeff*

### 4.210.7 Default

none

---

**(Warren)** Warren, Phys Rev E, 68, 066702 (2003).

**(Brennan)** Brennan, J Chem Phys Lett, 5, 2144-2149 (2014).

**(Moore)** Moore, J Chem Phys, 144, 104501 (2016).

## 4.211 pair_style multi/lucy/rx command

Accelerator Variants: *multi/lucy/rx/kk*

### 4.211.1 Syntax

```
pair_style multi/lucy/rx style N keyword ...
```

- style = *lookup* or *linear* = method of interpolation
- N = use N values in *lookup*, *linear* tables
- weighting = fractional or molecular (optional)

### 4.211.2 Examples

```
pair_style multi/lucy/rx linear 1000
pair_style multi/lucy/rx linear 1000 fractional
pair_style multi/lucy/rx linear 1000 molecular
pair_coeff * * multibody.table ENTRY1 h2o h2o 7.0
pair_coeff * * multibody.table ENTRY1 h2o 1fluid 7.0
```

### 4.211.3 Description

Style *multi/lucy/rx* is used in reaction DPD simulations, where the coarse-grained (CG) particles are composed of $m$ species whose reaction rate kinetics are determined from a set of $n$ reaction rate equations through the *fix rx* command. The species of one CG particle can interact with a species in a neighboring CG particle through a site-site interaction potential model. Style *multi/lucy/rx* computes the site-site density-dependent force following from the many-body form described in *(Moore)* and *(Warren)* as

$$F_i^{DD}(\rho_i, \rho_j, r_{ij}) = \frac{1}{2}\omega_{DD}\left(r_{ij}\right)\left[A\left(\rho_i\right) + A\left(\rho_j\right)\right]e_{ij}$$

which consists of a density-dependent function, $A(\rho)$, and a radial-dependent weight function, $\omega_{DD}(r_{ij})$. The radial-dependent weight function, $\omega_{DD}(r_{ij})$, is taken as the Lucy function:

$$\omega_{DD}\left(r_{ij}\right) = \left(1 + \frac{3r_{ij}}{r_{cut}}\right)\left(1 + \frac{r_{ij}}{r_{cut}}\right)^3$$

The density-dependent energy for a given particle is given by:

$$u_i^{DD}\left(\rho_i\right) = \frac{\pi r_{cut}^4}{84}\int_{\rho_0}^{\rho_i} A\left(\rho'\right)d\rho'$$

See the supporting information of *(Brennan)* or the publication by *(Moore)* for more details on the functional form.

An interpolation table is used to evaluate the density-dependent energy ($\int A(\rho')d\rho'$) and force ($A(\rho')$). Note that the prefactor to the energy is computed after the interpolation, thus the $\int A(\rho')d\rho'$ will have units of energy / length^4.

The interpolation table is created as a pre-computation by fitting cubic splines to the file values and interpolating the density-dependent energy and force at each of $N$ densities. During a simulation, the tables are used to interpolate the density-dependent energy and force as needed for each pair of particles separated by a distance $R$. The interpolation is done in one of 2 styles: *lookup* and *linear*.

For the *lookup* style, the density is used to find the nearest table entry, which is the density-dependent energy and force.

For the *linear* style, the density is used to find the 2 surrounding table values from which the density-dependent energy and force are computed by linear interpolation.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above.

- filename
- keyword
- species1
- species2
- cutoff (distance units)

The filename specifies a file containing the tabulated density-dependent energy and force. The keyword specifies a section of the file. The cutoff is an optional coefficient. If not specified, the outer cutoff in the table itself (see below)

will be used to build an interpolation table that extend to the largest tabulated distance. If specified, only file values up to the cutoff are used to create the interpolation table. The format of this file is described below.

The species tags define the site-site interaction potential between two species contained within two different particles. The species tags must either correspond to the species defined in the reaction kinetics files specified with the *fix rx* command or they must correspond to the tag "1fluid", signifying interaction with a product species mixture determined through a one-fluid approximation. The interaction potential is weighted by the geometric average of either the mole fraction concentrations or the number of molecules associated with the interacting coarse-grained particles (see the *fractional* or *molecular* weighting pair style options). The coarse-grained potential is stored before and after the reaction kinetics solver is applied, where the difference is defined to be the internal chemical energy (uChem).

---

The format of a tabulated file is a series of one or more sections, defined as follows (without the parenthesized comments):

```
# Density-dependent function (one or more comment or blank lines)

DD-FUNCTION                (keyword is first text on line)
N 500 R 1.0 10.0           (N, R, RSQ parameters)
                           (blank)
1 1.0 25.5 102.34          (index, density, energy/r^4, force)
2 1.02 23.4 98.5
...
500 10.0 0.001 0.003
```

A section begins with a non-blank line whose first character is not a "#"; blank lines or lines starting with "#" can be used as comments between sections. The first line begins with a keyword which identifies the section. The line can contain additional text, but the initial text must match the argument specified in the pair_coeff command. The next line lists (in any order) one or more parameters for the table. Each parameter is a keyword followed by one or more numeric values.

The parameter "N" is required and its value is the number of table entries that follow. Note that this may be different than the *N* specified in the *pair_style multi/lucy/rx* command. Let Ntable = *N* in the pair_style command, and Nfile = "N" in the tabulated file. What LAMMPS does is a preliminary interpolation by creating splines using the Nfile tabulated values as nodal points. It uses these to interpolate the density-dependent energy and force at Ntable different points. The resulting tables of length Ntable are then used as described above, when computing the density-dependent energy and force. This means that if you want the interpolation tables of length Ntable to match exactly what is in the tabulated file (with effectively no preliminary interpolation), you should set Ntable = Nfile, and use the "RSQ" parameter. This is because the internal table abscissa is always RSQ (separation distance squared), for efficient lookup.

All other parameters are optional. If "R" or "RSQ" does not appear, then the distances in each line of the table are used as-is to perform spline interpolation. In this case, the table values can be spaced in *density* uniformly or however you wish to position table values in regions of large gradients.

If used, the parameters "R" or "RSQ" are followed by 2 values *rlo* and *rhi*. If specified, the density associated with each density-dependent energy and force value is computed from these 2 values (at high accuracy), rather than using the (low-accuracy) value listed in each line of the table. The density values in the table file are ignored in this case. For "R", distances uniformly spaced between *rlo* and *rhi* are computed; for "RSQ", squared distances uniformly spaced between *rlo*rlo* and *rhi*rhi* are computed.

**Note**

If you use "R" or "RSQ", the tabulated distance values in the file are effectively ignored, and replaced by new values as described in the previous paragraph. If the density value in the table is not very close to the new value (i.e. round-off difference), then you will be assigning density-dependent energy and force values to a different density, which is probably not what you want. LAMMPS will warn if this is occurring.

Following a blank line, the next N lines list the tabulated values. On each line, the first value is the index from 1 to N, the second value is r (in density units), the third value is the density-dependent function value (in energy units / length^4), and the fourth is the force (in force units). The density values must increase from one line to the next.

Note that one file can contain many sections, each with a tabulated potential. LAMMPS reads the file section by section until it finds one that matches the specified keyword.

---

### 4.211.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

The *pair_modify* shift, table, and tail options are not relevant for this pair style.

This pair style writes the settings for the "pair_style multi/lucy/rx" command to *binary restart files*, so a pair_style command does not need to specified in an input script that reads a restart file. However, the coefficient information is not stored in the restart file, since it is tabulated in the potential files. Thus, pair_coeff commands do need to be specified in the restart input script.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.211.5 Restrictions

This command is part of the DPD-REACT package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.211.6 Related commands

*pair_coeff*

### 4.211.7 Default

fractional weighting

---

**(Warren)** Warren, Phys Rev E, 68, 066702 (2003).

**(Brennan)** Brennan, J Chem Phys Lett, 5, 2144-2149 (2014).

**(Moore)** Moore, J Chem Phys, 144, 104501 (2016).

## 4.212 pair_style nb3b/harmonic command

Accelerator Variants: *nb3b/harmonic/omp*

## 4.213 pair_style nb3b/screened command

### 4.213.1 Syntax

```
pair_style style
```

- style = *nb3b/harmonic* or *nb3b/screened*

### 4.213.2 Examples

```
pair_style nb3b/harmonic
pair_coeff * * MgOH.nb3bharmonic Mg O H

pair_style nb3b/screened
pair_coeff * * PO.nb3b.screened P NULL O
pair_coeff * * SiOH.nb3b.screened Si O H
```

### 4.213.3 Description

The pair style *nb3b/harmonic* computes a non-bonded 3-body harmonic potential for the energy E of a system of atoms as

$$E = K(\theta - \theta_0)^2$$

where $\theta_0$ is the equilibrium value of the angle and $K$ is a prefactor. Note that the usual 1/2 factor is included in $K$. The form of the potential is identical to that used in angle_style *harmonic*, but in this case, the atoms do not need to be explicitly bonded.

Style *nb3b/screened* adds an additional exponentially decaying factor to the harmonic term, given by

$$E = K(\theta - \theta_0)^2 \exp\left(-\frac{r_{ij}}{\rho_{ij}} - \frac{r_{ik}}{\rho_{ik}}\right)$$

where $\rho_i j$ and $\rho_i k$ are the screening factors along the two bonds. Note that the usual 1/2 factor is included in $K$.

Only a single pair_coeff command is used with these styles which specifies a potential file with parameters for specified elements. These are mapped to LAMMPS atom types by specifying N additional arguments after the filename in the pair_coeff command, where N is the number of LAMMPS atom types:

- filename
- N element names = mapping of elements to atom types

See the *pair_coeff* page for alternate ways to specify the path for the potential file.

As an example, imagine a file SiC.nb3b.harmonic has potential values for Si and C. If your LAMMPS simulation has 4 atoms types and you want the first 3 to be Si, and the fourth to be C, you would use the following pair_coeff command:

```
pair_coeff * * SiC.nb3b.harmonic Si Si Si C
```

The first 2 arguments must be * * so as to span all LAMMPS atom types. The first three Si arguments map LAMMPS atom types 1,2,3 to the Si element in the potential file. The final C argument maps LAMMPS atom type 4 to the C

element in the potential file. If a mapping value is specified as NULL, the mapping is not performed. This can be used when the potential is used as part of the *hybrid* pair style. The NULL values are placeholders for atom types that will be used with other potentials. Two examples of pair_coeff command for use with the *hybrid* pair style are:

```
pair_coeff * * nb3b/harmonic MgOH.nb3b.harmonic Mg O H
```

Three-body non-bonded harmonic files in the *potentials* directory of the LAMMPS distribution have a ".nb3b.harmonic" suffix. Lines that are not blank or comments (starting with #) define parameters for a triplet of elements.

Each entry has six arguments. The first three are atom types as referenced in the LAMMPS input file. The first argument specifies the central atom. The fourth argument indicates the $K$ parameter. The fifth argument indicates $\theta_0$. The sixth argument indicates a separation cutoff in Angstroms.

For a given entry, if the second and third arguments are identical, then the entry is for a cutoff for the distance between types 1 and 2 (values for $K$ and $\theta_0$ are irrelevant in this case).

For a given entry, if the first three arguments are all different, then the entry is for the $K$ and $\theta_0$ parameters (the cutoff in this case is irrelevant).

It is required that the potential file contains entries for *all* permutations of the elements listed in the pair_coeff command. If certain combinations are not parameterized the corresponding parameters should be set to zero. The potential file can also contain entries for additional elements which are not used in a particular simulation; LAMMPS ignores those entries.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.213.4 Restrictions

This pair style can only be used if LAMMPS was built with the MANYBODY package. See the *Build package* page for more info.

### 4.213.5 Related commands

*pair_coeff*

### 4.213.6 Default

none

## 4.214 pair_style nm/cut command

Accelerator Variants: *nm/cut/kk*, *nm/cut/omp*

## 4.215 pair_style nm/cut/split command

Accelerator Variants: *nm/cut/split/omp*

## 4.216 pair_style nm/cut/coul/cut command

Accelerator Variants: *nm/cut/coul/cut/kk*, *nm/cut/coul/cut/omp*

## 4.217 pair_style nm/cut/coul/long command

Accelerator Variants: *nm/cut/coul/long/kk*, *nm/cut/coul/long/omp*

### 4.217.1 Syntax

```
pair_style style args
```

- style = *nm/cut* or *nm/cut/split* or *nm/cut/coul/cut* or *nm/cut/coul/long*
- args = list of arguments for a particular style

  *nm/cut* args = cutoff
    cutoff = global cutoff for Pair interactions (distance units)
  *nm/cut/split* args = cutoff
    cutoff = global cutoff for Pair interactions (distance units)
  *nm/cut/coul/cut* args = cutoff (cutoff2)
    cutoff = global cutoff for Pair (and Coulombic if only 1 arg) (distance units)
    cutoff2 = global cutoff for Coulombic (optional) (distance units)
  *nm/cut/coul/long* args = cutoff (cutoff2)
    cutoff = global cutoff for Pair (and Coulombic if only 1 arg) (distance units)
    cutoff2 = global cutoff for Coulombic (optional) (distance units)

### 4.217.2 Examples

```
pair_style nm/cut 12.0
pair_coeff * * 0.01 5.4 8.0 7.0
pair_coeff 1 1 0.01 4.4 7.0 6.0

pair_style nm/cut/split 1.12246
pair_coeff 1 1 1.0 1.1246 12 6
pair_coeff * * 1.0 1.1246 11 6

pair_style nm/cut/coul/cut 12.0 15.0
pair_coeff * * 0.01 5.4 8.0 7.0
pair_coeff 1 1 0.01 4.4 7.0 6.0

pair_style nm/cut/coul/long 12.0 15.0
pair_coeff * * 0.01 5.4 8.0 7.0
pair_coeff 1 1 0.01 4.4 7.0 6.0
```

### 4.217.3 Description

Style *nm* computes site-site interactions based on the N-M potential by *Clarke*, mainly used for ionic liquids. A site can represent a single atom or a united-atom site. The energy of an interaction has the following form:

$$E = \frac{E_0}{(n-m)} \left[ m \left( \frac{r_0}{r} \right)^n - n \left( \frac{r_0}{r} \right)^m \right] \qquad r < r_c$$

where $r_c$ is the cutoff and $r_0$ is the minimum of the potential. Please note that this differs from the convention used for other Lennard-Jones potentials in LAMMPS where $\sigma$ represents the location where the energy is zero.

Style *nm/cut/split* applies the standard LJ (12-6) potential above $r_0 = 2^{\frac{1}{6}}\sigma$. Style *nm/cut/split* is employed in polymer equilibration protocols that combine core-softening approaches with topology-changing moves *Dietz*.

Style *nm/cut/coul/cut* adds a Coulombic pairwise interaction given by

$$E = \frac{C q_i q_j}{\varepsilon r} \qquad r < r_c$$

where $C$ is an energy-conversion constant, $q_i$ and $q_j$ are the charges on the two atoms, and epsilon is the dielectric constant which can be set by the *dielectric* command. If one cutoff is specified in the pair_style command, it is used for both the N-M and Coulombic terms. If two cutoffs are specified, they are used as cutoffs for the N-M and Coulombic terms respectively.

Styles *nm/cut/coul/long* compute the same Coulombic interactions as style *nm/cut/coul/cut* except that an additional damping factor is applied to the Coulombic term so it can be used in conjunction with the *kspace_style* command and its *ewald* or *pppm* option. The Coulombic cutoff specified for this style means that pairwise interactions within this distance are computed directly; interactions outside that distance are computed in reciprocal space.

For all of the *nm* pair styles, the following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands.

- $E_0$ (energy units)
- $r_0$ (distance units)
- $n$ (unitless)
- $m$ (unitless)
- cutoff1 (distance units)
- cutoff2 (distance units)

The latter 2 coefficients are optional. If not specified, the global N-M and Coulombic cutoffs specified in the pair_style command are used. If only one cutoff is specified, it is used as the cutoff for both N-M and Coulombic interactions for this type pair. If both coefficients are specified, they are used as the N-M and Coulombic cutoffs for this type pair. You cannot specify 2 cutoffs for style *nm*, since it has no Coulombic terms.

For *nm/cut/coul/long* only the N-M cutoff can be specified since a Coulombic cutoff cannot be specified for an individual I,J type pair. All type pairs use the same global Coulombic cutoff specified in the pair_style command.

---

### 4.217.4 Mixing, shift, table, tail correction, restart, rRESPA info

These pair styles do not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

All of the *nm* pair styles supports the *pair_modify* shift option for the energy of the pair interaction.

The *nm/cut/coul/long* pair styles support the *pair_modify* table option since they can tabulate the short-range portion of the long-range Coulombic interaction.

All of the *nm* pair styles support the *pair_modify* tail option for adding a long-range tail correction to the energy and pressure for the N-M portion of the pair interaction.

All of the *nm* pair styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

All of the *nm* pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 4.217.5 Restrictions

These pair styles are part of the EXTRA-PAIR package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.217.6 Related commands

*pair_coeff*, *pair style lj/cut*, *bond style fene/nm*

### 4.217.7 Default

none

---

**(Clarke)** Clarke and Smith, J Chem Phys, 84, 2290 (1986).

**(Dietz)** Dietz and Hoy, J. Chem Phys, 156, 014103 (2022).

## 4.218 pair_style none command

### 4.218.1 Syntax

```
pair_style none
```

### 4.218.2 Examples

```
pair_style none
```

### 4.218.3 Description

Using a pair style of *none* means that any previous pair style setting will be deleted and pairwise forces and energies are not computed.

As a consequence there will be a pairwise force cutoff of 0.0, which has implications for the default setting of the neighbor list and the communication cutoff. Those are the sum of the largest pairwise cutoff and the neighbor skin distance (see the documentation of the *neighbor* command and the *comm_modify* command). When you have bonds, angles, dihedrals, or impropers defined at the same time, you must set the communication cutoff so that communication cutoff distance is large enough to acquire and communicate sufficient ghost atoms from neighboring subdomains as needed for computing bonds, angles, etc.

A pair style of *none* will also not request a pairwise neighbor list. However if the *neighbor* style is *bin*, data structures for binning are still allocated. If the neighbor list cutoff is small, then these data structures can consume a large amount of memory. So you should either set the neighbor style to *nsq* or set the skin distance to a larger value.

See the *pair_style zero* for a way to set a pairwise cutoff and thus trigger the building of a neighbor lists and setting a corresponding communication cutoff, but compute no pairwise interactions.

### 4.218.4 Restrictions

You must not use a *pair_coeff* command with this pair style. Since there is no interaction computed, you cannot set any coefficients for it.

### 4.218.5 Related commands

*pair_style zero*

### 4.218.6 Default

none

## 4.219 pair_style oxdna/excv command

## 4.220 pair_style oxdna/stk command

## 4.221 pair_style oxdna/hbond command

## 4.222 pair_style oxdna/xstk command

## 4.223 pair_style oxdna/coaxstk command

### 4.223.1 Syntax

```
pair_style style1

pair_coeff * * style2 args
```

- style1 = *hybrid/overlay oxdna/excv oxdna/stk oxdna/hbond oxdna/xstk oxdna/coaxstk*
- style2 = *oxdna/excv* or *oxdna/stk* or *oxdna/hbond* or *oxdna/xstk* or *oxdna/coaxstk*
- args = list of arguments for these particular styles

```
oxdna/stk args = seq T xi kappa 6.0 0.4 0.9 0.32 0.75 1.3 0 0.8 0.9 0 0.95 0.9 0 0.95 2.
↪0 0.65 2.0 0.65
```

```
  seq = seqav (for average sequence stacking strength) or seqdep (for sequence-dependent␣
↪stacking strength)
  T = temperature (LJ units: 0.1 = 300 K, real units: 300 = 300 K)
  xi = 1.3448 (LJ units) or 8.01727944817084 (real units), temperature-independent␣
↪coefficient in stacking strength
  kappa = 2.6568 (LJ units) or 0.005279604 (real units), coefficient of linear␣
↪temperature dependence in stacking strength
oxdna/hbond args = seq eps 8.0 0.4 0.75 0.34 0.7 1.5 0 0.7 1.5 0 0.7 1.5 0 0.7 0.46 3.
↪141592653589793 0.7 4.0 1.5707963267948966 0.45 4.0 1.5707963267948966 0.45
  seq = seqav (for average sequence base-pairing strength) or seqdep (for␣
↪sequence-dependent base-pairing strength)
  eps = 1.077 (LJ units) or 6.42073911784652 (real units), average hydrogen bonding␣
↪strength between A-T and C-G Watson-Crick base pairs, 0 between all other pairs
```

### 4.223.2 Examples

```
# LJ units
pair_style hybrid/overlay oxdna/excv oxdna/stk oxdna/hbond oxdna/xstk oxdna/coaxstk
pair_coeff * * oxdna/excv    2.0 0.7 0.675 2.0 0.515 0.5 2.0 0.33 0.32
pair_coeff * * oxdna/stk     seqdep 0.1 1.3448 2.6568 6.0 0.4 0.9 0.32 0.75 1.3 0 0.8 0.
↪9 0 0.95 0.9 0 0.95 2.0 0.65 2.0 0.65
pair_coeff * * oxdna/hbond   seqdep 0.0 8.0 0.4 0.75 0.34 0.7 1.5 0 0.7 1.5 0 0.7 1.5 0␣
↪0.7 0.46 3.141592653589793 0.7 4.0 1.5707963267948966 0.45 4.0 1.5707963267948966 0.45
pair_coeff 1 4 oxdna/hbond   seqdep 1.077 8.0 0.4 0.75 0.34 0.7 1.5 0 0.7 1.5 0 0.7 1.5␣
↪0 0.7 0.46 3.141592653589793 0.7 4.0 1.5707963267948966 0.45 4.0 1.5707963267948966 0.
↪45
pair_coeff 2 3 oxdna/hbond   seqdep 1.077 8.0 0.4 0.75 0.34 0.7 1.5 0 0.7 1.5 0 0.7 1.5␣
↪0 0.7 0.46 3.141592653589793 0.7 4.0 1.5707963267948966 0.45 4.0 1.5707963267948966 0.
↪45
pair_coeff * * oxdna/xstk    47.5 0.575 0.675 0.495 0.655 2.25 0.791592653589793 0.58 1.
↪7 1.0 0.68 1.7 1.0 0.68 1.5 0 0.65 1.7 0.875 0.68 1.7 0.875 0.68
pair_coeff * * oxdna/coaxstk 46.0 0.4 0.6 0.22 0.58 2.0 2.541592653589793 0.65 1.3 0 0.8␣
↪0.9 0 0.95 0.9 0 0.95 2.0 -0.65 2.0 -0.65

pair_style hybrid/overlay oxdna/excv oxdna/stk oxdna/hbond oxdna/xstk oxdna/coaxstk
pair_coeff * * oxdna/excv    oxdna_lj.cgdna
pair_coeff * * oxdna/stk     seqav 0.1 oxdna_lj.cgdna
pair_coeff * * oxdna/hbond   seqav oxdna_lj.cgdna
pair_coeff 1 4 oxdna/hbond   seqav oxdna_lj.cgdna
pair_coeff 2 3 oxdna/hbond   seqav oxdna_lj.cgdna
pair_coeff * * oxdna/xstk    oxdna_lj.cgdna
pair_coeff * * oxdna/coaxstk oxdna_lj.cgdna

# Real units
pair_style hybrid/overlay oxdna/excv oxdna/stk oxdna/hbond oxdna/xstk oxdna/coaxstk
pair_coeff * * oxdna/excv    11.92337812042065 5.9626 5.74965 11.92337812042065 4.38677␣
↪4.259 11.92337812042065 2.81094 2.72576
pair_coeff * * oxdna/stk     seqdep 300.0 8.01727944817084 0.005279604 0.70439070204273␣
↪3.4072 7.6662 2.72576 6.3885 1.3 0.0 0.8 0.9 0.0 0.95 0.9 0.0 0.95 2.0 0.65 2.0 0.65
pair_coeff * * oxdna/hbond   seqdep 0.0 0.93918760272364 3.4072 6.3885 2.89612 5.9626 1.
↪5 0.0 0.7 1.5 0.0 0.7 1.5 0.0 0.7 0.46 3.141592654 0.7 4.0 1.570796327 0.45 4.0 1.
↪570796327 0.45
pair_coeff 1 4 oxdna/hbond   seqdep 6.42073911784652 0.93918760272364 3.4072 6.3885 2.
```

(continues on next page)

(continued from previous page)

```
↪89612 5.9626 1.5 0.0 0.7 1.5 0.0 0.7 1.5 0.0 0.7 0.46 3.141592654 0.7 4.0 1.570796327␣
↪0.45 4.0 1.570796327 0.45
pair_coeff 2 3 oxdna/hbond   seqdep 6.42073911784652 0.93918760272364 3.4072 6.3885 2.
↪89612 5.9626 1.5 0.0 0.7 1.5 0.0 0.7 1.5 0.0 0.7 0.46 3.141592654 0.7 4.0 1.570796327␣
↪0.45 4.0 1.570796327 0.45
pair_coeff * * oxdna/xstk    3.9029021145006 4.89785 5.74965 4.21641 5.57929 2.25 0.
↪791592654 0.58 1.7 1.0 0.68 1.7 1.0 0.68 1.5 0.0 0.65 1.7 0.875 0.68 1.7 0.875 0.68
pair_coeff * * oxdna/coaxstk 3.77965257404268 3.4072 5.1108 1.87396 4.94044 2.0 2.
↪541592654 0.65 1.3 0.0 0.8 0.9 0.0 0.95 0.9 0.0 0.95 2.0 -0.65 2.0 -0.65

pair_style hybrid/overlay oxdna/excv oxdna/stk oxdna/hbond oxdna/xstk oxdna/coaxstk
pair_coeff * * oxdna/excv    oxdna_real.cgdna
pair_coeff * * oxdna/stk     seqav 300.0 oxdna_real.cgdna
pair_coeff * * oxdna/hbond   seqav oxdna_real.cgdna
pair_coeff 1 4 oxdna/hbond   seqav oxdna_real.cgdna
pair_coeff 2 3 oxdna/hbond   seqav oxdna_real.cgdna
pair_coeff * * oxdna/xstk    oxdna_real.cgdna
pair_coeff * * oxdna/coaxstk oxdna_real.cgdna
```

**Note**

The coefficients in the above examples are provided in forms compatible with both *units lj* and *units real* (see documentation of *units*). These can also be read from a potential file with correct unit style by specifying the name of the file. Several potential files for each unit style are included in the `potentials` directory of the LAMMPS distribution.

### 4.223.3 Description

The *oxdna* pair styles compute the pairwise-additive parts of the oxDNA force field for coarse-grained modelling of DNA. The effective interaction between the nucleotides consists of potentials for the excluded volume interaction *oxdna/excv*, the stacking *oxdna/stk*, cross-stacking *oxdna/xstk* and coaxial stacking interaction *oxdna/coaxstk* as well as the hydrogen-bonding interaction *oxdna/hbond* between complementary pairs of nucleotides on opposite strands. Average sequence or sequence-dependent stacking and base-pairing strengths are supported *(Sulc)*.

The exact functional form of the pair styles is rather complex. The individual potentials consist of products of modulation factors, which themselves are constructed from a number of more basic potentials (Morse, Lennard-Jones, harmonic angle and distance) as well as quadratic smoothing and modulation terms. We refer to *(Ouldridge-DPhil)* and *(Ouldridge)* for a detailed description of the oxDNA force field.

**Note**

These pair styles have to be used together with the related oxDNA bond style *oxdna/fene* for the connectivity of the phosphate backbone (see also documentation of *bond_style oxdna/fene*). Most of the coefficients in the above example have to be kept fixed and cannot be changed without reparameterizing the entire model. Exceptions are the first two coefficients after *oxdna/stk* (seq=seqdep and T=0.1 and corresponding *real unit* equivalents in the above examples) and the first coefficient after *oxdna/hbond* (seq=seqdep in the above example). When using a Langevin thermostat, e.g. through *fix langevin* or *fix nve/dotc/langevin* the temperature coefficients have to be matched to the one used in the fix.

**Note**

These pair styles have to be used with the *atom_style hybrid bond ellipsoid oxdna* (see documentation of *atom_style*). The *atom_style oxdna* stores the 3'-to-5' polarity of the nucleotide strand, which is set through the bond topology in the data file. The first (second) atom in a bond definition is understood to point towards the 3'-end (5'-end) of the strand.

**Warning**

If data files are produced with *write_data*, then the *newton* command should be set to *newton on*. Otherwise the data files will not have the same 3'-to-5' polarity as the initial data file. This limitation does not apply to binary restart files produced with *write_restart*.

Example input and data files for DNA duplexes can be found in `examples/PACKAGES/cgdna/examples/lj_units/oxDNA/` or in the corresponding folder for real units. A simple python setup tool which creates single straight or helical DNA strands, DNA duplexes or arrays of DNA duplexes can be found in `examples/PACKAGES/cgdna/util/`.

---

### 4.223.4 Potential file reading

For each pair style above the first non-modifiable argument can be a filename, and if it is, no further arguments should be supplied. Therefore the following command:

```
pair_coeff 1 4 oxdna/hbond   seqav oxdna_lj.cgdna
```

will be interpreted as a request to read the corresponding hydrogen bonding potential parameters from the file with the given name. The file can define multiple potential parameters for both bonded and pair interactions, but for the example pair interaction above there must exist in the file a line of the form:

```
1 4 hbond     <coefficients>
```

If potential customization is required, the potential file reading can be mixed with the manual specification of the potential parameters. For example, the following command:

```
pair_style hybrid/overlay oxdna/excv oxdna/stk oxdna/hbond oxdna/xstk oxdna/coaxstk
pair_coeff * * oxdna/excv    oxdna_lj.cgdna
pair_coeff * * oxdna/stk     seqav 0.1 1.3448 2.6568 6.0 0.4 0.9 0.32 0.75 1.3 0 0.8 0.9␣
↪0 0.95 0.9 0 0.95 2.0 0.65 2.0 0.65
pair_coeff * * oxdna/hbond   seqav oxdna_lj.cgdna
pair_coeff 1 4 oxdna/hbond   seqav oxdna_lj.cgdna
pair_coeff 2 3 oxdna/hbond   seqav oxdna_lj.cgdna
pair_coeff * * oxdna/xstk    oxdna_lj.cgdna
pair_coeff * * oxdna/coaxstk 46.0 0.4 0.6 0.22 0.58 2.0 2.541592653589793 0.65 1.3 0 0.8␣
↪0.9 0 0.95 0.9 0 0.95 2.0 -0.65 2.0 -0.65
```

will read the stacking and coaxial stacking potential parameters from the manual specification and all others from the potential file *oxdna_lj.cgdna*.

There are sample potential files for each unit style in the `potentials` directory of the LAMMPS distribution. The potential file unit system must align with the units defined via the *units* command. For conversion between different *LJ* and *real* unit systems for oxDNA, the python tool *lj2real.py* located in the `examples/PACKAGES/cgdna/util/`

directory can be used. This tool assumes similar file structure to the examples found in `examples/PACKAGES/cgdna/examples/`.

---

### 4.223.5 Unique base pairing

Unique base pairing describes the restriction on the specific complementary nucleotide with which a particular base can pair. This can be used to prevent asymmetric base pairs or to simplify the free energy landscape. With unique base pairing enabled base pairs can only form between complementary nucleotides with specific atom IDs. This functionality draws on *fix property/atom* and a modified *read_data* command.

To use unique base pairing, the data file of a system with N nucleotides must contain a section like

```
Basepairs # i_idc

1 idc1
2 idc2
3 idc3
4 idc4
...
N idcN
```

where idc is the non-negative atom ID of a complementary nucleotide that binds uniquely to the preceding atom ID.

Unique base pairing can be combined with normal base pairing by setting a zero or negative value for idc. For instance, in a 4-mer with 8 nucleotides consisting of a ssDNA strand 3'-A-A-A-A-5' with atom IDs 3'-1-2-3-4-5' and a complementary strand 5'-T-T-T-T-3' with atom IDs 5'-8-7-6-5-3' set up as

```
Basepairs # i_idc

1 8
2 -1
3 -1
4 5
5 4
6 -1
7 -1
8 1
```

the A nucleotide with ID 1 can only hybridize with the T nucleotide with ID 8 and the A nucleotide with ID 4 can only hybridize with the T nucleotide with ID 5, whereas the A nucleotides with ID 2 and 3 can hybridize with either T nucleotide with ID 6 and 7.

The input file requires an instance of the *fix property/atom* and a *read_data* command as follows:

```
fix Basepairs all property/atom i_idc ghost yes
read_data file fix Basepairs NULL Basepairs
```

where *file* is the name of the data file and the only modifiable argument. An example input and data file for a dsDNA ring can be found in `examples/PACKAGES/cgdna/examples/lj_units/oxDNA3/unique_bp` or in the corresponding folder for real units.

Please cite *(Henrich)* in any publication that uses this implementation. An updated documentation that contains general information on the model, its implementation and performance as well as the structure of the data and input file can be found here.

Please cite also the relevant oxDNA publications *(Ouldridge)*, *(Ouldridge-DPhil)* and *(Sulc)*.

### 4.223.6 Restrictions

These pair styles can only be used if LAMMPS was built with the CG-DNA package and the MOLECULE and ASPHERE package. See the *Build package* page for more info.

### 4.223.7 Related commands

*bond_style oxdna/fene*, *pair_coeff*, *bond_style oxdna2/fene*, *pair_style oxdna2/excv*, *bond_style oxrna2/fene*, *pair_style oxrna2/excv*, *atom_style oxdna*, *fix nve/dotc/langevin*

### 4.223.8 Default

none

---

**(Henrich)** O. Henrich, Y. A. Gutierrez-Fosado, T. Curk, T. E. Ouldridge, Eur. Phys. J. E 41, 57 (2018).

**(Ouldridge-DPhil)** T.E. Ouldridge, Coarse-grained modelling of DNA and DNA self-assembly, DPhil. University of Oxford (2011).

**(Ouldridge)** T.E. Ouldridge, A.A. Louis, J.P.K. Doye, J. Chem. Phys. 134, 085101 (2011).

**(Sulc)** P. Sulc, F. Romano, T.E. Ouldridge, L. Rovigatti, J.P.K. Doye, A.A. Louis, J. Chem. Phys. 137, 135101 (2012).

## 4.224 pair_style oxdna2/excv command

## 4.225 pair_style oxdna2/stk command

## 4.226 pair_style oxdna2/hbond command

## 4.227 pair_style oxdna2/xstk command

## 4.228 pair_style oxdna2/coaxstk command

## 4.229 pair_style oxdna2/dh command

### 4.229.1 Syntax

```
pair_style style1

pair_coeff * * style2 args [keyword value]
```

- style1 = *hybrid/overlay oxdna2/excv oxdna2/stk oxdna2/hbond oxdna2/xstk oxdna2/coaxstk oxdna2/dh*
- style2 = *oxdna2/excv* or *oxdna2/stk* or *oxdna2/hbond* or *oxdna2/xstk* or *oxdna2/coaxstk* or *oxdna2/dh*
- args = list of arguments for these particular styles
- zero or one keyword/value pair may be appended to *oxdna2/dh*
- keyword = *half_charged_ends*

```
oxdna2/stk args = seq T xi kappa 6.0 0.4 0.9 0.32 0.75 1.3 0 0.8 0.9 0 0.95 0.9 0 0.95 2.
↪0 0.65 2.0 0.65
  seq = seqav (for average sequence stacking strength) or seqdep (for sequence-dependent␣
↪stacking strength)
  T = temperature (LJ units: 0.1 = 300 K, real units: 300 = 300 K)
  xi = 1.3523 (LJ units) or 8.06199211612242 (real units), temperature-independent␣
↪coefficient in stacking strength
  kappa = 2.6717 (LJ units) or 0.005309213 (real units), coefficient of linear␣
↪temperature dependence in stacking strength
oxdna2/hbond args = seq eps 8.0 0.4 0.75 0.34 0.7 1.5 0 0.7 1.5 0 0.7 1.5 0 0.7 0.46 3.
↪141592653589793 0.7 4.0 1.5707963267948966 0.45 4.0 1.5707963267948966 0.45
  seq = seqav (for average sequence base-pairing strength) or seqdep (for␣
↪sequence-dependent base-pairing strength)
  eps = 1.0678 (LJ units) or 6.36589157849259 (real units), average hydrogen bonding␣
↪strength between A-T and C-G Watson-Crick base pairs, 0 between all other pairs
oxdna2/dh args [keyword value] = T rhos qeff [half_charged_ends no|yes]
  T = temperature (LJ units: 0.1 = 300 K, real units: 300 = 300 K)
  rhos = salt concentration (mole per litre)
  qeff = 0.815 (effective charge in elementary charges)
  half_charged_ends yes = set half charge at terminal nucleotides
  half_charged_ends no  = set full charge at terminal nucleotides
```

### 4.229.2 Examples

```
# LJ units
pair_style hybrid/overlay oxdna2/excv oxdna2/stk oxdna2/hbond oxdna2/xstk oxdna2/coaxstk␣
↪oxdna2/dh
pair_coeff * * oxdna2/excv    2.0 0.7 0.675 2.0 0.515 0.5 2.0 0.33 0.32
pair_coeff * * oxdna2/stk     seqdep 0.1 1.3523 2.6717 6.0 0.4 0.9 0.32 0.75 1.3 0 0.8 0.
↪9 0 0.95 0.9 0 0.95 2.0 0.65 2.0 0.65
pair_coeff * * oxdna2/hbond   seqdep 0.0 8.0 0.4 0.75 0.34 0.7 1.5 0 0.7 1.5 0 0.7 1.5 0␣
↪0.7 0.46 3.141592653589793 0.7 4.0 1.5707963267948966 0.45 4.0 1.5707963267948966 0.45
pair_coeff 1 4 oxdna2/hbond   seqdep 1.0678 8.0 0.4 0.75 0.34 0.7 1.5 0 0.7 1.5 0 0.7 1.
↪5 0 0.7 0.46 3.141592653589793 0.7 4.0 1.5707963267948966 0.45 4.0 1.5707963267948966␣
↪0.45
pair_coeff 2 3 oxdna2/hbond   seqdep 1.0678 8.0 0.4 0.75 0.34 0.7 1.5 0 0.7 1.5 0 0.7 1.
↪5 0 0.7 0.46 3.141592653589793 0.7 4.0 1.5707963267948966 0.45 4.0 1.5707963267948966␣
↪0.45
pair_coeff * * oxdna2/xstk    47.5 0.575 0.675 0.495 0.655 2.25 0.791592653589793 0.58 1.
↪7 1.0 0.68 1.7 1.0 0.68 1.5 0 0.65 1.7 0.875 0.68 1.7 0.875 0.68
pair_coeff * * oxdna2/coaxstk 58.5 0.4 0.6 0.22 0.58 2.0 2.891592653589793 0.65 1.3 0 0.
↪8 0.9 0 0.95 0.9 0 0.95 40.0 3.116592653589793
pair_coeff * * oxdna2/dh      0.1 0.5 0.815

pair_style hybrid/overlay oxdna2/excv oxdna2/stk oxdna2/hbond oxdna2/xstk oxdna2/coaxstk␣
↪oxdna2/dh
pair_coeff * * oxdna2/excv    oxdna2_lj.cgdna
pair_coeff * * oxdna2/stk     seqdep 0.1 oxdna2_lj.cgdna
pair_coeff * * oxdna2/hbond   seqdep oxdna2_lj.cgdna
pair_coeff 1 4 oxdna2/hbond   seqdep oxdna2_lj.cgdna
pair_coeff 2 3 oxdna2/hbond   seqdep oxdna2_lj.cgdna
pair_coeff * * oxdna2/xstk    oxdna2_lj.cgdna
pair_coeff * * oxdna2/coaxstk oxdna2_lj.cgdna
```

(continues on next page)

(continued from previous page)

```
pair_coeff * * oxdna2/dh      0.1 0.5 oxdna2_lj.cgdna

# Real units
pair_style hybrid/overlay oxdna2/excv oxdna2/stk oxdna2/hbond oxdna2/xstk oxdna2/coaxstk␣
↪oxdna2/dh
pair_coeff * * oxdna2/excv    11.92337812042065 5.9626 5.74965 11.92337812042065 4.38677␣
↪4.259 11.92337812042065 2.81094 2.72576
pair_coeff * * oxdna2/stk     seqdep 300.0 8.06199211612242 0.005309213 0.70439070204273␣
↪3.4072 7.6662 2.72576 6.3885 1.3 0.0 0.8 0.9 0.0 0.95 0.9 0.0 0.95 2.0 0.65 2.0 0.65
pair_coeff * * oxdna2/hbond   seqdep 0.0 0.93918760272364 3.4072 6.3885 2.89612 5.9626 1.
↪5 0.0 0.7 1.5 0.0 0.7 1.5 0.0 0.7 0.46 3.141592654 0.7 4.0 1.570796327 0.45 4.0 1.
↪570796327 0.45
pair_coeff 1 4 oxdna2/hbond   seqdep 6.36589157849259 0.93918760272364 3.4072 6.3885 2.
↪89612 5.9626 1.5 0.0 0.7 1.5 0.0 0.7 1.5 0.0 0.7 0.46 3.141592654 0.7 4.0 1.570796327␣
↪0.45 4.0 1.570796327 0.45
pair_coeff 2 3 oxdna2/hbond   seqdep 6.36589157849259 0.93918760272364 3.4072 6.3885 2.
↪89612 5.9626 1.5 0.0 0.7 1.5 0.0 0.7 1.5 0.0 0.7 0.46 3.141592654 0.7 4.0 1.570796327␣
↪0.45 4.0 1.570796327 0.45
pair_coeff * * oxdna2/xstk    3.9029021145006 4.89785 5.74965 4.21641 5.57929 2.25 0.
↪791592654 0.58 1.7 1.0 0.68 1.7 1.0 0.68 1.5 0.0 0.65 1.7 0.875 0.68 1.7 0.875 0.68
pair_coeff * * oxdna2/coaxstk 4.80673207785863 3.4072 5.1108 1.87396 4.94044 2.0 2.
↪891592653589793 0.65 1.3 0.0 0.8 0.9 0.0 0.95 0.9 0.0 0.95 40.0 3.116592653589793
pair_coeff * * oxdna2/dh      300.0 0.5 0.815

pair_style hybrid/overlay oxdna2/excv oxdna2/stk oxdna2/hbond oxdna2/xstk oxdna2/coaxstk␣
↪oxdna2/dh
pair_coeff * * oxdna2/excv    oxdna2_real.cgdna
pair_coeff * * oxdna2/stk     seqdep 300.0 oxdna2_real.cgdna
pair_coeff * * oxdna2/hbond   seqdep oxdna2_real.cgdna
pair_coeff 1 4 oxdna2/hbond   seqdep oxdna2_real.cgdna
pair_coeff 2 3 oxdna2/hbond   seqdep oxdna2_real.cgdna
pair_coeff * * oxdna2/xstk    oxdna2_real.cgdna
pair_coeff * * oxdna2/coaxstk oxdna2_real.cgdna
pair_coeff * * oxdna2/dh      300.0 0.5 oxdna2_real.cgdna
```

**Note**

The coefficients in the above examples are provided in forms compatible with both *units lj* and *units real* (see documentation of *units*). These can also be read from a potential file with correct unit style by specifying the name of the file. Several potential files for each unit style are included in the `potentials` directory of the LAMMPS distribution.

### 4.229.3 Description

Added in version 30Mar2026.

The *oxdna2* pair styles compute the pairwise-additive parts of the oxDNA force field for coarse-grained modelling of DNA. The effective interaction between the nucleotides consists of potentials for the excluded volume interaction *oxdna2/excv*, the stacking *oxdna2/stk*, cross-stacking *oxdna2/xstk* and coaxial stacking interaction *oxdna2/coaxstk*, electrostatic Debye-Hueckel interaction *oxdna2/dh* as well as the hydrogen-bonding interaction *oxdna2/hbond* between complementary pairs of nucleotides on opposite strands. Average sequence or sequence-dependent stacking and base-pairing strengths are supported *(Sulc)*.

The exact functional form of the pair styles is rather complex. The individual potentials consist of products of modulation factors, which themselves are constructed from a number of more basic potentials (Morse, Lennard-Jones, harmonic angle and distance) as well as quadratic smoothing and modulation terms. We refer to *(Snodin)* and the original oxDNA publications *(Ouldridge-DPhil)* and *(Ouldridge)* for a detailed description of the oxDNA2 force field.

**Note**

These pair styles have to be used together with the related oxDNA2 bond style *oxdna2/fene* for the connectivity of the phosphate backbone (see also documentation of *bond_style oxdna2/fene*). Most of the coefficients in the above example have to be kept fixed and cannot be changed without reparameterizing the entire model. Exceptions are the first two coefficients after *oxdna2/stk* (seq=seqdep and T=0.1 and corresponding *real unit* equivalents in the above examples), the first coefficient after *oxdna2/hbond* (seq=seqdep in the above example) and the two coefficients after *oxdna2/dh* (T=0.1 and rhos=0.5 in the above example). *oxdna2/dh* has the option to set half a charge at terminal nucleotides (half_charged_ends yes) to aid coaxial stacking. When using a Langevin thermostat e.g. through *fix langevin* or *fix nve/dotc/langevin* the temperature coefficients have to be matched to the one used in the fix.

**Note**

These pair styles have to be used with the *atom_style hybrid bond ellipsoid oxdna* (see documentation of *atom_style*). The *atom_style oxdna* stores the 3'-to-5' polarity of the nucleotide strand, which is set through the bond topology in the data file. The first (second) atom in a bond definition is understood to point towards the 3'-end (5'-end) of the strand.

**Warning**

If data files are produced with *write_data*, then the *newton* command should be set to *newton on*. Otherwise the data files will not have the same 3'-to-5' polarity as the initial data file. This limitation does not apply to binary restart files produced with *write_restart*.

Example input and data files for DNA duplexes can be found in `examples/PACKAGES/cgdna/examples/lj_units/oxDNA2/` or in the corresponding folder for real units. A simple python setup tool which creates single straight or helical DNA strands, DNA duplexes or arrays of DNA duplexes can be found in `examples/PACKAGES/cgdna/util/`.

Please cite *(Henrich)* in any publication that uses this implementation. An updated documentation that contains general information on the model, its implementation and performance as well as the structure of the data and input file can be found here.

Please cite also the relevant oxDNA2 publications *(Snodin)* and *(Sulc)*.

---

### 4.229.4 Potential file reading

For each pair style above the first non-modifiable argument can be a filename (with exception of Debye-Hueckel, for which the effective charge argument can be a filename), and if it is, no further arguments should be supplied. Therefore the following command:

```
pair_coeff 1 4 oxdna2/hbond   seqdep oxdna_real.cgdna
```

will be interpreted as a request to read the corresponding hydrogen bonding potential parameters from the file with the given name. The file can define multiple potential parameters for both bonded and pair interactions, but for the example pair interaction above there must exist in the file a line of the form:

```
1 4 hbond     <coefficients>
```

If potential customization is required, the potential file reading can be mixed with the manual specification of the potential parameters. For example, the following command:

```
pair_style hybrid/overlay oxdna2/excv oxdna2/stk oxdna2/hbond oxdna2/xstk oxdna2/coaxstk␣
↪oxdna2/dh
pair_coeff * * oxdna2/excv    2.0 0.7 0.675 2.0 0.515 0.5 2.0 0.33 0.32
pair_coeff * * oxdna2/stk     seqdep 0.1 oxdna2_lj.cgdna
pair_coeff * * oxdna2/hbond   seqdep oxdna2_lj.cgdna
pair_coeff 1 4 oxdna2/hbond   seqdep oxdna2_lj.cgdna
pair_coeff 2 3 oxdna2/hbond   seqdep oxdna2_lj.cgdna
pair_coeff * * oxdna2/xstk    oxdna2_lj.cgdna
pair_coeff * * oxdna2/coaxstk oxdna2_lj.cgdna
pair_coeff * * oxdna2/dh      0.1 0.5 0.815
```

will read the excluded volume and Debye-Hueckel effective charge *qeff* parameters from the manual specification and all others from the potential file *oxdna2_lj.cgdna*.

There are sample potential files for each unit style in the `potentials` directory of the LAMMPS distribution. The potential file unit system must align with the units defined via the *units* command. For conversion between different *LJ* and *real* unit systems for oxDNA, the python tool *lj2real.py* located in the `examples/PACKAGES/cgdna/util/` directory can be used. This tool assumes similar file structure to the examples found in `examples/PACKAGES/cgdna/examples/`.

---

### 4.229.5 Unique base pairing

Unique base pairing describes the restriction on the specific complementary nucleotide with which a particular base can pair. This can be used to prevent asymmetric base pairs or to simplify the free energy landscape. With unique base pairing enabled base pairs can only form between complementary nucleotides with specific atom IDs. This functionality draws on *fix property/atom* and a modified *read_data* command.

To use unique base pairing, the data file of a system with N nucleotides must contain a section like

```
Basepairs # i_idc

1 idc1
2 idc2
3 idc3
4 idc4
...
N idcN
```

where idc is the non-negative atom ID of a complementary nucleotide that binds uniquely to the preceding atom ID.

Unique base pairing can be combined with normal base pairing by setting a zero or negative value for idc. For instance, in a 4-mer with 8 nucleotides consisting of a ssDNA strand 3'-A-A-A-A-5' with atom IDs 3'-1-2-3-4-5' and a complementary strand 5'-T-T-T-T-3' with atom IDs 5'-8-7-6-5-3' set up as

```
Basepairs # i_idc

1 8
2 -1
```

(continues on next page)

(continued from previous page)

```
3 -1
4 5
5 4
6 -1
7 -1
8 1
```

the A nucleotide with ID 1 can only hybridize with the T nucleotide with ID 8 and the A nucleotide with ID 4 can only hybridize with the T nucleotide with ID 5, whereas the A nucleotides with ID 2 and 3 can hybridize with either T nucleotide with ID 6 and 7.

The input file requires an instance of the *fix property/atom* and a *read_data* command as follows:

```
fix Basepairs all property/atom i_idc ghost yes
read_data file fix Basepairs NULL Basepairs
```

where *file* is the name of the data file and the only modifiable argument. An example input and data file for a dsDNA ring can be found in `examples/PACKAGES/cgdna/examples/lj_units/oxDNA3/unique_bp` or in the corresponding folder for real units.

---

### 4.229.6 Restrictions

These pair styles can only be used if LAMMPS was built with the CG-DNA package and the MOLECULE and ASPHERE package. See the *Build package* page for more info.

### 4.229.7 Related commands

*bond_style oxdna2/fene*, *pair_coeff* , *bond_style oxdna/fene*, *pair_style oxdna/excv*, *bond_style oxrna2/fene*, *pair_style oxrna2/excv*, *atom_style oxdna*, *fix nve/dotc/langevin*

### 4.229.8 Default

none

---

**(Henrich)** O. Henrich, Y. A. Gutierrez-Fosado, T. Curk, T. E. Ouldridge, Eur. Phys. J. E 41, 57 (2018).

**(Snodin)** B.E. Snodin, F. Randisi, M. Mosayebi, et al., J. Chem. Phys. 142, 234901 (2015).

**(Sulc)** P. Sulc, F. Romano, T.E. Ouldridge, L. Rovigatti, J.P.K. Doye, A.A. Louis, J. Chem. Phys. 137, 135101 (2012).

**(Ouldridge-DPhil)** T.E. Ouldridge, Coarse-grained modelling of DNA and DNA self-assembly, DPhil. University of Oxford (2011).

**(Ouldridge)** T.E. Ouldridge, A.A. Louis, J.P.K. Doye, J. Chem. Phys. 134, 085101 (2011).

## 4.230 pair_style oxdna3/excv command

## 4.231 pair_style oxdna3/stk command

## 4.232 pair_style oxdna3/hbond command

## 4.233 pair_style oxdna3/xstk command

## 4.234 pair_style oxdna3/coaxstk command

## 4.235 pair_style oxdna3/dh command

### 4.235.1 Syntax

```
pair_style style1

pair_coeff * * style2 args (keyword value)
```

- style1 = *hybrid/overlay oxdna3/excv oxdna3/stk oxdna3/hbond oxdna3/xstk oxdna3/coaxstk oxdna3/dh*
- style2 = *oxdna3/excv* or *oxdna3/stk* or *oxdna3/hbond* or *oxdna3/xstk* or *oxdna3/coaxstk* or *oxdna3/dh*
- args = list of arguments for these particular styles
- zero or one keyword/value pair may be appended to *oxdna3/dh*
- keyword = *half_charged_ends*

```
oxdna3/excv args = oxdna3_lj.cgdna or oxdna3_real.cgdna
oxdna3/stk args = T oxdna3_lj.cgdna or oxdna3_real.cgdna
  T = temperature (LJ units: 0.1 = 300 K, real units: 300 = 300 K)
oxdna3/hbond args = oxdna3_lj.cgdna or oxdna3_real.cgdna
oxdna3/xstk args = oxdna3_lj.cgdna or oxdna3_real.cgdna
oxdna3/coaxstk args = oxdna3_lj.cgdna or oxdna3_real.cgdna
oxdna3/dh args [keyword value] = T rhos oxdna3_lj.cgdna or oxdna3_real.cgdna [half_
↪charged_ends no|yes]
  T = temperature (LJ units: 0.1 = 300 K, real units: 300 = 300 K)
  rhos = salt concentration (mole per litre)
  half_charged_ends yes = set half charge at terminal nucleotides
  half_charged_ends no  = set full charge at terminal nucleotides
```

### 4.235.2 Examples

```
# LJ units
pair_style hybrid/overlay oxdna3/excv oxdna3/stk oxdna3/hbond oxdna3/xstk oxdna3/coaxstk␣
↪oxdna3/dh
pair_coeff * * oxdna3/excv     oxdna3_lj.cgdna
pair_coeff * * oxdna3/stk      0.1 oxdna3_lj.cgdna
pair_coeff * * oxdna3/hbond    oxdna3_lj.cgdna
pair_coeff 1 4 oxdna3/hbond    oxdna3_lj.cgdna
pair_coeff 2 3 oxdna3/hbond    oxdna3_lj.cgdna
pair_coeff * * oxdna3/xstk     oxdna3_lj.cgdna
pair_coeff * * oxdna3/coaxstk  oxdna3_lj.cgdna
pair_coeff * * oxdna3/dh       0.1 0.2 oxdna3_lj.cgdna
```

(continues on next page)

(continued from previous page)

```
# Real units
pair_style hybrid/overlay oxdna3/excv oxdna3/stk oxdna3/hbond oxdna3/xstk oxdna3/coaxstk␣
↪oxdna3/dh
pair_coeff * * oxdna3/excv     oxdna3_real.cgdna
pair_coeff * * oxdna3/stk      300.0 oxdna3_real.cgdna
pair_coeff * * oxdna3/hbond    oxdna3_real.cgdna
pair_coeff 1 4 oxdna3/hbond    oxdna3_real.cgdna
pair_coeff 2 3 oxdna3/hbond    oxdna3_real.cgdna
pair_coeff * * oxdna3/xstk     oxdna3_real.cgdna
pair_coeff * * oxdna3/coaxstk  oxdna3_real.cgdna
pair_coeff * * oxdna3/dh       300.0 0.2 oxdna3_real.cgdna
```

**Note**

The coefficients are provided in forms compatible with both *units lj* and *units real*. The potential file unit system must align with the units defined via the *units* command. In case of oxDNA3 almost all coefficients have to be read from a potential file with correct unit style by specifying the name of the file. The potential files for each unit style are included in the `potentials` directory of the LAMMPS distribution.

### 4.235.3 Description

Added in version 30Mar2026.

The *oxdna3* pair styles compute the pairwise-additive parts of the oxDNA force field for coarse-grained modelling of DNA. The effective interaction between the nucleotides consists of potentials for the excluded volume interaction *oxdna3/excv*, the stacking *oxdna3/stk*, cross-stacking *oxdna3/xstk* and coaxial stacking interaction *oxdna3/coaxstk*, electrostatic Debye-Hueckel interaction *oxdna3/dh* as well as the hydrogen-bonding interaction *oxdna3/hbond* between complementary pairs of nucleotides on opposite strands.

The exact functional form of the pair styles is rather complex. The individual potentials consist of products of modulation factors, which themselves are constructed from a number of more basic potentials (Morse, Lennard-Jones, harmonic angle and distance) as well as quadratic smoothing and modulation terms. We refer to *(Bonato)* and the original oxDNA publications *(Ouldridge-DPhil)* and *(Ouldridge)* for a detailed description of the oxDNA3 force field.

**Note**

These pair styles have to be used together with the related oxDNA3 bond style *oxdna3/fene* for the connectivity of the phosphate backbone (see also documentation of *bond_style oxdna3/fene*). All coefficients in the above mentioned potential files have to be kept fixed and cannot be changed without reparameterizing the entire model. The first coefficient after *oxdna3/stk* (T=0.1 and corresponding *real unit* equivalents in the above examples) and the two coefficients after *oxdna3/dh* (T=0.1 and rhos=0.2 in the above example) have to be set to the temperature and salt concentration of the system. *oxdna3/dh* has the option to set half a charge at terminal nucleotides (half_charged_ends yes) to aid coaxial stacking. When using a Langevin thermostat e.g. through *fix langevin* or *fix nve/dotc/langevin* the temperature coefficients have to be matched to the one used in the fix.

**Note**

These pair styles have to be used with the *atom_style hybrid bond ellipsoid oxdna* (see documentation of *atom_style*). The *atom_style oxdna* stores the 3'-to-5' polarity of the nucleotide strand, which is set through the bond topology

in the data file. The first (second) atom in a bond definition is understood to point towards the 3'-end (5'-end) of the strand.

**Warning**

If data files are produced with *write_data*, then the *newton* command should be set to *newton on*. Otherwise the data files will not have the same 3'-to-5' polarity as the initial data file. This limitation does not apply to binary restart files produced with *write_restart*.

Example input and data files for DNA duplexes can be found in `examples/PACKAGES/cgdna/examples/lj_units/oxDNA3/` or in the corresponding folder for real units. A simple python setup tool which creates single straight or helical DNA strands, DNA duplexes or arrays of DNA duplexes can be found in `examples/PACKAGES/cgdna/util/`.

---

### 4.235.4 Unique base pairing

Unique base pairing describes the restriction on the specific complementary nucleotide with which a particular base can pair. This can be used to prevent asymmetric base pairs or to simplify the free energy landscape. With unique base pairing enabled base pairs can only form between complementary nucleotides with specific atom IDs. This functionality draws on *fix property/atom* and a modified *read_data* command.

To use unique base pairing, the data file of a system with N nucleotides must contain a section like

```
Basepairs # i_idc

1 idc1
2 idc2
3 idc3
4 idc4
...
N idcN
```

where idc is the non-negative atom ID of a complementary nucleotide that binds uniquely to the preceding atom ID.

Unique base pairing can be combined with normal base pairing by setting a zero or negative value for idc. For instance, in a 4-mer with 8 nucleotides consisting of a ssDNA strand 3'-A-A-A-A-5' with atom IDs 3'-1-2-3-4-5' and a complementary strand 5'-T-T-T-T-3' with atom IDs 5'-8-7-6-5-3' set up as

```
Basepairs # i_idc

1 8
2 -1
3 -1
4 5
5 4
6 -1
7 -1
8 1
```

the A nucleotide with ID 1 can only hybridize with the T nucleotide with ID 8 and the A nucleotide with ID 4 can only hybridize with the T nucleotide with ID 5, whereas the A nucleotides with ID 2 and 3 can hybridize with either T nucleotide with ID 6 and 7.

The input file requires an instance of the *fix property/atom* and a *read_data* command as follows:

```
fix Basepairs all property/atom i_idc ghost yes
read_data file fix Basepairs NULL Basepairs
```

where *file* is the name of the data file and the only modifiable argument. An example input and data file for a dsDNA ring can be found in `examples/PACKAGES/cgdna/examples/lj_units/oxDNA3/unique_bp` or in the corresponding folder for real units.

Please cite *(Henrich)* in any publication that uses this implementation. An updated documentation that contains general information on the model, its implementation and performance as well as the structure of the data and input file can be found here.

Please cite also the relevant oxDNA3 publication *(Bonato)*.

---

### 4.235.5 Restrictions

These pair styles can only be used if LAMMPS was built with the CG-DNA package and the MOLECULE and ASPHERE package. See the *Build package* page for more info.

### 4.235.6 Related commands

*bond_style oxdna3/fene*, *bond_style oxdna/fene*, *pair_style oxdna/excv*, *bond_style oxdna2/fene*, *pair_style oxdna2/excv*, *bond_style oxrna2/fene*, *pair_style oxrna2/excv*, *pair_coeff*, *atom_style oxdna*, *fix nve/dotc/langevin*

### 4.235.7 Default

The option default is half_charged_ends = no.

---

**(Bonato)** A. Bonato, T.E. Ouldridge, A.A. Louis, J.P.K. Doye, L. Rovigatti, M. Matthies, O.Henrich, in preparation.

**(Ouldridge-DPhil)** T.E. Ouldridge, Coarse-grained modelling of DNA and DNA self-assembly, DPhil. University of Oxford (2011).

**(Ouldridge)** T.E. Ouldridge, A.A. Louis, J.P.K. Doye, J. Chem. Phys. 134, 085101 (2011).

**(Henrich)** O. Henrich, Y. A. Gutierrez-Fosado, T. Curk, T. E. Ouldridge, Eur. Phys. J. E 41, 57 (2018).

## 4.236 pair_style oxrna2/excv command

## 4.237 pair_style oxrna2/stk command

## 4.238 pair_style oxrna2/hbond command

## 4.239 pair_style oxrna2/xstk command

## 4.240 pair_style oxrna2/coaxstk command

## 4.241 pair_style oxrna2/dh command

### 4.241.1 Syntax

```
pair_style style1

pair_coeff * * style2 args
```

- style1 = *hybrid/overlay oxrna2/excv oxrna2/stk oxrna2/hbond oxrna2/xstk oxrna2/coaxstk oxrna2/dh*
- style2 = *oxrna2/excv* or *oxrna2/stk* or *oxrna2/hbond* or *oxrna2/xstk* or *oxrna2/coaxstk* or *oxrna2/dh*
- args = list of arguments for these particular styles

```
oxrna2/stk args = seq T xi kappa 6.0 0.43 0.93 0.35 0.78 0.9 0 0.95 0.9 0 0.95 1.3 0 0.8␣
↪1.3 0 0.8 2.0 0.65 2.0 0.65
  seq = seqav (for average sequence stacking strength) or seqdep (for sequence-dependent␣
↪stacking strength)
  T = temperature (LJ units: 0.1 = 300 K, real units: 300 = 300 K)
  xi = 1.40206 (LJ units) or 8.35864576375849 (real units), temperature-independent␣
↪coefficient in stacking strength
  kappa = 2.77 (LJ units) or 0.005504556 (real units), coefficient of linear temperature␣
↪dependence in stacking strength
oxrna2/hbond args = seq eps 8.0 0.4 0.75 0.34 0.7 1.5 0 0.7 1.5 0 0.7 1.5 0 0.7 0.46 3.
↪141592653589793 0.7 4.0 1.5707963267948966 0.45 4.0 1.5707963267948966 0.45
  seq = seqav (for average sequence base-pairing strength) or seqdep (for␣
↪sequence-dependent base-pairing strength)
  eps = 0.870439 (LJ units) or 5.18928666388042 (real units), average hydrogen bonding␣
↪strength between A-U and C-G Watson-Crick and G-U wobble base pairs, 0 between all␣
↪other pairs
oxrna2/dh args = T rhos qeff
  T = temperature (LJ units: 0.1 = 300 K, real units: 300 = 300 K)
  rhos = salt concentration (mole per litre)
  qeff = 1.02455 (effective charge in elementary charges)
```

### 4.241.2 Examples

```
# LJ units
pair_style hybrid/overlay oxrna2/excv oxrna2/stk oxrna2/hbond oxrna2/xstk oxrna2/coaxstk␣
↪oxrna2/dh
pair_coeff * * oxrna2/excv    2.0 0.7 0.675 2.0 0.515 0.5 2.0 0.33 0.32
pair_coeff * * oxrna2/stk     seqdep 0.1 1.40206 2.77 6.0 0.43 0.93 0.35 0.78 0.9 0 0.95␣
↪0.9 0 0.95 1.3 0 0.8 1.3 0 0.8 2.0 0.65 2.0 0.65
```

(continues on next page)

(continued from previous page)

```
pair_coeff * * oxrna2/hbond   seqdep 0.0 8.0 0.4 0.75 0.34 0.7 1.5 0 0.7 1.5 0 0.7 1.5 0␣
↪0.7 0.46 3.141592653589793 0.7 4.0 1.5707963267948966 0.45 4.0 1.5707963267948966 0.45
pair_coeff 1 4 oxrna2/hbond   seqdep 0.870439 8.0 0.4 0.75 0.34 0.7 1.5 0 0.7 1.5 0 0.7␣
↪1.5 0 0.7 0.46 3.141592653589793 0.7 4.0 1.5707963267948966 0.45 4.0 1.
↪5707963267948966 0.45
pair_coeff 2 3 oxrna2/hbond   seqdep 0.870439 8.0 0.4 0.75 0.34 0.7 1.5 0 0.7 1.5 0 0.7␣
↪1.5 0 0.7 0.46 3.141592653589793 0.7 4.0 1.5707963267948966 0.45 4.0 1.
↪5707963267948966 0.45
pair_coeff 3 4 oxrna2/hbond   seqdep 0.870439 8.0 0.4 0.75 0.34 0.7 1.5 0 0.7 1.5 0 0.7␣
↪1.5 0 0.7 0.46 3.141592653589793 0.7 4.0 1.5707963267948966 0.45 4.0 1.
↪5707963267948966 0.45
pair_coeff * * oxrna2/xstk    59.9626 0.5 0.6 0.42 0.58 2.25 0.505 0.58 1.7 1.266 0.68 1.
↪7 1.266 0.68 1.7 0.309 0.68 1.7 0.309 0.68
pair_coeff * * oxrna2/coaxstk 80 0.5 0.6 0.42 0.58 2.0 2.592 0.65 1.3 0.151 0.8 0.9 0.
↪685 0.95 0.9 0.685 0.95 2.0 -0.65 2.0 -0.65
pair_coeff * * oxrna2/dh      0.1 0.5 1.02455

pair_style hybrid/overlay oxrna2/excv oxrna2/stk oxrna2/hbond oxrna2/xstk oxrna2/coaxstk␣
↪oxrna2/dh
pair_coeff * * oxrna2/excv    oxrna2_lj.cgdna
pair_coeff * * oxrna2/stk     seqdep 0.1 oxrna2_lj.cgdna
pair_coeff * * oxrna2/hbond   seqdep oxrna2_lj.cgdna
pair_coeff 1 4 oxrna2/hbond   seqdep oxrna2_lj.cgdna
pair_coeff 2 3 oxrna2/hbond   seqdep oxrna2_lj.cgdna
pair_coeff 3 4 oxrna2/hbond   seqdep oxrna2_lj.cgdna
pair_coeff * * oxrna2/xstk    oxrna2_lj.cgdna
pair_coeff * * oxrna2/coaxstk oxrna2_lj.cgdna
pair_coeff * * oxrna2/dh      0.1 0.5 oxrna2_lj.cgdna

# Real units
pair_style hybrid/overlay oxrna2/excv oxrna2/stk oxrna2/hbond oxrna2/xstk oxrna2/coaxstk␣
↪oxrna2/dh
pair_coeff * * oxrna2/excv    11.92337812042065 5.9626 5.74965 11.92337812042065 4.38677␣
↪4.259 11.92337812042065 2.81094 2.72576
pair_coeff * * oxrna2/stk     seqdep 300.0 8.35864576375849 0.005504556 0.70439070204273␣
↪3.66274 7.92174 2.9813 6.64404 0.9 0.0 0.95 0.9 0.0 0.95 1.3 0.0 0.8 1.3 0.0 0.8 2.0 0.
↪65 2.0 0.65
pair_coeff * * oxrna2/hbond   seqdep 0.0 0.93918760272364 3.4072 6.3885 2.89612 5.9626 1.
↪5 0.0 0.7 1.5 0.0 0.7 1.5 0.0 0.7 0.46 3.141592653589793 0.7 4.0 1.5707963267948966 0.
↪45 4.0 1.5707963267948966 0.45
pair_coeff 1 4 oxrna2/hbond   seqdep 5.18928666388042 0.93918760272364 3.4072 6.3885 2.
↪89612 5.9626 1.5 0.0 0.7 1.5 0.0 0.7 1.5 0.0 0.7 0.46 3.141592653589793 0.7 4.0 1.
↪5707963267948966 0.45 4.0 1.5707963267948966 0.45
pair_coeff 2 3 oxrna2/hbond   seqdep 5.18928666388042 0.93918760272364 3.4072 6.3885 2.
↪89612 5.9626 1.5 0.0 0.7 1.5 0.0 0.7 1.5 0.0 0.7 0.46 3.141592653589793 0.7 4.0 1.
↪5707963267948966 0.45 4.0 1.5707963267948966 0.45
pair_coeff 3 4 oxrna2/hbond   seqdep 5.18928666388042 0.93918760272364 3.4072 6.3885 2.
↪89612 5.9626 1.5 0.0 0.7 1.5 0.0 0.7 1.5 0.0 0.7 0.46 3.141592653589793 0.7 4.0 1.
↪5707963267948966 0.45 4.0 1.5707963267948966 0.45
pair_coeff * * oxrna2/xstk    4.92690859644113 4.259 5.1108 3.57756 4.94044 2.25 0.505 0.
↪58 1.7 1.266 0.68 1.7 1.266 0.68 1.7 0.309 0.68 1.7 0.309 0.68
pair_coeff * * oxrna2/coaxstk 6.57330882442206 4.259 5.1108 3.57756 4.94044 2.0 2.592 0.
```

(continues on next page)

(continued from previous page)

```
↪65 1.3 0.151 0.8 0.9 0.685 0.95 0.9 0.685 0.95 2.0 -0.65 2.0 -0.65
pair_coeff * * oxrna2/dh      300.0 0.5 1.02455

pair_style hybrid/overlay oxrna2/excv oxrna2/stk oxrna2/hbond oxrna2/xstk oxrna2/coaxstk␣
↪oxrna2/dh
pair_coeff * * oxrna2/excv    oxrna2_real.cgdna
pair_coeff * * oxrna2/stk     seqdep 300.0 oxrna2_real.cgdna
pair_coeff * * oxrna2/hbond   seqdep oxrna2_real.cgdna
pair_coeff 1 4 oxrna2/hbond   seqdep oxrna2_real.cgdna
pair_coeff 2 3 oxrna2/hbond   seqdep oxrna2_real.cgdna
pair_coeff 3 4 oxrna2/hbond   seqdep oxrna2_real.cgdna
pair_coeff * * oxrna2/xstk    oxrna2_real.cgdna
pair_coeff * * oxrna2/coaxstk oxrna2_real.cgdna
pair_coeff * * oxrna2/dh      300.0 0.5 oxrna2_real.cgdna
```

**Note**

The coefficients in the above examples are provided in forms compatible with both *units lj* and *units real* (see documentation of *units*). These can also be read from a potential file with correct unit style by specifying the name of the file. Several potential files for each unit style are included in the `potentials` directory of the LAMMPS distribution.

### 4.241.3 Description

The *oxrna2* pair styles compute the pairwise-additive parts of the oxDNA force field for coarse-grained modelling of RNA. The effective interaction between the nucleotides consists of potentials for the excluded volume interaction *oxrna2/excv*, the stacking *oxrna2/stk*, cross-stacking *oxrna2/xstk* and coaxial stacking interaction *oxrna2/coaxstk*, electrostatic Debye-Hueckel interaction *oxrna2/dh* as well as the hydrogen-bonding interaction *oxrna2/hbond* between complementary pairs of nucleotides on opposite strands. Average sequence or sequence-dependent stacking and base-pairing strengths are supported *(Sulc2)*.

The exact functional form of the pair styles is rather complex. The individual potentials consist of products of modulation factors, which themselves are constructed from a number of more basic potentials (Morse, Lennard-Jones, harmonic angle and distance) as well as quadratic smoothing and modulation terms. We refer to *(Sulc1)* and the original oxDNA publications *(Ouldridge-DPhil)* and *(Ouldridge)* for a detailed description of the oxRNA2 force field.

**Note**

These pair styles have to be used together with the related oxDNA2 bond style *oxrna2/fene* for the connectivity of the phosphate backbone (see also documentation of *bond_style oxrna2/fene*). Most of the coefficients in the above example have to be kept fixed and cannot be changed without reparameterizing the entire model. Exceptions are the first two coefficients after *oxrna2/stk* (seq=seqdep and T=0.1 and corresponding *real unit* equivalents in the above examples), the first coefficient after *oxrna2/hbond* (seq=seqdep in the above example) and the two coefficients after *oxrna2/dh* (T=0.1 and rhos=0.5 in the above example). When using a Langevin thermostat e.g. through *fix langevin* or *fix nve/dotc/langevin* the temperature coefficients have to be matched to the one used in the fix.

**Note**

These pair styles have to be used with the *atom_style hybrid bond ellipsoid oxdna* (see documentation of *atom_style*). The *atom_style oxdna* stores the 3'-to-5' polarity of the nucleotide strand, which is set through the bond topology

in the data file. The first (second) atom in a bond definition is understood to point towards the 3'-end (5'-end) of the strand.

**Warning**

If data files are produced with *write_data*, then the *newton* command should be set to *newton on*. Otherwise the data files will not have the same 3'-to-5' polarity as the initial data file. This limitation does not apply to binary restart files produced with *write_restart*.

Example input and data files for DNA duplexes can be found in `examples/PACKAGES/cgdna/examples/lj_units/oxRNA2/` or in the corresponding folder for real units. A simple python setup tool which creates single straight or helical DNA strands, DNA duplexes or arrays of DNA duplexes can be found in `examples/PACKAGES/cgdna/util/`.

Please cite *(Henrich)* in any publication that uses this implementation. The article contains general information on the model, its implementation and performance as well as the structure of the data and input file. The preprint version of the article can be found here. Please cite also the relevant oxRNA2 publications *(Sulc1)* and *(Sulc2)*.

---

### 4.241.4 Potential file reading

For each pair style above the first non-modifiable argument can be a filename (with exception of Debye-Hueckel, for which the effective charge argument can be a filename), and if it is, no further arguments should be supplied. Therefore the following command:

```
pair_coeff 3 4 oxrna2/hbond   seqdep oxrna2_lj.cgdna
```

will be interpreted as a request to read the corresponding hydrogen bonding potential parameters from the file with the given name. The file can define multiple potential parameters for both bonded and pair interactions, but for the example pair interaction above there must exist in the file a line of the form:

```
3 4 hbond     <coefficients>
```

If potential customization is required, the potential file reading can be mixed with the manual specification of the potential parameters. For example, the following command:

```
pair_style hybrid/overlay oxrna2/excv oxrna2/stk oxrna2/hbond oxrna2/xstk oxrna2/coaxstk␣
↪oxrna2/dh
pair_coeff * * oxrna2/excv    2.0 0.7 0.675 2.0 0.515 0.5 2.0 0.33 0.32
pair_coeff * * oxrna2/stk     seqdep 0.1 oxrna2_lj.cgdna
pair_coeff * * oxrna2/hbond   seqdep oxrna2_lj.cgdna
pair_coeff 1 4 oxrna2/hbond   seqdep oxrna2_lj.cgdna
pair_coeff 2 3 oxrna2/hbond   seqdep oxrna2_lj.cgdna
pair_coeff 3 4 oxrna2/hbond   seqdep oxrna2_lj.cgdna
pair_coeff * * oxrna2/xstk    oxrna2_lj.cgdna
pair_coeff * * oxrna2/coaxstk oxrna2_lj.cgdna
pair_coeff * * oxrna2/dh      0.1 0.5 1.02455
```

will read the excluded volume and Debye-Hueckel effective charge *qeff* parameters from the manual specification and all others from the potential file *oxrna2_lj.cgdna*.

There are sample potential files for each unit style in the `potentials` directory of the LAMMPS distribution. The potential file unit system must align with the units defined via the *units* command. For conversion between different

*LJ* and *real* unit systems for oxDNA, the python tool *lj2real.py* located in the `examples/PACKAGES/cgdna/util/` directory can be used. This tool assumes similar file structure to the examples found in `examples/PACKAGES/cgdna/examples/`.

---

### 4.241.5 Unique base pairing

Unique base pairing describes the restriction on the specific complementary nucleotide with which a particular base can pair. This can be used to prevent asymmetric base pairs or to simplify the free energy landscape. With unique base pairing enabled base pairs can only form between complementary nucleotides with specific atom IDs. This functionality draws on *fix property/atom* and a modified *read_data* command.

To use unique base pairing, the data file of a system with N nucleotides must contain a section like

```
Basepairs # i_idc

1 idc1
2 idc2
3 idc3
4 idc4
...
N idcN
```

where idc is the non-negative atom ID of a complementary nucleotide that binds uniquely to the preceding atom ID.

Unique base pairing can be combined with normal base pairing by setting a zero or negative value for idc. For instance, in a 4-mer with 8 nucleotides consisting of a ssDNA strand 3'-A-A-A-A-5' with atom IDs 3'-1-2-3-4-5' and a complementary strand 5'-T-T-T-T-3' with atom IDs 5'-8-7-6-5-3' set up as

```
Basepairs # i_idc

1 8
2 -1
3 -1
4 5
5 4
6 -1
7 -1
8 1
```

the A nucleotide with ID 1 can only hybridize with the T nucleotide with ID 8 and the A nucleotide with ID 4 can only hybridize with the T nucleotide with ID 5, whereas the A nucleotides with ID 2 and 3 can hybridize with either T nucleotide with ID 6 and 7.

The input file requires an instance of the *fix property/atom* and a *read_data* command as follows:

```
fix Basepairs all property/atom i_idc ghost yes
read_data file fix Basepairs NULL Basepairs
```

where *file* is the name of the data file and the only modifiable argument. An example input and data file for a dsDNA ring can be found in `examples/PACKAGES/cgdna/examples/lj_units/oxDNA3/unique_bp` or in the corresponding folder for real units.

---

### 4.241.6 Restrictions

These pair styles can only be used if LAMMPS was built with the CG-DNA package and the MOLECULE and ASPHERE package. See the *Build package* page for more info.

### 4.241.7 Related commands

*bond_style oxrna2/fene*, *pair_coeff*, *bond_style oxdna/fene*, *pair_style oxdna/excv*, *bond_style oxdna2/fene*, *pair_style oxdna2/excv*, *atom_style oxdna*, *fix nve/dotc/langevin*

### 4.241.8 Default

none

---

**(Henrich)** O. Henrich, Y. A. Gutierrez-Fosado, T. Curk, T. E. Ouldridge, Eur. Phys. J. E 41, 57 (2018).

**(Sulc1)** P. Sulc, F. Romano, T. E. Ouldridge, et al., J. Chem. Phys. 140, 235102 (2014).

**(Sulc2)** P. Sulc, F. Romano, T.E. Ouldridge, L. Rovigatti, J.P.K. Doye, A.A. Louis, J. Chem. Phys. 137, 135101 (2012).

**(Ouldridge-DPhil)** T.E. Ouldridge, Coarse-grained modelling of DNA and DNA self-assembly, DPhil. University of Oxford (2011).

**(Ouldridge)** T.E. Ouldridge, A.A. Louis, J.P.K. Doye, J. Chem. Phys. 134, 085101 (2011).

## 4.242 pair_style pace command

Accelerator Variants: *pace/kk*

## 4.243 pair_style pace/extrapolation command

Accelerator Variants: *pace/extrapolation/kk*

### 4.243.1 Syntax

```
pair_style pace ... keyword values ...
```

- one or more keyword/value pairs may be appended

  keyword = *product* or *recursive* or *chunksize*
    *product* = use product algorithm for basis functions
    *recursive* = use recursive algorithm for basis functions
    *chunksize* value = number of atoms in each pass

```
pair_style pace/extrapolation
```

### 4.243.2 Examples

```
pair_style pace
pair_style pace product chunksize 2048
pair_coeff * * Cu-PBE-core-rep.ace Cu

pair_style pace
```

(continues on next page)

(continued from previous page)

```
pair_coeff * * Cu.yaml Cu

pair_style pace/extrapolation
pair_coeff * * Cu.yaml Cu.asi Cu
```

### 4.243.3 Description

Pair style *pace* computes interactions using the Atomic Cluster Expansion (ACE), which is a general expansion of the atomic energy in multi-body basis functions. *(Drautz19)*. The *pace* pair style provides an efficient implementation that is described in this paper *(Lysogorskiy21)*.

In ACE, the total energy is decomposed into a sum over atomic energies. The energy of atom $i$ is expressed as a linear or non-linear function of one or more density functions. By projecting the density onto a local atomic base, the lowest order contributions to the energy can be expressed as a set of scalar polynomials in basis function contributions summed over neighbor atoms.

Only a single pair_coeff command is used with the *pace* style which specifies an ACE coefficient file followed by N additional arguments specifying the mapping of ACE elements to LAMMPS atom types, where N is the number of LAMMPS atom types:

- ACE coefficient file (.yaml or .yace/.ace format)
- N element names = mapping of ACE elements to atom types

Only a single pair_coeff command is used with the *pace* style which specifies an ACE file that fully defines the potential. Note that unlike for other potentials, cutoffs are not set in the pair_style or pair_coeff command; they are specified in the ACE file.

The pair_style *pace* command may be followed by the optional keyword *product* or *recursive*, which determines which of two algorithms is used for the calculation of basis functions and derivatives. The default is *recursive*.

The keyword *chunksize* is only applicable when using the pair style *pace* with the KOKKOS package on GPUs and is ignored otherwise. This keyword controls the number of atoms in each pass used to compute the atomic cluster expansion and is used to avoid running out of memory. For example if there are 8192 atoms in the simulation and the *chunksize* is set to 4096, the ACE calculation will be broken up into two passes (running on a single GPU).

### 4.243.4 Extrapolation grade

Calculation of extrapolation grade in PACE is implemented in *pair_style pace/extrapolation*. It is based on the MaxVol algorithm similar to Moment Tensor Potential (MTP) by Shapeev et al. and is described in *(Lysogorskiy23)*. In order to compute extrapolation grade one needs to provide:

1. ACE potential in B-basis form (*.yaml* format) and
2. Active Set Inverted (ASI) file for corresponding potential (*.asi* format)

Calculation of extrapolation grades requires matrix-vector multiplication for each atom and is slower than the usual *pair_style pace recursive*, therefore it is *not* computed by default. Extrapolation grade calculation is involved by *fix pair*, which requests to compute *gamma*, as shown in example below:

```
pair_style  pace/extrapolation
pair_coeff  * * Cu.yaml Cu.asi Cu

fix pace_gamma all pair 10 pace/extrapolation gamma 1

compute max_pace_gamma all reduce max f_pace_gamma
variable dump_skip equal "c_max_pace_gamma < 5"
```

(continues on next page)

(continued from previous page)

```
dump pace_dump all custom 20 extrapolative_structures.dump id type x y z f_pace_gamma
dump_modify pace_dump skip v_dump_skip

variable max_pace_gamma equal c_max_pace_gamma
fix extreme_extrapolation all halt 10 v_max_pace_gamma > 25
```

Here extrapolation grade gamma is computed every 10 steps and is stored in *f_pace_gamma* per-atom variable. The largest value of extrapolation grade among all atoms in a structure is reduced to *c_max_pace_gamma* variable. Only if this value exceeds extrapolation threshold 5, then the structure will be dumped into *extrapolative_structures.dump* file, but not more often than every 20 steps.

On all other steps *pair_style pace recursive* will be used.

When using the pair style *pace/extrapolation* with the KOKKOS package on GPUs product B-basis evaluator is always used and only *linear* ASI is supported.

---

See the *pair_coeff* page for alternate ways to specify the path for the ACE coefficient file.

### 4.243.5 Core repulsion

The ACE potential can be configured to initiate core-repulsion from an inner cutoff, seamlessly transitioning from ACE to ZBL. The core repulsion factor can be accessed as a per-atom quantity, as demonstrated in the example below:

```
pair_style  pace
pair_coeff  * * CuNi.yaml Cu Ni

fix pace_corerep all pair 1 pace corerep 1
```

In this case, per-atom *f_pace_corerep* quantities represent the fraction of ZBL core-repulsion for each atom.

### 4.243.6 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, where types I and J correspond to two different element types, mixing is performed by LAMMPS with user-specifiable parameters as described above. You never need to specify a pair_coeff command with I != J arguments for this style.

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.243.7 Restrictions

This pair style is part of the ML-PACE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.243.8 Related commands

*pair_style snap*, *fix pair*

### 4.243.9 Default

recursive, chunksize = 4096,

**(Drautz19)** Drautz, Phys Rev B, 99, 014104 (2019).

**(Lysogorskiy21)** Lysogorskiy, van der Oord, Bochkarev, Menon, Rinaldi, Hammerschmidt, Mrovec, Thompson, Csanyi, Ortner, Drautz, npj Comp Mat, 7, 97 (2021).

**(Lysogorskiy23)** Lysogorskiy, Bochkarev, Mrovec, Drautz, Phys Rev Mater, 7, 043801 (2023) / arXiv:2212.08716 (2022).

## 4.244 pair_style pace/apip command

## 4.245 pair_style pace/fast/apip command

## 4.246 pair_style pace/precise/apip command

Constant precision variant: *pace*

### 4.246.1 Syntax

```
pair_style pace/apip ... keyword values ...
pair_style pace/fast/apip ... keyword values ...
pair_style pace/precise/apip ... keyword values ...
```

- one or more keyword/value pairs may be appended

  keyword = keywords of *pair pace*

### 4.246.2 Examples

```
pair_style hybrid/overlay pace/fast/apip pace/precise/apip lambda/input/csp/apip fcc␣
↪cutoff 5.0 lambda/zone/apip 12.0
pair_coeff * * pace/fast/apip Cu_fast.yace Cu
pair_coeff * * pace/precise/apip Cu_precise.yace Cu
pair_coeff * * lambda/input/csp/apip
pair_coeff * * lambda/zone/apip

pair_style hybrid/overlay eam/fs/apip pace/precise/apip lambda/input/csp/apip fcc cutoff␣
↪5.0 lambda/zone/apip 12.0
```

(continues on next page)

(continued from previous page)

```
pair_coeff * * eam/fs/apip Cu.eam.fs Cu
pair_coeff * * pace/precise/apip Cu_precise.yace Cu
pair_coeff * * lambda/input/csp/apip
pair_coeff * * lambda/zone/apip
```

### 4.246.3 Description

Pair style *pace* computes interactions using the Atomic Cluster Expansion (ACE), which is a general expansion of the atomic energy in multi-body basis functions *(Drautz19)*. The *pace* pair style provides an efficient implementation that is described in this paper *(Lysogorskiy21)*.

The potential energy $E_i$ of an atom $i$ of an adaptive-precision interatomic potential (APIP) according to *(Immel25)* is given by

$$E_i^{\mathrm{APIP}} = \lambda_i E_i^{(\mathrm{fast})} + (1 - \lambda_i) E_i^{(\mathrm{precise})} ,$$

whereas the switching parameter $\lambda_i$ is computed dynamically during a simulation by *fix lambda/apip* or set prior to a simulation via *set*.

The pair style *pace/precise/apip* computes the potential energy $(1 - \lambda_i)E_i^{(\mathrm{pace})}$ and the corresponding force and should be combined with a fast potential that computes the potential energy $\lambda_i E_i^{(\mathrm{fast})}$ and the corresponding force via *pair_style hybrid/overlay*.

The pair style *pace/fast/apip* computes the potential energy $\lambda_i E_i^{(\mathrm{pace})}$ and the corresponding force and should be combined with a precise potential that computes the potential energy $(1 - \lambda_i)E_i^{(\mathrm{precise})}$ and the corresponding force via *pair_style hybrid/overlay*.

The pair_styles *pace/fast/apip* and *pace/precise/apip* commands may be followed by the optional keywords of *pair_style pace*, which are described *here*.

### 4.246.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, where types I and J correspond to two different element types, mixing is performed by LAMMPS with user-specifiable parameters as described above. You never need to specify a pair_coeff command with I != J arguments for this style.

These pair styles do not support the *pair_modify* shift, table, and tail options.

These pair styles do not write their information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

These pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

---

### 4.246.5 Restrictions

These pair styles are part of the APIP package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.246.6 Related commands

*pair_style pace*, *pair_style hybrid/overlay*, *fix lambda/apip*, *fix lambda_thermostat/apip*, *pair_style lambda/zone/apip*, *pair_style lambda/input/apip*, *pair_style eam/apip*, *fix atom_weight/apip*

### 4.246.7 Default

See *pair_style pace*.

---

**(Drautz19)** Drautz, Phys Rev B, 99, 014104 (2019).

**(Lysogorskiy21)** Lysogorskiy, van der Oord, Bochkarev, Menon, Rinaldi, Hammerschmidt, Mrovec, Thompson, Csanyi, Ortner, Drautz, npj Comp Mat, 7, 97 (2021).

**(Immel25)** Immel, Drautz and Sutmann, J Chem Phys, 162, 114119 (2025)

## 4.247 pair_style pedone command

Accelerator Variants: *pedone/omp*, *pedone/kk*

### 4.247.1 Syntax

```
pair_style style args
```

- style = pedone*
- args = list of arguments for a particular style

```
pedone args = cutoff
  cutoff = global cutoff for Pedone interactions (distance units)
```

### 4.247.2 Examples

```
pair_style hybrid/overlay pedone 15.0 coul/long 15.0
kspace_style pppm 1.0e-5

pair_coeff * * coul/long
pair_coeff 1 2 pedone 0.030211 2.241334 2.923245 5.0
pair_coeff 2 2 pedone 0.042395 1.379316 3.618701 22.0
```

Used in input scripts:

```
examples/PACKAGES/pedone/in.pedone.relax
examples/PACKAGES/pedone/in.pedone.melt
```

### 4.247.3 Description

Added in version 17Apr2024.

Pair style *pedone* computes the **non-Coulomb** interactions of the Pedone (or PMMCS) potential *(Pedone)* which combines Coulomb interactions, Morse potential, and repulsive $r^{-12}$ Lennard-Jones terms (see below). The *pedone* pair style is meant to be used in addition to a *Coulomb pair style* via pair style *hybrid/overlay* (see example above). Using *coul/long* or *could/dsf* (for solids) is recommended.

The full Pedone potential function from *(Pedone)* for each pair of atoms is:

$$E = \frac{Cq_iq_j}{\varepsilon r} + D_0\left[e^{-2\alpha(r-r_0)} - 2e^{-\alpha(r-r_0)}\right] + \frac{B_0}{r^{12}} \qquad r < r_c$$

$r_c$ is the cutoff and $C$ is a conversion factor that is specific to the choice of *units* so that the entire Coulomb term is in energy units with $q_i$ and $q_j$ as the assigned charges in multiples of the elementary charge.

The following coefficients must be defined for the selected pairs of atom types via the *pair_coeff* command as in the example above:

- $D_0$ (energy units)
- $\alpha$ (1/distance units)
- $r_0$ (distance units)
- $C_0$ (energy units)
- cutoff (distance units)

The last coefficient is optional. If not specified, the global *pedone* cutoff is used.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.247.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support mixing.

This pair style support the *pair_modify* shift option for the energy of the pair interaction.

This pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands does not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, or *outer* keywords.

---

### 4.247.5 Restrictions

The *pedone* pair style is only enabled if LAMMPS was built with the EXTRA-PAIR package. See the *Build package* page for more info.

### 4.247.6 Related commands

*pair_coeff*, *pair_style*, *pair style coul/long and coul/dsf*, *pair style morse*

### 4.247.7 Default

none

---

**(Pedone)** A. Pedone, G. Malavasi, M. C. Menziani, A. N. Cormack, and U. Segre, J. Phys. Chem. B, 110, 11780 (2006)

## 4.248 pair_style peri/pmb command

Accelerator Variants: *peri/pmb/omp*

## 4.249 pair_style peri/lps command

Accelerator Variants: *peri/lps/omp*

## 4.250 pair_style peri/ves command

## 4.251 pair_style peri/eps command

### 4.251.1 Syntax

```
pair_style style
```

- style = *peri/pmb* or *peri/lps* or *peri/ves* or *peri/eps*

### 4.251.2 Examples

```
pair_style peri/pmb
pair_coeff * * 1.6863e22 0.0015001 0.0005 0.25

pair_style peri/lps
pair_coeff * * 14.9e9 14.9e9 0.0015001 0.0005 0.25

pair_style peri/ves
pair_coeff * * 14.9e9 14.9e9 0.0015001 0.0005 0.25 0.5 0.001

pair_style peri/eps
pair_coeff * * 14.9e9 14.9e9 0.0015001 0.0005 0.25 118.43
```

### 4.251.3 Description

The peridynamic pair styles implement material models that can be used at the mesoscopic and macroscopic scales. See this document for an overview of LAMMPS commands for Peridynamics modeling.

Style *peri/pmb* implements the Peridynamic bond-based prototype microelastic brittle (PMB) model.

Style *peri/lps* implements the Peridynamic state-based linear peridynamic solid (LPS) model.

Style *peri/ves* implements the Peridynamic state-based linear peridynamic viscoelastic solid (VES) model.

Style *peri/eps* implements the Peridynamic state-based elastic-plastic solid (EPS) model.

The canonical papers on Peridynamics are *(Silling 2000)* and *(Silling 2007)*. The implementation of Peridynamics in LAMMPS is described in *(Parks)*. Also see the *Peridynamics Howto* for more details about its implementation.

The peridynamic VES and EPS models in PDLAMMPS were implemented by R. Rahman and J. T. Foster at University of Texas at San Antonio. The original VES formulation is described in "(Mitchell2011)" and the original EPS formulation is in "(Mitchell2011a)". Additional PDF docs that describe the VES and EPS implementations are include in the LAMMPS distribution in doc/PDF/PDLammps_VES.pdf and doc/PDF/PDLammps_EPS.pdf. For questions regarding the VES and EPS models in LAMMPS you can contact R. Rahman (rezwanur.rahman at utsa.edu).

The following coefficients must be defined for each pair of atom types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below.

For the *peri/pmb* style:

- c (energy/distance/volume^2 units)
- horizon (distance units)
- s00 (unitless)
- $\alpha$ (unitless)

C is the effectively a spring constant for Peridynamic bonds, the horizon is a cutoff distance for truncating interactions, and s00 and $\alpha$ are used as a bond breaking criteria. The units of c are such that c/distance = stiffness/volume^2, where stiffness is energy/distance^2 and volume is distance^3. See the users guide for more details.

For the *peri/lps* style:

- K (force/area units)
- G (force/area units)
- horizon (distance units)
- s00 (unitless)
- $\alpha$ (unitless)

K is the bulk modulus and G is the shear modulus. The horizon is a cutoff distance for truncating interactions, and s00 and $\alpha$ are used as a bond breaking criteria. See the users guide for more details.

For the *peri/ves* style:

- K (force/area units)
- G (force/area units)
- horizon (distance units)
- s00 (unitless)
- $\alpha$ (unitless)
- m_lambdai (unitless)
- m_taubi (unitless)

K is the bulk modulus and G is the shear modulus. The horizon is a cutoff distance for truncating interactions, and s00 and $\alpha$ are used as a bond breaking criteria. m_lambdai and m_taubi are the viscoelastic relaxation parameter and time constant, respectively. m_lambdai varies within zero to one. For very small values of m_lambdai the viscoelastic model responds very similar to a linear elastic model. For details please see the description in "(Mitchell2011)".

For the *peri/eps* style:

- K (force/area units)
- G (force/area units)
- horizon (distance units)
- s00 (unitless)
- $\alpha$ (unitless)
- m_yield_stress (force/area units)

K is the bulk modulus and G is the shear modulus. The horizon is a cutoff distance and s00 and $\alpha$ are used as a bond breaking criteria. m_yield_stress is the yield stress of the material. For details please see the description in "(Mitchell2011a)".

Changed in version 4Jul2026.

**Note**

Prior versions of LAMMPS, had an incorrect the plasticity model in style *peri/eps* relative to the source report *(Mitchell2011a)*. These affected the evolution of the plastic deviatoric extension and caused significant overshooting of the yield surface. These have since been corrected, however, there is still no radial return rule to ensure the plastic deviatoric extension does not leave the yield surface. This may cause some drift off the surface during long simulations. This possibility for future improvement is tracked as issue #5064.

---

### 4.251.4 Bond breaking criterion

For all of these styles a peridynamic bond between particles $i$ and $j$ breaks irreversibly once its stretch $s = (r - r_0)/r_0$ exceeds a critical stretch. Following *(Parks)* (eq. 9), the critical stretch of a particle is $s_0 = s_{00} - \alpha\, s_{min}$, where $s_{min}$ is the minimum (most compressive) stretch over all of the particle's bonds, and a bond breaks when $s > \min(s_{0,i}, s_{0,j})$, i.e. symmetrically from the point of view of both particles.

The s00 and $\alpha$ coefficients may be chosen differently for each pair of atom types. This makes it possible, for example, to assign a smaller s00 to the bonds across an interface between two materials so that a crack preferentially initiates there. The critical stretch is evaluated *per bond* using that bond's own s00 and $\alpha$ together with the geometric $s_{min}$ of each endpoint.

Changed in version 4Jul2026.

In previous versions the critical stretch was stored as a single per-particle value computed as the maximum of $s_{00} - \alpha s$ over a particle's bonds. That is only equivalent to the criterion above when s00 and $\alpha$ are identical for all type pairs; with type-dependent coefficients a bond would incorrectly inherit the critical stretch of the surrounding bulk bonds and a weakened interface would not fracture. The criterion is now evaluated per bond so that type-dependent s00 and $\alpha$ behave as intended.

The *compute property/atom s0* per-atom property reports this per-particle critical stretch. Earlier versions of this documentation incorrectly described *s0* as the maximum stretch of any bond a particle is part of.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.251.5 Mixing, shift, table, tail correction, restart, rRESPA info

These pair styles do not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

These pair styles do not support the *pair_modify* shift option.

The *pair_modify* table and tail options are not relevant for these pair styles.

These pair styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

These pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

---

### 4.251.6 Restrictions

All of these styles are part of the PERI package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.251.7 Related commands

*pair_coeff*

### 4.251.8 Default

none

---

**(Parks)** Parks, Lehoucq, Plimpton, Silling, Comp Phys Comm, 179(11), 777-783 (2008).

**(Silling 2000)** Silling, J Mech Phys Solids, 48, 175-209 (2000).

**(Silling 2007)** Silling, Epton, Weckner, Xu, Askari, J Elasticity, 88, 151-184 (2007).

**(Mitchell2011)** Mitchell. A non-local, ordinary-state-based viscoelasticity model for peridynamics. Sandia National Lab Report, 8064:1-28 (2011).

**(Mitchell2011a)** Mitchell. A Nonlocal, Ordinary, State-Based Plasticity Model for Peridynamics. Sandia National Lab Report, 3166:1-34 (2011).

## 4.252 pair_style pod command

Accelerator Variants: *pod/kk*

### 4.252.1 Syntax

```
pair_style pod
```

### 4.252.2 Examples

```
pair_style pod
pair_coeff * * Ta_param.pod Ta_coefficients.pod Ta
```

### 4.252.3 Description

Added in version 22Dec2022.

Pair style *pod* defines the proper orthogonal descriptor (POD) potential *(Nguyen and Rohskopf)*, *(Nguyen2023)*, *(Nguyen2024)*, and *(Nguyen and Sema)*. The *fitpod* is used to fit the POD potential.

Only a single pair_coeff command is used with the *pod* style which specifies a POD parameter file followed by a coefficient file, a projection matrix file, and a centroid file.

The POD parameter file (`Ta_param.pod`) can contain blank and comment lines (start with #) anywhere. Each non-blank non-comment line must contain one keyword/value pair. See *fitpod* for the description of all the keywords that can be assigned in the parameter file.

The coefficient file (`Ta_coefficients.pod`) contains coefficients for the POD potential. The top of the coefficient file can contain any number of blank and comment lines (start with #), but follows a strict format after that. The first non-blank non-comment line must contain:

- model_coefficients: *ncoeff nproj ncentroid*

This is followed by *ncoeff* coefficients, *nproj* projection matrix entries, and *ncentroid* centroid coordinates, one per line. The coefficient file is generated after training the POD potential using *fitpod*.

As an example, if a LAMMPS indium phosphide simulation has 4 atoms types, with the first two being indium and the third and fourth being phophorous, the pair_coeff command would look like this:

```
pair_coeff * * pod InP_param.pod InP_coefficients.pod In In P P
```

The first 2 arguments must be * * so as to span all LAMMPS atom types. The two filenames are for the parameter and coefficient files, respectively. The two trailing 'In' arguments map LAMMPS atom types 1 and 2 to the POD 'In' element. The two trailing 'P' arguments map LAMMPS atom types 3 and 4 to the POD 'P' element.

If a POD mapping value is specified as NULL, the mapping is not performed. This can be used when a *pod* potential is used as part of the *hybrid* pair style. The NULL values are placeholders for atom types that will be used with other potentials.

Examples about training and using POD potentials are found in the directory lammps/examples/PACKAGES/pod and the Github repo https://github.com/cesmix-mit/pod-examples.

---

### 4.252.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, where types I and J correspond to two different element types, mixing is performed by LAMMPS with user-specifiable parameters as described above. You never need to specify a pair_coeff command with I != J arguments for this style.

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.252.5 Restrictions

This style is part of the ML-POD package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.252.6 Related commands

*fitpod*, *compute pod/atom*, *compute podd/atom*, *compute pod/local*, *compute pod/global*

### 4.252.7 Default

none

---

**(Nguyen and Rohskopf)** Nguyen and Rohskopf, Journal of Computational Physics, 480, 112030, (2023).

**(Nguyen2023)** Nguyen, Physical Review B, 107(14), 144103, (2023).

**(Nguyen2024)** Nguyen, Journal of Computational Physics, 113102, (2024).

**(Nguyen and Sema)** Nguyen and Sema, https://arxiv.org/abs/2405.00306, (2024).

## 4.253 pair_style polymorphic command

### 4.253.1 Syntax

```
pair_style polymorphic
```

style = *polymorphic*

### 4.253.2 Examples

```
pair_style polymorphic
pair_coeff * * FeCH_BOP_I.poly Fe C H
pair_coeff * * TlBr_msw.poly Tl Br
pair_coeff * * CuTa_eam.poly Cu Ta
pair_coeff * * GaN_tersoff.poly Ga N
pair_coeff * * GaN_sw.poly Ga N
```

### 4.253.3 Description

The *polymorphic* pair style computes a 3-body free-form potential (*Zhou3*) for the energy E of a system of atoms as

$$E = \frac{1}{2}\sum_{i=1}^{i=N}\sum_{j=1}^{j=N}\left[\left(1-\delta_{ij}\right)\cdot U_{IJ}\left(r_{ij}\right) - \left(1-\eta_{ij}\right)\cdot F_{IJ}\left(X_{ij}\right)\cdot V_{IJ}\left(r_{ij}\right)\right]$$

$$X_{ij} = \sum_{k=i_1,k\neq j}^{i_N} W_{IK}\left(r_{ik}\right)\cdot G_{JIK}\left(\cos\theta_{jik}\right)\cdot P_{JIK}\left(\Delta r_{jik}\right)$$

$$\Delta r_{jik} = r_{ij} - \xi_{IJ}\cdot r_{ik}$$

where I, J, K represent species of atoms i, j, and k, $i_1, ..., i_N$ represents a list of $i$'s neighbors, $\delta_{ij}$ is a Dirac constant (i.e., $\delta_{ij} = 1$ when $i = j$, and $\delta_{ij} = 0$ otherwise), $\eta_{ij}$ is similar constant that can be set either to $\eta_{ij} = \delta_{ij}$ or $\eta_{ij} = 1 - \delta_{ij}$ depending on the potential type, $U_{IJ}(r_{ij})$, $V_{IJ}(r_{ij})$, $W_{IK}(r_{ik})$ are pair functions, $G_{JIK}(\cos\theta_{jik})$ is an angular function, $P_{JIK}(\Delta r_{jik})$ is a function of atomic spacing differential $\Delta r_{jik} = r_{ij} - \xi_{IJ}\cdot r_{ik}$ with $\xi_{IJ}$ being a pair-dependent parameter, and $F_{IJ}(X_{ij})$ is a function of the local environment variable $X_{ij}$. This generic potential is fully defined once the constants $\eta_{ij}$ and $\xi_{IJ}$, and the six functions $U_{IJ}(r_{ij})$, $V_{IJ}(r_{ij})$, $W_{IK}(r_{ik})$, $G_{JIK}(\cos\theta_{jik})$, $P_{JIK}(\Delta r_{jik})$, and $F_{IJ}(X_{ij})$ are given. Here LAMMPS uses a global parameter $\eta$ to represent $\eta_{ij}$. When $\eta = 1$, $\eta_{ij} = 1 - \delta_{ij}$, otherwise $\eta_{ij} = \delta_{ij}$. Additionally, $\eta = 3$ indicates that the function $P_{JIK}(\Delta r)$ depends on species I, J and K, otherwise $P_{JIK}(\Delta r) = P_{IK}(\Delta r)$ only depends on species I and K. Note that these six functions are all one dimensional, and hence can be provided in a tabular form. This allows users to design different potentials solely based on a manipulation of these functions. For instance, the potential reduces to a Stillinger-Weber potential (*SW*) if we set

$$\eta_{ij} = \delta_{ij}(\eta = 2\ or\ \eta = 0), \xi_{IJ} = 0$$

$$U_{IJ}(r) = A_{IJ}\cdot\varepsilon_{IJ}\cdot\left(\frac{\sigma_{IJ}}{r}\right)^{q}\cdot\left[B_{IJ}\cdot\left(\frac{\sigma_{IJ}}{r}\right)^{p-q} - 1\right]\cdot exp\left(\frac{\sigma_{IJ}}{r - a_{IJ}\cdot\sigma_{IJ}}\right)$$

$$V_{IJ}(r) = \sqrt{\lambda_{IJ}\cdot\varepsilon_{IJ}}\cdot exp\left(\frac{\gamma_{IJ}\cdot\sigma_{IJ}}{r - a_{IJ}\cdot\sigma_{IJ}}\right)$$

$$F_{IJ}(X) = -X$$

$$P_{JIK}(\Delta r) = P_{IK}(\Delta r) = 1$$

$$W_{IJ}(r) = \sqrt{\lambda_{IJ}\cdot\varepsilon_{IJ}}\cdot exp\left(\frac{\gamma_{IJ}\cdot\sigma_{IJ}}{r - a_{IJ}\cdot\sigma_{IJ}}\right)$$

$$G_{JIK}(\cos\theta) = \left(\cos\theta + \frac{1}{3}\right)^2$$

The potential reduces to a Tersoff potential (*Tersoff* or *Albe1*) if we set

$$\eta_{ij} = \delta_{ij}(\eta = 2\ or\ \eta = 0), \xi_{IJ} = 1$$

$$U_{IJ}(r) = \frac{D_{e,IJ}}{S_{IJ} - 1}\cdot exp\left[-\beta_{IJ}\sqrt{2S_{IJ}}\left(r - r_{e,IJ}\right)\right]\cdot f_{c,IJ}(r)$$

$$V_{IJ}(r) = \frac{S_{IJ}\cdot D_{e,IJ}}{S_{IJ} - 1}\cdot exp\left[-\beta_{IJ}\sqrt{\frac{2}{S_{IJ}}}\left(r - r_{e,IJ}\right)\right]\cdot f_{c,IJ}(r)$$

$$F_{IJ}(X) = (1+X)^{-\frac{1}{2}}$$

$$P_{JIK}(\Delta r) = P_{IK}(\Delta r) = exp(2\mu_{IK}\cdot\Delta r)$$

$$W_{IJ}(r) = f_{c,IJ}(r)$$

$$G_{JIK}(\cos\theta) = \gamma_{IK}\left[1 + \frac{c_{IK}^2}{d_{IK}^2} - \frac{c_{IK}^2}{d_{IK}^2 + (h_{IK} + \cos\theta)^2}\right]$$

where

$$f_{c,IJ}(r) = \begin{cases} 1, r \leq R_{IJ} - D_{IJ} \\ \frac{1}{2} + \frac{1}{2}cos\left[\frac{\pi(r+D_{IJ}-R_{IJ})}{2D_{IJ}}\right], R_{IJ} - D_{IJ} < r < R_{IJ} + D_{IJ} \\ 0, r \geq R_{IJ} + D_{IJ} \end{cases}$$

The potential reduces to a modified Stillinger-Weber potential (*Zhou3*) if we set

$$\begin{aligned}
\eta_{ij} &= \delta_{ij}(\eta = 2 \; or \; \eta = 0), \xi_{IJ} = 0 \\
U_{IJ}(r) &= \varphi_{R,IJ}(r) - \varphi_{A,IJ}(r) \\
V_{IJ}(r) &= u_{IJ}(r) \\
F_{IJ}(X) &= -X \\
P_{JIK}(\Delta r) &= P_{IK}(\Delta r) = 1 \\
W_{IJ}(r) &= u_{IJ}(r) \\
G_{JIK}(\cos\theta) &= g_{JIK}(\cos\theta)
\end{aligned}$$

The potential reduces to a Rockett-Tersoff potential (*Wang3*) if we set

$$\begin{aligned}
\eta_{ij} &= \delta_{ij}(\eta = 2 \; or \; \eta = 0), \xi_{IJ} = 1 \\
U_{IJ}(r) &= A_{IJ} exp\left(-\lambda_{1,IJ} \cdot r\right) f_{c,IJ}(r) f_{ca,IJ}(r) \\
V_{IJ}(r) &= \left\{ \begin{array}{l} B_{IJ} exp\left(-\lambda_{2,IJ} \cdot r\right) f_{c,IJ}(r) + \\ A_{IJ} exp\left(-\lambda_{1,IJ} \cdot r\right) f_{c,IJ}(r) \left[1 - f_{ca,IJ}(r)\right] \end{array} \right\} \\
F_{IJ}(X) &= \left[1 + \left(\beta_{IJ} X\right)^{n_{IJ}}\right]^{-\frac{1}{2n_{IJ}}} \\
P_{JIK}(\Delta r) &= P_{IK}(\Delta r) = exp\left(\lambda_{3,IK} \cdot \Delta r^3\right) \\
W_{IJ}(r) &= f_{c,IJ}(r) \\
G_{JIK}(\cos\theta) &= 1 + \frac{c_{IK}^2}{d_{IK}^2} - \frac{c_{IK}^2}{d_{IK}^2 + (h_{IK} + \cos\theta)^2}
\end{aligned}$$

where $f_{ca,IJ}(r)$ is similar to the $f_{c,IJ}(r)$ defined above:

$$f_{ca,IJ}(r) = \left\{ \begin{array}{l} 1, r \leq R_{a,IJ} - D_{a,IJ} \\ \frac{1}{2} + \frac{1}{2} cos\left[\frac{\pi\left(r + D_{a,IJ} - R_{a,IJ}\right)}{2D_{a,IJ}}\right], R_{a,IJ} - D_{a,IJ} < r < R_{a,IJ} + D_{a,IJ} \\ 0, r \geq R_{a,IJ} + D_{a,IJ} \end{array} \right.$$

The potential becomes the embedded atom method (*Daw*) if we set

$$\begin{aligned}
\eta_{ij} &= 1 - \delta_{ij}(\eta = 1), \xi_{IJ} = 0 \\
U_{IJ}(r) &= \phi_{IJ}(r) \\
V_{IJ}(r) &= 1 \\
F_{II}(X) &= -2F_I(X) \\
P_{JIK}(\Delta r) &= P_{IK}(\Delta r) = 1 \\
W_{IJ}(r) &= f_J(r) \\
G_{JIK}(\cos\theta) &= 1
\end{aligned}$$

In the embedded atom method case, $\phi_{IJ}(r)$ is the pair energy, $F_I(X)$ is the embedding energy, $X$ is the local electron density, and $f_J(r)$ is the atomic electron density function.

The potential reduces to another type of Tersoff potential (*Zhou4*) if we set

$$\eta_{ij} = \delta_{ij}(\eta = 3), \xi_{IJ} = 1$$

$$U_{IJ}(r) = \frac{D_{e,IJ}}{S_{IJ} - 1} \cdot exp\left[-\beta_{IJ}\sqrt{2S_{IJ}}\left(r - r_{e,IJ}\right)\right] \cdot f_{c,IJ}(r) \cdot T_{IJ}(r) + V_{ZBL,IJ}(r)\left[1 - T_{IJ}(r)\right]$$

$$V_{IJ}(r) = \frac{S_{IJ} \cdot D_{e,IJ}}{S_{IJ} - 1} \cdot exp\left[-\beta_{IJ}\sqrt{\frac{2}{S_{IJ}}}\left(r - r_{e,IJ}\right)\right] \cdot f_{c,IJ}(r) \cdot T_{IJ}(r)$$

$$F_{IJ}(X) = (1 + X)^{-\frac{1}{2}}$$

$$P_{JIK}(\Delta r) = \omega_{JIK} \cdot exp(\alpha_{JIK} \cdot \Delta r)$$

$$W_{IJ}(r) = f_{c,IJ}(r)$$

$$G_{JIK}(\cos\theta) = \gamma_{JIK}\left[1 + \frac{c_{JIK}^2}{d_{JIK}^2} - \frac{c_{JIK}^2}{d_{JIK}^2 + (h_{JIK} + \cos\theta)^2}\right]$$

$$T_{IJ}(r) = \frac{1}{1 + exp\left[-b_{f,IJ}\left(r - r_{f,IJ}\right)\right]}$$

$$V_{ZBL,IJ}(r) = 14.4 \cdot \frac{Z_I \cdot Z_J}{r}\sum_{k=1}^{4}\mu_k \cdot exp\left[-\nu_k\left(Z_I^{0.23} + Z_J^{0.23}\right)r\right]$$

where $f_{c,IJ}(r)$ is the same as defined above. This Tersoff potential differs from the one above because the $P_{JIK}(\Delta r)$ function is now dependent on all three species I, J, and K.

If the tabulated functions are created using the parameters of Stillinger-Weber, Tersoff, and EAM potentials, the polymorphic pair style will produce the same global properties (energies and stresses) and the same forces as the *sw*, *tersoff*, and *eam* pair styles. The polymorphic pair style also produces the same per-atom properties (energies and stresses) as the corresponding *tersoff* and *eam* pair styles. However, due to a different partitioning of global properties to per-atom properties, the polymorphic pair style will produce different per-atom properties (energies and stresses) as the *sw* pair style. This does not mean that polymorphic pair style is different from the sw pair style. It just means that the definitions of the atom energies and atom stresses are different.

Only a single pair_coeff command is used with the polymorphic pair style which specifies a potential file for all needed elements. These are mapped to LAMMPS atom types by specifying N additional arguments after the filename in the pair_coeff command, where N is the number of LAMMPS atom types:

- filename
- N element names = mapping of polymorphic potential elements to atom types

See the pair_coeff page for alternate ways to specify the path for the potential file. Several files for polymorphic potentials are included in the potentials directory of the LAMMPS distribution. They have a "poly" suffix.

As an example, imagine the GaN_tersoff.poly file has tabulated functions for Ga-N tersoff potential. If your LAMMPS simulation has 4 atom types and you want the first 3 to be Ga, and the fourth to be N, you would use the following pair_coeff command:

```
pair_coeff * * GaN_tersoff.poly Ga Ga Ga N
```

The first two arguments must be * * to span all pairs of LAMMPS atom types. The first three Ga arguments map LAMMPS atom types 1,2,3 to the Ga element in the polymorphic file. The final N argument maps LAMMPS atom type 4 to the N element in the polymorphic file. If a mapping value is specified as NULL, the mapping is not performed. This can be used when an polymorphic potential is used as part of the hybrid pair style. The NULL values are placeholders for atom types that will be used with other potentials.

Potential files in the potentials directory of the LAMMPS distribution have a ".poly" suffix. At the beginning of the files, an unlimited number of lines starting with '#' are used to describe the potential and are ignored by LAMMPS. The next line lists two numbers:

```
ntypes eta
```

Here *ntypes* represent total number of species defined in the potential file, $\eta = 1$ reduces to embedded atom method, $\eta = 3$ assumes a three species dependent $P_{JIK}(\Delta r)$ function, and all other values of $\eta$ assume a two species dependent $P_{JK}(\Delta r)$ function. The value of *ntypes* must equal the total number of different species defined in the pair_coeff command. The next *ntypes* lines each lists two numbers and a character string representing atomic number, atomic mass, and name of the species of the ntypes elements:

```
atomic-number atomic-mass element-name(1)
atomic-number atomic-mass element-name(2)
...
atomic-number atomic-mass element-name(ntypes)
```

The next line contains four numbers:

```
nr ntheta nx xmax
```

Here nr is total number of tabular points for radial functions U, V, W, P, ntheta is total number of tabular points for the angular function G, nx is total number of tabular points for the function F, xmax is a maximum value of the argument of function F. Note that the pair functions $U_{IJ}(r)$, $V_{IJ}(r)$, $W_{IJ}(r)$ are uniformly tabulated between 0 and cutoff distance of the IJ pair, $G_{JIK}(\cos\theta)$ is uniformly tabulated between -1 and 1, $P_{JIK}(\Delta r)$ is uniformly tabulated between -rcmax and rcmax where rcmax is the maximum cutoff distance of all pairs, and $F_{IJ}(X)$ is uniformly tabulated between 0 and xmax. Linear extrapolation is assumed if actual simulations exceed these ranges.

The next ntypes*(ntypes+1)/2 lines contain two numbers:

```
cut xi(1)
cut xi(2)
...
cut xi(ntypes*(ntypes+1)/2)
```

Here cut means the cutoff distance of the pair functions, "xi" is $\xi$ as defined in the potential functions above. The ntypes*(ntypes+1)/2 lines are related to the pairs according to the sequence of first ii (self) pairs, i = 1, 2, …, ntypes, and then ij (cross) pairs, i = 1, 2, …, ntypes-1, and j = i+1, i+2, …, ntypes (i.e., the sequence of the ij pairs follows 11, 22, …, 12, 13, 14, …, 23, 24, …).

In the final blocks of the potential file, U, V, W, P, G, and F functions are listed sequentially. First, U functions are given for each of the ntypes*(ntypes+1)/2 pairs according to the sequence described above. For each of the pairs, nr values are listed. Next, similar arrays are given for V and W functions. If P functions depend only on pair species, i.e., $\eta \neq 3$, then P functions are also listed the same way the next. If P functions depend on three species, i.e., $\eta = 3$, then P functions are listed for all the ntypes*ntypes*ntypes IJK triplets in a natural sequence I from 1 to ntypes, J from 1 to ntypes, and K from 1 to ntypes (i.e., IJK = 111, 112, 113, …, 121, 122, 123 …, 211, 212, …). Next, G functions are listed for all the ntypes*ntypes*ntypes IJK triplets similarly. For each of the G functions, ntheta values are listed. Finally, F functions are listed for all the ntypes*(ntypes+1)/2 pairs in the same sequence as described above. For each of the F functions, nx values are listed.

### 4.253.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write their information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

### 4.253.5 Restrictions

If using create_atoms command, atomic masses must be defined in the input script. If using read_data, atomic masses must be defined in the atomic structure data file.

This pair style is part of the MANYBODY package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair potential requires the *newton* setting to be "on" for pair interactions.

The potential files provided with LAMMPS (see the potentials directory) are parameterized for metal *units*. You can use any LAMMPS units, but you would need to create your own potential files.

### 4.253.6 Related commands

*pair_coeff*

---

**(Zhou3)** X. W. Zhou, M. E. Foster, R. E. Jones, P. Yang, H. Fan, and F. P. Doty, J. Mater. Sci. Res., 4, 15 (2015).

**(Zhou4)** X. W. Zhou, M. E. Foster, J. A. Ronevich, and C. W. San Marchi, J. Comp. Chem., 41, 1299 (2020).

**(SW)** F. H. Stillinger, and T. A. Weber, Phys. Rev. B, 31, 5262 (1985).

**(Tersoff)** J. Tersoff, Phys. Rev. B, 39, 5566 (1989).

**(Albe1)** K. Albe, K. Nordlund, J. Nord, and A. Kuronen, Phys. Rev. B, 66, 035205 (2002).

**(Wang)** J. Wang, and A. Rockett, Phys. Rev. B, 43, 12571 (1991).

**(Daw)** M. S. Daw, and M. I. Baskes, Phys. Rev. B, 29, 6443 (1984).

## 4.254 pair_style python command

### 4.254.1 Syntax

```
pair_style python cutoff
```

cutoff = global cutoff for interactions in python potential classes

### 4.254.2 Examples

```
pair_style python 2.5
pair_coeff * * py_pot.LJCutMelt lj

pair_style python 10.0
pair_coeff * * py_pot.HarmonicCut A B

pair_style hybrid/overlay coul/long 12.0 python 12.0
pair_coeff * * coul/long
pair_coeff * * python py_pot.LJCutSPCE OW NULL
```

### 4.254.3 Description

The *python* pair style provides a way to define pairwise additive potential functions as python script code that is loaded into LAMMPS from a python file which must contain specific python class definitions. This allows to rapidly evaluate different potential functions without having to modify and re-compile LAMMPS. Due to python being an interpreted language, however, the performance of this pair style is going to be significantly slower (often between 20x and 100x)

than corresponding compiled code. This penalty can be significantly reduced through generating tabulations from the python code through the *pair_write* command, which is supported by this style.

Only a single *pair_coeff* command is used with the *python* pair style which specifies a python class inside a python module or a file that LAMMPS will look up in the current directory, a folder pointed to by the LAMMPS_POTENTIALS environment variable or somewhere in your python path. A single python module can hold multiple python pair class definitions. The class definitions itself have to follow specific rules that are explained below.

Atom types in the python class are specified through symbolic constants, typically strings. These are mapped to LAMMPS atom types by specifying N additional arguments after the class name in the pair_coeff command, where N must be the number of currently defined atom types:

As an example, imagine a file *py_pot.py* has a python potential class names *LJCutMelt* with parameters and potential functions for a two Lennard-Jones atom types labeled as 'LJ1' and 'LJ2'. In your LAMMPS input and you would have defined 3 atom types, out of which the first two are supposed to be using the 'LJ1' parameters and the third the 'LJ2' parameters, then you would use the following pair_coeff command:

```
pair_coeff * * py_pot.LJCutMelt LJ1 LJ1 LJ2
```

The first two arguments **must** be * * so as to span all LAMMPS atom types. The first two LJ1 arguments map LAMMPS atom types 1 and 2 to the LJ1 atom type in the LJCutMelt class of the py_pot.py file. The final LJ2 argument maps LAMMPS atom type 3 to the LJ2 atom type the python file. If a mapping value is specified as NULL, the mapping is not performed, any pair interaction with this atom type will be skipped. This can be used when a *python* potential is used as part of the *hybrid* or *hybrid/overlay* pair style. The NULL values are then placeholders for atom types that will be used with other potentials.

---

The python potential file has to start with the following code:

```
from __future__ import print_function

class LAMMPSPairPotential(object):
    def __init__(self):
        self.pmap=dict()
        self.units='lj'
    def map_coeff(self,name,ltype):
        self.pmap[ltype]=name
    def check_units(self,units):
        if (units != self.units):
           raise Exception("Conflicting units: %s vs. %s" % (self.units,units))
```

Any classes with definitions of specific potentials have to be derived from this class and should be initialize in a similar fashion to the example given below.

**Note**

The class constructor has to set up a data structure containing the potential parameters supported by this class. It should also define a variable *self.units* containing a string matching one of the options of LAMMPS' *units* command, which is used to verify, that the potential definition in the python class and in the LAMMPS input match.

Here is an example for a single type Lennard-Jones potential class *LJCutMelt* in reduced units, which defines an atom type *lj* for which the parameters epsilon and sigma are both 1.0:

```
class LJCutMelt(LAMMPSPairPotential):
    def __init__(self):
        super(LJCutMelt,self).__init__()
        # set coeffs: 48*eps*sig**12, 24*eps*sig**6,
        #              4*eps*sig**12,  4*eps*sig**6
        self.units = 'lj'
        self.coeff = {'lj'  : {'lj'  : (48.0,24.0,4.0,4.0)}}
```

The class also has to provide two methods for the computation of the potential energy and forces, which have be named *compute_force*, and *compute_energy*, which both take 3 numerical arguments:

- rsq = the square of the distance between a pair of atoms (float)
- itype = the (numerical) type of the first atom
- jtype = the (numerical) type of the second atom

This functions need to compute the (scaled) force and the energy, respectively, and use the result as return value. The functions need to use the *pmap* dictionary to convert the LAMMPS atom type number to the symbolic value of the internal potential parameter data structure. Following the *LJCutMelt* example, here are the two functions:

```
def compute_force(self,rsq,itype,jtype):
     coeff = self.coeff[self.pmap[itype]][self.pmap[jtype]]
     r2inv  = 1.0/rsq
     r6inv  = r2inv*r2inv*r2inv
     lj1 = coeff[0]
     lj2 = coeff[1]
     return (r6inv * (lj1*r6inv - lj2))*r2inv

 def compute_energy(self,rsq,itype,jtype):
     coeff = self.coeff[self.pmap[itype]][self.pmap[jtype]]
     r2inv  = 1.0/rsq
     r6inv  = r2inv*r2inv*r2inv
     lj3 = coeff[2]
     lj4 = coeff[3]
     return (r6inv * (lj3*r6inv - lj4))
```

**Note**

for consistency with the C++ pair styles in LAMMPS, the *compute_force* function follows the conventions of the Pair::single() methods and does not return the pairwise force directly, but the force divided by the distance between the two atoms, so this value only needs to be multiplied by delta x, delta y, and delta z to conveniently obtain the three components of the force vector between these two atoms.

Below is a more complex example using *real* units and defines an interaction equivalent to:

```
units real
pair_style harmonic/cut
pair_coeff 1 1 0.2 9.0
pair_coeff 2 2 0.4 9.0
```

This uses the default geometric mixing. The equivalent input with pair style *python* is:

```
units real
pair_style python 10.0
pair_coeff * * py_pot.Harmonic A B
```

Note that while for pair style *harmonic/cut* the cutoff is implicitly set to the minimum of the harmonic potential, for pair style python a global cutoff must be set and it must be equal or larger to the implicit cutoff of the potential in python, which has to explicitly return zero force and energy beyond the cutoff. Also, the mixed parameters have to be explicitly provided. The corresponding python code is:

```
class Harmonic(LAMMPSPairPotential):
    def __init__(self):
        super(Harmonic,self).__init__()
        self.units = 'real'
        # set coeffs: K, r0
        self.coeff = {'A'  : {'A'  : (0.2,9.0),
                              'B'  : (math.sqrt(0.2*0.4),9.0)},
                      'B'  : {'A'  : (math.sqrt(0.2*0.4),9.0),
                              'B'  : (0.4,9.0)}}

    def compute_force(self,rsq,itype,jtype):
        coeff = self.coeff[self.pmap[itype]][self.pmap[jtype]]
        r = math.sqrt(rsq)
        delta = coeff[1]-r
        if (r < coeff[1]):
            return 2.0*delta*coeff[0]/r
        else:
            return 0.0

    def compute_energy(self,rsq,itype,jtype):
        coeff = self.coeff[self.pmap[itype]][self.pmap[jtype]]
        r = math.sqrt(rsq)
        delta = coeff[1]-r
        if (r < coeff[1]):
            return delta*delta*coeff[0]
        else:
            return 0.0
```

**Performance Impact**

The evaluation of scripted python code will slow down the computation of pairwise interactions quite significantly. However, this performance penalty can be worked around through using the python pair style not for the actual simulation, but to generate tabulated potentials using the *pair_write* command. This will also enable GPU or multi-thread acceleration through the GPU, KOKKOS, or OPENMP package versions of the *table* pair style. Please see below for a LAMMPS input example demonstrating how to build a table file:

```
pair_style python 2.5
pair_coeff * * py_pot.LJCutMelt lj
shell rm -f lj.table
pair_write  1 1 2000 rsq 0.01 2.5 lj.table lj
```

Note that it is strongly recommended to try to **delete** the potential table file before generating it. Since the *pair_write* command will always **append** to a table file, while pair style table will use the **first match**. Thus when changing the potential function in the python class, the table pair style will still read the old variant unless the table file is first deleted.

After switching the pair style to *table*, the potential tables need to be assigned to the LAMMPS atom types like this:

```
pair_style      table linear 2000
pair_coeff      1  1  lj.table lj
```

This can also be done for more complex systems. Please see the *examples/python* folders for a few more examples.

---

### 4.254.4 Mixing, shift, table, tail correction, restart, rRESPA info

Mixing of potential parameters has to be handled inside the provided python module. The python pair style simply assumes that force and energy computation can be correctly performed for all pairs of atom types as they are mapped to the atom type labels inside the python potential class.

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.254.5 Restrictions

This pair style is part of the PYTHON package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.254.6 Related commands

*pair_coeff* , *pair_write*, *pair style table*

### 4.254.7 Default

none

## 4.255 pair_style quip command

### 4.255.1 Syntax

```
pair_style quip
```

### 4.255.2 Examples

```
pair_style      quip
pair_coeff      * * gap_example.xml "Potential xml_label=GAP_2014_5_8_60_17_10_38_466" 14
pair_coeff      * * sw_example.xml "IP SW" 14
```

### 4.255.3 Description

Style *quip* provides an interface for calling potential routines from the QUIP package. QUIP is built separately, and then linked to LAMMPS. The most recent version of the QUIP package can be downloaded from GitHub: https://github.com/libAtoms/QUIP. The interface is chiefly intended to be used to run Gaussian Approximation Potentials (GAP), which are described in the following publications: *(Bartok et al)* and *(PhD thesis of Bartok)*.

Only a single pair_coeff command is used with the *quip* style that specifies a QUIP potential file containing the parameters of the potential for all needed elements in XML format. This is followed by a QUIP initialization string. Finally, the QUIP elements are mapped to LAMMPS atom types by specifying N atomic numbers, where N is the number of LAMMPS atom types:

- QUIP filename
- QUIP initialization string
- N atomic numbers = mapping of QUIP elements to atom types

See the *pair_coeff* page for alternate ways to specify the path for the potential file.

A QUIP potential is fully specified by the filename which contains the parameters of the potential in XML format, the initialization string, and the map of atomic numbers.

GAP potentials can be obtained from the GAP models and databases page on the libAtoms homepage `https://libatoms.github.io, where the appropriate initialization strings are also advised. The list of atomic numbers must be matched to the LAMMPS atom types specified in the LAMMPS data file or elsewhere.

Two examples input scripts are provided in the examples/PACKAGES/quip directory.

### 4.255.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support the *pair_modify* mix, shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.255.5 Restrictions

This pair style is part of the ML-QUIP package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

QUIP potentials are parameterized in electron-volts and Angstroms and therefore should be used with LAMMPS metal *units*.

QUIP potentials are generally not designed to work with the scaling factors set by the *special_bonds* command. The recommended setting in molecular systems is to include all interactions, i.e. to use *special_bonds lj/coul 1.0 1.0 1.0*. Scaling factors > 0.0 will be ignored and treated as 1.0. The only exception to this rule is if you know that your QUIP potential needs to exclude bonded, 1-3, or 1-4 interactions and does not already do this exclusion within QUIP. Then a factor 0.0 needs to be used which will remove such pairs from the neighbor list. This needs to be very carefully tested, because it may remove pairs from the neighbor list that are still required.

### 4.255.6 Related commands

*pair_coeff*

---

**(Bartok2010)** AP Bartok, MC Payne, R Kondor, and G Csanyi, Physical Review Letters 104, 136403 (2010).

**(Bartok_PhD)** A Bartok-Partay, PhD Thesis, University of Cambridge, (2010).

# 4.256 pair_style rann command

## 4.256.1 Syntax

```
pair_style rann
pair_coeff file Type1_element Type2_element Type3_element...
```

## 4.256.2 Examples

```
pair_style rann
pair_coeff * * Mg.rann Mg
pair_coeff * * MgAlalloy.rann Mg Mg Al Mg
```

## 4.256.3 Description

Pair style *rann* computes pairwise interactions for a variety of materials using rapid atomistic neural network (RANN) potentials (*Dickel* , *Nitol*). Neural network potentials work by first generating a series of symmetry functions i.e. structural fingerprints from the neighbor list and then using these values as the input layer of a neural network. There is a single output neuron in the final layer which is the energy. Atomic forces are found by analytical derivatives computed via back-propagation. For alloy systems, each element has a unique network.

## 4.256.4 Potential file syntax

The RANN potential is defined by a single text file which contains all the fitting parameters for the alloy system. The potential file also defines the active fingerprints, network architecture, activation functions, etc. The potential file is divided into several sections which are identified by one of the following keywords:

- atomtypes
- mass
- fingerprintsperelement
- fingerprints
- fingerprintconstants
- screening (optional)
- networklayers
- layersize
- weight
- bias
- activationfunctions
- calibrationparameters (ignored)

The '#' character is treated as a comment marker, similar to LAMMPS input scripts. Sections are not required to follow a rigid ordering, but do require previous definition of prerequisite information. E.g., fingerprintconstants for a particular fingerprint must follow the fingerprints definition; layersize for a particular layer must follow the declaration of network layers.

*atomtypes* are defined as follows using element keywords separated by spaces.

```
atomtypes:
Fe Mg Al etc.
```

*mass* must be specified for each element keyword as follows:

```
mass:Mg:
24.305
mass:Fe:
55.847
mass:Al:
26.982
```

*fingerprintsperelement* specifies how many fingerprints are active for computing the energy of a given atom. This number must be specified for each element keyword. Active elements for each fingerprint depend upon the type of the central atom and the neighboring atoms. Pairwise fingerprints may be defined for a Mg atom based exclusively on its Al neighbors, for example. Bond fingerprints may use two neighbor lists of different element types. In computing fingerprintsperelement from all defined fingerprints, only the fingerprints defined for atoms of a particular element should be considered, regardless of the elements used in its neighbor list. In the following code, for example, some fingerprints may compute pairwise fingerprints summing contributions about Fe atoms based on a neighbor list of exclusively Al atoms, but if there are no fingerprints summing contributions of all neighbors about a central Al atom, then fingerprintsperelement of Al is zero:

```
fingerprintsperelement:Mg:
5
fingerprintsperelement:Fe:
2
fingerprintsperelement:Al:
0
```

*fingerprints* specifies the active fingerprints for a certain element combination. Pair fingerprints are specified for two elements, while bond fingerprints are specified for three elements. Only one fingerprints header should be used for an individual combination of elements. The ordering of the fingerprints in the network input layer is determined by the order of element combinations specified by subsequent *fingerprints* lines, and the order of the fingerprints defined for each element combination. Multiple fingerprints of the same style or different ones may be specified. If the same style and element combination is used for multiple fingerprints, they should have different id numbers. The first element specifies the atoms for which this fingerprint is computed while the other(s) specify which atoms to use in the neighbor lists for the computation. Switching the second and third element type in bond fingerprints has no effect on the computation:

```
fingerprints:Mg_Mg:
radial_0 radialscreened_0 radial_1
fingerprints:Mg_Al_Fe:
bond_0 bondspin_0
fingerprints:Mg_Al:
radial_0 radialscreened_0
```

The following fingerprint styles are currently defined. See the *formulation section* below for their definitions:

- radial
- radialscreened
- radialspin
- radialscreenedspin
- bond
- bondscreened
- bondspin

- bondscreenedspin

*fingerprintconstants* specifies the meta-parameters for a defined fingerprint. For all radial styles, re, rc, alpha, dr, o, and n must be specified. re should usually be the stable interatomic distance, rc is the cutoff radius, dr is the cutoff smoothing distance, o is the lowest radial power term (which may be negative), and n is the highest power term. The total length of the fingerprint vector is (n-o+1). alpha is a list of decay parameters used for exponential decay of radial contributions. It may be set proportionally to the bulk modulus similarly to MEAM potentials, but other values may provided better fitting in special cases. Bond style fingerprints require specification of re, rc, alphak, dr, k, and m. Here m is the power of the bond cosines and k is the number of decay parameters. Cosine powers go from 0 to m-1 and are each computed for all values of alphak. Thus the total length of the fingerprint vector is m*k.

```
fingerprintconstants:Mg_Mg:radialscreened_0:re:
3.193592
fingerprintconstants:Mg_Mg:radialscreened_0:rc:
6.000000
fingerprintconstants:Mg_Mg:radialscreened_0:alpha:
5.520000 5.520000 5.520000 5.520000 5.520000
fingerprintconstants:Mg_Mg:radialscreened_0:dr:
2.806408
fingerprintconstants:Mg_Mg:radialscreened_0:o:
-1
fingerprintconstants:Mg_Mg:radialscreened_0:n:
3
```

*screening* specifies the Cmax and Cmin values used in the screening fingerprints. Contributions form neighbors to the fingerprint are omitted if they are blocked by a closer neighbor, and reduced if they are partially blocked. Larger values of Cmin correspond to neighbors being blocked more easily. Cmax cannot be greater than 3, and Cmin cannot be greater than Cmax or less than zero. Screening may be omitted in which case the default values Cmax = 2.8, Cmin = 0.8 are used. Since screening is a bond computation, it is specified separately for each combination of three elements in which the latter two may be interchanged with no effect.

```
screening:Mg_Mg_Mg:Cmax:
2.700000
screening:Mg_Mg_Mg:Cmin:
0.400000
```

*networklayers* species the size of the neural network for each atom. It counts both the input and output layer and so is 2 + <hidden layers>.

```
networklayers:Mg:
3
```

*layersize* specifies the length of each layer, including the input layer and output layer. The input layer is layer 0. The size of the input layer size must match the summed length of all the fingerprints for that element, and the output layer size must be 1:

```
layersize:Mg:0:
14
layersize:Mg:1:
20
layersize:Mg:2:
1
```

*weight* specifies the weight for a given element and layer. Weight cannot be specified for the output layer. The weight of layer i is a *m* x *n* matrix where *m* is the layer size of *i* and *n* is the layer size of *i*+1:

```
weight:Mg:0:
w11 w12 w13 ...
w21 w22 w23 ...
...
```

*bias* specifies the bias for a given element and layer. Bias cannot be specified for the output layer. The bias of layer i is a nx1 vector where n is the layer size of i+1:

```
bias:Mg:0:
b1
b2
b3
...
```

*activationfunctions* specifies the activation function for a given element and layer. Activation functions cannot be specified for the output layer:

```
activationfunctions:Mg:0:
sigI
activationfunctions:Mg:1:
linear
```

The following activation styles are currently specified. See the *formulation section* below for their definitions.

- sigI
- linear

*calibrationparameters* specifies a number of parameters used to calibrate the potential. These are ignored by LAMMPS.

### 4.256.5 Formulation

In the RANN formulation, the total energy of a system of atoms is given by:

$$E = \sum_{\alpha} E^{\alpha}$$

$$E^{\alpha} = {}^{N}A^{\alpha}$$

$${}^{n+1}A_i^{\alpha} = {}^{n}F\left({}^{n}W_{ij}\ {}^{n}A_j^{\alpha} + {}^{n}B_i\right)$$

$${}^{0}A_i^{\alpha} = \begin{bmatrix} {}^{1}Sf^{\alpha} \\ {}^{2}Sf^{\alpha} \\ \cdots \end{bmatrix}$$

Here $E^{\alpha}$ is the energy of atom $\alpha$, ${}^{n}F()$, ${}^{n}W_{ij}$ and ${}^{n}B_i$ are the activation function, weight matrix and bias vector of the n-th layer respectively. The inputs to the first layer are a collection of structural fingerprints which are collected and reshaped into a single long vector. The individual fingerprints may be defined in any order and have various shapes and sizes. Multiple fingerprints of the same type and varying parameters may also be defined in the input layer.

Eight types of structural fingerprints are currently defined. In the following, $\beta$ and $\gamma$ span the full neighbor list of atom $\alpha$. $\delta_i$ are decay meta-parameters, and $r_e$ is a meta-parameter roughly proportional to the first neighbor distance. $r_c$ and $dr$ are the neighbor cutoff distance and cutoff smoothing distance respectively. $S^{\alpha\beta}$ is the MEAM screening function *(Baskes)*, $s_i^{\alpha}$ and $s_i^{\beta}$ are the atom spin vectors *(Tranchida)*. $r^{\alpha\beta}$ is the distance from atom $\alpha$ to atom $\beta$, and $\theta^{\alpha\beta\gamma}$ is the

bond angle:

$$cos\left(\theta^{\alpha\beta\gamma}\right) = \frac{\mathbf{r}^{\alpha\beta} \cdot \mathbf{r}^{\alpha\gamma}}{r^{\alpha\beta} r^{\alpha\gamma}}$$

$S^{\alpha\beta}$ is defined as *(Baskes)*:

$$X^{\gamma\beta} = \left(\frac{r^{\gamma\beta}}{r^{\alpha\beta}}\right)^2$$

$$X^{\alpha\gamma} = \left(\frac{r^{\alpha\gamma}}{r^{\alpha\beta}}\right)^2$$

$$C = \frac{2\left(X^{\alpha\gamma} + X^{\gamma\beta}\right) - \left(X^{\alpha\gamma} - X^{\gamma\beta}\right)^2 - 1}{1 - \left(X^{\alpha\gamma} - X^{\gamma\beta}\right)^2}$$

$$f_c(x) = \begin{cases} 1 & x \geq 1 \\ \left(1 - (1-x)^4\right)^2 & 0 < x < 1 \\ 0 & x \leq 0 \end{cases}$$

$$S^{\alpha\beta\gamma} = f_c\left(\frac{C - C_{min}}{C_{max} - C_{min}}\right)$$

$$S^{\alpha\beta} = \prod_{\gamma} S^{\alpha\beta\gamma}$$

The structural fingerprints are computed as follows:

- **radial**

$$^{r}Sf_i^{\alpha} = \sum_{\beta} \left(\frac{r^{\alpha\beta}}{r_e}\right)^i e^{-\delta_i \frac{r^{\alpha\beta}}{r_e}} f_c\left(\frac{r_c - r^{\alpha\beta}}{dr}\right)$$

- **bond**

$$^{b}Sf_{ij}^{\alpha} = \sum_{\beta}\sum_{\gamma} \left(cos(\theta_{\alpha\beta\gamma})\right)^i e^{-\delta_j \frac{r^{\alpha\beta}}{r_e}} e^{-\delta_j \frac{r^{\alpha\gamma}}{r_e}} f_c\left(\frac{r_c - r^{\alpha\beta}}{dr}\right) f_c\left(\frac{r_c - r^{\alpha\gamma}}{dr}\right)$$

- **radialscreened**

$$^{rsc}Sf_i^{\alpha} = \sum_{\beta} \left(\frac{r^{\alpha\beta}}{r_e}\right)^i e^{-\delta_i \frac{r^{\alpha\beta}}{r_e}} S^{\alpha\beta} f_c\left(\frac{r_c - r^{\alpha\beta}}{dr}\right)$$

- **bondscreened**

$$^{bsc}Sf_{ij}^{\alpha} = \sum_{\beta}\sum_{\gamma} \left(cos(\theta_{\alpha\beta\gamma})\right)^i e^{-\delta_j \frac{r^{\alpha\beta}}{r_e}} e^{-\delta_j \frac{r^{\alpha\gamma}}{r_e}} S^{\alpha\beta} S^{\alpha\gamma} f_c\left(\frac{r_c - r^{\alpha\beta}}{dr}\right) f_c\left(\frac{r_c - r^{\alpha\gamma}}{dr}\right)$$

- **radialspin**

$$^{rsp}Sf_i^{\alpha} = \sum_{\beta} \left(\frac{r^{\alpha\beta}}{r_e}\right)^i e^{-\delta_i \frac{r^{\alpha\beta}}{r_e}} \left(\mathbf{s}^{\alpha} \cdot \mathbf{s}^{\beta}\right) f_c\left(\frac{r_c - r^{\alpha\beta}}{dr}\right)$$

- **bondspin**

$$^{bsp}Sf_{ij}^{\alpha} = \sum_{\beta}\sum_{\gamma}\left(cos(\theta_{\alpha\beta\gamma})\right)^{i} e^{-\delta_j \frac{r^{\alpha\beta}}{r_e}} e^{-\delta_j \frac{r^{\alpha\gamma}}{r_e}} \left(\mathbf{s}^{\alpha}\cdot\mathbf{s}^{\beta}\right)\left(\mathbf{s}^{\alpha}\cdot\mathbf{s}^{\gamma}\right) f_c\left(\frac{r_c - r^{\alpha\beta}}{dr}\right) f_c\left(\frac{r_c - r^{\alpha\gamma}}{dr}\right)$$

- **radialscreenedspin**

$$^{rscsp}Sf_{i}^{\alpha} = \sum_{\beta}\left(\frac{r^{\alpha\beta}}{r_e}\right)^{i} e^{-\delta_i \frac{r^{\alpha\beta}}{r_e}} S^{\alpha\beta}\left(\mathbf{s}^{\alpha}\cdot\mathbf{s}^{\beta}\right) f_c\left(\frac{r_c - r^{\alpha\beta}}{dr}\right)$$

- **bondscreenedspin**

$$^{bscsp}Sf_{ij}^{\alpha} = \sum_{\beta}\sum_{\gamma}\left(cos(\theta_{\alpha\beta\gamma})\right)^{i} e^{-\delta_j \frac{r^{\alpha\beta}}{r_e}} e^{-\delta_j \frac{r^{\alpha\gamma}}{r_e}} S^{\alpha\beta}S^{\alpha\gamma}\left(\mathbf{s}^{\alpha}\cdot\mathbf{s}^{\beta}\right)\left(\mathbf{s}^{\alpha}\cdot\mathbf{s}^{\gamma}\right) f_c\left(\frac{r_c - r^{\alpha\beta}}{dr}\right) f_c\left(\frac{r_c - r^{\alpha\gamma}}{dr}\right)$$

The activation functions are computed as follows:

- **sigI**

$$F^{sigI}(x) = 0.1x + 0.9ln\left(e^{x} + 1\right)$$

- **linear**

$$F^{linear}(x) = x$$

### 4.256.6 Restrictions

Pair style *rann* is part of the ML-RANN package. It is only enabled if LAMMPS was built with that package. Additionally, if any spin fingerprint styles are used LAMMPS must be built with the SPIN package as well.

Pair style *rann* does not support computing per-atom stress or using *pair_modify nofdotr*.

### 4.256.7 Defaults

Cmin = 0.8, Cmax = 2.8.

---

**(Baskes)** Baskes, Materials Chemistry and Physics, 50(2), 152-158, (1997).

**(Dickel)** Dickel, Francis, and Barrett, Computational Materials Science 171 (2020): 109157.

**(Nitol)** Nitol, Dickel, and Barrett, Computational Materials Science 188 (2021): 110207.

**(Tranchida)** Tranchida, Plimpton, Thibaudeau and Thompson, Journal of Computational Physics, 372, 406-425, (2018).

## 4.257 pair_style reaxff command

Accelerator Variants: *reaxff/kk*, *reaxff/omp*

### 4.257.1 Syntax

```
pair_style reaxff cfile keyword value
```

- cfile = NULL or name of a control file
- zero or more keyword/value pairs may be appended

```
keyword = checkqeq or lgvdw or safezone or mincap or minhbonds or tabulate or list/
 ↪blocking
  checkqeq value = yes or no = whether or not to require one of fix qeq/reaxff, fix␣
 ↪acks2/reaxff or fix qtpie/reaxff
  enobonds value = yes or no = whether or not to tally energy of atoms with no bonds
  lgvdw value = yes or no = whether or not to use a low gradient vdW correction
  safezone = factor used for array allocation
  mincap = minimum size for array allocation
  minhbonds = minimum size use for storing hydrogen bonds
  tabulate value = size of interpolation table for Lennard-Jones and Coulomb␣
 ↪interactions
  list/blocking value = yes or no = whether or not to use "blocking" scheme for␣
 ↪bond list build
```

### 4.257.2 Examples

```
pair_style reaxff NULL
pair_style reaxff controlfile checkqeq no
pair_style reaxff NULL lgvdw yes
pair_style reaxff NULL safezone 1.6 mincap 100
pair_coeff * * ffield.reax C H O N
```

### 4.257.3 Description

Pair style *reaxff* computes the ReaxFF potential of van Duin, Goddard and co-workers. ReaxFF uses distance-dependent bond-order functions to represent the contributions of chemical bonding to the potential energy. There is more than one version of ReaxFF. The version implemented in LAMMPS uses the functional forms documented in the supplemental information of the following paper: *(Chenoweth et al., 2008)* and matches the version of the reference ReaxFF implementation from Summer 2010. For more technical details about the implementation of ReaxFF in pair style *reaxff*, see the *(Aktulga)* paper. The *reaxff* style was initially implemented as a stand-alone C code and is now converted to C++ and integrated into LAMMPS as a package.

The *reaxff/kk* style is a Kokkos version of the ReaxFF potential that is derived from the *reaxff* style. The Kokkos version can run on GPUs and can also use OpenMP multithreading. For more information about the Kokkos package, see *Packages details* and *Speed kokkos* doc pages. One important consideration when using the *reaxff/kk* style is the choice of either half or full neighbor lists. This setting can be changed using the Kokkos *package* command.

The *reaxff* style differs from the (obsolete) "pair_style reax" command in the implementation details. The *reax* style was a Fortran library, linked to LAMMPS. The *reax* style has been removed from LAMMPS after the 12 December 2018 version.

LAMMPS provides several different versions of ffield.reax in its potentials dir, each called potentials/ffield.reax.label. These are documented in potentials/README.reax.

The format of these files is identical to that used originally by van Duin. We have tested the accuracy of *pair_style reaxff* potential against the original ReaxFF code for the systems mentioned above. You can use other ffield files for specific chemical systems that may be available elsewhere (but note that their accuracy may not have been tested).

**Note**

We do not distribute a wide variety of ReaxFF force field files with LAMMPS. Adri van Duin's group at PSU is the central repository for this kind of data as they are continuously deriving and updating parameterizations for different classes of materials. You can submit a contact request at the Materials Computation Center (MCC) website https://www.mri.psu.edu/materials-computation-center/connect-mcc, describing the material(s) you are interested

in modeling with ReaxFF. They can tell you what is currently available or what it would take to create a suitable ReaxFF parameterization.

The *cfile* setting can be specified as NULL, in which case default settings are used. A control file can be specified which defines values of control variables. Some control variables are global parameters for the ReaxFF potential. Others define certain performance and output settings. Each line in the control file specifies the value for a control variable. The format of the control file is described below.

**Note**

The LAMMPS default values for the ReaxFF global parameters correspond to those used by Adri van Duin's stand-alone serial code. If these are changed by setting control variables in the control file, the results from LAMMPS and the serial code will not agree.

Examples using *pair_style reaxff* are provided in the examples/reax directory and its subdirectories.

Use of this pair style requires using an *atom_style* that includes a per-atom charge property *or* using *fix property/atom q*. Charges can be set via *read_data* or *set*. Using an initial charge that is close to the result of charge equilibration will speed up that process.

The ReaxFF parameter files provided were created using a charge equilibration (QEq) model for handling the electrostatic interactions. Therefore, by default, LAMMPS requires that *fix qeq/reaxff* or *fix qeq/shielded* or *fix acks2/reaxff* or *fix qtpie/reaxff* is used with *pair_style reaxff* when simulating a ReaxFF model, to equilibrate the charges at each timestep. See the *fix qeq/reaxff* or *fix qeq/shielded* or *fix acks2/reaxff* or *fix qtpie/reaxff* command documentation for more details.

Using the keyword *checkqeq* with the value *no* turns off the check for the QEq fixes, allowing a simulation to be run without charge equilibration. In this case, the static charges you assign to each atom will be used for computing the electrostatic interactions in the system.

Using the optional keyword *lgvdw* with the value *yes* turns on the low-gradient correction of ReaxFF for long-range London Dispersion, as described in the *(Liu)* paper. The bundled force field file *ffield.reax.lg* is designed for this correction, and is trained for several energetic materials (see "Liu"). When using *lgvdw yes*, the recommended value for parameter *thb* is 0.01, which can be set in the control file. Note: Force field files are different for the original or lg corrected pair styles, using the wrong ffield file generates an error.

Using the optional keyword *enobonds* with the value *yes*, the energy of atoms with no bonds (i.e. isolated atoms) is included in the total potential energy and the per-atom energy of that atom. If the value *no* is specified then the energy of atoms with no bonds is set to zero. The latter behavior is usual not desired, as it causes discontinuities in the potential energy when the bonding of an atom drops to zero.

Optional keywords *safezone*, *mincap*, and *minhbonds* are used for allocating reaxff arrays. Increasing these values can avoid memory problems, such as segmentation faults and bondchk failed errors, that could occur under certain conditions. These keywords are **not** used by the Kokkos version, which instead uses a more robust memory allocation scheme that checks if the sizes of the arrays have been exceeded and automatically allocates more memory.

**Memory management problems with ReaxFF**

The LAMMPS implementation of ReaxFF is adapted from a standalone MD program written in C called PuReMD. It inherits from this code a heuristic memory management that is different from what the rest of LAMMPS uses. It assumes that a system is dense and already well equilibrated, so that there are no large changes in how many and what types of neighbors atoms have. However, not all systems are like that, and thus there can be errors or segmentation faults if the system changes too much. If you run into problems, here are three options to avoid them:

- Use the KOKKOS version of ReaxFF (KOKKOS is not only for GPUs, but can also be compiled for serial or OpenMP execution) which uses a different memory management approach.
- Break down a run command during which memory related errors happen into multiple smaller segments so that the memory management heuristics are re-initialized for each segment before they become invalid.
- Increase the values for *safezone*, *mincap*, and *minhbonds* as needed. This can lead to significant increase of memory consumption through.

The keyword *tabulate* controls the size of interpolation table for Lennard-Jones and Coulomb interactions. Tabulation may also be set in the control file (see below). If tabulation is set in both the input script and the control file, the value in the control file will be ignored. A size of 10000 is typically used for the interpolation table. A value of 0 means no tabulation will be used.

The keyword *list/blocking* is only supported by the Kokkos version of ReaxFF and ignored otherwise. Setting the value to *yes* enables the "blocking" scheme (dynamically building interaction lists) for the ReaxFF bond neighbor list. This reduces the number of empty interactions and can improve performance in some cases (e.g. large number of atoms/GPU on AMD hardware). It is also enabled by default when running the CPU with Kokkos.

The thermo variable *evdwl* stores the sum of all the ReaxFF potential energy contributions, with the exception of the Coulombic and charge equilibration contributions which are stored in the thermo variable *ecoul*. The output of these quantities is controlled by the *thermo* command.

This pair style tallies a breakdown of the total ReaxFF potential energy into sub-categories, which can be accessed via the *compute pair* command as a vector of values of length 14. The 14 values correspond to the following sub-categories (the variable names in italics match those used in the original FORTRAN ReaxFF code):

1. *eb* = bond energy
2. *ea* = atom energy
3. *elp* = lone-pair energy
4. *emol* = molecule energy (always 0.0)
5. *ev* = valence angle energy
6. *epen* = double-bond valence angle penalty
7. *ecoa* = valence angle conjugation energy
8. *ehb* = hydrogen bond energy
9. *et* = torsion energy
10. *eco* = conjugation energy
11. *ew* = van der Waals energy
12. *ep* = Coulomb energy
13. *efi* = electric field energy (always 0.0)
14. *eqeq* = charge equilibration energy

To print these quantities to the log file (with descriptive column headings) the following commands could be included in an input script:

```
compute reax all pair reaxff
variable eb      equal c_reax[1]
variable ea      equal c_reax[2]
[...]
```

(continues on next page)

(continued from previous page)

```
variable eqeq    equal c_reax[14]
thermo_style custom step temp epair v_eb v_ea [...] v_eqeq
```

Only a single pair_coeff command is used with the *reaxff* style which specifies a ReaxFF potential file with parameters for all needed elements. These are mapped to LAMMPS atom types by specifying N additional arguments after the filename in the pair_coeff command, where N is the number of LAMMPS atom types:

- filename
- N indices = ReaxFF elements

The filename is the ReaxFF potential file.

In the ReaxFF potential file, near the top, after the general parameters, is the atomic parameters section that contains element names, each with a couple dozen numeric parameters. If there are M elements specified in the *ffield* file, think of these as numbered 1 to M. Each of the N indices you specify for the N atom types of LAMMPS atoms must be an integer from 1 to M. Atoms with LAMMPS type 1 will be mapped to whatever element you specify as the first index value, etc. If a mapping value is specified as NULL, the mapping is not performed. This can be used when the *reaxff* style is used as part of the *hybrid* pair style. The NULL values are placeholders for atom types that will be used with other potentials.

As an example, say your LAMMPS simulation has 4 atom types and the elements are ordered as C, H, O, N in the *ffield* file. If you want the LAMMPS atom type 1 and 2 to be C, type 3 to be N, and type 4 to be H, you would use the following pair_coeff command:

```
pair_coeff * * ffield.reax C C N H
```

---

### 4.257.4 Control file

The format of a line in the control file is as follows:

```
variable_name value
```

and it may be followed by an "!" character and a trailing comment.

If the value of a control variable is not specified, then default values are used. What follows is the list of variables along with a brief description of their use and default values.

***simulation_name***
: Output files produced by *pair_style reaxff* carry this name + extensions specific to their contents. Partial energies are reported with a ".pot" extension, while the trajectory file has ".trj" extension.

***tabulate_long_range***
: To improve performance, long range interactions can optionally be tabulated (0 means no tabulation). Value of this variable denotes the size of the long range interaction table. The range from 0 to long range cutoff (defined in the *ffield* file) is divided into *tabulate_long_range* points. Then at the start of simulation, we fill in the entries of the long range interaction table by computing the energies and forces resulting from van der Waals and Coulomb interactions between every possible atom type pairs present in the input system. During the simulation we consult to the long range interaction table to estimate the energy and forces between a pair of atoms. Linear interpolation is used for estimation. (default value = 0)

***energy_update_freq***
: Denotes the frequency (in number of steps) of writes into the partial energies file. (default value = 0)

***nbrhood_cutoff***
: Denotes the near neighbors cutoff (in Angstroms) regarding the bonded interactions. (default value = 5.0)

***hbond_cutoff***
: Denotes the cutoff distance (in Angstroms) for hydrogen bond interactions.(default value = 7.5. A value of 0.0 turns off hydrogen bonds)

***bond_graph_cutoff***
: is the threshold used in determining what is a physical bond, what is not. Bonds and angles reported in the trajectory file rely on this cutoff. (default value = 0.3)

***thb_cutoff***
: cutoff value for the strength of bonds to be considered in three body interactions. (default value = 0.001)

***thb_cutoff_sq***
: cutoff value for the strength of bond order products to be considered in three body interactions. (default value = 0.00001)

***write_freq***
: Frequency of writes into the trajectory file. (default value = 0)

***traj_title***
: Title of the trajectory - not the name of the trajectory file.

***atom_info***
: 1 means print only atomic positions + charge (default = 0)

***atom_forces***
: 1 adds net forces to atom lines in the trajectory file (default = 0)

***atom_velocities***
: 1 adds atomic velocities to atoms line (default = 0)

***bond_info***
: 1 prints bonds in the trajectory file (default = 0)

***angle_info***
: 1 prints angles in the trajectory file (default = 0)

---

### 4.257.5 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support the *pair_modify* mix, shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.257.6 Restrictions

This pair style is part of the REAXFF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The ReaxFF potential files provided with LAMMPS in the potentials directory are parameterized for *real units*. You can use the ReaxFF pair style with any LAMMPS units, but you would need to create your own potential file with coefficients listed in the appropriate units if your simulation does not use "real" units.

### 4.257.7 Related commands

*pair_coeff* , *fix qeq/reaxff* , *fix acks2/reaxff* , *fix qtpie/reaxff* , *fix reaxff/bonds*, *fix reaxff/species*, *compute reaxff/atom*

### 4.257.8 Default

The keyword defaults are checkqeq = yes, enobonds = yes, lgvdw = no, safezone = 1.2, mincap = 50, minhbonds = 25, tabulate = 0, list/blocking = yes on CPU, no on GPU.

---

**(Chenoweth_2008)** Chenoweth, van Duin and Goddard, Journal of Physical Chemistry A, 112, 1040-1053 (2008).

(Aktulga) Aktulga, Fogarty, Pandit, Grama, Parallel Computing, 38, 245-259 (2012).

**(Liu)** L. Liu, Y. Liu, S. V. Zybin, H. Sun and W. A. Goddard, Journal of Physical Chemistry A, 115, 11016-11022 (2011).

## 4.258 pair_style rebomos command

Accelerator Variants: *rebomos/omp*

### 4.258.1 Syntax

```
pair_style rebomos
```

- rebomos = name of this pair style

### 4.258.2 Examples

```
pair_style rebomos
pair_coeff * * ../potentials/MoS.rebomos Mo S
```

Example input scripts available: examples/threebody/

### 4.258.3 Description

Added in version 17Apr2024.

The *rebomos* pair style computes the interactions between molybdenum and sulfur atoms *(Stewart)* utilizing an adaptive interatomic reactive empirical bond order potential that is similar in form to the AIREBO potential *(Stuart)*. The potential is based on an earlier parameterizations for $MoS_2$ developed by *(Liang)*.

The REBOMoS potential consists of two terms:

$$E = \frac{1}{2} \sum_i \sum_{j \neq i} \left[ E_{ij}^{\mathrm{REBO}} + E_{ij}^{\mathrm{LJ}} \right]$$

The $E^{\mathrm{REBO}}$ term describes the covalently bonded interactions between Mo and S atoms while the $E^{\mathrm{LJ}}$ term describes longer range dispersion forces between layers. A cubic spline function is applied to smoothly switch between covalent bonding at short distances to dispersion interactions at longer distances. This allows the model to capture bond formation and breaking events which may occur between adjacent MoS2 layers, edges, defects, and more.

---

Only a single pair_coeff command is used with the *rebomos* pair style which specifies an REBOMoS potential file with parameters for Mo and S. These are mapped to LAMMPS atom types by specifying N additional arguments after the filename in the pair_coeff command, where N is the number of LAMMPS atom types:

- filename
- *N* element names = mapping of REBOMoS elements to atom types

See the *pair_coeff* page for alternate ways to specify the path for the potential file.

As an example, if your LAMMPS simulation has three atom types and you want the first two to be Mo, and the third to be S, you would use the following pair_coeff command:

```
pair_coeff * * MoS.rebomos Mo Mo S
```

The first 2 arguments must be * * so as to span all LAMMPS atom types. The first two Mo arguments map LAMMPS atom types 1 and 2 to the Mo element in the REBOMoS file. The final S argument maps LAMMPS atom type 3 to the S element in the REBOMoS file. If a mapping value is specified as NULL, the mapping is not performed. This can be used when a *rebomos* potential is used as part of the *hybrid* pair style. The NULL values are placeholders for atom types that will be used with other potentials.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.258.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support the *pair_modify* mix, shift, table, and tail options.

This pair style does not write their information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair styles can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.258.5 Restrictions

This pair style is part of the MANYBODY package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

These pair potentials require the *newton* setting to be "on" for pair interactions.

The MoS.rebomos potential file provided with LAMMPS (see the potentials directory) is parameterized for metal *units*. You can use the *rebomos* pair style with any LAMMPS units setting, but you would need to create your own REBOMoS potential file with coefficients listed in the appropriate units.

The pair style provided here **only** supports potential files parameterized for the elements molybdenum and sulfur (designated with "Mo" and "S" in the *pair_coeff* command. Using potential files for other elements will trigger an error.

### 4.258.6 Related commands

*pair_coeff*, *pair style rebo*

### 4.258.7 Default

none

---

**(Steward)** Stewart, Spearot, Modelling Simul. Mater. Sci. Eng. 21, 045003, (2013).

**(Stuart)** Stuart, Tutein, Harrison, J Chem Phys, 112, 6472-6486, (2000).

**(Liang)** Liang, Phillpot, Sinnott Phys. Rev. B79 245110, (2009), Erratum: Phys. Rev. B85 199903(E), (2012)

## 4.259 pair_style resquared command

Accelerator Variants: *resquared/gpu*, *resquared/omp*

### 4.259.1 Syntax

```
pair_style resquared cutoff
```

- cutoff = global cutoff for interactions (distance units)

### 4.259.2 Examples

```
pair_style resquared 10.0
pair_coeff * * 1.0 1.0 1.7 3.4 3.4 1.0 1.0 1.0
```

### 4.259.3 Description

Style *resquared* computes the RE-squared anisotropic interaction *(Everaers)*, *(Babadi)* between pairs of ellipsoidal and/or spherical Lennard-Jones particles. For ellipsoidal interactions, the potential considers the ellipsoid as being comprised of small spheres of size $\sigma$. LJ particles are a single sphere of size $\sigma$. The distinction is made to allow the pair style to make efficient calculations of ellipsoid/solvent interactions.

Details for the equations used are given in the references below and in this supplementary document.

Use of this pair style requires the NVE, NVT, or NPT fixes with the *asphere* extension (e.g. *fix nve/asphere*) in order to integrate particle rotation. Additionally, *atom_style ellipsoid* should be used since it defines the rotational state and the size and shape of each ellipsoidal particle.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- A12 = Energy Prefactor/Hamaker constant (energy units)
- $\sigma$ = atomic interaction diameter (distance units)

- $\varepsilon_{i,a}$ = relative well depth of type I for side-to-side interactions
- $\varepsilon_{i,b}$ = relative well depth of type I for face-to-face interactions
- $\varepsilon_{i,c}$ = relative well depth of type I for end-to-end interactions
- $\varepsilon_{j,a}$ = relative well depth of type J for side-to-side interactions
- $\varepsilon_{j,b}$ = relative well depth of type J for face-to-face interactions
- $\varepsilon_{j,c}$ = relative well depth of type J for end-to-end interactions
- cutoff (distance units)

The parameters used depend on the type of the interacting particles, i.e. ellipsoids or LJ spheres. The type of a particle is determined by the diameters specified for its 3 shape parameters. If all 3 shape parameters = 0.0, then the particle is treated as an LJ sphere. The $\varepsilon_{i,*}$ or $\varepsilon_{j,*}$ parameters are ignored for LJ spheres. If the 3 shape parameters are > 0.0, then the particle is treated as an ellipsoid (even if the 3 parameters are equal to each other).

A12 specifies the energy prefactor which depends on the types of the two interacting particles.

For ellipsoid/ellipsoid interactions, the interaction is computed by the formulas in the supplementary document referenced above. A12 is the Hamaker constant as described in *(Everaers)*. In LJ units:

$$A_{12} = 4\pi^2 \varepsilon_{\mathrm{LJ}} (\rho \sigma^3)^2$$

where $\rho$ gives the number density of the spherical particles composing the ellipsoids and $\varepsilon_{\mathrm{LJ}}$ determines the interaction strength of the spherical particles.

For ellipsoid/LJ sphere interactions, the interaction is also computed by the formulas in the supplementary document referenced above. A12 has a modified form (see here for details):

$$A_{12} = 4\pi^2 \varepsilon_{\mathrm{LJ}} (\rho \sigma^3)$$

For ellipsoid/LJ sphere interactions, a correction to the distance- of-closest approach equation has been implemented to reduce the error from two particles of disparate sizes; see this supplementary document.

For LJ sphere/LJ sphere interactions, the interaction is computed using the standard Lennard-Jones formula, which is much cheaper to compute than the ellipsoidal formulas. A12 is used as epsilon in the standard LJ formula:

$$A_{12} = \varepsilon_{\mathrm{LJ}}$$

and the specified $\sigma$ is used as the $\sigma$ in the standard LJ formula.

When one of both of the interacting particles are ellipsoids, then $\sigma$ specifies the diameter of the continuous distribution of constituent particles within each ellipsoid used to model the RE-squared potential. Note that this is a different meaning for $\sigma$ than the *pair_style gayberne* potential uses.

The $\varepsilon_i$ and $\varepsilon_j$ coefficients are defined for atom types, not for pairs of atom types. Thus, in a series of pair_coeff commands, they only need to be specified once for each atom type.

Specifically, if any of $\varepsilon_{i,a}$, $\varepsilon_{i,b}$, $\varepsilon_{i,c}$ are non-zero, the three values are assigned to atom type I. If all the $\varepsilon_i$ values are zero, they are ignored. If any of $\varepsilon_{j,a}$, $\varepsilon_{j,b}$, $\varepsilon_{j,c}$ are non-zero, the three values are assigned to atom type J. If all three $\varepsilon_i$ values are zero, they are ignored. Thus the typical way to define the $\varepsilon_i$ and $\varepsilon_j$ coefficients is to list their values in "pair_coeff I J" commands when I = J, but set them to 0.0 when I != J. If you do list them when I != J, you should ensure they are consistent with their values in other pair_coeff commands.

Note that if this potential is being used as a sub-style of *pair_style hybrid*, and there is no "pair_coeff I I" setting made for RE-squared for a particular type I (because I-I interactions are computed by another hybrid pair potential), then you still need to ensure the epsilon a,b,c coefficients are assigned to that type in a "pair_coeff I J" command.

For large uniform molecules it has been shown that the $\varepsilon_{*,*}$ energy parameters are approximately representable in terms of local contact curvatures *(Everaers)*:

$$\varepsilon_a = \sigma \cdot \frac{a}{b \cdot c}; \varepsilon_b = \sigma \cdot \frac{b}{a \cdot c}; \varepsilon_c = \sigma \cdot \frac{c}{a \cdot b}$$

where a, b, and c give the particle diameters.

The last coefficient is optional. If not specified, the global cutoff specified in the pair_style command is used.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.259.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon and sigma coefficients and cutoff distance can be mixed, but only for sphere pairs. The default mix value is *geometric*. See the "pair_modify" command for details. Other type pairs cannot be mixed, due to the different meanings of the energy prefactors used to calculate the interactions and the implicit dependence of the ellipsoid-sphere interaction on the equation for the Hamaker constant presented here. Mixing of sigma and epsilon followed by calculation of the energy prefactors using the equations above is recommended.

This pair style supports the *pair_modify* shift option for the energy of the Lennard-Jones portion of the pair interaction, but only for sphere-sphere interactions. There is no shifting performed for ellipsoidal interactions due to the anisotropic dependence of the interaction.

The *pair_modify* table option is not relevant for this pair style.

This pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords of the *run_style command*.

---

### 4.259.5 Restrictions

This style is part of the ASPHERE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires that atoms be ellipsoids as defined by the *atom_style ellipsoid* command.

Particles acted on by the potential can be finite-size aspherical or spherical particles, or point particles. Spherical particles have all 3 of their shape parameters equal to each other. Point particles have all 3 of their shape parameters equal to 0.0.

The distance-of-closest-approach approximation used by LAMMPS becomes less accurate when high-aspect ratio ellipsoids are used.

### 4.259.6 Related commands

*pair_coeff*, *fix nve/asphere*, *compute temp/asphere*, *pair_style gayberne*

### 4.259.7 Default

none

---

**(Everaers)** Everaers and Ejtehadi, Phys Rev E, 67, 041710 (2003).

**(Babadi)** Babadi, Ejtehadi, Everaers, J Comp Phys, 219, 770-779 (2006).

## 4.260 pair_style rheo command

### 4.260.1 Syntax

```
pair_style rheo cutoff keyword values
```

- cutoff = global cutoff for kernel (distance units)
- zero or more keyword/value pairs may be appended to args
- keyword = *rho/damp* or *artificial/visc* or *harmonic/means*

*rho/damp* args = density damping prefactor $\xi$
*artificial/visc* args = artificial viscosity prefactor $\zeta$
*harmonic/means* args = none

### 4.260.2 Examples

```
pair_style rheo 3.0 rho/damp 1.0 artificial/visc 2.0
pair_coeff * *
```

### 4.260.3 Description

Added in version 29Aug2024.

Pair style *rheo* computes pressure and viscous forces between particles in the *rheo package*. If thermal evolution is turned on in *fix rheo*, then the pair style also calculates heat exchanged between particles.

The *artificial/viscosity* keyword is used to specify the magnitude $\zeta$ of an optional artificial viscosity contribution to forces. This factor can help stabilize simulations by smoothing out small length scale variations in velocity fields. Artificial viscous forces typically are only exchanged by fluid particles. However, if interfaces are not reconstructed in fix rheo, fluid particles will also exchange artificial viscous forces with solid particles to improve stability.

The *rho/damp* keyword is used to specify the magnitude $\xi$ of an optional pairwise damping term between the density of particles. This factor can help stabilize simulations by smoothing out small length scale variations in density fields. However, in systems that develop a density gradient in equilibrium (e.g. in a hydrostatic column underlying gravity), this option may be inappropriate.

If particles have different viscosities or conductivities, the *harmonic/means* keyword changes how they are averaged before calculating pairwise forces or heat exchanges. By default, an arithmetic averaged is used, however, a harmonic mean may improve stability in systems with multiple fluid phases with large disparities in viscosities.

No coefficients are defined for each pair of atoms types via the *pair_coeff* command as in the examples above.

---

### 4.260.4 Mixing, shift, table, tail correction, restart, rRESPA info

This style does not support the *pair_modify* shift, table, and tail options.

This style does not write information to *binary restart files*. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.260.5 Restrictions

This fix is part of the RHEO package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.260.6 Related commands

*fix rheo*, *fix rheo/pressure*, *fix rheo/thermal*, *fix rheo/viscosity*, *compute rheo/property/atom*

### 4.260.7 Default

Density damping and artificial viscous forces are not calculated. Arithmetic means are used for mixing particle properties.

## 4.261 pair_style rheo/solid command

### 4.261.1 Syntax

```
pair_style rheo/solid
```

### 4.261.2 Examples

```
pair_style rheo/solid
pair_coeff * * 1.0 1.5 1.0
```

### 4.261.3 Description

Added in version 29Aug2024.

Style *rheo/solid* is effectively a copy of pair style *bpm/spring* except it only applies forces between solid RHEO particles, determined by checking the status of each pair of neighboring particles before calculating forces.

The style computes pairwise forces with the formula

$$F = k(r - r_c)$$

where $k$ is a stiffness and $r_c$ is the cutoff length. An additional damping force is also applied to interacting particles. The force is proportional to the difference in the normal velocity of particles

$$F_D = -\gamma w(\hat{r} \bullet \vec{v})$$

where $\gamma$ is the damping strength, $\hat{r}$ is the displacement normal vector, $\vec{v}$ is the velocity difference between the two particles, and $w$ is a smoothing factor. This smoothing factor is constructed such that damping forces go to zero as particles come out of contact to avoid discontinuities. It is given by

$$w = 1.0 - \left(\frac{r}{r_c}\right)^8.$$

The following coefficients must be defined for each pair of atom types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- $k$ (force/distance units)
- $r_c$ (distance units)
- $\gamma$ (force/velocity units)

---

### 4.261.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the A coefficient and cutoff distance for this pair style can be mixed. A is always mixed via a *geometric* rule. The cutoff is mixed according to the pair_modify mix value. The default mix value is *geometric*. See the "pair_modify" command for details.

This pair style does not support the *pair_modify* shift option, since the pair interaction goes to 0.0 at the cutoff.

The *pair_modify* table and tail options are not relevant for this pair style.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.261.5 Restrictions

This pair style is part of the RHEO package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.261.6 Related commands

*fix rheo*, *fix rheo/thermal*, *pair bpm/spring*

### 4.261.7 Default

none

## 4.262 pair_style runner command

### 4.262.1 Syntax

```
pair_style runner keyword value ...
```

- zero or more keyword/value pairs may be appended
- keyword = *dir* or *cflength* or *cfenergy* or *f_comm* or *q_comm* or *committee_size* or *total_charge* or *use_prev_q* or *check_extrap* or *max_extrap* or *show_ew* or *sum_ew_freq* or *reset_ew_freq*
- value depends on the preceding keyword:

| Keyword | Value | Description |
|---|---|---|
| *dir* | path | Path to RuNNer configuration files |
| *cflength* | factor | Length unit conversion factor |
| *cfenergy* | factor | Energy unit conversion factor |
| *committee_size* | N | Number of committee members |
| *f_comm* | *yes* or *no* | Write individual committee forces to per-atom array |
| *q_comm* | *yes* or *no* | Write individual committee charges to per-atom array |
| *total_charge* | Q | Total system charge |
| *use_prev_q* | *yes* or *no* | Use charges from previous step as QEq guess |
| *check_extrap* | *yes* or *no* | Enable extrapolation warning (EW) monitoring |
| *max_extrap* | N | Stop simulation if EW count exceeds N |
| *show_ew* | *yes* or *no* | Write EWs to the log file |
| *sum_ew_freq* | N | Write EW summary every N steps |
| *reset_ew_freq* | N | Reset EW counters every N steps |

### 4.262.2 Examples

```
pair_style runner dir "./potential_files"
pair_coeff * * 1 8

fix 1 all property/atom d2_f_comm 24 ghost yes
pair_style runner dir "./potential_files" cflength 1.8897261328 &
   cfenergy 0.0367493254 committee_size 8 f_comm yes
pair_coeff * * 1 3 8 25

fix 1 all property/atom d2_q_comm 4 ghost yes
pair_style runner dir "./potential_files" committee_size 4 q_comm yes total_charge 0.0
pair_coeff * * 8 12 79
```

### 4.262.3 Description

Added in version 4Jul2026.

This pair style provides an interface to the RuNNer 2 (Ruhr University Neural Network Energy Representation) library. It implements High-Dimensional Neural Network Potentials (HDNNPs) as introduced in (*Behler and Parrinello 2007*). HDNNPs are machine learning potentials that represent the total energy of a system as a sum of environment-dependent atomic contributions.

The pair style supports several "generations" of HDNNPs as categorized in (*Behler 2021*):

- **Second-generation (2G):** Short-range many-body potentials where the total energy is the sum of atomic energies predicted from local chemical environments (*Behler and Parrinello 2007*).
- **Third-generation (3G):** Extends 2G by adding explicit long-range electrostatic interactions based on environment-dependent atomic partial charges (*Artrith, Morawietz and Behler 2011*).
- **Fourth-generation (4G):** Includes global charge equilibration (QEq) based on environment-dependent electronegativities. These charges are used to calculate long-range electrostatics and serve as a global descriptor for the atomic energies, allowing for the description of nonlocal charge transfer (*Ko et al 2021*; *Gubler et al 2024*).

Additionally, all generations can be augmented with:

- **Hirshfeld-based dispersion:** Long-range van der Waals interactions based on the Tkatchenko-Scheffler dispersion model (*Tkatchenko and Scheffler 2009*).

- **Repulsive potentials:** Screened nuclear repulsion at short interatomic distances based on the Ziegler-Biersack-Littmark (ZBL) model.

Only a single *pair_coeff* command with two asterisk wildcards is used with this pair style. Its additional arguments define the mapping of LAMMPS atom types to RuNNer atomic numbers.

```
pair_coeff * * 1 8
```

The example above maps LAMMPS atom types 1 and 2 to atomic numbers 1 ("H") and 8 ("O") in RuNNer.

---

### General

For more information on the options of RuNNer, visit the official RuNNer documentation..

Use the *dir* keyword to specify the directory containing the RuNNer configuration files. The directory must contain:

- `input.nn`: The HDNNP generation, architecture and feature map specifications.
- `scaling_*.data`: Feature map scaling data for the model.
- `weights_*.???.data`: Neural network parameters (weights and biases) for each element and model.

The RuNNer library is unit-agnostic. Use *cflength* and *cfenergy* to scale LAMMPS coordinates and energies to the units in which the potential was trained (typically Bohr and Hartree). If the HDNNP was trained in Bohr/Hartree and the LAMMPS simulation uses *metal* units (Angstroms, eV), then *cflength* and *cfenergy* must be the multiplicative factors required to convert LAMMPS units to the respective quantities in native HDNNP units:

$$R_{\text{native}} = R_{\text{LAMMPS}} \times \text{cflength}$$

$$E_{\text{native}} = E_{\text{LAMMPS}} \times \text{cfenergy}$$

Example for *metal* units to a Bohr/Hartree potential:

```
cflength 1.8897261328   # Angstrom to Bohr
cfenergy 0.0367493254   # eV to Hartree
```

Since machine learning potentials are most reliable within their training data range, the *runner* pair style can monitor whether the features representing local atomic environments extrapolate beyond the training range.

- Set *check_extrap* to *yes* to enable monitoring.
- The keyword *show_ew* enables writing individual extrapolation warnings (EWs) to the log file.
- Use *sum_ew_freq* to write a summary of EW counts at specific intervals instead of logging every occurrence.
- The *max_extrap* threshold allows termination of a simulation if the total EW count exceeds this value. Setting *max_extrap* to a negative number disables the termination threshold.
- Use *reset_ew_freq* to reset the EW counters at specific intervals.

### Committees

The pair style supports **Committees**, where multiple HDNNPs sharing atomic descriptors are evaluated simultaneously. The forces, energies, and virials used to propagate the simulation are the average of all committee members. This is useful for Query-by-Committee-based Active Learning approaches and uncertainty estimation for production simulations.

In the case of a *committee_size* greater than 1, the *dir* keyword must point to a directory that contains the global configuration files (`input.nn` and `scaling_*.data`) and a set of subdirectories named **1** to **N** (where N is the *committee_size*). Each subdirectory must contain the `weights_*.???.data` files for that specific committee member.

```
potential_files/
|-- 1/
|   |-- weights_short.001.data
|   `-- weights_short.008.data
|-- 2/
|   |-- weights_short.001.data
|   `-- weights_short.008.data
|-- input.nn
`-- scaling.data
```

**Accessing Member Energies**

The individual potential energies of each committee member can be accessed using the *compute pair* command:

```
compute e_comm all pair runner
```

The energies are stored in a global vector *e_comm* of length *committee_size*. They can be accessed in subsequent commands (like *thermo_style*) as:

- `c_e_comm[1]`: Potential energy of member 1
- `c_e_comm[2]`: Potential energy of member 2
- `c_e_comm[N]`: Potential energy of member N

**Accessing Member Forces and Charges**

To extract individual member forces or charges (e.g., to compute per-atom variance), you must define a custom per-atom array using the *fix property/atom* command **before** the *pair_style* command.

For forces, set *f_comm* to *yes*. The array **must** be a floating-point array (type `d2`) named *f_comm* with three times *committee_size* columns. Ghost atom communication must be enabled.

```
# Example: committee_size 8 requires 8 * 3 = 24 columns
fix 1 all property/atom d2_f_comm 24 ghost yes
pair_style runner committee_size 8 f_comm yes
```

Forces are stored sequentially by member and dimension (*fx1*, *fy1*, *fz1*, *fx2*, ...) and can be accessed by subsequent commands (like *dump*) as:

- `d2_f_comm[1]`, `d2_f_comm[2]`, `d2_f_comm[3]`: Atomic forces (x,y,z) for member 1
- `d2_f_comm[4]`, `d2_f_comm[5]`, `d2_f_comm[6]`: Atomic forces (x,y,z) for member 2

For 3G and 4G potentials, set *q_comm* to *yes* to extract individual member atomic charges. The array **must** be a floating-point array (type `d2`) named *q_comm* with *committee_size* columns.

```
# Example: committee_size 2
fix 2 all property/atom d2_q_comm 2 ghost yes
pair_style runner committee_size 2 q_comm yes
```

The committee atomic charges can be accessed by subsequent commands (like *dump*) as:

- `d2_q_comm[1]`: Atomic charges for member 1
- `d2_q_comm[2]`: Atomic charges for member 2
- `d2_q_comm[N]`: Atomic charges for member N

#### 3G / 4G only

For 3G and 4G HDNNPs, the total charge of the system can be specified with the *total_charge* keyword. For periodic systems, the system must be charge neutral.

For 4G HDNNPs only, the *use_prev_q* keyword controls the initial guess for the iterative Charge Equilibration (QEq) solver. Setting this to *yes* uses the predicted charges from the previous time step as the starting point for the current time step, which can reduce the number of iterations required for convergence.

**Periodicity limitations**

3G and 4G HDNNPs require either full 3D periodicity (`boundary p p p`) or no periodicity (`boundary f f f`). Partial periodicity (e.g., `boundary p p f`) is currently not supported for long-range electrostatics.

**MPI scaling**

3G and 4G HDNNPs require a global collection of the atomic structure on a single process to perform long-range electrostatics and global QEq calculations. This creates an MPI bottleneck that leads to inferior strong scaling as the number of MPI tasks increases. Since 3G and 4G HDNNPs are heavily optimized for OpenMP in RuNNer, it is highly recommended to use a small number of MPI tasks and a large number of OpenMP threads per task to achieve the best performance.

---

### 4.262.4 Mixing, shift, table, tail correction, restart, rRESPA info

This style does not support mixing. The *pair_coeff* command should only be invoked with asterisk wildcards to define the mapping for all atom types.

This style does not support the *pair_modify* shift, table, and tail options.

This style does not write information to *binary restart files*. You must re-specify the *pair_style* and *pair_coeff* commands in any input script that reads a restart file.

This style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, or *outer* keywords.

### 4.262.5 Restrictions

This pair style is part of the ML-RUNNER package. It is only enabled if LAMMPS was built with that package. See the *Build package* doc page for more info.

Currently, only one instance of `pair_style runner` can be initialized per simulation. The style does not support the use of *pair_style hybrid* where multiple `runner` instances are defined.

### 4.262.6 Related commands

*pair_coeff* , *fix property/atom*, *compute pair*

### 4.262.7 Default

The default options are:

- *dir* = "./"
- *cflength* = 1.0

- *cfenergy* = 1.0
- *committee_size* = 1
- *f_comm* = no
- *q_comm* = no
- *total_charge* = 0.0
- *use_prev_q* = no
- *check_extrap* = no
- *max_extrap* = 100
- *show_ew* = no
- *sum_ew_freq* = 0
- *reset_ew_freq* = 0

---

### 4.262.8 References

**(Behler and Parrinello 2007)** Behler, J.; Parrinello, M., Phys. Rev. Lett. 2007, 98 (14), 146401.

**(Tkatchenko and Scheffler 2009)** Tkatchenko, A.; Scheffler, M., Phys. Rev. Lett. 2009, 102 (7), 073005.

**(Behler 2011)** Behler, J., J. Chem. Phys. 2011, 134 (7), 074106.

**(Artrith, Morawietz and Behler 2011)** Artrith, N.; Morawietz, T.; Behler, J., Phys. Rev. B 2011, 83 (15), 153101.

**(Ko et al 2021)** Ko, T. W.; Finkler, J. A.; Goedecker, S.; Behler, J., Nat. Commun. 2021, 12, 398.

**(Behler 2021)** Behler, J., Chem. Rev. 2021, 121 (16), 10037-10072.

**(Gubler et al 2024)** Gubler, M.; Finkler, J. A.; Schaefer, M. R.; Behler, J.; Goedecker S., J. Chem. Theory Comput. 2024, 20 (16), 7264-7271.

## 4.263 pair_style saip/metal command

Accelerator Variants: *saip/metal/opt*

## 4.264 pair_style saip/metal/tmd command

Accelerator Variants: *saip/metal/tmd/opt*

### 4.264.1 Syntax

```
pair_style [hybrid/overlay ...] style cutoff tap_flag
```

- style = *saip/metal* or *saip/metal/tmd*
- cutoff = global cutoff (distance units)
- tap_flag = 0/1 to turn off/on the taper function

### 4.264.2 Examples

```
pair_style  hybrid/overlay saip/metal 16.0 1
pair_coeff  * * saip/metal CHAu.ILP Au C H

pair_style  hybrid/overlay eam rebo saip/metal 16.0
pair_coeff  1 1 eam  Au_u3.eam  Au NULL NULL
pair_coeff  * * rebo CH.rebo    NULL  C H
pair_coeff  * * saip/metal      CHAu.ILP  Au C H

pair_style  hybrid/overlay eam sw/mod saip/metal/tmd 16.0
pair_coeff  1 1 eam  Au_u3.eam    Au NULL NULL
pair_coeff  * * sw/mod tmd.sw.mod NULL  S Mo S
pair_coeff  * * saip/metal/tmd    TMDAu.SAIP  Au S Mo S
```

### 4.264.3 Description

Added in version 17Feb2022.

The *saip/metal* style computes the semi-anisotropic interfacial potential (SAIP) potential for hetero-junctions formed with hexagonal 2D materials and metal surfaces, as described in *(Ouyang6)* and *(Yao1)*.

Added in version 10Dec2025.

The *saip/metal/tmd* style computes the semi-anisotropic interfacial potential (SAIP) potential for hetero-junctions formed with transition metal dichalcogenides (TMDCs) and metal surfaces, as described in *(Yao2)*.

$$
\begin{aligned}
E =& \frac{1}{2}\sum_i \sum_{j\neq i} V_{ij} \\
V_{ij} =& \mathrm{Tap}(r_{ij}) \left\{ e^{-\alpha(r_{ij}/\beta-1)} \left[\varepsilon + f(\rho_{ij}) + f(\rho_{ji})\right] - \frac{1}{1+e^{-d[(r_{ij}/(s_R\cdot r^{eff}))-1]}} \cdot \frac{C_6}{r_{ij}^6} \right\} \\
\rho_{ij}^2 =& r_{ij}^2 - (\mathbf{r}_{ij}\cdot\mathbf{n}_i)^2 \\
\rho_{ji}^2 =& r_{ij}^2 - (\mathbf{r}_{ij}\cdot\mathbf{n}_j)^2 \\
f(\rho) =& Ce^{-(\rho/\delta)^2} \\
\mathrm{Tap}(r_{ij}) =& 20\left(\frac{r_{ij}}{R_{cut}}\right)^7 - 70\left(\frac{r_{ij}}{R_{cut}}\right)^6 + 84\left(\frac{r_{ij}}{R_{cut}}\right)^5 - 35\left(\frac{r_{ij}}{R_{cut}}\right)^4 + 1
\end{aligned}
$$

Where $\mathrm{Tap}(r_{ij})$ is the taper function which provides a continuous cutoff (up to third derivative) for interatomic separations larger than $r_c$ *pair_style ilp_graphene_hbn*.

It is important to include all the pairs to build the neighbor list for calculating the normals.

**Note**

To account for the isotropic nature of the isolated gold atom electron cloud, their corresponding normal vectors (*mathbf{n}_i*) are assumed to lie along the interatomic vector *mathbf{r}_ij*. Notably, this assumption is suitable for many bulk material surfaces, for example, for systems possessing s-type valence orbitals or metallic surfaces, whose valence electrons are mostly delocalized, such that their Pauli repulsion with the electrons of adjacent surfaces are isotropic. Caution should be used in the case of very small gold contacts, for example, nano-clusters, where edge effects may become relevant.

The parameter file (e.g. CHAu.ILP), is intended for use with *metal* *units*, with energies in meV. Two additional parameters, *S*, and *rcut* are included in the parameter file. *S* is designed to facilitate scaling of energies. *rcut* is designed to

build the neighbor list for calculating the normals for each atom pair.

> **Note**
>
> The parameters presented in the parameter file (e.g. TMDAu.SAIP), are fitted with taper function by setting the cutoff equal to 16.0 Angstrom. Using different cutoff or taper function should be careful.

This potential must be used in combination with hybrid/overlay. Other interactions can be set to zero using pair_style *none*.

This pair style tallies a breakdown of the total interlayer potential energy into sub-categories, which can be accessed via the *compute pair* command as a vector of values of length 2. The 2 values correspond to the following sub-categories:

1. *E_vdW* = vdW (attractive) energy
2. *E_Rep* = Repulsive energy

To print these quantities to the log file (with descriptive column headings) the following commands could be included in an input script:

```
compute 0 all pair saip/metal
variable Evdw  equal c_0[1]
variable Erep  equal c_0[2]
thermo_style custom step temp epair v_Erep v_Evdw
```

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.264.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support the pair_modify mix, shift, table, and tail options.

This pair style does not write their information to binary restart files, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

### 4.264.5 Restrictions

This pair style is part of the INTERLAYER package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires the newton setting to be *on* for pair interactions.

The CHAu.ILP and TMDAu.SAIP potential file provided with LAMMPS (see the potentials directory) are parameterized for *metal* units. You can use this potential with any LAMMPS units, but you would need to create your own custom CHAu.ILP potential file with coefficients listed in the appropriate units, if your simulation does not use *metal* units.

### 4.264.6 Related commands

*pair_coeff*, *pair_none*, *pair_style hybrid/overlay*, *pair_style drip*, *pair_style ilp_tmd*, *pair_style ilp_graphene_hbn*, *pair_style pair_kolmogorov_crespi_z*, *pair_style pair_kolmogorov_crespi_full*, *pair_style pair_lebedeva_z*, *pair_style pair_coul_shield*.

### 4.264.7 Default

tap_flag = 1

---

**(Ouyang6)** W. Ouyang, O. Hod, and R. Guerra, J. Chem. Theory Comput. 17, 7215 (2021).

**(Yao1)** Y. Yao, ..., W. Ouyang, J. Phys. Chem. C, 128, 6836 (2024).

**(Yao2)** Y. Yao, ..., W. Ouyang, Adv. Sci. 12, 2415884 (2025).

## 4.265 pair_style sdpd/taitwater/isothermal command

### 4.265.1 Syntax

```
pair_style sdpd/taitwater/isothermal temperature viscosity seed
```

- temperature = temperature of the fluid (temperature units)
- viscosity = dynamic viscosity of the fluid (mass*distance/time units)
- seed = random number generator seed (positive integer, optional)

### 4.265.2 Examples

```
pair_style sdpd/taitwater/isothermal 300. 1. 28681
pair_coeff * * 1000.0 1430.0 2.4
```

### 4.265.3 Description

The sdpd/taitwater/isothermal style computes forces between mesoscopic particles according to the Smoothed Dissipative Particle Dynamics model described in this paper by *(Espanol and Revenga)* under the following assumptions:

1. The temperature is constant and uniform.
2. The shear viscosity is constant and uniform.
3. The volume viscosity is negligible before the shear viscosity.
4. The Boltzmann constant is negligible before the heat capacity of a single mesoscopic particle of fluid.

The third assumption is true for water in nearly incompressible flows. The fourth holds true for water for any reasonable size one can imagine for a mesoscopic particle.

The pressure forces between particles will be computed according to Tait's equation of state:

$$p = B\left[(\frac{\rho}{\rho_0})^\gamma - 1\right]$$

where $\gamma = 7$ and $B = c_0^2\rho_0/\gamma$, with $\rho_0$ being the reference density and $c_0$ the reference speed of sound.

The laminar viscosity and the random forces will be computed according to formulas described in *(Espanol and Revenga)*.

**Warning**

Similar to *brownian* and *dpd* styles, the *newton* setting for pairwise interactions needs to be on when running LAMMPS in parallel if you want to ensure linear momentum conservation. Otherwise random forces generated for pairs straddling processor boundary will not be equal and opposite.

**Note**

The actual random seed used will be a mix of what you specify and other parameters like the MPI ranks. This is to ensure that different MPI tasks have distinct seeds.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above.

- $\rho_0$ reference density (mass/volume units)
- $c_0$ reference soundspeed (distance/time units)
- h kernel function cutoff (distance units)

---

### 4.265.4 Mixing, shift, table, tail correction, restart, rRESPA info

This style does not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

This style does not support the *pair_modify* shift, table, and tail options.

This style does not write information to *binary restart files*. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.265.5 Restrictions

This pair style is part of the DPD-SMOOTH package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.265.6 Related commands

*pair coeff*, *pair sph/rhosum*, *pair sph/taitwater*

### 4.265.7 Default

The default seed is 0 (before mixing).

---

**(Espanol and Revenga)** Espanol, Revenga, Physical Review E, 67, 026705 (2003).

# 4.266 pair_style smatb command

# 4.267 pair_style smatb/single command

## 4.267.1 Syntax

```
pair_style style args
```

- style = *smatb* or *smatb/single*
- args = none

## 4.267.2 Examples

```
pair_style smatb
pair_coeff 1 1 2.88 10.35 4.178 0.210 1.818 4.07293506 4.9883063257983666

pair_style smatb/single
pair_coeff 1 1 2.88 10.35 4.178 0.210 1.818 4.07293506 4.9883063257983666
```

## 4.267.3 Description

Added in version 4May2022.

The *smatb* and *smatb/single* styles compute the Second Moment Approximation to the Tight Binding *(Cyrot)*, *(Gupta)*, *(Rosato)*, given by

$$E_i = \sum_{j,R_{ij} \leq R_c} \alpha(R_{ij}) - \sqrt{\sum_{j,R_{ij} \leq R_c} \Xi^2(R_{ij})}$$

$R_{ij}$ is the distance between the atom $i$ and $j$. And the two functions $\alpha\,(r)$ and $\Xi\,(r)$ are:

$$\alpha\,(r) = \begin{cases} Ae^{-p\left(\frac{r}{R_0}-1\right)} & r < R_{sc} \\ a_3\,(r-R_c)^3 + a_4\,(r-R_c)^4 + a_5\,(r-R_c)^5 & R_{sc} < r < R_c \end{cases}$$

$$\Xi\,(r) = \begin{cases} \xi e^{-q\left(\frac{r}{R_0}-1\right)} & r < R_{sc} \\ x_3\,(r-R_c)^3 + x_4\,(r-R_c)^4 + x_5\,(r-R_c)^5 & R_{sc} < r < R_c \end{cases}$$

The polynomial coefficients $a_3$, $a_4$, $a_5$, $x_3$, $x_4$, $x_5$ are computed by LAMMPS: the two exponential terms and their first and second derivatives are smoothly reduced to zero, from the inner cutoff $R_{sc}$ to the outer cutoff $R_c$.

The *smatb/single* style is an optimization when using only a single atom type.

## 4.267.4 Coefficients

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- $R_0$ (distance units)
- $p$ (dimensionless)
- $q$ (dimensionless)
- $A$ (energy units)
- $\xi$ (energy units)

- $R_{cs}$ (distance units)
- $R_c$ (distance units)

Note that: $R_0$ is the nearest neighbor distance, usually coincides with the diameter of the atoms

See the *run_style* command for details.

---

### 4.267.5 Mixing info

For atom type pairs I,J and I != J the coefficients are not automatically mixed.

---

### 4.267.6 Restrictions

These pair styles are part of the SMTBQ package and are only enabled if LAMMPS is built with that package. See the *Build package* page for more info.

These pair styles require the *newton* setting to be "on" for pair interactions.

### 4.267.7 Related commands

- *pair_coeff*

### 4.267.8 Default

none

---

**(Cyrot)** Cyrot-Lackmann and Ducastelle, Phys Rev. B, 4, 2406-2412 (1971).

**(Gupta)** Gupta ,Phys Rev. B, 23, 6265-6270 (1981).

**(Rosato)** Rosato and Guillope and Legrand, Philosophical Magazine A, 59.2, 321-336 (1989).

## 4.268 pair_style smd/hertz command

### 4.268.1 Syntax

```
pair_style smd/hertz scale_factor
```

### 4.268.2 Examples

```
pair_style smd/hertz 1.0
pair_coeff 1 1 <contact_stiffness>
```

### 4.268.3 Description

The *smd/hertz* style calculates contact forces between SPH particles belonging to different physical bodies.

The contact forces are calculated using a Hertz potential, which evaluates the overlap between two particles (whose spatial extents are defined via its contact radius). The effect is that a particles cannot penetrate into each other. The parameter <contact_stiffness> has units of pressure and should equal roughly one half of the Young's modulus (or bulk modulus in the case of fluids) of the material model associated with the SPH particles.

The parameter *scale_factor* can be used to scale the particles' contact radii. This can be useful to control how close particles can approach each other. Usually, *scale_factor* =1.0.

---

### 4.268.4 Mixing, shift, table, tail correction, restart, rRESPA info

No mixing is performed automatically. Currently, no part of MACHDYN supports restarting nor minimization. rRESPA does not apply to this pair style.

---

### 4.268.5 Restrictions

This fix is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.268.6 Related commands

*pair_coeff*

### 4.268.7 Default

none

## 4.269 pair_style smd/tlsph command

### 4.269.1 Syntax

```
pair_style smd/tlsph args
```

### 4.269.2 Examples

```
pair_style smd/tlsph
```

### 4.269.3 Description

The *smd/tlsph* style computes particle interactions according to continuum mechanics constitutive laws and a Total-Lagrangian Smooth-Particle Hydrodynamics algorithm.

This pair style is invoked with the following command:

```
pair_style smd/tlsph
pair_coeff i j *COMMON rho0 E nu Q1 Q2 hg Cp &
               *END
```

Here, *i* and *j* denote the *LAMMPS* particle types for which this pair style is defined. Note that *i* and *j* must be equal, i.e., no *tlsph* cross interactions between different particle types are allowed. In contrast to the usual *LAMMPS pair coeff* definitions, which are given solely a number of floats and integers, the *tlsph pair coeff* definition is organized using keywords. These keywords mark the beginning of different sets of parameters for particle properties, material constitutive models, and damage models. The *pair coeff* line must be terminated with the **END* keyword. The use the line continuation operator *&* is recommended. A typical invocation of the *tlsph* for a solid body would consist of an equation of state for computing the pressure (the diagonal components of the stress tensor), and a material model to compute shear stresses (the off-diagonal components of the stress tensor). Damage and failure models can also be added.

Please see the SMD user guide for a complete listing of the possible keywords and material models.

---

### 4.269.4 Mixing, shift, table, tail correction, restart, rRESPA info

No mixing is performed automatically. Currently, no part of MACHDYN supports restarting nor minimization. rRESPA does not apply to this pair style.

---

### 4.269.5 Restrictions

This fix is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.269.6 Related commands

*pair_coeff*

### 4.269.7 Default

none

## 4.270 pair_style smd/tri_surface command

### 4.270.1 Syntax

```
pair_style smd/tri_surface scale_factor
```

### 4.270.2 Examples

```
pair_style smd/tri_surface 1.0
pair_coeff 1 1 <contact_stiffness>
```

### 4.270.3 Description

The *smd/tri_surface* style calculates contact forces between SPH particles and a rigid wall boundary defined via the *smd/wall_surface* fix.

The contact forces are calculated using a Hertz potential, which evaluates the overlap between a particle (whose spatial extents are defined via its contact radius) and the triangle. The effect is that a particle cannot penetrate into the triangular surface. The parameter <contact_stiffness> has units of pressure and should equal roughly one half of the Young's modulus (or bulk modulus in the case of fluids) of the material model associated with the SPH particle

The parameter *scale_factor* can be used to scale the particles' contact radii. This can be useful to control how close particles can approach the triangulated surface. Usually, *scale_factor* =1.0.

---

### 4.270.4 Mixing, shift, table, tail correction, restart, rRESPA info

No mixing is performed automatically. Currently, no part of MACHDYN supports restarting nor minimization. rRESPA does not apply to this pair style.

---

### 4.270.5 Restrictions

This fix is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.270.6 Related commands

*pair_coeff*

### 4.270.7 Default

none

## 4.271 pair_style smd/ulsph command

### 4.271.1 Syntax

```
pair_style smd/ulsph args
```

- these keywords must be given

keyword = **DENSITY_SUMMATION* or **DENSITY_CONTINUITY* and **VELOCITY_GRADIENT* or **NO_VELOCITY_GRADIENT* and **GRADIENT_CORRECTION* or **NO_GRADIENT_CORRECTION*

### 4.271.2 Examples

```
pair_style smd/ulsph *DENSITY_CONTINUITY *VELOCITY_GRADIENT *NO_GRADIENT_CORRECTION
```

### 4.271.3 Description

The *smd/ulsph* style computes particle interactions according to continuum mechanics constitutive laws and an updated Lagrangian Smooth-Particle Hydrodynamics algorithm.

This pair style is invoked similar to the following command:

```
pair_style smd/ulsph *DENSITY_CONTINUITY *VELOCITY_GRADIENT *NO_GRADIENT_CORRECTION
pair_coeff i j *COMMON rho0 c0 Q1 Cp hg &
               *END
```

Here, *i* and *j* denote the *LAMMPS* particle types for which this pair style is defined. Note that *i* and *j* can be different, i.e., *ulsph* cross interactions between different particle types are allowed. However, *i–i* respectively *j–j* pair_coeff lines have to precede a cross interaction. In contrast to the usual *LAMMPS pair coeff* definitions, which are given solely a number of floats and integers, the *ulsph pair coeff* definition is organized using keywords. These keywords mark the

beginning of different sets of parameters for particle properties, material constitutive models, and damage models. The *pair coeff* line must be terminated with the **END* keyword. The use the line continuation operator *&* is recommended. A typical invocation of the *ulsph* for a solid body would consist of an equation of state for computing the pressure (the diagonal components of the stress tensor), and a material model to compute shear stresses (the off-diagonal components of the stress tensor).

Note that the use of *GRADIENT_CORRECTION can lead to severe numerical instabilities. For a general fluid simulation, *NO_GRADIENT_CORRECTION is recommended.

Please see the SMD user guide for a complete listing of the possible keywords and material models.

---

### 4.271.4 Mixing, shift, table, tail correction, restart, rRESPA info

No mixing is performed automatically. Currently, no part of MACHDYN supports restarting nor minimization. rRESPA does not apply to this pair style.

---

### 4.271.5 Restrictions

This fix is part of the MACHDYN package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.271.6 Related commands

*pair_coeff*

### 4.271.7 Default

none

## 4.272 pair_style smtbq command

### 4.272.1 Syntax

```
pair_style smtbq
```

### 4.272.2 Examples

```
pair_style smtbq
pair_coeff * * ffield.smtbq.Al2O3 O Al
```

### 4.272.3 Description

This pair style computes a variable charge SMTB-Q (Second-Moment tight-Binding QEq) potential as described in *SMTB-Q_1* and *SMTB-Q_2*. This potential was first proposed in *SMTB-Q_0*. Briefly, the energy of metallic-oxygen

systems is given by three contributions:

$$
\begin{aligned}
E_{tot} &= E_{ES} + E_{OO} + E_{MO} \\
E_{ES} &= \sum_i \left[ \chi_i^0 Q_i + \frac{1}{2} J_i^0 Q_i^2 + \frac{1}{2} \sum_{j \neq i} J_{ij}(r_{ij}) f_{cut}^{R_{coul}}(r_{ij}) Q_i Q_j \right] \\
E_{OO} &= \sum_{i,j}^{i,j=O} \left[ Cexp(-\frac{r_{ij}}{\rho}) - D f_{cut}^{r_1^{OO} r_2^{OO}}(r_{ij}) exp(Br_{ij}) \right] \\
E_{MO} &= \sum_i E_{cov}^i + \sum_{j \neq i} A f_{cut}^{r_{c1} r_{c2}}(r_{ij}) exp\left[ -p(\frac{r_{ij}}{r_0} - 1) \right]
\end{aligned}
$$

where $E_{tot}$ is the total potential energy of the system, $E_{ES}$ is the electrostatic part of the total energy, $E_{OO}$ is the interaction between oxygen atoms and $E_{MO}$ is a short-range interaction between metal and oxygen atoms. This interactions depend on interatomic distance $r_{ij}$ and/or the charge $Q_i$ of atoms $i$. Cut-off function enables smooth convergence to zero interaction.

The parameters appearing in the upper expressions are set in the ffield.SMTBQ.Syst file where Syst corresponds to the selected system (e.g. field.SMTBQ.Al2O3). Examples for $TiO_2$, $Al_2O_3$ are provided. A single pair_coeff command is used with the SMTBQ styles which provides the path to the potential file with parameters for needed elements. These are mapped to LAMMPS atom types by specifying additional arguments after the potential filename in the pair_coeff command. Note that atom type 1 must always correspond to oxygen atoms. As an example, to simulate a $TiO_2$ system, atom type 1 has to be oxygen and atom type 2 Ti. The following pair_coeff command should then be used:

```
pair_coeff * * PathToLammps/potentials/ffield.smtbq.TiO2 O Ti
```

The electrostatic part of the energy consists of two components

self-energy of atom $i$ in the form of a second order charge dependent polynomial and a long-range Coulombic electrostatic interaction. The latter uses the wolf summation method described in *Wolf*, spherically truncated at a longer cutoff, $R_{coul}$. The charge of each ion is modeled by an orbital Slater which depends on the principal quantum number ($n$) of the outer orbital shared by the ion.

Interaction between oxygen, $E_{OO}$, consists of two parts, an attractive and a repulsive part. The attractive part is effective only at short range ($< r_2^{OO}$). The attractive contribution was optimized to study surfaces reconstruction (e.g. *SMTB-Q_2* in $TiO_2$) and is not necessary for oxide bulk modeling. The repulsive part is the Pauli interaction between the electron clouds of oxygen. The Pauli repulsion and the coulombic electrostatic interaction have same cut off value. In the ffield.SMTBQ.Syst, the keyword *'buck'* allows to consider only the repulsive O-O interactions. The keyword *'buckPlusAttr'* allows to consider the repulsive and the attractive O-O interactions.

The short-range interaction between metal-oxygen, $E_{MO}$ is based on the second moment approximation of the density of states with a N-body potential for the band energy term, $E_{cov}^i$, and a Born-Mayer type repulsive terms as indicated by the keyword *'second_moment'* in the ffield.SMTBQ.Syst. The energy band term is given by:

$$
\begin{aligned}
E_{cov}^{i(i=M,O)} &= -\left\{ \eta_i (\mu \xi^0)^2 f_{cut}^{r_{c1} r_{c2}}(r_{ij}) \left( \sum_{j(j=O,M)} exp[-2q(\frac{r_{ij}}{r_0} - 1)] \right) \delta Q_i \left(2\frac{n_0}{\eta_i} - \delta Q_i\right) \right\}^{1/2} \\
\delta Q_i &= |Q_i^F| - |Q_i|
\end{aligned}
$$

where $\eta_i$ is the stoichiometry of atom $i$, $\delta Q_i$ is the charge delocalization of atom $i$, compared to its formal charge $Q_i^F$. $n_0$, the number of hybridized orbitals, is calculated with to the atomic orbitals shared $d_i$ and the stoichiometry $\eta_i$. $r_{c1}$ and $r_{c2}$ are the two cutoff radius around the fourth neighbors in the cutoff function.

In the formalism used here, $\xi^0$ is the energy parameter. $\xi^0$ is in tight-binding approximation the hopping integral between the hybridized orbitals of the cation and the anion. In the literature we find many ways to write the hopping integral depending on whether one takes the point of view of the anion or cation. These are equivalent vision. The correspondence between the two visions is explained in appendix A of the article in the SrTiO3 *SMTB-Q_3* (parameter

$\beta$ shown in this article is in fact the $\beta_O$). To summarize the relationship between the hopping integral $\xi^O$ and the others, we have in an oxide $C_nO_m$ the following relationship:

$$\xi^0 = \frac{\xi_O}{m} = \frac{\xi_C}{n}$$
$$\frac{\beta_O}{\sqrt{m}} = \frac{\beta_C}{\sqrt{n}} = \xi^0 \frac{\sqrt{m}+\sqrt{n}}{2}$$

Thus parameter $\mu$, indicated above, is given by $\mu = \frac{1}{2}(\sqrt{n}+\sqrt{m})$

The potential offers the possibility to consider the polarizability of the electron clouds of oxygen by changing the slater radius of the charge density around the oxygen atoms through the parameters *rBB, rB and rS* in the ffield.SMTBQ.Syst. This change in radius is performed according to the method developed by E. Maras *SMTB-Q_2*. This method needs to determine the number of nearest neighbors around the oxygen. This calculation is based on first ($r_{1n}$) and second ($r_{2n}$) distances neighbors.

The SMTB-Q potential is a variable charge potential. The equilibrium charge on each atom is calculated by the electronegativity equalization (QEq) method. See *Rick* for further detail. One can adjust the frequency, the maximum number of iterative loop and the convergence of the equilibrium charge calculation. To obtain the energy conservation in NVE thermodynamic ensemble, we recommend to use a convergence parameter in the interval 10e-5 - 10e-6 eV.

The ffield.SMTBQ.Syst files are provided for few systems. They consist of nine parts and the lines beginning with '#' are comments (note that the number of comment lines matter). The first sections are on the potential parameters and others are on the simulation options and might be modified. Keywords are character type and must be enclosed in quotation marks ('').

1) Number of different element in the oxide:

- N_elem= 2 or 3
- Divider line

2) Atomic parameters

For the anion (oxygen)

- Name of element (char) and stoichiometry in oxide
- Formal charge and mass of element
- Principal quantum number of outer orbital n), electronegativity ($\chi_i^0$) and hardness ($J_i^0$)
- Ionic radius parameters : max coordination number (*coordBB* = 6 by default), bulk coordination number *(coordB)*, surface coordination number *(coordS)* and *rBB, rB and rS* the slater radius for each coordination number. (**note : If you don't want to change the slater radius, use three identical radius values**)
- Number of orbital shared by the element in the oxide ($d_i$)
- Divider line

For each cations (metal):

- Name of element (char) and stoichiometry in oxide
- Formal charge and mass of element
- Number of electron in outer orbital *(ne)*, electronegativity ($\chi_i^0$), hardness ($J_i^0$) and $r_{Slater}$ the slater radius for the cation.
- Number of orbitals shared by the elements in the oxide ($d_i$)
- Divider line

3) Potential parameters:

- Keyword for element1, element2 and interaction potential ('second_moment' or 'buck' or 'buckPlusAttr') between element 1 and 2. If the potential is 'second_moment', specify 'oxide' or 'metal' for metal-oxygen or metal-metal interactions respectively.
- Potential parameter:
  - If type of potential is 'second_moment' : A (eV), $p$, $\zeta^0$ (eV) and $q$, $r_{c1}$(Å), $r_{c2}$(Å) and $r_0$(Å)
  - If type of potential is 'buck' : $C$ (eV) and $\rho$(Å)
  - If type of potential is 'buckPlusAttr' : $C$ (eV) and $\rho$(Å) $D$ (eV), $B$ ($Å^{-1}$), $r_1^{OO}$(Å) and $r_2^{OO}$(Å)
- Divider line

4) Tables parameters:

- Cutoff radius for the Coulomb interaction ($R_{coul}$)
- Starting radius ($r_{min} = 1,18845$Å) and increments ($dr = 0.001$Å) for creating the potential table.
- Divider line

5) Rick model parameter:

- *Nevery* : parameter to set the frequency of the charge resolution. The charges are evaluated each *Nevery* time steps.
- Max number of iterative loop (*loopmax*) and convergence criterion (*prec*) in eV of the charge resolution
- Divider line

6) Coordination parameter:

- First ($r_{1n}$) and second ($r_{2n}$) neighbor distances in angstroms
- Divider line

7) Charge initialization mode:

- Keyword (*QInitMode*) and initial oxygen charge ($Q_{init}$). If keyword = 'true', all oxygen charges are initially set equal to $Q_{init}$. The charges on the cations are initially set in order to respect the neutrality of the box. If keyword = 'false', all atom charges are initially set equal to 0 if you use the *create_atoms* command or the charge specified in the file structure using *read_data* command.
- Divider line

8) Mode for the electronegativity equalization (Qeq)

- Keyword (*mode*) followed by:
  - QEqAll (one QEq group) | no parameters
  - QEqAllParallel (several QEq groups) | no parameters
  - Surface | zlim (QEq only for z>zlim)
- Parameter if necessary
- Divider line

9) Verbose

- If you want the code to work in verbose mode or not : 'true' or 'false'
- If you want to print or not in the file 'Energy_component.txt' the three main contributions to the energy of the system according to the description presented above : 'true' or 'false' and $N_{Energy}$. This option writes to the file every $N_{Energy}$ time steps. If the value is 'false' then $N_{Energy} = 0$. The file takes into account the possibility to have

several QEq groups $g$ then it writes: time step, number of atoms in group $g$, electrostatic part of energy, $E_{ES}$, the interaction between oxygen, $E_{OO}$, and short range metal-oxygen interaction, $E_{MO}$.

- If you want to print to the file 'Electroneg_component.txt' the electronegativity component ($\frac{\partial E_{tot}}{\partial Q_i}$) or not: 'true' or 'false' and $N_{Electroneg}$. This option writes to the file every $N_{Electroneg}$ time steps. If the value is 'false' then $N_{Electroneg} = 0$. The file consist of atom number $i$, atom type (1 for oxygen and # higher than 1 for metal), atom position: $x$, $y$ and $z$, atomic charge of atom $i$, electrostatic part of atom $i$ electronegativity, covalent part of atom $i$ electronegativity, the hopping integral of atom $i$ $(Z\beta^2)_i$ and box electronegativity.

**Note**

This last option slows down the calculation dramatically. Use only with a single processor simulation.

---

### 4.272.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support the *pair_modify* mix, shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in potential files. Thus, you needs to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.272.5 Restrictions

This pair style is part of the SMTBQ package and is only enabled if LAMMPS is built with that package. See the *Build package* page for more info.

This potential requires using atom type 1 for oxygen and atom type higher than 1 for metal atoms.

This pair style requires the *newton* setting to be "on" for pair interactions.

The SMTB-Q potential files provided with LAMMPS (see the potentials directory) are parameterized for metal *units*.

---

### 4.272.6 Citing this work

Please cite related publication: N. Salles, O. Politano, E. Amzallag and R. Tetot, Comput. Mater. Sci. 111 (2016) 181-189

---

**(SMTB-Q_0)** A. Hallil, E. Amzallag, S. Landron, R. Tetot, Surface Science 605 738-745 (2011); R. Tetot, A. Hallil, J. Creuze and I. Braems, EPL, 83 40001 (2008)

**(SMTB-Q_1)** N. Salles, O. Politano, E. Amzallag, R. Tetot, Comput. Mater. Sci. 111 (2016) 181-189

**(SMTB-Q_2)** E. Maras, N. Salles, R. Tetot, T. Ala-Nissila, H. Jonsson, J. Phys. Chem. C 2015, 119, 10391-10399

**(SMTB-Q_3)** R. Tetot, N. Salles, S. Landron, E. Amzallag, Surface Science 616, 19-8722 28 (2013)

**(Wolf)** D. Wolf, P. Keblinski, S. R. Phillpot, J. Eggebrecht, J Chem Phys, 110, 8254 (1999).

**(Rick)** S. W. Rick, S. J. Stuart, B. J. Berne, J Chem Phys 101, 6141 (1994).

# 4.273 pair_style snap command

Accelerator Variants: *snap/intel*, *snap/kk*

## 4.273.1 Syntax

```
pair_style snap
```

## 4.273.2 Examples

```
pair_style snap
pair_coeff * * InP.snapcoeff InP.snapparam In In P P
```

## 4.273.3 Description

Pair style *snap* defines the spectral neighbor analysis potential (SNAP), a machine-learning interatomic potential *(Thompson)*. Like the GAP framework of Bartok et al. *(Bartok2010)*, SNAP uses bispectrum components to characterize the local neighborhood of each atom in a very general way. The mathematical definition of the bispectrum calculation and its derivatives w.r.t. atom positions is identical to that used by *compute snap*, which is used to fit SNAP potentials to *ab initio* energy, force, and stress data. In SNAP, the total energy is decomposed into a sum over atom energies. The energy of atom $i$ is expressed as a weighted sum over bispectrum components.

$$E^i_{SNAP}(B^i_1, ..., B^i_K) = \beta^{\mu_i}_0 + \sum_{k=1}^{K} \beta^{\mu_i}_k B^i_k$$

where $B^i_k$ is the $k$-th bispectrum component of atom $i$, and $\beta^{\mu_i}_k$ is the corresponding linear coefficient that depends on $\mu_i$, the SNAP element of atom $i$. The number of bispectrum components used and their definitions depend on the value of *twojmax* and other parameters defined in the SNAP parameter file described below. The bispectrum calculation is described in more detail in *compute sna/atom*.

Note that unlike for other potentials, cutoffs for SNAP potentials are not set in the pair_style or pair_coeff command; they are specified in the SNAP potential files themselves.

Only a single pair_coeff command is used with the *snap* style which specifies a SNAP coefficient file followed by a SNAP parameter file and then N additional arguments specifying the mapping of SNAP elements to LAMMPS atom types, where N is the number of LAMMPS atom types:

- SNAP coefficient file
- SNAP parameter file
- N element names = mapping of SNAP elements to atom types

As an example, if a LAMMPS indium phosphide simulation has 4 atoms types, with the first two being indium and the third and fourth being phophorous, the pair_coeff command would look like this:

```
pair_coeff * * snap InP.snapcoeff InP.snapparam In In P P
```

The first 2 arguments must be * * so as to span all LAMMPS atom types. The two filenames are for the coefficient and parameter files, respectively. The two trailing 'In' arguments map LAMMPS atom types 1 and 2 to the SNAP 'In' element. The two trailing 'P' arguments map LAMMPS atom types 3 and 4 to the SNAP 'P' element.

If a SNAP mapping value is specified as NULL, the mapping is not performed. This can be used when a *snap* potential is used as part of the *hybrid* pair style. The NULL values are placeholders for atom types that will be used with other potentials.

The name of the SNAP coefficient file usually ends in the ".snapcoeff" extension. It may contain coefficients for many SNAP elements. The only requirement is that each of the unique element names appearing in the LAMMPS pair_coeff command appear exactly once in the SNAP coefficient file. It is okay if the SNAP coefficient file contains additional elements not in the pair_coeff command, except when using *chemflag* (see below). The name of the SNAP parameter file usually ends in the ".snapparam" extension. It contains a small number of parameters that define the overall form of the SNAP potential. See the *pair_coeff* page for alternate ways to specify the path for these files.

SNAP potentials are quite commonly combined with one or more other LAMMPS pair styles using the *hybrid/overlay* pair style. As an example, the SNAP tantalum potential provided in the LAMMPS potentials directory combines the *snap* and *zbl* pair styles. It is invoked by the following commands:

```
variable zblcutinner equal 4
variable zblcutouter equal 4.8
variable zblz equal 73
pair_style hybrid/overlay &
zbl ${zblcutinner} ${zblcutouter} snap
pair_coeff * * zbl 0.0
pair_coeff 1 1 zbl ${zblz}
pair_coeff * * snap Ta06A.snapcoeff Ta06A.snapparam Ta
```

It is convenient to keep these commands in a separate file that can be inserted in any LAMMPS input script using the *include* command.

The top of the SNAP coefficient file can contain any number of blank and comment lines (start with #), but follows a strict format after that. The first non-blank non-comment line must contain two integers:

- nelem = Number of elements
- ncoeff = Number of coefficients

This is followed by one block for each of the *nelem* elements. The first line of each block contains three entries:

- Element name (text string)
- R = Element radius (distance units)
- w = Element weight (dimensionless)

This line is followed by *ncoeff* coefficients, one per line.

The SNAP parameter file can contain blank and comment lines (start with #) anywhere. Each non-blank non-comment line must contain one keyword/value pair. The required keywords are *rcutfac* and *twojmax*. Optional keywords are *rfac0*, *rmin0*, *switchflag*, *bzeroflag*, *quadraticflag*, *chemflag*, *bnormflag*, *wselfallflag*, *switchinnerflag*, *sinner*, *dinner*, *chunksize*, and *parallelthresh*.

The default values for these keywords are

- *rfac0* = 0.99363
- *rmin0* = 0.0
- *switchflag* = 1
- *bzeroflag* = 1
- *quadraticflag* = 0
- *chemflag* = 0
- *bnormflag* = 0
- *wselfallflag* = 0
- *switchinnerflag* = 0

- *chunksize* = 32768
- *parallelthresh* = 8192

For detailed definitions of all of these keywords, see the *compute sna/atom* doc page.

If *quadraticflag* is set to 1, then the SNAP energy expression includes additional quadratic terms that have been shown to increase the overall accuracy of the potential without much increase in computational cost *(Wood)*.

$$E^i_{SNAP}(\mathbf{B}^i) = \beta^{\mu_i}_0 + \beta^{\mu_i} \cdot \mathbf{B}_i + \frac{1}{2}\mathbf{B}^t_i \cdot \alpha^{\mu_i} \cdot \mathbf{B}_i$$

where $\mathbf{B}_i$ is the $K$-vector of bispectrum components, $\beta^{\mu_i}$ is the $K$-vector of linear coefficients for element $\mu_i$, and $\alpha^{\mu_i}$ is the symmetric $K$ by $K$ matrix of quadratic coefficients. The SNAP coefficient file should contain $K(K+1)/2$ additional coefficients in each element block, the upper-triangular elements of $\alpha^{\mu_i}$.

If *chemflag* is set to 1, then the energy expression is written in terms of explicit multi-element bispectrum components indexed on ordered triplets of elements, which has been shown to increase the ability of the SNAP potential to capture energy differences in chemically complex systems, at the expense of a significant increase in computational cost *(Cusentino)*.

$$E^i_{SNAP}(\mathbf{B}^i) = \beta^{\mu_i}_0 + \sum_{\kappa,\lambda,\mu} \beta^{\kappa\lambda\mu}_{\mu_i} \cdot \mathbf{B}^{\kappa\lambda\mu}_i$$

where $\mathbf{B}^{\kappa\lambda\mu}_i$ is the $K$-vector of bispectrum components for neighbors of elements $\kappa$, $\lambda$, and $\mu$ and $\beta^{\kappa\lambda\mu}_{\mu_i}$ is the corresponding $K$-vector of linear coefficients for element $\mu_i$. The SNAP coefficient file should contain a total of $KN^3_{elem}$ coefficients in each element block, where $N_{elem}$ is the number of elements in the SNAP coefficient file, which must equal the number of unique elements appearing in the LAMMPS pair_coeff command, to avoid ambiguity in the number of coefficients.

The keyword *switchinnerflag* activates an additional switching function that smoothly turns off contributions to the SNAP potential from neighbor atoms at short separations. If *switchinnerflag* is set to 1 then the additional keywords *sinner* and *dinner* must also be provided. Each of these is followed by *nelements* values, where *nelements* is the number of unique elements appearing in appearing in the LAMMPS pair_coeff command. The element order should correspond to the order in which elements first appear in the pair_coeff command reading from left to right.

The keywords *chunksize* and *parallelthresh* are only applicable when using the pair style *snap* with the KOKKOS package on GPUs and are ignored otherwise. The *chunksize* keyword controls the number of atoms in each pass used to compute the bispectrum components and is used to avoid running out of memory. For example if there are 8192 atoms in the simulation and the *chunksize* is set to 4096, the bispectrum calculation will be broken up into two passes (running on a single GPU). The *parallelthresh* keyword controls a crossover threshold for performing extra parallelism. For small systems, exposing additional parallelism can be beneficial when there is not enough work to fully saturate the GPU threads otherwise. However, the extra parallelism also leads to more divergence and can hurt performance when the system is already large enough to saturate the GPU threads. Extra parallelism will be performed if the *chunksize* (or total number of atoms per GPU) is smaller than *parallelthresh*.

**Note**

The previously used *diagonalstyle* keyword was removed in 2019, since all known SNAP potentials use the default value of 3.

### 4.273.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, where types I and J correspond to two different element types, mixing is performed by LAMMPS with user-specifiable parameters as described above. You never need to specify a pair_coeff command with I != J arguments for this style.

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.273.5 Restrictions

This style is part of the ML-SNAP package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The *snap/intel* accelerator variant will *only* be available if LAMMPS is built with Intel *compilers* and for CPUs with AVX-512 support. While the INTEL package in general allows multiple floating point precision modes to be selected, *snap/intel* will currently always use full double precision regardless of the precision mode selected. Additionally, the *intel* variant of snap will **NOT** use multiple threads with OpenMP.

### 4.273.6 Related commands

*compute sna/atom*, *compute snad/atom*, *compute snav/atom*, *compute snap*

### 4.273.7 Default

none

---

**(Thompson)** Thompson, Swiler, Trott, Foiles, Tucker, J Comp Phys, 285, 316 (2015).

**(Bartok2010)** Bartok, Payne, Kondor, Csanyi, Phys Rev Lett, 104, 136403 (2010).

**(Wood)** Wood and Thompson, J Chem Phys, 148, 241721, (2018)

**(Cusentino)** Cusentino, Wood, Thompson, J Phys Chem A, 124, 5456, (2020)

## 4.274 pair_style soft command

Accelerator Variants: *soft/gpu*, *soft/kk*, *soft/omp*

### 4.274.1 Syntax

```
pair_style soft cutoff
```

- cutoff = global cutoff for soft interactions (distance units)

### 4.274.2 Examples

```
pair_style soft 1.0
pair_coeff * * 10.0
pair_coeff 1 1 10.0 3.0

pair_style soft 1.0
pair_coeff * * 0.0
variable prefactor equal ramp(0,30)
fix 1 all adapt 1 pair soft a * * v_prefactor
```

### 4.274.3 Description

Style *soft* computes pairwise interactions with the formula

$$E = A \left[1 + \cos\left(\frac{\pi r}{r_c}\right)\right] \qquad r < r_c$$

It is useful for pushing apart overlapping atoms, since it does not blow up as r goes to 0. A is a prefactor that can be made to vary in time from the start to the end of the run (see discussion below), e.g. to start with a very soft potential and slowly harden the interactions over time. $r_c$ is the cutoff. See the *fix nve/limit* command for another way to push apart overlapping atoms.

The following coefficients must be defined for each pair of atom types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- A (energy units)
- cutoff (distance units)

The last coefficient is optional. If not specified, the global soft cutoff is used.

**Note**

The syntax for *pair_coeff* with a single A coeff is different in the current version of LAMMPS than in older versions which took two values, Astart and Astop, to ramp between them. This functionality is now available in a more general form through the *fix adapt* command, as explained below. Note that if you use an old input script and specify Astart and Astop without a cutoff, then LAMMPS will interpret that as A and a cutoff, which is probably not what you want.

The *fix adapt* command can be used to vary A for one or more pair types over the course of a simulation, in which case pair_coeff settings for A must still be specified, but will be overridden. For example these commands will vary the prefactor A for all pairwise interactions from 0.0 at the beginning to 30.0 at the end of a run:

```
variable prefactor equal ramp(0,30)
fix 1 all adapt 1 pair soft a * * v_prefactor
```

Note that a formula defined by an *equal-style variable* can use the current timestep, elapsed time in the current run, elapsed time since the beginning of a series of runs, as well as access other variables.

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 4.274.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the A coefficient and cutoff distance for this pair style can be mixed. A is always mixed via a *geometric* rule. The cutoff is mixed according to the pair_modify mix value. The default mix value is *geometric*. See the "pair_modify" command for details.

This pair style does not support the *pair_modify* shift option, since the pair interaction goes to 0.0 at the cutoff.

The *pair_modify* table and tail options are not relevant for this pair style.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.274.5 Restrictions

none

### 4.274.6 Related commands

*pair_coeff*, *fix nve/limit*, *fix adapt*

### 4.274.7 Default

none

## 4.275 pair_style sph/heatconduction command

Accelerator Variants: *sph/heatconduction/gpu*

### 4.275.1 Syntax

```
pair_style sph/heatconduction
```

### 4.275.2 Examples

```
pair_style sph/heatconduction
pair_coeff * * 1.0 2.4
```

### 4.275.3 Description

The sph/heatconduction style computes heat transport between SPH particles. The transport model is the diffusion equation for the internal energy.

See this PDF guide to using SPH in LAMMPS.

**Note**

Please note that the SPH PDF guide file has not been updated for many years and thus does not reflect the current *syntax* of the SPH package commands. For that please refer to the LAMMPS manual.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above.

- D diffusion coefficient (length^2/time units)
- h kernel function cutoff (distance units)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.275.4 Mixing, shift, table, tail correction, restart, rRESPA info

This style does not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

This style does not support the *pair_modify* shift, table, and tail options.

This style does not write information to *binary restart files*. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.275.5 Restrictions

This pair style is part of the SPH package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.275.6 Related commands

*pair_coeff*, pair_sph/rhosum

### 4.275.7 Default

none

## 4.276 pair_style sph/idealgas command

### 4.276.1 Syntax

```
pair_style sph/idealgas
```

### 4.276.2 Examples

```
pair_style sph/idealgas
pair_coeff * * 1.0 2.4
```

### 4.276.3 Description

The sph/idealgas style computes pressure forces between particles according to the ideal gas equation of state:

$$p = (\gamma - 1)\rho e$$

where $\gamma = 1.4$ is the heat capacity ratio, $\rho$ is the local density, and e is the internal energy per unit mass. This pair style also computes Monaghan's artificial viscosity to prevent particles from interpenetrating *(Monaghan)*.

See this PDF guide to using SPH in LAMMPS.

**Note**

Please note that the SPH PDF guide file has not been updated for many years and thus does not reflect the current *syntax* of the SPH package commands. For that please refer to the LAMMPS manual.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above.

- $\nu$ artificial viscosity (no units)
- h kernel function cutoff (distance units)

---

### 4.276.4 Mixing, shift, table, tail correction, restart, rRESPA info

This style does not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

This style does not support the *pair_modify* shift, table, and tail options.

This style does not write information to *binary restart files*. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.276.5 Restrictions

This pair style is part of the SPH package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.276.6 Related commands

*pair_coeff*, pair_sph/rhosum

### 4.276.7 Default

none

---

**(Monaghan)** Monaghan and Gingold, Journal of Computational Physics, 52, 374-389 (1983).

## 4.277 pair_style sph/lj command

Accelerator Variants: *sph/lj/gpu*

### 4.277.1 Syntax

```
pair_style sph/lj
```

### 4.277.2 Examples

```
pair_style sph/lj
pair_coeff * * 1.0 2.4
```

### 4.277.3 Description

The sph/lj style computes pressure forces between particles according to the Lennard-Jones equation of state, which is computed according to Ree's 1980 polynomial fit *(Ree)*. The Lennard-Jones parameters epsilon and sigma are set to unity. This pair style also computes Monaghan's artificial viscosity to prevent particles from interpenetrating *(Monaghan)*.

See this PDF guide to using SPH in LAMMPS.

> **Note**
>
> Please note that the SPH PDF guide file has not been updated for many years and thus does not reflect the current *syntax* of the SPH package commands. For that please refer to the LAMMPS manual.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above.

- $\nu$ artificial viscosity (no units)
- h kernel function cutoff (distance units)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages*

page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.277.4 Mixing, shift, table, tail correction, restart, rRESPA info

This style does not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

This style does not support the *pair_modify* shift, table, and tail options.

This style does not write information to *binary restart files*. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.277.5 Restrictions

As noted above, the Lennard-Jones parameters epsilon and sigma are set to unity.

This pair style is part of the SPH package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.277.6 Related commands

*pair_coeff*, pair_sph/rhosum

### 4.277.7 Default

none

---

**(Ree)** Ree, Journal of Chemical Physics, 73, 5401 (1980).

**(Monaghan)** Monaghan and Gingold, Journal of Computational Physics, 52, 374-389 (1983).

## 4.278 pair_style sph/rhosum command

### 4.278.1 Syntax

```
pair_style sph/rhosum Nstep
```

- Nstep = timestep interval

### 4.278.2 Examples

```
pair_style sph/rhosum 10
pair_coeff * * 2.4
```

### 4.278.3 Description

The sph/rhosum style computes the local particle mass density rho for SPH particles by kernel function interpolation, every Nstep timesteps.

See this PDF guide to using SPH in LAMMPS.

> **Note**
>
> Please note that the SPH PDF guide file has not been updated for many years and thus does not reflect the current *syntax* of the SPH package commands. For that please refer to the LAMMPS manual.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above.

- h (distance units)

---

### 4.278.4 Mixing, shift, table, tail correction, restart, rRESPA info

This style does not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

This style does not support the *pair_modify* shift, table, and tail options.

This style does not write information to *binary restart files*. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.278.5 Restrictions

This pair style is part of the SPH package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.278.6 Related commands

*pair_coeff*, pair_sph/taitwater

### 4.278.7 Default

none

## 4.279 pair_style sph/taitwater command

Accelerator Variants: *sph/taitwater/gpu*

### 4.279.1 Syntax

```
pair_style sph/taitwater
```

### 4.279.2 Examples

```
pair_style sph/taitwater
pair_coeff * * 1000.0 1430.0 1.0 2.4
```

### 4.279.3 Description

The sph/taitwater style computes pressure forces between SPH particles according to Tait's equation of state:

$$p = B\left[\left(\frac{\rho}{\rho_0}\right)^{\gamma} - 1\right]$$

where $\gamma = 7$ and $B = c_0^2\rho_0/\gamma$, with $\rho_0$ being the reference density and $c_0$ the reference speed of sound.

This pair style also computes Monaghan's artificial viscosity to prevent particles from interpenetrating *(Monaghan)*.

See this PDF guide to using SPH in LAMMPS.

**Note**

Please note that the SPH PDF guide file has not been updated for many years and thus does not reflect the current *syntax* of the SPH package commands. For that please refer to the LAMMPS manual.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above.

- $\rho_0$ reference density (mass/volume units)
- $c_0$ reference soundspeed (distance/time units)
- $\nu$ artificial viscosity (no units)
- h kernel function cutoff (distance units)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.279.4 Mixing, shift, table, tail correction, restart, rRESPA info

This style does not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

This style does not support the *pair_modify* shift, table, and tail options.

This style does not write information to *binary restart files*. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.279.5 Restrictions

This pair style is part of the SPH package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.279.6 Related commands

*pair_coeff*, pair_sph/rhosum

### 4.279.7 Default

none

---

**(Monaghan)** Monaghan and Gingold, Journal of Computational Physics, 52, 374-389 (1983).

## 4.280 pair_style sph/taitwater/morris command

### 4.280.1 Syntax

```
pair_style sph/taitwater/morris
```

### 4.280.2 Examples

```
pair_style sph/taitwater/morris
pair_coeff * * 1000.0 1430.0 1.0 2.4
```

### 4.280.3 Description

The sph/taitwater/morris style computes pressure forces between SPH particles according to Tait's equation of state:

$$p = B\left[\left(\frac{\rho}{\rho_0}\right)^{\gamma} - 1\right]$$

where $\gamma = 7$ and $B = c_0^2\rho_0/\gamma$, with $\rho_0$ being the reference density and $c_0$ the reference speed of sound.

This pair style also computes laminar viscosity *(Morris)*.

See this PDF guide to using SPH in LAMMPS.

**Note**

Please note that the SPH PDF guide file has not been updated for many years and thus does not reflect the current *syntax* of the SPH package commands. For that please refer to the LAMMPS manual.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above.

- $\rho_0$ reference density (mass/volume units)
- $c_0$ reference soundspeed (distance/time units)
- $\nu$ dynamic viscosity (mass*distance/time units)
- h kernel function cutoff (distance units)

---

### 4.280.4 Mixing, shift, table, tail correction, restart, rRESPA info

This style does not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

This style does not support the *pair_modify* shift, table, and tail options.

This style does not write information to *binary restart files*. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.280.5 Restrictions

This pair style is part of the SPH package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.280.6 Related commands

*pair_coeff*, pair_sph/rhosum

### 4.280.7 Default

none

---

**(Morris)** Morris, Fox, Zhu, J Comp Physics, 136, 214-226 (1997).

## 4.281 pair_style lj/spica command

Accelerator Variants: *lj/spica/gpu*, *lj/spica/kk*, *lj/spica/omp*

## 4.282 pair_style lj/spica/coul/long command

Accelerator Variants: *lj/spica/coul/long/gpu*, *lj/spica/coul/long/omp*, *lj/spica/coul/long/kk*

## 4.283 pair_style lj/spica/coul/msm command

Accelerator Variants: *lj/spica/coul/msm/omp*

### 4.283.1 Syntax

```
pair_style style args
```

- style = *lj/spica* or *lj/spica/coul/long*
- args = list of arguments for a particular style

```
lj/spica args = cutoff
  cutoff = global cutoff for Lennard Jones interactions (distance units)
lj/spica/coul/long args = cutoff (cutoff2)
  cutoff = global cutoff for LJ (and Coulombic if only 1 arg) (distance units)
  cutoff2 = global cutoff for Coulombic (optional) (distance units)
```

### 4.283.2 Examples

```
pair_style lj/spica 2.5
pair_coeff 1 1 lj12_6 1 1.1 2.8

pair_style lj/spica/coul/long 10.0
pair_style lj/spica/coul/long 10.0 12.0
pair_coeff 1 1 lj9_6 100.0 3.5 12.0

pair_style lj/spica/coul/msm 10.0
pair_style lj/spica/coul/msm 10.0 12.0
pair_coeff 1 1 lj9_6 100.0 3.5 12.0
```

### 4.283.3 Description

The *lj/spica* styles compute a 9/6, 12/4, 12/5, or 12/6 Lennard-Jones potential, given by

$$E = \frac{27}{4}\varepsilon\left[\left(\frac{\sigma}{r}\right)^9 - \left(\frac{\sigma}{r}\right)^6\right] \qquad r < r_c$$

$$E = \frac{3\sqrt{3}}{2}\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^4\right] \qquad r < r_c$$

$$E = \frac{12}{7}\left(\frac{12}{5}\right)^{\left(\frac{5}{7}\right)}\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^5\right] \qquad r < r_c$$

$$E = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^6\right] \qquad r < r_c$$

as required for the SPICA (formerly called SDK) and the pSPICA Coarse-grained MD parameterization discussed in *(Shinoda)*, *(DeVane)*, *(Seo)*, and *(Miyazaki)*. $r_c$ is the cutoff. Summary information on these force fields can be found at https://www.spica-ff.org

Style *lj/spica/coul/long* computes the adds Coulombic interactions with an additional damping factor applied so it can be used in conjunction with the *kspace_style* command and its *ewald* or *pppm* or *pppm/cg* option. The Coulombic cutoff specified for this style means that pairwise interactions within this distance are computed directly; interactions outside that distance are computed in reciprocal space.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- cg_type (lj9_6, lj12_4, lj12_5, or lj12_6)
- epsilon (energy units)
- sigma (distance units)
- cutoff1 (distance units)

Note that sigma is defined in the LJ formula as the zero-crossing distance for the potential, not as the energy minimum. The prefactors are chosen so that the potential minimum is at -epsilon.

The latter 2 coefficients are optional. If not specified, the global LJ and Coulombic cutoffs specified in the pair_style command are used. If only one cutoff is specified, it is used as the cutoff for both LJ and Coulombic interactions for this type pair. If both coefficients are specified, they are used as the LJ and Coulombic cutoffs for this type pair.

For *lj/spica/coul/long* and *lj/spica/coul/msm* only the LJ cutoff can be specified since a Coulombic cutoff cannot be specified for an individual I,J type pair. All type pairs use the same global Coulombic cutoff specified in the pair_style command.

The original implementation of the above styles are style *lj/sdk*, *lj/sdk/coul/long*, and *lj/sdk/coul/msm*, and available for backward compatibility.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.283.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon and sigma coefficients and cutoff distance for all of the lj/spica pair styles *cannot* be mixed, since different pairs may have different exponents. So all parameters for all pairs have to be specified explicitly through the "pair_coeff" command. Defining then in a data file is also not supported, due to limitations of that file format.

All of the lj/spica pair styles support the *pair_modify* shift option for the energy of the Lennard-Jones portion of the pair interaction.

The *lj/spica/coul/long* pair styles support the *pair_modify* table option since they can tabulate the short-range portion of the long-range Coulombic interaction.

All of the lj/spica pair styles write their information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

The lj/spica and lj/cut/coul/long pair styles do not support the use of the *inner*, *middle*, and *outer* keywords of the *run_style respa* command.

---

### 4.283.5 Restrictions

All of the lj/spica pair styles are part of the CG-SPICA package. The *lj/spica/coul/long* style also requires the KSPACE package to be built (which is enabled by default). They are only enabled if LAMMPS was built with that package. See the *Build package* doc page for more info.

### 4.283.6 Related commands

*pair_coeff* , *angle_style spica*

### 4.283.7 Default

none

---

**(Shinoda)** Shinoda, DeVane, Klein, Mol Sim, 33, 27-36 (2007).

**(DeVane)** Shinoda, DeVane, Klein, Soft Matter, 4, 2453-2462 (2008).

**(Seo)** Seo, Shinoda, J Chem Theory Comput, 15, 762-774 (2019).

**(Miyazaki)** Miyazaki, Okazaki, Shinoda, J Chem Theory Comput, 16, 782-793 (2020).

# 4.284 pair_style spin/dipole/cut command

# 4.285 pair_style spin/dipole/long command

## 4.285.1 Syntax

```
pair_style spin/dipole/cut cutoff
pair_style spin/dipole/long cutoff
```

- cutoff = global cutoff for magnetic dipole energy and forces (optional) (distance units)

## 4.285.2 Examples

```
pair_style spin/dipole/cut 10.0
pair_coeff * * 10.0
pair_coeff 2 3 8.0

pair_style spin/dipole/long 9.0
pair_coeff * * 10.0
pair_coeff 2 3 6.0
```

## 4.285.3 Description

Style *spin/dipole/cut* computes a short-range dipole-dipole interaction between pairs of magnetic particles that each have a magnetic spin. The magnetic dipole-dipole interactions are computed by the following formulas for the magnetic energy, magnetic precession vector omega and mechanical force between particles I and J.

$$\mathcal{H}_{\text{long}} = -\frac{\mu_0 \left(\mu_B\right)^2}{4\pi} \sum_{i,j,i\neq j}^{N} \frac{g_i g_j}{r_{ij}^3} \left( 3 \left(\vec{e}_{ij} \cdot \vec{s}_i\right) \left(\vec{e}_{ij} \cdot \vec{s}_j\right) - \vec{s}_i \cdot \vec{s}_j \right)$$

$$\omega_i = \frac{\mu_0 (\mu_B)^2}{4\pi\hbar} \sum_j \frac{g_i g_j}{r_{ij}^3} \left( 3 \left(\vec{e}_{ij} \cdot \vec{s}_j\right) \vec{e}_{ij} - \vec{s}_j \right)$$

$$\mathbf{F}_i = \frac{3\,\mu_0 (\mu_B)^2}{4\pi} \sum_j \frac{g_i g_j}{r_{ij}^4} \left[ \left( (\vec{s}_i \cdot \vec{s}_j) - 5 (\vec{e}_{ij} \cdot \vec{s}_i)(\vec{e}_{ij} \cdot \vec{s}_j) \right) \vec{e}_{ij} + \left( (\vec{e}_{ij} \cdot \vec{s}_i) \vec{s}_j + (\vec{e}_{ij} \cdot \vec{s}_j) \vec{s}_i \right) \right]$$

where $\vec{s}_i$ and $\vec{s}_j$ are the spin on two magnetic particles, r is their separation distance, and the vector $\vec{e}_{ij} = \frac{r_i - r_j}{|r_i - r_j|}$ is the direction vector between the two particles.

Style *spin/dipole/long* computes long-range magnetic dipole-dipole interaction. A *kspace_style* must be defined to use this pair style. Currently, *kspace_style ewald/dipole/spin* and *kspace_style pppm/dipole/spin* support long-range magnetic dipole-dipole interactions.

---

The *pair_modify* table option is not relevant for this pair style.

This pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

### 4.285.4 Restrictions

The *spin/dipole/cut* and *spin/dipole/long* styles are part of the SPIN package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Using dipole/spin pair styles with *electron units* is not currently supported.

### 4.285.5 Related commands

*pair_coeff* , *kspace_style fix nve/spin*

### 4.285.6 Default

none

## 4.286 pair_style spin/dmi command

### 4.286.1 Syntax

```
pair_style spin/dmi cutoff
```

- cutoff = global cutoff pair (distance in metal units)

### 4.286.2 Examples

```
pair_style spin/dmi 4.0
pair_coeff * * dmi 2.6 0.001 1.0 0.0 0.0
pair_coeff 1 2 dmi 4.0 0.00109 0.0 0.0 1.0
```

### 4.286.3 Description

Style *spin/dmi* computes the Dzyaloshinskii-Moriya (DM) interaction between pairs of magnetic spins. According to the expression reported in *(Rohart)*, one has the following DM energy:

$$\mathbf{H}_{dm} = \sum_{i,j=1,i\neq j}^{N} \left(\vec{e}_{ij} \times \vec{D}\right) \cdot \left(\vec{s}_i \times \vec{s}_j\right),$$

where $\vec{s}_i$ and $\vec{s}_j$ are two neighboring magnetic spins of two particles, $\vec{e}_i j = \frac{r_i - r_j}{|r_i - r_j|}$ is the unit vector between sites $i$ and $j$, and $\vec{D}$ is the DM vector defining the intensity (in eV) and the direction of the interaction.

In *(Rohart)*, $\vec{D}$ is defined as the direction normal to the film oriented from the high spin-orbit layer to the magnetic ultra-thin film.

The application of a spin-lattice Poisson bracket to this energy (as described in *(Tranchida)*) allows to derive a magnetic torque omega, and a mechanical force F (for spin-lattice calculations only) for each magnetic particle i:

$$\vec{\omega}_i = -\frac{1}{\hbar} \sum_{j}^{Neighb} \vec{s}_j \times \left(\vec{e}_{ij} \times \vec{D}\right) \quad \text{and} \quad \vec{F}_i = -\sum_{j}^{Neighb} \frac{1}{r_{ij}} \vec{D} \times \left(\vec{s}_i \times \vec{s}_j\right)$$

More details about the derivation of these torques/forces are reported in *(Tranchida)*.

For the *spin/dmi* pair style, the following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, and set in the following order:

- rc (distance units)

- |D| (energy units)
- Dx, Dy, Dz (direction of D)

Note that rc is the radius cutoff of the considered DM interaction, |D| is the norm of the DM vector (in eV), and Dx, Dy and Dz define its direction.

None of those coefficients is optional. If not specified, the *spin/dmi* pair style cannot be used.

---

### 4.286.4 Restrictions

All the *pair/spin* styles are part of the SPIN package. These styles are only enabled if LAMMPS was built with this package, and if the atom_style "spin" was declared. See the *Build package* page for more info.

### 4.286.5 Related commands

*atom_style spin*, *pair_coeff*, *pair_eam*,

### 4.286.6 Default

none

---

**(Rohart)** Rohart and Thiaville, Physical Review B, 88(18), 184422. (2013).

**(Tranchida)** Tranchida, Plimpton, Thibaudeau and Thompson, Journal of Computational Physics, 372, 406-425, (2018).

## 4.287 pair_style spin/exchange command

## 4.288 pair_style spin/exchange/biquadratic command

### 4.288.1 Syntax

```
pair_style spin/exchange cutoff
pair_style spin/exchange/biquadratic cutoff
```

- cutoff = global cutoff pair (distance in metal units)

### 4.288.2 Examples

```
pair_style spin/exchange 4.0
pair_coeff * * exchange 4.0 0.0446928 0.003496 1.4885
pair_coeff 1 2 exchange 6.0 -0.01575 0.0 1.965 offset yes

pair_style spin/exchange/biquadratic 4.0
pair_coeff * * biquadratic 4.0 0.05 0.03 1.48 0.05 0.03 1.48 offset no
pair_coeff 1 2 biquadratic 6.0 -0.01 0.0 1.9 0.0 0.1 19
```

### 4.288.3 Description

Style *spin/exchange* computes the exchange interaction between pairs of magnetic spins:

$$H_{ex} = -\sum_{i,j}^{N} J_{ij}(r_{ij})\,\vec{s}_i \cdot \vec{s}_j$$

where $\vec{s}_i$ and $\vec{s}_j$ are two unit vectors representing the magnetic spins of two particles (usually atoms), and $r_{ij} = |\vec{r}_i - \vec{r}_j|$ is the inter-atomic distance between those two particles. The summation is over pairs of nearest neighbors. $J(r_{ij})$ is a function defining the intensity and the sign of the exchange interaction for different neighboring shells.

Style *spin/exchange/biquadratic* computes a biquadratic exchange interaction between pairs of magnetic spins:

$$H_{bi} = -\sum_{i,j}^{N} J_{ij}\left(r_{ij}\right)\,\vec{s}_i \cdot \vec{s}_j - \sum_{i,j}^{N} K_{ij}\left(r_{ij}\right)\,\left(\vec{s}_i \cdot \vec{s}_j\right)^2$$

where $\vec{s}_i$, $\vec{s}_j$, $r_{ij}$ and $J(r_{ij})$ have the same definitions as above, and $K(r_{ij})$ is a second function, defining the intensity and the sign of the biquadratic term.

The interatomic dependence of $J(r_{ij})$ and $K(r_{ij})$ in both interactions above is defined by the following function:

$$f\left(r_{ij}\right) = 4a\left(\frac{r_{ij}}{d}\right)^2\left(1 - b\left(\frac{r_{ij}}{d}\right)^2\right)e^{-\left(\frac{r_{ij}}{d}\right)^2}\Theta(R_c - r_{ij})$$

where $a$, $b$ and $d$ are the three constant coefficients defined in the associated "pair_coeff" command, and $R_c$ is the radius cutoff associated to the pair interaction (see below for more explanations).

The coefficients $a$, $b$, and $d$ need to be fitted so that the function above matches with the value of the exchange interaction for the $N$ neighbor shells taken into account. Examples and more explanations about this function and its parameterization are reported in *(Tranchida)*.

When a *spin/exchange/biquadratic* pair style is defined, six coefficients (three for $J(r_{ij})$, and three for $K(r_{ij})$) have to be fitted.

From this exchange interaction, each spin $i$ will be submitted to a magnetic torque $\vec{\omega}_i$, and its associated atom can be submitted to a force $\vec{F}_i$ for spin-lattice calculations (see *fix nve/spin*), such as:

$$\vec{\omega}_i = \frac{1}{\hbar}\sum_{j}^{Neighb} J\left(r_{ij}\right)\,\vec{s}_j \quad \text{and} \quad \vec{F}_i = \sum_{j}^{Neighb} \frac{\partial J\left(r_{ij}\right)}{\partial r_{ij}}\left(\vec{s}_i \cdot \vec{s}_j\right)\vec{e}_{ij}$$

with $\hbar$ the Planck constant (in metal units), and $\vec{e}_{ij} = \frac{\vec{r}_i - \vec{r}_j}{|\vec{r}_i - \vec{r}_j|}$ the unit vector between sites $i$ and $j$. Equivalent forces and magnetic torques are generated for the biquadratic term when a *spin/exchange/biquadratic* pair style is defined.

More details about the derivation of these torques/forces are reported in *(Tranchida)*.

For the *spin/exchange* and *spin/exchange/biquadratic* pair styles, the following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, and set in the following order:

- $R_c$ (distance units)
- $a$ (energy units)
- $b$ (adim parameter)
- $d$ (distance units)

for the *spin/exchange* pair style, and:

- $R_c$ (distance units)
- $a_j$ (energy units)

- $b_j$ (adim parameter)
- $d_j$ (distance units)
- $a_k$ (energy units)
- $b_k$ (adim parameter)
- $d_k$ (distance units)

for the *spin/exchange/biquadratic* pair style.

Note that $R_c$ is the radius cutoff of the considered exchange interaction, and $a$, $b$ and $d$ are the three coefficients performing the parameterization of the function $J(r_{ij})$ defined above (in the *biquadratic* style, $a_j$, $b_j$, $d_j$ and $a_k$, $b_k$, $d_k$ are the coefficients of $J(r_{ij})$ and $K(r_{ij})$ respectively).

None of those coefficients is optional. If not specified, the *spin/exchange* pair style cannot be used.

---

**Offsetting magnetic forces and energies**:

For spin-lattice simulation, it can be useful to offset the mechanical forces and energies generated by the exchange interaction. The *offset* keyword allows to apply this offset. By setting *offset* to *yes*, the energy definitions above are replaced by:

$$H_{ex} = -\sum_{i,j}^{N} J_{ij}(r_{ij}) \left[\vec{s}_i \cdot \vec{s}_j - 1\right]$$

for the *spin/exchange* pair style, and:

$$H_{bi} = -\sum_{i,j}^{N} J_{ij}\left(r_{ij}\right) \left[\vec{s}_i \cdot \vec{s}_j - 1\right] - \sum_{i,j}^{N} K_{ij}\left(r_{ij}\right) \left[\left(\vec{s}_i \cdot \vec{s}_j\right)^2 - 1\right]$$

for the *spin/exchange/biquadratic* pair style.

Note that this offset only affects the calculation of the energy and mechanical forces. It does not modify the calculation of the precession vectors (and thus does no impact the purely magnetic properties). This ensures that when all spins are aligned, the magnetic energy and the associated mechanical forces (and thus the pressure generated by the magnetic potential) are null.

**Note**

This offset term can be very important when calculations such as equations of state (energy vs volume, or energy vs pressure) are being performed. Indeed, setting the *offset* term ensures that at the ground state of the crystal and at the equilibrium magnetic configuration (typically ferromagnetic), the pressure is null, as expected. Otherwise, magnetic forces could generate a residual pressure.

When the *offset* option is set to *no*, no offset is applied (also corresponding to the default option).

---

## 4.288.4 Restrictions

All the *pair/spin* styles are part of the SPIN package. These styles are only enabled if LAMMPS was built with this package, and if the atom_style "spin" was declared. See the *Build package* page for more info.

### 4.288.5 Related commands

*atom_style spin*, *pair_coeff*, *pair_eam*,

### 4.288.6 Default

The default *offset* keyword value is *no*.

---

**(Tranchida)** Tranchida, Plimpton, Thibaudeau and Thompson, Journal of Computational Physics, 372, 406-425, (2018).

## 4.289 pair_style spin/magelec command

### 4.289.1 Syntax

```
pair_style spin/magelec cutoff
```

- cutoff = global cutoff pair (distance in metal units)

### 4.289.2 Examples

```
pair_style spin/magelec 4.5
pair_coeff * * magelec 4.5 0.00109 1.0 1.0 1.0
```

### 4.289.3 Description

Style *spin/me* computes a magneto-electric interaction between pairs of magnetic spins. According to the derivation reported in *(Katsura)*, this interaction is defined as:

$$\vec{\omega}_i = -\frac{1}{\hbar}\sum_{j}^{Neighb} \vec{s}_j \times \vec{D}(r_{ij})$$

$$\vec{F}_i = -\sum_{j}^{Neighb} \frac{\partial D(r_{ij})}{\partial r_{ij}} \left(\vec{s}_i \times \vec{s}_j\right) \cdot \vec{r}_{ij}$$

where $\vec{s}_i$ and $\vec{s}_j$ are neighboring magnetic spins of two particles.

From this magneto-electric interaction, each spin i will be submitted to a magnetic torque omega, and its associated atom can be submitted to a force F for spin-lattice calculations (see *fix nve/spin*), such as:

$$\vec{F}^i = -\sum_{j}^{Neighbor} \left(\vec{s}_i \times \vec{s}_j\right) \times \vec{E}$$

$$\vec{\omega}^i = -\frac{1}{\hbar}\sum_{j}^{Neighbor} \vec{s}_j \times \left(\vec{E} \times r_{ij}\right)$$

with h the Planck constant (in metal units) and $\vec{E}$ an electric polarization vector. The norm and direction of E are giving the intensity and the direction of a screened dielectric atomic polarization (in eV).

More details about the derivation of these torques/forces are reported in *(Tranchida)*.

---

### 4.289.4 Restrictions

All the *pair/spin* styles are part of the SPIN package. These styles are only enabled if LAMMPS was built with this package, and if the atom_style "spin" was declared. See the *Build package* page for more info.

### 4.289.5 Related commands

*atom_style spin*, *pair_coeff*, *pair_style spin/exchange*, *pair_eam*,

### 4.289.6 Default

none

---

**(Katsura)** H. Katsura, N. Nagaosa, A.V. Balatsky. Phys. Rev. Lett., 95(5), 057205. (2005)

**(Tranchida)** Tranchida, Plimpton, Thibaudeau, and Thompson, Journal of Computational Physics, 372, 406-425, (2018).

## 4.290 pair_style spin/neel command

### 4.290.1 Syntax

```
pair_style spin/neel cutoff
```

- cutoff = global cutoff pair (distance in metal units)

### 4.290.2 Examples

```
pair_style spin/neel 4.0
pair_coeff * * neel 4.0 0.0048 0.234 1.168 2.6905 0.705 0.652
pair_coeff 1 2 neel 4.0 0.0048 0.234 1.168 0.0 0.0 1.0
```

### 4.290.3 Description

Style *spin/neel* computes the Neel pair anisotropy model between pairs of magnetic spins:

$$\mathcal{H}_{N\acute{e}el} = -\sum_{i,j=1,i\neq j}^{N} g_1(r_{ij})\left((\mathbf{e}_{ij}\cdot\mathbf{s}_i)(\mathbf{e}_{ij}\cdot\mathbf{s}_j)-\frac{\mathbf{s}_i\cdot\mathbf{s}_j}{3}\right)+q_1(r_{ij})\left((\mathbf{e}_{ij}\cdot\mathbf{s}_i)^2-\frac{\mathbf{s}_i\cdot\mathbf{s}_j}{3}\right)\left((\mathbf{e}_{ij}\cdot\mathbf{s}_i)^2-\frac{\mathbf{s}_i\cdot\mathbf{s}_j}{3}\right)+q_2(r_{ij})\Big((\mathbf{e}_{ij}\cdot\mathbf{s}_i)(\mathbf{e}_{ij}\cdot\mathbf{s}_j)^3+(\mathbf{e}_{ij}$$

where $\mathbf{s}_i$ and $\mathbf{s}_j$ are two neighboring magnetic spins of two particles, $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$ is the inter-atomic distance between the two particles, $\mathbf{e}_{ij} = \frac{\mathbf{r}_i-\mathbf{r}_j}{|\mathbf{r}_i-\mathbf{r}_j|}$ is their normalized separation vector and $g_1$, $q_1$ and $q_2$ are three functions defining the intensity of the dipolar and quadrupolar contributions, with:

$$g_1(r_{ij}) = g(r_{ij}) + \frac{12}{35}q(r_{ij})$$
$$q_1(r_{ij}) = \frac{9}{5}q(r_{ij})$$
$$q_2(r_{ij}) = -\frac{2}{5}q(r_{ij})$$

With the functions $g(r_{ij})$ and $q(r_{ij})$ defined and fitted according to the same Bethe-Slater function used to fit the exchange interaction:

$$J\left(r_{ij}\right) = 4a\left(\frac{r_{ij}}{d}\right)^2\left(1-b\left(\frac{r_{ij}}{d}\right)^2\right)e^{-\left(\frac{r_{ij}}{d}\right)^2}\Theta(R_c - r_{ij})$$

where $a$, $b$ and $d$ are the three constant coefficients defined in the associated "pair_coeff" command.

The coefficients $a$, $b$, and $d$ need to be fitted so that the function above matches with the values of the magneto-elastic constant of the materials at stake.

Examples and more explanations about this function and its parameterization are reported in *(Tranchida)*. More examples of parameterization will be provided in future work.

From this DM interaction, each spin $i$ will be submitted to a magnetic torque $\omega$ and its associated atom to a force $\mathbf{F}$ (for spin-lattice calculations only).

More details about the derivation of these torques/forces are reported in *(Tranchida)*.

---

### 4.290.4 Restrictions

All the *pair/spin* styles are part of the SPIN package. These styles are only enabled if LAMMPS was built with this package, and if the atom_style "spin" was declared. See the *Build package* page for more info.

### 4.290.5 Related commands

*atom_style spin*, *pair_coeff*, *pair_eam*,

### 4.290.6 Default

none

---

**(Tranchida)** Tranchida, Plimpton, Thibaudeau and Thompson, Journal of Computational Physics, 372, 406-425, (2018).

## 4.291 pair_style srp command

## 4.292 pair_style srp/react command

### 4.292.1 Syntax

```
pair_style srp cutoff btype dist keyword value ...
pair_style srp/react cutoff btype dist react-id keyword value ...
```

- cutoff = global cutoff for SRP interactions (distance units)
- btype = bond type (numeric, type label, or wildcard) to apply SRP interactions to
- distance = *min* or *mid*
- react-id = id of either fix bond/break or fix bond/create
- zero or more keyword/value pairs may be appended
- keyword = *exclude*

  *bptype* value = atom type (numeric or type label) for bond particles
  *exclude* value = *yes* or *no*

### 4.292.2 Examples

```
pair_style hybrid dpd 1.0 1.0 12345 srp 0.8 1 mid exclude yes
pair_coeff 1 1 dpd 60.0 4.5 1.0
pair_coeff 1 2 none
pair_coeff 2 2 srp 100.0 0.8

pair_style hybrid dpd 1.0 1.0 12345 srp 0.8 * min exclude yes
pair_coeff 1 1 dpd 60.0 50 1.0
pair_coeff 1 2 none
pair_coeff 2 2 srp 40.0

fix        create all bond/create   100  1  2  1.0 1 prob  0.2  19852
pair_style hybrid dpd 1.0 1.0 12345 srp/react 0.8 * min create exclude yes
pair_coeff 1 1 dpd 60.0 50 1.0
pair_coeff 1 2 none
pair_coeff 2 2 srp/react 40.0

pair_style hybrid srp 0.8 2 mid
pair_coeff 1 1 none
pair_coeff 1 2 none
pair_coeff 2 2 srp 100.0 0.8

labelmap bond 1 C-C
pair_style hybrid srp 0.8 C-C mid
```

### 4.292.3 Description

Style *srp* computes a soft segmental repulsive potential (SRP) that acts between pairs of bonds. This potential is useful for preventing bonds from passing through one another when a soft non-bonded potential acts between beads in, for example, DPD polymer chains. An example input script that uses this command is provided in examples/PACKAGES/srp.

Bonds of specified type *btype* interact with one another through a bond-pairwise potential, such that the force on bond $i$ due to bond $j$ is as follows

$$F_{ij}^{\mathrm{SRP}} = C(1 - r/r_c)\hat{r}_{ij} \qquad r < r_c$$

where $r$ and $\hat{r}_{ij}$ are the distance and unit vector between the two bonds. Note that *btype* can be specified as an asterisk "*", which case the interaction is applied to all bond types. The *mid* option computes $r$ and $\hat{r}_{ij}$ from the midpoint distance between bonds. The *min* option computes $r$ and $\hat{r}_{ij}$ from the minimum distance between bonds. The force acting on a bond is mapped onto the two bond atoms according to the lever rule,

$$F_{i1}^{\mathrm{SRP}} = F_{ij}^{\mathrm{SRP}}(L)$$
$$F_{i2}^{\mathrm{SRP}} = F_{ij}^{\mathrm{SRP}}(1 - L)$$

where $L$ is the normalized distance from the atom to the point of closest approach of bond $i$ and $j$. The *mid* option takes $L$ as 0.5 for each interaction as described in *(Sirk)*.

The following coefficients must be defined via the *pair_coeff* command as in the examples above, or in the data file or restart file read by the *read_data* or *read_restart* commands:

- $C$ (force units)
- $r_c$ (distance units)

The last coefficient is optional. If not specified, the global cutoff is used.

**Note**

Pair style srp considers each bond of type *btype* to be a fictitious "particle" of type *bptype*, where *bptype* is either the largest atom type in the system, or the type set by the *bptype* flag. Any actual existing particles with this atom type will be deleted at the beginning of a run. This means you must specify the number of types in your system accordingly; usually to be one larger than what would normally be the case, e.g. via the *create_box* or by changing the header in your *data file*. The fictitious "bond particles" are inserted at the beginning of the run, and serve as placeholders that define the position of the bonds. This allows neighbor lists to be constructed and pairwise interactions to be computed in almost the same way as is done for actual particles. Because bonds interact only with other bonds, *pair_style hybrid* should be used to turn off interactions between atom type *bptype* and all other types of atoms. An error will be flagged if *pair_style hybrid* is not used.

**Note**

If using type labels, the type labels must be defined before calling the *pair_coeff* command.

The optional *exclude* keyword determines if forces are computed between first neighbor (directly connected) bonds. For a setting of *no*, first neighbor forces are computed; for *yes* they are not computed. A setting of *no* cannot be used with the *min* option for distance calculation because the minimum distance between directly connected bonds is zero.

Pair style *srp* turns off normalization of thermodynamic properties by particle number, as if the command *thermo_modify norm no* had been issued.

The pairwise energy associated with style *srp* is shifted to be zero at the cutoff distance $r_c$.

---

Added in version 3Aug2022.

Pair style *srp/react* interfaces the pair style *srp* with the bond breaking and formation mechanisms provided by fix *bond/break* and fix *bond/create*, respectively. When using this pair style, whenever a bond breaking (or formation) reaction occurs, the corresponding fictitious particle is deleted (or inserted) during the same simulation time step as the reaction. This is useful in the simulation of reactive systems involving large polymeric molecules *(Palkar)* where the segmental repulsive potential is necessary to minimize topological violations, and also needs to be turned on and off according to the progress of the reaction.

---

### 4.292.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support mixing.

This pair style does not support the *pair_modify* shift option for the energy of the pair interaction. Note that as discussed above, the energy term is already shifted to be 0.0 at the cutoff distance $r_c$.

The *pair_modify* table option is not relevant for this pair style.

This pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

This pair style writes global and per-atom information to *binary restart files*. Pair srp should be used with *pair_style hybrid*, thus the pair_coeff commands need to be specified in the input script when reading a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.292.5 Restrictions

This pair style is part of the MISC package. It is only enabled if LAMMPS was built with that package. See the Making LAMMPS section for more info.

This pair style must be used with *pair_style hybrid*.

This pair style requires the *newton* command to be *on* for non-bonded interactions.

This pair style is not compatible with *rigid body integrators*

### 4.292.6 Related commands

*pair_style hybrid*, *pair_coeff*, *pair dpd*

### 4.292.7 Default

The default keyword value is exclude = yes.

---

**(Sirk)** Sirk TW, Sliozberg YR, Brennan JK, Lisal M, Andzelm JW, J Chem Phys, 136 (13) 134903, 2012.

**(Palkar)** Palkar V, Kuksenok O, J. Phys. Chem. B, 126 (1), 336-346, 2022

## 4.293 pair_style surf/granular command

### 4.293.1 Syntax

```
pair_style surf/granular cutoff
```

- cutoff = global cutoff (optional). See discussion below.

### 4.293.2 Examples

```
pair_style surf/granular
pair_coeff * * hooke 1000.0 50.0 tangential linear_nohistory 1.0 0.4 damping mass_
↪velocity

pair_style surf/granular
pair_coeff * * hooke 1000.0 50.0 tangential linear_history 500.0 1.0 0.4 damping mass_
↪velocity
```

### 4.293.3 Description

Added in version 4Jul2026.

The *surf/granular* pair style is designed for interactions between granular surfaces made out of lines/triangles and finite-sized atoms and must be used in conjunction with *fix surface/local*. See the *Howto granular surfaces* doc page for more information on granular surfaces.

The equation for the force between a triangle/line and a particle touching it is the same as the corresponding equation on the *pair_style granular* doc page in the limit of one of the two particles going to infinite radius and mass (flat surface). Specifically, $\delta$ = overlap of particle with triangle/line, $m_eff$ = mass of particle, and the effective radius of contact *R_{eff}= R_i R_j/R_i + R_j* is set to the radius of the particle. See the *Howto granular surfaces* doc page for information on how overlaps and normal vectors are calculated based on the geometry of the surface and when friction is transferred between lines/triangles.

All model choices and parameters are entered in the *pair_coeff* command. Coefficient values are not global, but can be set to different values for different combinations of particle types, as determined by the *pair_coeff* command. If the contact model choice is the same for two particle types, the mixing for the cross-coefficients can be carried out automatically. For additional flexibility, coefficients as well as model forms can vary between particle types. Available pair coefficients and contact models are identical to those in the *granular* pair style. The only exception is that forces cannot extend beyond contact as in the JKR contact model.

---

LAMMPS automatically sets pairwise cutoff values for *pair_style granular* based on the atom radii. In the vast majority of situations, this is adequate. However, a cutoff value can optionally be appended to the *pair_style granular* command to specify a global cutoff (i.e. a cutoff for all atom types). Additionally, the optional *cutoff* keyword can be passed to the *pair_coeff* command, followed by a cutoff value. This will set a pairwise cutoff for the atom types in the *pair_coeff* command. These options may be useful in some rare cases where the automatic cutoff determination is not sufficient, e.g. if particle diameters are being modified via the *fix adapt* command. In that case, the global cutoff specified as part of the *pair_style granular* command is applied to all atom types, unless it is overridden for a given atom type combination by the *cutoff* value specified in the *pair coeff* command. If *cutoff* is only specified in the *pair coeff* command and no global cutoff is appended to the *pair_style granular* command, then LAMMPS will use that cutoff for the specified atom type combination, and automatically set pairwise cutoffs for the remaining atom types.

---

### 4.293.4 Mixing, shift, table, tail correction, restart, rRESPA info

The *pair_modify* mix, shift, table, and tail options are not relevant for granular pair styles.

Mixing of coefficients is carried out using geometric averaging for most quantities. See the doc page for *pair granular* discussion of exceptions.

These pair styles write their information to *binary restart files*, so a pair_style command does not need to be specified in an input script that reads a restart file.

These pair styles can only be used via the *pair* keyword of the *run_style respa* command. They do not support the *inner*, *middle*, *outer* keywords.

The single() function of these pair styles returns 0.0 for the energy of a pairwise interaction, since energy is not conserved in these dissipative potentials. It also returns only the normal component of the pairwise interaction force. However, the single() function also calculates extra pairwise quantities. See the doc page for *pair granular* for additional information on what these quantities are and how to access them.

---

### 4.293.5 Restrictions

This pair style is part of the GRANSURF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style must be in used in conjunction with *fix surface/local*.

This pair style requires the data structures in *atom_style line* in 2d and *atom_style line* in 3d. Additionally, this pair style requires that atoms store per-particle radius, torque, and angular velocity (omega). These quantities are defined by the line/tri atom styles, however, the radii are not written to data files so it is recommended to hybridize these atom styles with atom style sphere using atom style hybrid.

This pair style requires you to use the *comm_modify vel yes* command so that velocities are stored by ghost atoms.

This pair style will not restart exactly when using the *read_restart* command, though it should provide statistically similar results. This is because the forces it computes depend on atom velocities and the atom velocities have been propagated half a timestep between the force computation and when the restart is written, due to using Velocity Verlet time integration. See the *read_restart* command for more details.

Accumulated values for individual contacts are saved to restart files but are not saved to data files. Therefore, forces may differ significantly when a system is reloaded using the *read_data* command.

### 4.293.6 Related commands

- *fix surface/local*
- *fix pair/granular*

### 4.293.7 Default

For the *pair_coeff* settings: *damping viscoelastic*, *rolling none*, *twisting none*.

## 4.294 pair_style sw command

Accelerator Variants: *sw/gpu*, *sw/intel*, *sw/kk*, *sw/omp*

## 4.295 pair_style sw/mod command

Accelerator Variants: *sw/mod/omp*

### 4.295.1 Syntax

```
pair_style style keyword values
```

- style = *sw* or *sw/mod*
- keyword = *maxdelcs* or *threebody*

```
maxdelcs value = delta1 delta2 (optional, sw/mod only)
  delta1 = The minimum thershold for the variation of cosine of three-body angle
  delta2 = The maximum threshold for the variation of cosine of three-body angle
threebody value = on or off (optional, sw only)
  on (default) = Compute both the three-body and two-body terms of the potential
  off = Compute only the two-body term of the potential
```

### 4.295.2 Examples

```
pair_style sw
pair_coeff * * si.sw Si
pair_coeff * * GaN.sw Ga N Ga

pair_style sw/mod maxdelcs 0.25 0.35
pair_coeff * * tmd.sw.mod Mo S S

pair_style hybrid sw threebody on sw threebody off
pair_coeff * * sw 1 mW_xL.sw mW NULL
pair_coeff 1 2 sw 2 mW_xL.sw mW xL
pair_coeff 2 2 sw 2 mW_xL.sw mW xL
```

### 4.295.3 Description

The *sw* style computes a 3-body *Stillinger-Weber* potential for the energy E of a system of atoms as

$$E = \sum_i \sum_{j>i} \phi_2(r_{ij}) + \sum_i \sum_{j \neq i} \sum_{k>j} \phi_3(r_{ij}, r_{ik}, \theta_{ijk})$$

$$\phi_2(r_{ij}) = A_{ij}\varepsilon_{ij} \left[ B_{ij}(\frac{\sigma_{ij}}{r_{ij}})^{p_{ij}} - (\frac{\sigma_{ij}}{r_{ij}})^{q_{ij}} \right] \exp\left( \frac{\sigma_{ij}}{r_{ij} - a_{ij}\sigma_{ij}} \right)$$

$$\phi_3(r_{ij}, r_{ik}, \theta_{ijk}) = \lambda_{ijk}\varepsilon_{ijk} \left[\cos\theta_{ijk} - \cos\theta_{0ijk}\right]^2 \exp\left( \frac{\gamma_{ij}\sigma_{ij}}{r_{ij} - a_{ij}\sigma_{ij}} \right) \exp\left( \frac{\gamma_{ik}\sigma_{ik}}{r_{ik} - a_{ik}\sigma_{ik}} \right)$$

where $\phi_2$ is a two-body term and $\phi_3$ is a three-body term. The summations in the formula are over all neighbors J and K of atom I within a cutoff distance $a`\sigma$.

Added in version 14Dec2021.

The *sw/mod* style is designed for simulations of materials when distinguishing three-body angles are necessary, such as borophene and transition metal dichalcogenides, which cannot be described by the original code for the Stillinger-Weber potential. For instance, there are several types of angles around each Mo atom in *MoS_2*, and some unnecessary angle types should be excluded in the three-body interaction. Such exclusion may be realized by selecting proper angle types directly. The exclusion of unnecessary angles is achieved here by the cut-off function (*f_C(delta)*), which induces only minimum modifications for LAMMPS.

Validation, benchmark tests, and applications of the *sw/mod* style can be found in *(Jiang2)* and *(Jiang3)*.

The *sw/mod* style computes the energy E of a system of atoms, whose potential function is mostly the same as the Stillinger-Weber potential. The only modification is in the three-body term, where the value of $\delta = \cos\theta_{ijk} - \cos\theta_{0ijk}$ used in the original energy and force expression is scaled by a switching factor $f_C(\delta)$:

$$f_C(\delta) = \begin{cases} 1 & : \quad |\delta| < \delta_1 \\ \frac{1}{2} + \frac{1}{2}\cos\left(\pi \frac{|\delta| - \delta_1}{\delta_2 - \delta_1}\right) & : \quad \delta_1 < |\delta| < \delta_2 \\ 0 & : \quad |\delta| > \delta_2 \end{cases}$$

This cut-off function decreases smoothly from 1 to 0 over the range $[\delta_1, \delta_2]$. This smoothly turns off the energy and force contributions for $|\delta| > \delta_2$. It is suggested that $\delta 1$ and $\delta_2$ to be the value around $0.5\,|\cos\theta_1 - \cos\theta_2|$, with $\theta_1$ and $\theta_2$ as the different types of angles around an atom. For borophene and transition metal dichalcogenides, $\delta_1 = 0.25$ and $\delta_2 = 0.35$. This value enables the cut-off function to exclude unnecessary angles in the three-body SW terms.

**Note**

The cut-off function is just to be used as a technique to exclude some unnecessary angles, and it has no physical meaning. It should be noted that the force and potential are inconsistent with each other in the decaying range of the cut-off function, as the angle dependence for the cut-off function is not implemented in the force (first derivation of potential). However, the angle variation is much smaller than the given threshold value for actual simulations, so the inconsistency between potential and force can be neglected in actual simulations.

Added in version 3Aug2022.

The *threebody* keyword is optional and determines whether or not the three-body term of the potential is calculated. The default value is "on" and it is only available for the plain *sw* pair style variants, but not available for the *sw/mod* and *sw/angle/table* pair style variants. To turn off the threebody contributions all $\lambda_{ijk}$ parameters from the potential file are forcibly set to 0. In addition the pair style implementation may employ code optimizations for the *threebody off* setting that can result in significant speedups versus the default. These code optimizations are currently only available for the MANYBODY and OPENMP packages.

Only a single pair_coeff command is used with the *sw* and *sw/mod* styles which specifies a Stillinger-Weber potential file with parameters for all needed elements, except for when the *threebody off* setting is used (see note below). These are

mapped to LAMMPS atom types by specifying N additional arguments after the filename in the pair_coeff command, where N is the number of LAMMPS atom types:

- filename
- N element names = mapping of SW elements to atom types

See the *pair_coeff* page for alternate ways to specify the path for the potential file.

As an example, imagine a file SiC.sw has Stillinger-Weber values for Si and C. If your LAMMPS simulation has 4 atoms types and you want the first 3 to be Si, and the fourth to be C, you would use the following pair_coeff command:

```
pair_style sw
pair_coeff * * SiC.sw Si Si Si C
```

The first 2 arguments must be * * so as to span all LAMMPS atom types. The first three Si arguments map LAMMPS atom types 1, 2, and 3 to the Si element in the SW file. The final C argument maps LAMMPS atom type 4 to the C element in the SW file. If an argument value is specified as NULL, the mapping is not performed. This can be used when an *sw* potential is used as part of the *hybrid* pair style. The NULL values are placeholders for atom types that will be used with other potentials.

**Note**

When the *threebody off* keyword is used, multiple pair_coeff commands may be used to specific the pairs of atoms which don't require three-body term. In these cases, the first 2 arguments are not required to be * *, the potential parameter file is only read by the first *pair_coeff command* and the element to atom type mappings must be consistent across all *pair_coeff* statements. If not LAMMPS will abort with an error.

Stillinger-Weber files in the *potentials* directory of the LAMMPS distribution have a ".sw" suffix. Lines that are not blank or comments (starting with #) define parameters for a triplet of elements. The parameters in a single entry correspond to the two-body and three-body coefficients in the formula above:

- element 1 (the center atom in a 3-body interaction)
- element 2
- element 3
- $\varepsilon$ (energy units)
- $\sigma$ (distance units)
- a
- $\lambda$
- $\gamma$
- $\cos \theta_0$
- A
- B
- p
- q
- tol

The A, B, p, and q parameters are used only for two-body interactions. The $\lambda$ and $\cos \theta_0$ parameters are used only for three-body interactions. The $\varepsilon$, $\sigma$ and $a$ parameters are used for both two-body and three-body interactions. $\gamma$ is used only in the three-body interactions, but is defined for pairs of atoms. The non-annotated parameters are unitless.

LAMMPS introduces an additional performance-optimization parameter tol that is used for both two-body and three-body interactions. In the Stillinger-Weber potential, the interaction energies become negligibly small at atomic separations substantially less than the theoretical cutoff distances. LAMMPS therefore defines a virtual cutoff distance based on a user defined tolerance tol. The use of the virtual cutoff distance in constructing atom neighbor lists can significantly reduce the neighbor list sizes and therefore the computational cost. LAMMPS provides a *tol* value for each of the three-body entries so that they can be separately controlled. If tol = 0.0, then the standard Stillinger-Weber cutoff is used.

The Stillinger-Weber potential file must contain entries for all the elements listed in the pair_coeff command. It can also contain entries for additional elements not being used in a particular simulation; LAMMPS ignores those entries.

For a single-element simulation, only a single entry is required (e.g. SiSiSi). For a two-element simulation, the file must contain 8 entries (for SiSiSi, SiSiC, SiCSi, SiCC, CSiSi, CSiC, CCSi, CCC), that specify SW parameters for all permutations of the two elements interacting in three-body configurations. Thus for 3 elements, 27 entries would be required, etc.

As annotated above, the first element in the entry is the center atom in a three-body interaction. Thus an entry for SiCC means a Si atom with 2 C atoms as neighbors. The parameter values used for the two-body interaction come from the entry where the second and third elements are the same. Thus the two-body parameters for Si interacting with C, comes from the SiCC entry. The three-body parameters can in principle be specific to the three elements of the configuration. In the literature, however, the three-body parameters are usually defined by simple formulas involving two sets of pairwise parameters, corresponding to the ij and ik pairs, where i is the center atom. The user must ensure that the correct combining rule is used to calculate the values of the three-body parameters for alloys. Note also that the function $\phi_3$ contains two exponential screening factors with parameter values from the ij pair and ik pairs. So $\phi_3$ for a C atom bonded to a Si atom and a second C atom will depend on the three-body parameters for the CSiC entry, and also on the two-body parameters for the CCC and CSiSi entries. Since the order of the two neighbors is arbitrary, the three-body parameters for entries CSiC and CCSi should be the same. Similarly, the two-body parameters for entries SiCC and CSiSi should also be the same. The parameters used only for two-body interactions (A, B, p, and q) in entries whose second and third element are different (e.g. SiCSi) are not used for anything and can be set to 0.0 if desired. This is also true for the parameters in $\phi_3$ that are taken from the ij and ik pairs ($\sigma$, $a$, $\gamma$)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

**Note**

When using the INTEL package with this style, there is an additional 5 to 10 percent performance improvement when the Stillinger-Weber parameters p and q are set to 4 and 0 respectively. These parameters are common for modeling silicon and water.

---

### 4.295.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, where types I and J correspond to two different element types, mixing is performed by LAMMPS as described above from values in the potential file.

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

The single() function of the *sw* pair style is only enabled and supported for the case of the *threebody off* setting.

---

### 4.295.5 Restrictions

This pair style is part of the MANYBODY package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires the *newton* setting to be "on" for pair interactions.

The Stillinger-Weber potential files provided with LAMMPS (see the potentials directory) are parameterized for metal *units*. You can use the sw or sw/mod pair styles with any LAMMPS units, but you would need to create your own SW potential file with coefficients listed in the appropriate units if your simulation does not use "metal" units. If the potential file contains a 'UNITS:' metadata tag in the first line of the potential file, then LAMMPS can convert it transparently between "metal" and "real" units.

### 4.295.6 Related commands

*pair_coeff*

### 4.295.7 Default

The default value for the *threebody* setting of the "sw" pair style is "on", the default values for the "*maxdelcs* setting of the *sw/mod* pair style are *delta1* = 0.25 and *delta2* = 0.35`.

---

**(Stillinger)** Stillinger and Weber, Phys Rev B, 31, 5262 (1985).

**(Jiang2)** J.-W. Jiang, Nanotechnology 26, 315706 (2015).

**(Jiang3)** J.-W. Jiang, Acta Mech. Solida. Sin 32, 17 (2019).

## 4.296 pair_style sw/angle/table command

### 4.296.1 Syntax

```
pair_style style
```

- style = *sw/angle/table*

### 4.296.2 Examples

```
pair_style sw/angle/table
pair_coeff * * spce.sw type
```

Used in example input script:

```
examples/PACKAGES/manybody_table/in.spce_sw
```

### 4.296.3 Description

Added in version 2Jun2022.

The *sw/angle/table* style is a modification of the original *pair_style sw*. It has been developed for coarse-grained simulations (of water) (*Scherer1*), but can be employed for all kinds of systems. It computes a modified 3-body *Stillinger-Weber* potential for the energy E of a system of atoms as

$$E = \sum_i \sum_{j>i} \phi_2(r_{ij}) + \sum_i \sum_{j \neq i} \sum_{k>j} \phi_3(r_{ij}, r_{ik}, \theta_{ijk})$$

$$\phi_2(r_{ij}) = A_{ij}\varepsilon_{ij} \left[ B_{ij}(\frac{\sigma_{ij}}{r_{ij}})^{p_{ij}} - (\frac{\sigma_{ij}}{r_{ij}})^{q_{ij}} \right] \exp\left(\frac{\sigma_{ij}}{r_{ij} - a_{ij}\sigma_{ij}}\right)$$

$$\phi_3(r_{ij}, r_{ik}, \theta_{ijk}) = f^{3b}\left(\theta_{ijk}\right) \exp\left(\frac{\gamma_{ij}\sigma_{ij}}{r_{ij} - a_{ij}\sigma_{ij}}\right) \exp\left(\frac{\gamma_{ik}\sigma_{ik}}{r_{ik} - a_{ik}\sigma_{ik}}\right)$$

where $\phi_2$ is a two-body term and $\phi_3$ is a three-body term. The summations in the formula are over all neighbors J and K of atom I within a cutoff distance $a\sigma$. In contrast to the original *sw* style, *sw/angle/table* allows for a flexible three-body term $f^{3b}\left(\theta_{ijk}\right)$ which is read in as a tabulated interaction. It can be parameterized with the csg_fmatch app of VOTCA as available at: https://gitlab.mpcdf.mpg.de/votca/votca.

Only a single pair_coeff command is used with the *sw/angle/table* style which specifies a modified Stillinger-Weber potential file with parameters for all needed elements. These are mapped to LAMMPS atom types by specifying N_el additional arguments after the ".sw" filename in the pair_coeff command, where N_el is the number of LAMMPS atom types:

- ".sw" filename
- N_el element names = mapping of SW elements to atom types

See the *pair_coeff* page for alternate ways to specify the path for the potential file.

As an example, imagine a file SiC.sw has Stillinger-Weber values for Si and C. If your LAMMPS simulation has 4 atoms types and you want the first 3 to be Si, and the fourth to be C, you would use the following pair_coeff command:

```
pair_coeff * * SiC.sw Si Si Si C
```

The first 2 arguments must be * * so as to span all LAMMPS atom types. The first three Si arguments map LAMMPS atom types 1,2,3 to the Si element in the SW file. The final C argument maps LAMMPS atom type 4 to the C element in the SW file. If a mapping value is specified as NULL, the mapping is not performed. This can be used when a *sw/angle/table* potential is used as part of the *hybrid* pair style. The NULL values are placeholders for atom types that will be used with other potentials.

The (modified) Stillinger-Weber files have a ".sw" suffix. Lines that are not blank or comments (starting with #) define parameters for a triplet of elements. The parameters in a single entry correspond to the two-body and three-body coefficients in the formula above. Here, also the suffix ".sw" is used though the original Stillinger-Weber file format is supplemented with four additional lines per parameter block to specify the tabulated three-body interaction. A single entry then contains:

- element 1 (the center atom in a 3-body interaction)

- element 2
- element 3
- $\varepsilon$ (energy units)
- $\sigma$ (distance units)
- a
- $\lambda$
- $\gamma$
- $\cos\theta_0$
- A
- B
- p
- q
- tol
- filename
- keyword
- style
- N

The A, B, p, and q parameters are used only for two-body interactions. The $\lambda$ and $\cos\theta_0$ parameters, only used for three-body interactions in the original Stillinger-Weber style, are read in but ignored in this modified pair style. The $\varepsilon$ parameter is only used for two-body interactions in this modified pair style and not for the three-body terms. The $\sigma$ and $a$ parameters are used for both two-body and three-body interactions. $\gamma$ is used only in the three-body interactions, but is defined for pairs of atoms. The non-annotated parameters are unitless.

LAMMPS introduces an additional performance-optimization parameter tol that is used for both two-body and three-body interactions. In the Stillinger-Weber potential, the interaction energies become negligibly small at atomic separations substantially less than the theoretical cutoff distances. LAMMPS therefore defines a virtual cutoff distance based on a user defined tolerance tol. The use of the virtual cutoff distance in constructing atom neighbor lists can significantly reduce the neighbor list sizes and therefore the computational cost. LAMMPS provides a *tol* value for each of the three-body entries so that they can be separately controlled. If tol = 0.0, then the standard Stillinger-Weber cutoff is used.

The additional parameters *filename*, *keyword*, *style*, and *N* refer to the tabulated angular potential $f^{3b}\left(\theta_{ijk}\right)$. The tabulated angular potential has to be of the format as used in the *angle_style table* command:

An interpolation tables of length *N* is created. The interpolation is done in one of 2 *styles*: *linear* or *spline*. For the *linear* style, the angle is used to find 2 surrounding table values from which an energy or its derivative is computed by linear interpolation. For the *spline* style, a cubic spline coefficients are computed and stored at each of the *N* values in the table. The angle is used to find the appropriate set of coefficients which are used to evaluate a cubic polynomial which computes the energy or derivative.

The *filename* specifies the file containing the tabulated energy and derivative values of $f^{3b}\left(\theta_{ijk}\right)$. The *keyword* then specifies a section of the file. The format of this file is as follows (without the parenthesized comments):

```
# Angle potential for harmonic (one or more comment or blank lines)

HAM                          (keyword is the first text on line)
N 181 FP 0 0 EQ 90.0         (N, FP, EQ parameters)
```

(continues on next page)

(continued from previous page)

```
                                (blank line)
1 0.0 200.5 2.5                 (index, angle, energy, derivative)
2 1.0 198.0 2.5
...
181 180.0 0.0 0.0
```

A section begins with a non-blank line whose first character is not a "#"; blank lines or lines starting with "#" can be used as comments between sections. The first line begins with a keyword which identifies the section. The next line lists (in any order) one or more parameters for the table. Each parameter is a keyword followed by one or more numeric values.

The parameter "N" is required and its value is the number of table entries that follow. Note that this may be different than the $N$ specified in the Stillinger-Weber potential file. Let Nsw = $N$ in the ".sw" file, and Nfile = "N" in the tabulated angular file. What LAMMPS does is a preliminary interpolation by creating splines using the Nfile tabulated values as nodal points. It uses these to interpolate as needed to generate energy and derivative values at Ntable different points. The resulting tables of length Nsw are then used as described above, when computing energy and force for individual angles and their atoms. This means that if you want the interpolation tables of length Nsw to match exactly what is in the tabulated file (with effectively no preliminary interpolation), you should set Nsw = Nfile.

The "FP" parameter is optional. If used, it is followed by two values fplo and fphi, which are the second derivatives at the innermost and outermost angle settings. These values are needed by the spline construction routines. If not specified by the "FP" parameter, they are estimated (less accurately) by the first two and last two derivative values in the table.

The "EQ" parameter is also optional. If used, it is followed by a the equilibrium angle value, which is used, for example, by the *fix shake* command. If not used, the equilibrium angle is set to 180.0.

Following a blank line, the next N lines of the angular table file list the tabulated values. On each line, the first value is the index from 1 to N, the second value is the angle value (in degrees), the third value is the energy (in energy units), and the fourth is -dE/d(theta) (also in energy units). The third term is the energy of the 3-atom configuration for the specified angle. The last term is the derivative of the energy with respect to the angle (in degrees, not radians). Thus the units of the last term are still energy, not force. The angle values must increase from one line to the next. The angle values must also begin with 0.0 and end with 180.0, i.e. span the full range of possible angles.

Note that one angular potential file can contain many sections, each with a tabulated potential. LAMMPS reads the file section by section until it finds one that matches the specified *keyword* of appropriate section of the ".sw" file.

The Stillinger-Weber potential file must contain entries for all the elements listed in the pair_coeff command. It can also contain entries for additional elements not being used in a particular simulation; LAMMPS ignores those entries.

For a single-element simulation, only a single entry is required (e.g. SiSiSi). For a two-element simulation, the file must contain 8 entries (for SiSiSi, SiSiC, SiCSi, SiCC, CSiSi, CSiC, CCSi, CCC), that specify SW parameters for all permutations of the two elements interacting in three-body configurations. Thus for 3 elements, 27 entries would be required, etc.

As annotated above, the first element in the entry is the center atom in a three-body interaction. Thus an entry for SiCC means a Si atom with 2 C atoms as neighbors. The parameter values used for the two-body interaction come from the entry where the second and third elements are the same. Thus the two-body parameters for Si interacting with C, comes from the SiCC entry. The three-body angular potential $f^{3b}\left(\theta_{ijk}\right)$ can in principle be specific to the three elements of the configuration. However, the user must ensure that it makes physically sense. Note also that the function $\phi_3$ contains two exponential screening factors with parameter values from the ij pair and ik pairs. So $\phi_3$ for a C atom bonded to a Si atom and a second C atom will depend on the three-body parameters for the CSiC entry, and also on the two-body parameters for the CCC and CSiSi entries. Since the order of the two neighbors is arbitrary, the three-body parameters and the tabulated angular potential for entries CSiC and CCSi should be the same. Similarly, the two-body parameters for entries SiCC and CSiSi should also be the same. The parameters used only for two-body interactions (A, B, p, and q) in entries whose second and third element are different (e.g. SiCSi) are not used for anything and can be set to 0.0 if desired. This is also true for the parameters in $\phi_3$ that are taken from the ij and ik pairs ($\sigma$, $a$, $\gamma$)

Additional input files and reference data can be found at: https://gitlab.mpcdf.mpg.de/votca/votca/-/tree/master/csg-tutorials/spce/3body_sw

---

### 4.296.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, where types I and J correspond to two different element types, mixing is performed by LAMMPS as described above from values in the potential file, but not for the tabulated angular potential file.

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.296.5 Restrictions

This pair style is part of the MANYBODY package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires the *newton* setting to be "on" for pair interactions.

### 4.296.6 Related commands

*pair_coeff* , *pair_style sw*, *pair_style threebody/table*

---

**(Stillinger)** Stillinger and Weber, Phys Rev B, 31, 5262 (1985).

**(Scherer1)** C. Scherer and D. Andrienko, Phys. Chem. Chem. Phys. 20, 22387-22394 (2018).

## 4.297 pair_style table command

Accelerator Variants: *table/gpu*, *table/kk*, *table/omp*

### 4.297.1 Syntax

```
pair_style table style N keyword ...
```

- style = *lookup* or *linear* or *spline* or *bitmap* = method of interpolation
- N = use N values in *lookup*, *linear*, *spline* tables
- N = use 2^N values in *bitmap* tables
- zero or more keywords may be appended
- keyword = *ewald* or *pppm* or *msm* or *dispersion* or *tip4p*

### 4.297.2 Examples

```
pair_style table linear 1000
pair_style table linear 1000 pppm
pair_style table bitmap 12
pair_coeff * 3 morse.table ENTRY1
pair_coeff * 3 morse.table ENTRY1 7.0
```

### 4.297.3 Description

Style *table* creates interpolation tables from potential energy and force values listed in a file(s) as a function of distance. When performing dynamics or minimization, the interpolation tables are used to evaluate energy and forces for pairwise interactions between particles, similar to how analytic formulas are used for other pair styles.

The interpolation tables are created as a pre-computation by fitting cubic splines to the file values and interpolating energy and force values at each of *N* distances. During a simulation, the tables are used to interpolate energy and force values as needed for each pair of particles separated by a distance *R*. The interpolation is done in one of 4 styles: *lookup*, *linear*, *spline*, or *bitmap*.

For the *lookup* style, the distance *R* is used to find the nearest table entry, which is the energy or force.

For the *linear* style, the distance *R* is used to find the 2 surrounding table values from which an energy or force is computed by linear interpolation.

For the *spline* style, cubic spline coefficients are computed and stored for each of the *N* values in the table, one set of splines for energy, another for force. Note that these splines are different than the ones used to pre-compute the *N* values. Those splines were fit to the *Nfile* values in the tabulated file, where often *Nfile* < *N*. The distance *R* is used to find the appropriate set of spline coefficients which are used to evaluate a cubic polynomial which computes the energy or force.

For the *bitmap* style, the specified *N* is used to create interpolation tables that are 2^N in length. The distance *R* is used to index into the table via a fast bit-mapping technique due to *(Wolff)*, and a linear interpolation is performed between adjacent table values.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above.

- filename
- keyword
- cutoff (distance units)

The filename specifies a file containing tabulated energy and force values. The keyword specifies a section of the file. The cutoff is an optional coefficient. If not specified, the outer cutoff in the table itself (see below) will be used to build an interpolation table that extend to the largest tabulated distance. If specified, only file values up to the cutoff are used to create the interpolation table. The format of this file is described below.

If your tabulated potential(s) are designed to be used as the short-range part of one of the long-range solvers specified by the *kspace_style* command, then you must use one or more of the optional keywords listed above for the pair_style command. These are *ewald* or *pppm* or *msm* or *dispersion* or *tip4p*. This is so LAMMPS can ensure the short-range potential and long-range solver are compatible with each other, as it does for other short-range pair styles, such as *pair_style lj/cut/coul/long*. Note that it is up to you to ensure the tabulated values for each pair of atom types has the correct functional form to be compatible with the matching long-range solver.

---

Here are some guidelines for using the pair_style table command to best effect:

- Vary the number of table points; you may need to use more than you think to get good resolution.
- Always use the *pair_write* command to produce a plot of what the final interpolated potential looks like. This can show up interpolation "features" you may not like.

- Start with the linear style; it's the style least likely to have problems.
- Use *N* in the pair_style command equal to the "N" in the tabulation file, and use the "RSQ" or "BITMAP" parameter, so additional interpolation is not needed. See discussion below.
- Make sure that your tabulated forces and tabulated energies are consistent (dE/dr = -F) over the entire range of r values. LAMMPS will warn if this is not the case.
- Use as large an inner cutoff as possible. This avoids fitting splines to very steep parts of the potential.

---

Suitable tables in the correct format for use with these pair styles can be created by LAMMPS itself using the *pair_write* command. In combination with the pair styles *python*, *lepton, or lepton/coul* this can be a powerful mechanism to implement and test tables for use with LAMMPS. Another option to generate tables is the Python code in the `tools/tabulate` folder of the LAMMPS source code distribution.

The format of a tabulated file has an (optional) header followed by a series of one or more sections, defined as follows (without the parenthesized comments). The header must start with a # character and the DATE: and UNITS: tags will be parsed and used:

```
# DATE: 2020-06-10  UNITS: real  CONTRIBUTOR: ... (header line)
# Morse potential for Fe   (one or more comment or blank lines)

MORSE_FE                   (keyword is first text on line)
N 500 R 1.0 10.0           (N, R, RSQ, BITMAP, FPRIME parameters)
                           (blank)
1 1.0 25.5 102.34          (index, r, energy, force)
2 1.02 23.4 98.5
...
500 10.0 0.001 0.003
```

A section begins with a non-blank line whose first character is not a "#"; blank lines or lines starting with "#" can be used as comments between sections. The first line begins with a keyword which identifies the section. The line can contain additional text, but the initial text must match the argument specified in the pair_coeff command. The next line lists (in any order) one or more parameters for the table. Each parameter is a keyword followed by one or more numeric values.

The parameter "N" is required and its value is the number of table entries that follow. Note that this may be different than the *N* specified in the *pair_style table* command. Let Ntable = *N* in the pair_style command, and Nfile = "N" in the tabulated file. What LAMMPS does is a preliminary interpolation by creating splines using the Nfile tabulated values as nodal points. It uses these to interpolate energy and force values at Ntable different points. The resulting tables of length Ntable are then used as described above, when computing energy and force for individual pair distances. This means that if you want the interpolation tables of length Ntable to match exactly what is in the tabulated file (with effectively no preliminary interpolation), you should set Ntable = Nfile, and use the "RSQ" or "BITMAP" parameter. This is because the internal table abscissa is always RSQ (separation distance squared), for efficient lookup.

All other parameters are optional. If "R" or "RSQ" or "BITMAP" does not appear, then the distances in each line of the table are used as-is to perform spline interpolation. In this case, the table values can be spaced in *r* uniformly or however you wish to position table values in regions of large gradients.

If used, the parameters "R" or "RSQ" are followed by 2 values *rlo* and *rhi*. If specified, the distance associated with each energy and force value is computed from these 2 values (at high accuracy), rather than using the (low-accuracy) value listed in each line of the table. The distance values in the table file are ignored in this case. For "R", distances uniformly spaced between *rlo* and *rhi* are computed; for "RSQ", squared distances uniformly spaced between *rlo*rlo* and *rhi*rhi* are computed.

**Note**

If you use "R" or "RSQ", the tabulated distance values in the file are effectively ignored, and replaced by new values as described in the previous paragraph. If the distance value in the table is not very close to the new value (i.e. round-off difference), then you will be assigning energy/force values to a different distance, which is probably not what you want. LAMMPS will warn if this is occurring.

If used, the parameter "BITMAP" is also followed by 2 values *rlo* and *rhi*. These values, along with the "N" value determine the ordering of the N lines that follow and what distance is associated with each. This ordering is complex, so it is not documented here, since this file is typically produced by the *pair_write* command with its *bitmap* option. When the table is in BITMAP format, the "N" parameter in the file must be equal to 2^M where M is the value specified in the pair_style command. Also, a cutoff parameter cannot be used as an optional third argument in the pair_coeff command; the entire table extent as specified in the file must be used.

If used, the parameter "FPRIME" is followed by 2 values *fplo* and *fphi* which are the derivative of the force at the innermost and outermost distances listed in the table. These values are needed by the spline construction routines. If not specified by the "FPRIME" parameter, they are estimated (less accurately) by the first 2 and last 2 force values in the table. This parameter is not used by BITMAP tables.

Following a blank line, the next N lines list the tabulated values. On each line, the first value is the index from 1 to N, the second value is r (in distance units), the third value is the energy (in energy units), and the fourth is the force (in force units). The r values must increase from one line to the next (unless the BITMAP parameter is specified).

Note that one file can contain many sections, each with a tabulated potential. LAMMPS reads the file section by section until it finds one that matches the specified keyword.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.297.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

The *pair_modify* shift, table, and tail options are not relevant for this pair style.

This pair style writes the settings for the "pair_style table" command to *binary restart files*, so a pair_style command does not need to specified in an input script that reads a restart file. However, the coefficient information is not stored in the restart file, since it is tabulated in the potential files. Thus, pair_coeff commands do need to be specified in the restart input script.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.297.5 Restrictions

none

### 4.297.6 Related commands

*pair_coeff*, *pair_write*

### 4.297.7 Default

none

---

**(Wolff)** Wolff and Rudd, Comp Phys Comm, 120, 200-32 (1999).

## 4.298 pair_style table/rx command

Accelerator Variants: *table/rx/kk*

### 4.298.1 Syntax

```
pair_style table style N ...
```

- style = *lookup* or *linear* or *spline* or *bitmap* = method of interpolation
- N = use N values in *lookup*, *linear*, *spline* tables
- weighting = fractional or molecular (optional)

### 4.298.2 Examples

```
pair_style table/rx linear 1000
pair_style table/rx linear 1000 fractional
pair_style table/rx linear 1000 molecular
pair_coeff * * rxn.table ENTRY1 h2o h2o 10.0
pair_coeff * * rxn.table ENTRY1 1fluid 1fluid 10.0
pair_coeff * 3 rxn.table ENTRY1 h2o no2 10.0
```

### 4.298.3 Description

Style *table/rx* is used in reaction DPD simulations,where the coarse-grained (CG) particles are composed of *m* species whose reaction rate kinetics are determined from a set of *n* reaction rate equations through the *fix rx* command. The species of one CG particle can interact with a species in a neighboring CG particle through a site-site interaction potential model. Style *table/rx* creates interpolation tables of length *N* from pair potential and force values listed in a file(s) as a function of distance. The files are read by the *pair_coeff* command.

The interpolation tables are created by fitting cubic splines to the file values and interpolating energy and force values at each of *N* distances. During a simulation, these tables are used to interpolate energy and force values as needed. The interpolation is done in one of 4 styles: *lookup*, *linear*, *spline*, or *bitmap*.

For the *lookup* style, the distance between two atoms is used to find the nearest table entry, which is the energy or force.

For the *linear* style, the pair distance is used to find 2 surrounding table values from which an energy or force is computed by linear interpolation.

For the *spline* style, a cubic spline coefficients are computed and stored at each of the *N* values in the table. The pair distance is used to find the appropriate set of coefficients which are used to evaluate a cubic polynomial which computes the energy or force.

For the *bitmap* style, the N means to create interpolation tables that are 2^N in length. The pair distance is used to index into the table via a fast bit-mapping technique *(Wolff)* and a linear interpolation is performed between adjacent table values.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above.

- filename
- keyword
- species1
- species2
- cutoff (distance units)

The filename specifies a file containing tabulated energy and force values. The keyword specifies a section of the file. The cutoff is an optional coefficient. If not specified, the outer cutoff in the table itself (see below) will be used to build an interpolation table that extend to the largest tabulated distance. If specified, only file values up to the cutoff are used to create the interpolation table. The format of this file is described below.

The species tags define the site-site interaction potential between two species contained within two different particles. The species tags must either correspond to the species defined in the reaction kinetics files specified with the *fix rx* command or they must correspond to the tag "1fluid", signifying interaction with a product species mixture determined through a one-fluid approximation. The interaction potential is weighted by the geometric average of either the mole fraction concentrations or the number of molecules associated with the interacting coarse-grained particles (see the *fractional* or *molecular* weighting pair style options). The coarse-grained potential is stored before and after the reaction kinetics solver is applied, where the difference is defined to be the internal chemical energy (uChem).

---

Here are some guidelines for using the pair_style table/rx command to best effect:

- Vary the number of table points; you may need to use more than you think to get good resolution.
- Always use the *pair_write* command to produce a plot of what the final interpolated potential looks like. This can show up interpolation "features" you may not like.
- Start with the linear style; it's the style least likely to have problems.
- Use *N* in the pair_style command equal to the "N" in the tabulation file, and use the "RSQ" or "BITMAP" parameter, so additional interpolation is not needed. See discussion below.
- Make sure that your tabulated forces and tabulated energies are consistent (dE/dr = -F) along the entire range of r values.
- Use as large an inner cutoff as possible. This avoids fitting splines to very steep parts of the potential.

---

The format of a tabulated file is a series of one or more sections, defined as follows (without the parenthesized comments):

```
# Morse potential for Fe   (one or more comment or blank lines)

MORSE_FE                   (keyword is first text on line)
N 500 R 1.0 10.0           (N, R, RSQ, BITMAP, FPRIME parameters)
                           (blank)
```

(continues on next page)

(continued from previous page)

```
1 1.0 25.5 102.34          (index, r, energy, force)
2 1.02 23.4 98.5
...
500 10.0 0.001 0.003
```

A section begins with a non-blank line whose first character is not a "#"; blank lines or lines starting with "#" can be used as comments between sections. The first line begins with a keyword which identifies the section. The line can contain additional text, but the initial text must match the argument specified in the pair_coeff command. The next line lists (in any order) one or more parameters for the table. Each parameter is a keyword followed by one or more numeric values.

The parameter "N" is required and its value is the number of table entries that follow. Note that this may be different than the *N* specified in the *pair_style table/rx* command. Let Ntable = *N* in the pair_style command, and Nfile = "N" in the tabulated file. What LAMMPS does is a preliminary interpolation by creating splines using the Nfile tabulated values as nodal points. It uses these to interpolate as needed to generate energy and force values at Ntable different points. The resulting tables of length Ntable are then used as described above, when computing energy and force for individual pair distances. This means that if you want the interpolation tables of length Ntable to match exactly what is in the tabulated file (with effectively no preliminary interpolation), you should set Ntable = Nfile, and use the "RSQ" or "BITMAP" parameter. The internal table abscissa is RSQ (separation distance squared).

All other parameters are optional. If "R" or "RSQ" or "BITMAP" does not appear, then the distances in each line of the table are used as-is to perform spline interpolation. In this case, the table values can be spaced in *r* uniformly or however you wish to position table values in regions of large gradients.

If used, the parameters "R" or "RSQ" are followed by 2 values *rlo* and *rhi*. If specified, the distance associated with each energy and force value is computed from these 2 values (at high accuracy), rather than using the (low-accuracy) value listed in each line of the table. The distance values in the table file are ignored in this case. For "R", distances uniformly spaced between *rlo* and *rhi* are computed; for "RSQ", squared distances uniformly spaced between *rlo***rlo* and *rhi***rhi* are computed.

If used, the parameter "BITMAP" is also followed by 2 values *rlo* and *rhi*. These values, along with the "N" value determine the ordering of the N lines that follow and what distance is associated with each. This ordering is complex, so it is not documented here, since this file is typically produced by the *pair_write* command with its *bitmap* option. When the table is in BITMAP format, the "N" parameter in the file must be equal to 2^M where M is the value specified in the pair_style command. Also, a cutoff parameter cannot be used as an optional third argument in the pair_coeff command; the entire table extent as specified in the file must be used.

If used, the parameter "FPRIME" is followed by 2 values *fplo* and *fphi* which are the derivative of the force at the innermost and outermost distances listed in the table. These values are needed by the spline construction routines. If not specified by the "FPRIME" parameter, they are estimated (less accurately) by the first 2 and last 2 force values in the table. This parameter is not used by BITMAP tables.

Following a blank line, the next N lines list the tabulated values. On each line, the first value is the index from 1 to N, the second value is r (in distance units), the third value is the energy (in energy units), and the fourth is the force (in force units). The r values must increase from one line to the next (unless the BITMAP parameter is specified).

Note that one file can contain many sections, each with a tabulated potential. LAMMPS reads the file section by section until it finds one that matches the specified keyword.

---

### 4.298.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

The *pair_modify* shift, table, and tail options are not relevant for this pair style.

This pair style writes the settings for the "pair_style table/rx" command to *binary restart files*, so a pair_style command does not need to specified in an input script that reads a restart file. However, the coefficient information is not stored in the restart file, since it is tabulated in the potential files. Thus, pair_coeff commands do need to be specified in the restart input script.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.298.5 Restrictions

This command is part of the DPD-REACT package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.298.6 Related commands

*pair_coeff*

### 4.298.7 Default

fractional weighting

---

**(Wolff)** Wolff and Rudd, Comp Phys Comm, 120, 200-32 (1999).

## 4.299 pair_style tersoff command

Accelerator Variants: *tersoff/gpu*, *tersoff/intel*, *tersoff/kk*, *tersoff/omp*

## 4.300 pair_style tersoff/table command

Accelerator Variants: *tersoff/table/omp*

### 4.300.1 Syntax

```
pair_style style keywords values
```

- style = *tersoff* or *tersoff/table*
- keyword = *shift*

```
shift value = delta
  delta = negative shift in equilibrium bond length
```

### 4.300.2 Examples

```
pair_style tersoff
pair_coeff * * Si.tersoff Si
pair_coeff * * SiC.tersoff Si C Si

pair_style tersoff/table
pair_coeff * * SiCGe.tersoff Si(D)

pair_style tersoff shift 0.05
pair_coeff * * Si.tersoff Si
```

### 4.300.3 Description

The *tersoff* style computes a 3-body Tersoff potential *(Tersoff_1)* for the energy E of a system of atoms as

$$
\begin{aligned}
E &= \frac{1}{2}\sum_i \sum_{j \neq i} V_{ij} \\
V_{ij} &= f_C(r_{ij}+\delta)\left[f_R(r_{ij}+\delta) + b_{ij} f_A(r_{ij}+\delta)\right] \\
f_C(r) &= \begin{cases} 1 & : \quad r < R-D \\ \frac{1}{2} - \frac{1}{2}\sin\left(\frac{\pi}{2}\frac{r-R}{D}\right) & : \quad R-D < r < R+D \\ 0 & : \quad r > R+D \end{cases} \\
f_R(r) &= A\exp(-\lambda_1 r) \\
f_A(r) &= -B\exp(-\lambda_2 r) \\
b_{ij} &= \left(1+\beta^n {\zeta_{ij}}^n\right)^{-\frac{1}{2n}} \\
\zeta_{ij} &= \sum_{k \neq i,j} f_C(r_{ik}+\delta) g\left[\theta_{ijk}(r_{ij},r_{ik})\right] \exp\left[{\lambda_3}^m (r_{ij}-r_{ik})^m\right] \\
g(\theta) &= \gamma_{ijk}\left(1 + \frac{c^2}{d^2} - \frac{c^2}{\left[d^2 + (\cos\theta - \cos\theta_0)^2\right]}\right)
\end{aligned}
$$

where $f_R$ is a two-body term and $f_A$ includes three-body interactions. The summations in the formula are over all neighbors J and K of atom I within a cutoff distance = R + D. $\delta$ is an optional negative shift of the equilibrium bond length, as described below.

The *tersoff/table* style uses tabulated forms for the two-body, environment and angular functions. Linear interpolation is performed between adjacent table entries. The table length is chosen to be accurate within 10^-6 with respect to the *tersoff* style energy. The *tersoff/table* should give better performance in terms of speed.

Only a single pair_coeff command is used with the *tersoff* style which specifies a Tersoff potential file with parameters for all needed elements. These are mapped to LAMMPS atom types by specifying N additional arguments after the filename in the pair_coeff command, where N is the number of LAMMPS atom types:

- filename
- N element names = mapping of Tersoff elements to atom types

See the *pair_coeff* page for alternate ways to specify the path for the potential file.

As an example, imagine the SiC.tersoff file has Tersoff values for Si and C. If your LAMMPS simulation has 4 atoms types and you want the first 3 to be Si, and the fourth to be C, you would use the following pair_coeff command:

```
pair_coeff * * SiC.tersoff Si Si Si C
```

The first 2 arguments must be * * so as to span all LAMMPS atom types. The first three Si arguments map LAMMPS atom types 1,2,3 to the Si element in the Tersoff file. The final C argument maps LAMMPS atom type 4 to the C

element in the Tersoff file. If a mapping value is specified as NULL, the mapping is not performed. This can be used when a *tersoff* potential is used as part of the *hybrid* pair style. The NULL values are placeholders for atom types that will be used with other potentials.

Tersoff files in the *potentials* directory of the LAMMPS distribution have a ".tersoff" suffix. Lines that are not blank or comments (starting with #) define parameters for a triplet of elements. The parameters in a single entry correspond to coefficients in the formula above:

- element 1 (the center atom in a 3-body interaction)
- element 2 (the atom bonded to the center atom)
- element 3 (the atom influencing the 1-2 bond in a bond-order sense)
- m
- $\gamma$
- $\lambda_3$ (1/distance units)
- c
- d
- $\cos \theta_0$ (can be a value < -1 or > 1)
- n
- $\beta$
- $\lambda_2$ (1/distance units)
- B (energy units)
- R (distance units)
- D (distance units)
- $\lambda_1$ (1/distance units)
- A (energy units)

The n, $\beta$, $\lambda_2$, B, $\lambda_1$, and A parameters are only used for two-body interactions. The m, $\gamma$, $\lambda_3$, c, d, and $\cos \theta_0$ parameters are only used for three-body interactions. The R and D parameters are used for both two-body and three-body interactions. The non-annotated parameters are unitless. The value of m must be 3 or 1.

The Tersoff potential file must contain entries for all the elements listed in the pair_coeff command. It can also contain entries for additional elements not being used in a particular simulation; LAMMPS ignores those entries.

For a single-element simulation, only a single entry is required (e.g. SiSiSi). For a two-element simulation, the file must contain 8 entries (for SiSiSi, SiSiC, SiCSi, SiCC, CSiSi, CSiC, CCSi, CCC), that specify Tersoff parameters for all permutations of the two elements interacting in three-body configurations. Thus for 3 elements, 27 entries would be required, etc.

As annotated above, the first element in the entry is the center atom in a three-body interaction and it is bonded to the second atom and the bond is influenced by the third atom. Thus an entry for SiCC means Si bonded to a C with another C atom influencing the bond. Thus three-body parameters for SiCSi and SiSiC entries will not, in general, be the same. The parameters used for the two-body interaction come from the entry where the second element is repeated. Thus the two-body parameters for Si interacting with C, comes from the SiCC entry.

The parameters used for a particular three-body interaction come from the entry with the corresponding three elements. The parameters used only for two-body interactions (n, $\beta$, $\lambda_2$, B, $\lambda_1$, and A) in entries whose second and third element are different (e.g. SiCSi) are not used for anything and can be set to 0.0 if desired.

Note that the twobody parameters in entries such as SiCC and CSiSi are often the same, due to the common use of symmetric mixing rules, but this is not always the case. For example, the beta and n parameters in Tersoff_2 *(Tersoff_2)*

are not symmetric. Similarly, the threebody parameters in entries such as SiCSi and SiSiC are often the same, but this is not always the case, particularly the value of R, which is sometimes typed on the first and second elements, sometimes on the first and third elements. Hence the need to specify R and D explicitly for all element triples. For example, while Tersoff's notation in Tersoff_2 *(Tersoff_2)* is ambiguous on this point, and properties of the zincblende lattice are the same for either choice, Tersoff's results for rocksalt are consistent with typing on the first and third elements. *Albe et al.* adopts the same convention. Conversely, the potential for B/N/C from the Cagin group uses the opposite convention, typing on the first and second elements.

We chose the above form so as to enable users to define all commonly used variants of the Tersoff potential. In particular, our form reduces to the original Tersoff form when m = 3 and gamma = 1, while it reduces to the form of *Albe et al.* when beta = 1 and m = 1. Note that in the current Tersoff implementation in LAMMPS, m must be specified as either 3 or 1. Tersoff used a slightly different but equivalent form for alloys, which we will refer to as Tersoff_2 potential *(Tersoff_2)*. The *tersoff/table* style implements Tersoff_2 parameterization only.

LAMMPS parameter values for Tersoff_2 can be obtained as follows: $\gamma_{ijk} = \omega_{ik}$, $\lambda_3 = 0$ and the value of m has no effect. The parameters for species i and j can be calculated using the Tersoff_2 mixing rules:

$$\lambda_1^{i,j} = \frac{1}{2}(\lambda_1^i + \lambda_1^j)$$
$$\lambda_2^{i,j} = \frac{1}{2}(\lambda_2^i + \lambda_2^j)$$
$$A_{i,j} = (A_i A_j)^{1/2}$$
$$B_{i,j} = \chi_{ij}(B_i B_j)^{1/2}$$
$$R_{i,j} = (R_i R_j)^{1/2}$$
$$S_{i,j} = (S_i S_j)^{1/2}$$

Tersoff_2 parameters R and S must be converted to the LAMMPS parameters R and D (R is different in both forms), using the following relations: R=(R'+S')/2 and D=(S'-R')/2, where the primes indicate the Tersoff_2 parameters.

In the potentials directory, the file SiCGe.tersoff provides the LAMMPS parameters for Tersoff's various versions of Si, as well as his alloy parameters for Si, C, and Ge. This file can be used for pure Si, (three different versions), pure C, pure Ge, binary SiC, and binary SiGe. LAMMPS will generate an error if this file is used with any combination involving C and Ge, since there are no entries for the GeC interactions (Tersoff did not publish parameters for this cross-interaction.) Tersoff files are also provided for the SiC alloy (SiC.tersoff) and the GaN (GaN.tersoff) alloys.

Many thanks to Rutuparna Narulkar, David Farrell, and Xiaowang Zhou for helping clarify how Tersoff parameters for alloys have been defined in various papers.

The *shift* keyword computes the energy E of a system of atoms, whose formula is the same as the Tersoff potential. The only modification is that the original equilibrium bond length ( $r_0$) of the system is shifted to $r_0 - \delta$. The minus sign arises because each radial distance $r$ is replaced by $r + \delta$.

The *shift* keyword is designed for simulations of closely matched van der Waals heterostructures. For instance, consider the case of a system with few-layers graphene atop a thick hexagonal boron nitride (h-BN) substrate simulated using periodic boundary conditions. The experimental lattice mismatch of ~1.8% between graphene and h-BN is not well captured by the equilibrium lattice constants of available potentials, thus a small in-plane strain will be introduced in the system when building a periodic supercell. To minimize the effect of strain on simulation results, the *shift* keyword allows adjusting the equilibrium bond length of one of the two materials (e.g., h-BN). Validation, benchmark tests, and applications of the *shift* keyword can be found in *(Mandelli_1)* and *(Ouyang_1)*.

For the specific case discussed above, the force field can be defined as

```
pair_style  hybrid/overlay rebo tersoff shift -0.00407 ilp/graphene/hbn 16.0 coul/shield␣
↪16.0
pair_coeff  * * rebo              CH.rebo     NULL NULL C
pair_coeff  * * tersoff           BNC.tersoff B    N    NULL
```

(continues on next page)

(continued from previous page)

```
pair_coeff  * * ilp/graphene/hbn  BNCH.ILP    B    N    C
pair_coeff  1 1 coul/shield 0.70
pair_coeff  1 2 coul/shield 0.695
pair_coeff  2 2 coul/shield 0.69
```

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.300.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, where types I and J correspond to two different element types, mixing is performed by LAMMPS as described above from values in the potential file.

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.300.5 Restrictions

This pair style is part of the MANYBODY package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires the *newton* setting to be "on" for pair interactions.

The *shift* keyword is not supported by the *tersoff/gpu*, *tersoff/intel*, *tersoff/kk*, *tersoff/table* or *tersoff/table/omp* variants.

The *tersoff/intel* pair style is only available when compiling LAMMPS with the Intel compilers.

The Tersoff potential files provided with LAMMPS (see the potentials directory) are parameterized for *"metal" units*. In addition the pair style supports converting potential parameters on-the-fly between "metal" and "real" units. You can use the *tersoff* pair style variants with any LAMMPS units setting, but you would need to create your own Tersoff potential file with coefficients listed in the appropriate units if your simulation does not use "metal" or "real" units.

### 4.300.6 Related commands

*pair_coeff*

### 4.300.7 Default

shift delta = 0.0

---

**(Tersoff_1)** J. Tersoff, Phys Rev B, 37, 6991 (1988).

**(Albe)** J. Nord, K. Albe, P. Erhart, and K. Nordlund, J. Phys.: Condens. Matter, 15, 5649(2003).

**(Tersoff_2)** J. Tersoff, Phys Rev B, 39, 5566 (1989); errata (PRB 41, 3248)

**(Mandelli_1)** D. Mandelli, W. Ouyang, M. Urbakh, and O. Hod, ACS Nano 13(7), 7603-7609 (2019).

**(Ouyang_1)** W. Ouyang et al., J. Chem. Theory Comput. 16(1), 666-676 (2020).

## 4.301 pair_style tersoff/mod command

Accelerator Variants: *tersoff/mod/gpu*, *tersoff/mod/kk*, *tersoff/mod/omp*

## 4.302 pair_style tersoff/mod/c command

Accelerator Variants: *tersoff/mod/c/omp*

### 4.302.1 Syntax

```
pair_style style keywords values
```

- style = *tersoff/mod* or *tersoff/mod/c*
- keyword = *shift*

  *shift* value = delta
    delta = negative shift in equilibrium bond length

### 4.302.2 Examples

```
pair_style tersoff/mod
pair_coeff * * Si.tersoff.mod Si Si

pair_style tersoff/mod/c
pair_coeff * * Si.tersoff.modc Si Si
```

### 4.302.3 Description

The *tersoff/mod* and *tersoff/mod/c* styles computes a bond-order type interatomic potential *(Kumagai)* based on a 3-body Tersoff potential *(Tersoff_1)*, *(Tersoff_2)* with modified cutoff function and angular-dependent term, giving the

energy E of a system of atoms as

$$
\begin{aligned}
E &= \frac{1}{2}\sum_i \sum_{j \neq i} V_{ij} \\
V_{ij} &= f_C(r_{ij}+\delta)\left[f_R(r_{ij}+\delta) + b_{ij} f_A(r_{ij}+\delta)\right] \\
f_C(r) &= \begin{cases} 1 & : \quad r < R-D \\ \frac{1}{2} - \frac{9}{16}\sin\left(\frac{\pi}{2}\frac{r-R}{D}\right) - \frac{1}{16}\sin\left(\frac{3\pi}{2}\frac{r-R}{D}\right) & : \quad R-D < r < R+D \\ 0 & : \quad r > R+D \end{cases} \\
f_R(r) &= A\exp(-\lambda_1 r) \\
f_A(r) &= -B\exp(-\lambda_2 r) \\
b_{ij} &= \left(1+{\zeta_{ij}}^{\eta}\right)^{-\frac{1}{2n}} \\
\zeta_{ij} &= \sum_{k \neq i,j} f_C(r_{ik}+\delta) g(\theta_{ijk}) \exp\left[\alpha(r_{ij}-r_{ik})^{\beta}\right] \\
g(\theta) &= c_1 + g_o(\theta) g_a(\theta) \\
g_o(\theta) &= \frac{c_2(h-\cos\theta)^2}{c_3 + (h-\cos\theta)^2} \\
g_a(\theta) &= 1 + c_4 \exp\left[-c_5(h-\cos\theta)^2\right]
\end{aligned}
$$

where $f_R$ is a two-body term and $f_A$ includes three-body interactions. $\delta$ is an optional negative shift of the equilibrium bond length, as described below.

The summations in the formula are over all neighbors J and K of atom I within a cutoff distance = R + D. The *tersoff/mod/c* style differs from *tersoff/mod* only in the formulation of the V_ij term, where it contains an additional c0 term.

$$
V_{ij} = f_C(r_{ij}+\delta)\left[f_R(r_{ij}+\delta) + b_{ij} f_A(r_{ij}+\delta) + c_0\right]
$$

The modified cutoff function $f_C$ proposed by *(Murty)* and having a continuous second-order differential is employed. The angular-dependent term $g(\theta)$ was modified to increase the flexibility of the potential.

The *tersoff/mod* potential is fitted to both the elastic constants and melting point by employing the modified Tersoff potential function form in which the angular-dependent term is improved. The model performs extremely well in describing the crystalline, liquid, and amorphous phases *(Schelling)*.

Only a single pair_coeff command is used with the *tersoff/mod* style which specifies a Tersoff/MOD potential file with parameters for all needed elements. These are mapped to LAMMPS atom types by specifying N additional arguments after the filename in the pair_coeff command, where N is the number of LAMMPS atom types:

- filename
- N element names = mapping of Tersoff/MOD elements to atom types

As an example, imagine the Si.tersoff_mod file has Tersoff values for Si. If your LAMMPS simulation has 3 Si atoms types, you would use the following pair_coeff command:

```
pair_coeff * * Si.tersoff_mod Si Si Si
```

The first 2 arguments must be * * so as to span all LAMMPS atom types. The three Si arguments map LAMMPS atom types 1,2,3 to the Si element in the Tersoff/MOD file. If a mapping value is specified as NULL, the mapping is not performed. This can be used when a *tersoff/mod* potential is used as part of the *hybrid* pair style. The NULL values are placeholders for atom types that will be used with other potentials.

Tersoff/MOD file in the *potentials* directory of the LAMMPS distribution have a ".tersoff.mod" suffix. Potential files for the *tersoff/mod/c* style have the suffix ".tersoff.modc". Lines that are not blank or comments (starting with #) define parameters for a triplet of elements. The parameters in a single entry correspond to coefficients in the formulae above:

- element 1 (the center atom in a 3-body interaction)
- element 2 (the atom bonded to the center atom)
- element 3 (the atom influencing the 1-2 bond in a bond-order sense)
- $\beta$
- $\alpha$
- h
- $\eta$
- $\beta_{ters}$ = 1 (dummy parameter)
- $\lambda_2$ (1/distance units)
- B (energy units)
- R (distance units)
- D (distance units)
- $\lambda_1$ (1/distance units)
- A (energy units)
- n
- c1
- c2
- c3
- c4
- c5
- c0 (energy units, tersoff/mod/c only)

The n, $\eta$, $\lambda_2$, B, $\lambda_1$, and A parameters are only used for two-body interactions. The $\beta$, $\alpha$, c1, c2, c3, c4, c5, h parameters are only used for three-body interactions. The R and D parameters are used for both two-body and three-body interactions. The c0 term applies to *tersoff/mod/c* only. The non-annotated parameters are unitless.

The Tersoff/MOD potential file must contain entries for all the elements listed in the pair_coeff command. It can also contain entries for additional elements not being used in a particular simulation; LAMMPS ignores those entries.

For a single-element simulation, only a single entry is required (e.g. SiSiSi). As annotated above, the first element in the entry is the center atom in a three-body interaction and it is bonded to the second atom and the bond is influenced by the third atom. Thus an entry for SiSiSi means Si bonded to a Si with another Si atom influencing the bond.

The *shift* keyword computes the energy E of a system of atoms, whose formula is the same as the Tersoff potential. The only modification is that the original equilibrium bond length ( $r_0$) of the system is shifted to $r_0 - \delta$. The minus sign arises because each radial distance $r$ is replaced by $r + \delta$. More information on this option is given on the main *pair_tersoff* page.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.302.4 Mixing, shift, table, tail correction, restart, rRESPA info

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.302.5 Restrictions

This pair style is part of the MANYBODY package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires the *newton* setting to be "on" for pair interactions.

The *shift* keyword is not supported by the *tersoff/gpu*, *tersoff/intel*, *tersoff/kk*, *tersoff/table* or *tersoff/table/omp* variants.

The *tersoff/mod* potential files provided with LAMMPS (see the potentials directory) are parameterized for metal *units*. You can use the *tersoff/mod* pair style with any LAMMPS units, but you would need to create your own Tersoff/MOD potential file with coefficients listed in the appropriate units if your simulation does not use "metal" units.

### 4.302.6 Related commands

*pair_coeff*

### 4.302.7 Default

none

---

**(Kumagai)** T. Kumagai, S. Izumi, S. Hara, S. Sakai, Comp. Mat. Science, 39, 457 (2007).

**(Tersoff_1)** J. Tersoff, Phys Rev B, 37, 6991 (1988).

**(Tersoff_2)** J. Tersoff, Phys Rev B, 38, 9902 (1988).

**(Murty)** M.V.R. Murty, H.A. Atwater, Phys Rev B, 51, 4889 (1995).

**(Schelling)** Patrick K. Schelling, Comp. Mat. Science, 44, 274 (2008).

## 4.303 pair_style tersoff/zbl command

Accelerator Variants: *tersoff/zbl/gpu*, *tersoff/zbl/kk*, *tersoff/zbl/omp*

### 4.303.1 Syntax

```
pair_style tersoff/zbl keywords values
```

- keyword = *shift*

```
shift value = delta
  delta = negative shift in equilibrium bond length
```

### 4.303.2 Examples

```
pair_style tersoff/zbl
pair_coeff * * SiC.tersoff.zbl Si C Si
```

### 4.303.3 Description

The *tersoff/zbl* style computes a 3-body Tersoff potential *(Tersoff_1)* with a close-separation pairwise modification based on a Coulomb potential and the Ziegler-Biersack-Littmark universal screening function *(ZBL)*, giving the energy E of a system of atoms as

$$
\begin{aligned}
E &= \frac{1}{2}\sum_i \sum_{j \neq i} V_{ij} \\
V_{ij} &= (1 - f_F(r_{ij}+\delta))V^{ZBL}(r_{ij}+\delta) + f_F(r_{ij}+\delta)V^{Tersoff}(r_{ij}+\delta) \\
f_F(r) &= \frac{1}{1+e^{-A_F(r-r_C)}}
\end{aligned}
$$

$$
\begin{aligned}
V^{ZBL}(r) &= \frac{1}{4\pi\varepsilon_0}\frac{Z_1 Z_2\, e^2}{r}\phi(r/a) \\
a &= \frac{0.8854\, a_0}{Z_1^{0.23} + Z_2^{0.23}} \\
\phi(x) &= 0.1818e^{-3.2x} + 0.5099e^{-0.9423x} + 0.2802e^{-0.4029x} + 0.02817e^{-0.2016x}
\end{aligned}
$$

$$
\begin{aligned}
V^{Tersoff}(r) &= f_C(r)\left[f_R(r) + b_{ij} f_A(r)\right] \\
f_C(r) &= \begin{cases} 1 & : \quad r < R - D \\ \frac{1}{2} - \frac{1}{2}\sin\left(\frac{\pi}{2}\frac{r-R}{D}\right) & : \quad R - D < r < R + D \\ 0 & : \quad r > R + D \end{cases} \\
f_R(r) &= A\exp(-\lambda_1 r) \\
f_A(r) &= -B\exp(-\lambda_2 r) \\
b_{ij} &= \left(1 + \beta^n {\zeta_{ij}}^n\right)^{-\frac{1}{2n}} \\
\zeta_{ij} &= \sum_{k \neq i,j} f_C(r_{ik}+\delta) g(\theta_{ijk}) \exp\left[{\lambda_3}^m (r_{ij} - r_{ik})^m\right] \\
g(\theta) &= \gamma_{ijk}\left(1 + \frac{c^2}{d^2} - \frac{c^2}{\left[d^2 + (\cos\theta - \cos\theta_0)^2\right]}\right)
\end{aligned}
$$

The $f_F$ term is a fermi-like function used to smoothly connect the ZBL repulsive potential with the Tersoff potential. There are 2 parameters used to adjust it: $A_F$ and $r_C$. $A_F$ controls how "sharp" the transition is between the two, and $r_C$ is essentially the cutoff for the ZBL potential.

For the ZBL portion, there are two terms. The first is the Coulomb repulsive term, with Z1, Z2 as the number of protons in each nucleus, e as the electron charge (1 for metal and real units) and $\varepsilon_0$ as the permittivity of vacuum. The second part is the ZBL universal screening function, with a0 being the Bohr radius (typically 0.529 Angstroms), and the remainder of the coefficients provided by the original paper. This screening function should be applicable to most systems. However, it is only accurate for small separations (i.e. less than 1 Angstrom).

For the Tersoff portion, $f_R$ is a two-body term and $f_A$ includes three-body interactions. The summations in the formula are over all neighbors J and K of atom I within a cutoff distance = R + D.

$\delta$ is an optional negative shift of the equilibrium bond length, as described below.

Only a single pair_coeff command is used with the *tersoff/zbl* style which specifies a Tersoff/ZBL potential file with parameters for all needed elements. These are mapped to LAMMPS atom types by specifying N additional arguments after the filename in the pair_coeff command, where N is the number of LAMMPS atom types:

- filename
- N element names = mapping of Tersoff/ZBL elements to atom types

See the *pair_coeff* page for alternate ways to specify the path for the potential file.

As an example, imagine the SiC.tersoff.zbl file has Tersoff/ZBL values for Si and C. If your LAMMPS simulation has 4 atoms types and you want the first 3 to be Si, and the fourth to be C, you would use the following pair_coeff command:

```
pair_coeff * * SiC.tersoff Si Si Si C
```

The first 2 arguments must be * * so as to span all LAMMPS atom types. The first three Si arguments map LAMMPS atom types 1,2,3 to the Si element in the Tersoff/ZBL file. The final C argument maps LAMMPS atom type 4 to the C element in the Tersoff/ZBL file. If a mapping value is specified as NULL, the mapping is not performed. This can be used when a *tersoff/zbl* potential is used as part of the *hybrid* pair style. The NULL values are placeholders for atom types that will be used with other potentials.

Tersoff/ZBL files in the *potentials* directory of the LAMMPS distribution have a ".tersoff.zbl" suffix. Lines that are not blank or comments (starting with #) define parameters for a triplet of elements. The parameters in a single entry correspond to coefficients in the formula above:

- element 1 (the center atom in a 3-body interaction)
- element 2 (the atom bonded to the center atom)
- element 3 (the atom influencing the 1-2 bond in a bond-order sense)
- m
- $\gamma$
- $\lambda_3$ (1/distance units)
- c
- d
- cos $\theta_0$ (can be a value < -1 or > 1)
- n
- $\beta$
- $\lambda_2$ (1/distance units)
- B (energy units)
- R (distance units)
- D (distance units)

- $\lambda_1$ (1/distance units)
- A (energy units)
- $Z_i$
- $Z_j$
- ZBLcut (distance units)
- ZBLexpscale (1/distance units)

The n, $\beta$, $\lambda_2$, B, $\lambda_1$, and A parameters are only used for two-body interactions. The m, $\gamma$, $\lambda_3$, c, d, and $\cos\theta_0$ parameters are only used for three-body interactions. The R and D parameters are used for both two-body and three-body interactions. The $Z_i$, $Z_j$, ZBLcut, ZBLexpscale parameters are used in the ZBL repulsive portion of the potential and in the Fermi-like function. The non-annotated parameters are unitless. The value of m must be 3 or 1.

The Tersoff/ZBL potential file must contain entries for all the elements listed in the pair_coeff command. It can also contain entries for additional elements not being used in a particular simulation; LAMMPS ignores those entries.

For a single-element simulation, only a single entry is required (e.g. SiSiSi). For a two-element simulation, the file must contain 8 entries (for SiSiSi, SiSiC, SiCSi, SiCC, CSiSi, CSiC, CCSi, CCC), that specify Tersoff parameters for all permutations of the two elements interacting in three-body configurations. Thus for 3 elements, 27 entries would be required, etc.

As annotated above, the first element in the entry is the center atom in a three-body interaction and it is bonded to the second atom and the bond is influenced by the third atom. Thus an entry for SiCC means Si bonded to a C with another C atom influencing the bond. Thus three-body parameters for SiCSi and SiSiC entries will not, in general, be the same. The parameters used for the two-body interaction come from the entry where the second element is repeated. Thus the two-body parameters for Si interacting with C, comes from the SiCC entry.

The parameters used for a particular three-body interaction come from the entry with the corresponding three elements. The parameters used only for two-body interactions (n, $\beta$, $\lambda_2$, B, $\lambda_1$, and A) in entries whose second and third element are different (e.g. SiCSi) are not used for anything and can be set to 0.0 if desired.

Note that the twobody parameters in entries such as SiCC and CSiSi are often the same, due to the common use of symmetric mixing rules, but this is not always the case. For example, the beta and n parameters in Tersoff_2 *(Tersoff_2)* are not symmetric.

We chose the above form so as to enable users to define all commonly used variants of the Tersoff portion of the potential. In particular, our form reduces to the original Tersoff form when m = 3 and gamma = 1, while it reduces to the form of *Albe et al.* when beta = 1 and m = 1. Note that in the current Tersoff implementation in LAMMPS, m must be specified as either 3 or 1. Tersoff used a slightly different but equivalent form for alloys, which we will refer to as Tersoff_2 potential *(Tersoff_2)*.

LAMMPS parameter values for Tersoff_2 can be obtained as follows: $\gamma = \omega_{ijk}$, $\lambda_3 = 0$ and the value of m has no effect. The parameters for species i and j can be calculated using the Tersoff_2 mixing rules:

$$\lambda_1^{i,j} = \frac{1}{2}(\lambda_1^i + \lambda_1^j)$$
$$\lambda_2^{i,j} = \frac{1}{2}(\lambda_2^i + \lambda_2^j)$$
$$A_{i,j} = (A_i A_j)^{1/2}$$
$$B_{i,j} = \chi_{ij}(B_i B_j)^{1/2}$$
$$R_{i,j} = (R_i R_j)^{1/2}$$
$$S_{i,j} = (S_i S_j)^{1/2}$$

Tersoff_2 parameters R and S must be converted to the LAMMPS parameters R and D (R is different in both forms), using the following relations: R=(R'+S')/2 and D=(S'-R')/2, where the primes indicate the Tersoff_2 parameters.

In the potentials directory, the file SiCGe.tersoff provides the LAMMPS parameters for Tersoff's various versions of Si, as well as his alloy parameters for Si, C, and Ge. This file can be used for pure Si, (three different versions), pure C, pure Ge, binary SiC, and binary SiGe. LAMMPS will generate an error if this file is used with any combination involving C and Ge, since there are no entries for the GeC interactions (Tersoff did not publish parameters for this cross-interaction.) Tersoff files are also provided for the SiC alloy (SiC.tersoff) and the GaN (GaN.tersoff) alloys.

Many thanks to Rutuparna Narulkar, David Farrell, and Xiaowang Zhou for helping clarify how Tersoff parameters for alloys have been defined in various papers. Also thanks to Ram Devanathan for providing the base ZBL implementation.

The *shift* keyword computes the energy E of a system of atoms, whose formula is the same as the Tersoff potential. The only modification is that the original equilibrium bond length ( $r_0$) of the system is shifted to $r_0 - \delta$. The minus sign arises because each radial distance $r$ is replaced by $r + \delta$. More information on this option is given on the main *pair_tersoff* page.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.303.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, where types I and J correspond to two different element types, mixing is performed by LAMMPS as described above from values in the potential file.

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.303.5 Restrictions

This pair style is part of the MANYBODY package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires the *newton* setting to be "on" for pair interactions.

The *shift* keyword is currently not supported for the *tersoff/gpu* and *tersoff/kk* variants of this pair style.

The tersoff/zbl potential files provided with LAMMPS (see the potentials directory) are parameterized for *"metal" units*. Also the pair style supports converting potential file parameters on-the-fly between "metal" and "real" units. You can use the tersoff/zbl pair style with any LAMMPS units, but you would need to create your own tersoff/zbl potential file with coefficients listed in the appropriate units if your simulation does not use "metal" or "real" units.

### 4.303.6 Related commands

*pair_coeff*

### 4.303.7 Default

none

---

**(Tersoff_1)** J. Tersoff, Phys Rev B, 37, 6991 (1988).

**(ZBL)** J.F. Ziegler, J.P. Biersack, U. Littmark, 'Stopping and Ranges of Ions in Matter' Vol 1, 1985, Pergamon Press.

**(Albe)** J. Nord, K. Albe, P. Erhart and K. Nordlund, J. Phys.: Condens. Matter, 15, 5649(2003).

**(Tersoff_2)** J. Tersoff, Phys Rev B, 39, 5566 (1989); errata (PRB 41, 3248)

## 4.304 pair_style thole command

## 4.305 pair_style lj/cut/thole/long command

Accelerator Variants: *lj/cut/thole/long/omp*

### 4.305.1 Syntax

```
pair_style style args
```

- style = *thole* or *lj/cut/thole/long*
- args = list of arguments for a particular style

```
thole args = damp cutoff
  damp = global damping parameter
  cutoff = global cutoff (distance units)
lj/cut/thole/long args = damp cutoff (cutoff2)
  damp = global damping parameter
  cutoff = global cutoff for LJ (and Thole if only 1 arg) (distance units)
  cutoff2 = global cutoff for Thole (optional) (distance units)
```

### 4.305.2 Examples

```
pair_style hybrid/overlay ... thole 2.6 12.0
pair_coeff 1 1 thole 1.0
pair_coeff 1 2 thole 1.0 2.6 10.0
pair_coeff * 2 thole 1.0 2.6

pair_style lj/cut/thole/long 2.6 12.0
```

Example input scripts available: examples/PACKAGES/drude

### 4.305.3 Description

The *thole* pair styles are meant to be used with force fields that include explicit polarization through Drude dipoles. This link describes how to use the *thermalized Drude oscillator model* in LAMMPS and polarizable models in LAMMPS are discussed on the *Howto polarizable* doc page.

The *thole* pair style should be used as a sub-style within in the *pair_style hybrid/overlay* command, in conjunction with a main pair style including Coulomb interactions, i.e. any pair style containing *coul/cut* or *coul/long* in its style name.

The *lj/cut/thole/long* pair style is equivalent to, but more convenient that the frequent combination *hybrid/overlay lj/cut/coul/long cutoff thole damp cutoff2*. It is not only a shorthand for this pair_style combination, but it also allows for mixing pair coefficients instead of listing them all. The *lj/cut/thole/long* pair style is also a bit faster because it avoids an overlay and can benefit from OMP acceleration. Moreover, it uses a more precise approximation of the direct Coulomb interaction at short range similar to *coul/long/cs*, which stabilizes the temperature of Drude particles.

The *thole* pair styles compute the Coulomb interaction damped at short distances by a function

$$T_{ij}(r_{ij}) = 1 - \left(1 + \frac{s_{ij} r_{ij}}{2}\right) \exp\left(-s_{ij} r_{ij}\right)$$

This function results from an adaptation to point charges *(Noskov)* of the dipole screening scheme originally proposed by *Thole*. The scaling coefficient $s_{ij}$ is determined by the polarizability of the atoms, $\alpha_i$, and by a Thole damping parameter $a$. This Thole damping parameter usually takes a value of 2.6, but in certain force fields the value can depend upon the atom types. The mixing rule for Thole damping parameters is the arithmetic average, and for polarizabilities the geometric average between the atom-specific values.

$$s_{ij} = \frac{a_{ij}}{(\alpha_{ij})^{1/3}} = \frac{(a_i + a_j)/2}{[(\alpha_i \alpha_j)^{1/2}]^{1/3}}$$

The damping function is only applied to the interactions between the point charges representing the induced dipoles on polarizable sites, that is, charges on Drude particles, $q_{D,i}$, and opposite charges, $-q_{D,i}$, located on the respective core particles (to which each Drude particle is bonded). Therefore, Thole screening is not applied to the full charge of the core particle $q_i$, but only to the $-q_{D,i}$ part of it.

The interactions between core charges are subject to the weighting factors set by the *special_bonds* command. The interactions between Drude particles and core charges or non-polarizable atoms are also subject to these weighting factors. The Drude particles inherit the 1-2, 1-3 and 1-4 neighbor relations from their respective cores.

For pair_style *thole*, the following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the example above.

- $\alpha$ (distance units^3)
- damp
- cutoff (distance units)

The last two coefficients are optional. If not specified the global Thole damping parameter or global cutoff specified in the pair_style command are used. In order to specify a cutoff (third argument) a damp parameter (second argument) must also be specified.

For pair style *lj/cut/thole/long*, the following coefficients must be defined for each pair of atoms types via the *pair_coeff* command.

- $\varepsilon$ (energy units)
- $\sigma$ (length units)
- $\alpha$ (distance units^3)
- damp
- LJ cutoff (distance units)

The last two coefficients are optional and default to the global values from the *pair_style* command.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages*

page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 4.305.4 Mixing, shift, table, tail correction, restart, rRESPA info

The *thole* pair style does not support mixing. Thus, coefficients for all I,J pairs must be specified explicitly.

The *lj/cut/thole/long* pair style does support mixing. Mixed coefficients are defined using

$$\alpha_{ij} = \sqrt{\alpha_i \alpha_j}$$

$$a_{ij} = \frac{1}{2}(a_i + a_j)$$

### 4.305.5 Restrictions

These pair styles are part of the DRUDE package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair_style should currently not be used with the *charmm dihedral style* if the latter has non-zero 1-4 weighting factors. This is because the *thole* pair style does not know which pairs are 1-4 partners of which dihedrals.

The *lj/cut/thole/long* pair style should be used with a *Kspace solver* like PPPM or Ewald, which is only enabled if LAMMPS was built with the kspace package.

### 4.305.6 Related commands

*fix drude*, *fix langevin/drude*, *fix drude/transform*, *compute temp/drude pair_style lj/cut/coul/long*

### 4.305.7 Default

none

---

**(Noskov)** Noskov, Lamoureux and Roux, J Phys Chem B, 109, 6705 (2005).

**(Thole)** Chem Phys, 59, 341 (1981).

## 4.306 pair_style threebody/table command

Accelerator Variants: *threebody/table/omp*

### 4.306.1 Syntax

```
pair_style style
```

- style = *threebody/table*

### 4.306.2 Examples

```
pair_style threebody/table
pair_coeff * * spce2.3b type1 type2

pair_style hybrid/overlay table linear 1200 threebody/table
pair_coeff      1 1 table table_CG_CG.txt VOTCA
pair_coeff      * * threebody/table spce.3b type
```

Used in example input scripts:

```
examples/PACKAGES/manybody_table/in.spce
examples/PACKAGES/manybody_table/in.spce2
```

### 4.306.3 Description

Added in version 2Jun2022.

The *threebody/table* style is a pair style for generic tabulated three-body interactions. It has been developed for (coarse-grained) simulations (of water) with Kernel-based machine learning (ML) potentials (*Scherer2*). As for many other MANYBODY package pair styles the energy of a system is computed as a sum over three-body terms:

$$E = \sum_i \sum_{j \neq i} \sum_{k>j} \phi_3(r_{ij}, r_{ik}, \theta_{ijk})$$

The summations in the formula are over all neighbors J and K of atom I within a cutoff distance *cut*. In contrast to the Stillinger-Weber potential, all forces are not calculated analytically, but read in from a three-body force/energy table which can be generated with the csg_ml app of VOTCA as available at: https://gitlab.mpcdf.mpg.de/votca/votca.

Only a single pair_coeff command is used with the *threebody/table* style which specifies a threebody potential (".3b") file with parameters for all needed elements. These are then mapped to LAMMPS atom types by specifying N_el additional arguments after the ".3b" filename in the pair_coeff command, where N_el is the number of LAMMPS atom types:

- ".3b" filename
- N_el element names = mapping of threebody elements to atom types

See the *pair_coeff* page for alternate ways to specify the path for the potential file.

As an example, imagine a file SiC.3b has three-body values for Si and C. If your LAMMPS simulation has 4 atoms types and you want the first 3 to be Si, and the fourth to be C, you would use the following pair_coeff command:

```
pair_coeff * * SiC.3b Si Si Si C
```

The first 2 arguments must be * * so as to span all LAMMPS atom types. The first three Si arguments map LAMMPS atom types 1,2,3 to the Si element in the ".3b" file. The final C argument maps LAMMPS atom type 4 to the C element in the threebody file. If a mapping value is specified as NULL, the mapping is not performed. This can be used when a *threebody/table* potential is used as part of the *hybrid* pair style. The NULL values are placeholders for atom types that will be used with other potentials.

The three-body files have a ".3b" suffix. Lines that are not blank or comments (starting with #) define parameters for a triplet of elements. The parameters in a single entry specify to the (three-body) cutoff distance and the tabulated three-body interaction. A single entry then contains:

- element 1 (the center atom in a 3-body interaction)
- element 2
- element 3

- cut (distance units)
- filename
- keyword
- style
- N

The parameter *cut* is the (three-body) cutoff distance. When set to 0, no interaction is calculated for this element triplet. The parameters *filename*, *keyword*, *style*, and *N* refer to the tabulated three-body potential.

The tabulation is done on a three-dimensional grid of the two distances $r_{ij}$ and $r_{ik}$ as well as the angle $\theta_{ijk}$ which is constructed in the following way. There are two different cases. If element 2 and element 3 are of the same type (e.g. SiCC), the distance $r_{ij}$ is varied in "N" steps from rmin to rmax and the distance $r_{ik}$ is varied from $r_{ij}$ to rmax. This can be done, due to the symmetry of the triplet. If element 2 and element 3 are not of the same type (e.g. SiCSi), there is no additional symmetry and the distance $r_{ik}$ is also varied from rmin to rmax in "N" steps. The angle $\theta_{ijk}$ is always varied in "2N" steps from (0.0 + 180.0/(4N)) to (180.0 - 180.0/(4N)). Therefore, the total number of table entries is "M = N * N * (N+1)" for the symmetric (element 2 and element 3 are of the same type) and "M = 2 * N * N * N" for the general case (element 2 and element 3 are not of the same type).

The forces on all three particles I, J, and K of a triplet of this type of three-body interaction potential ($\phi_3(r_{ij}, r_{ik}, \theta_{ijk})$) lie within the plane defined by the three inter-particle distance vectors $\mathbf{r}_{ij}$, $\mathbf{r}_{ik}$, and $\mathbf{r}_{jk}$. This property is used to project the forces onto the inter-particle distance vectors as follows

$$\begin{pmatrix} \mathbf{f}_i \\ \mathbf{f}_j \\ \mathbf{f}_k \end{pmatrix} = \begin{pmatrix} f_{i1} & f_{i2} & 0 \\ f_{j1} & 0 & f_{j2} \\ 0 & f_{k1} & f_{k2} \end{pmatrix} \begin{pmatrix} \mathbf{r}_{ij} \\ \mathbf{r}_{ik} \\ \mathbf{r}_{jk} \end{pmatrix}$$

and then tabulate the 6 force constants $f_{i1}$, $f_{i2}$, $f_{j1}$, $f_{j2}$, $f_{k1}$, and $f_{k2}$, as well as the energy of a triplet e. Due to symmetry reasons, the following relations hold: $f_{i1} = -f_{j1}$, $f_{i2} = -f_{k1}$, and $f_{j2} = -f_{k2}$. As in this pair style the forces are read in directly, a correct MD simulation is also performed in the case that the triplet energies are set to e=0.

The *filename* specifies the file containing the tabulated energy and derivative values of $\phi_3(r_{ij}, r_{ik}, \theta_{ijk})$. The *keyword* then specifies a section of the file. The format of this file is as follows (without the parenthesized comments):

```
# Tabulated three-body potential for spce water (one or more comment or blank lines)

ENTRY1                                                                    (keyword is␣
↪the first text on line)
N 12 rmin 2.55 rmax 3.65                                                  (N, rmin,␣
↪rmax parameters)
                                                                          (blank line)
1 2.55 2.55 3.75 -867.212 -611.273 867.212 21386.8 611.273 -21386.8  0.0    (index, r_ij,
↪ r_ik, theta, f_i1, f_i2, f_j1, f_j2, f_k1, f_k2, e)
2 2.55 2.55 11.25 -621.539 -411.189 621.539 5035.95 411.189 -5035.95  0.0
...
1872 3.65 3.65 176.25 -0.00215132 -0.00412886 0.00215137 0.00111754 0.00412895 -0.
↪00111757  0.0
```

A section begins with a non-blank line whose first character is not a "#"; blank lines or lines starting with "#" can be used as comments between sections. The first line begins with a keyword which identifies the section. The next line lists (in any order) one or more parameters for the table. Each parameter is a keyword followed by one or more numeric values.

The parameter "N" is required. It should be the same than the parameter "N" of the ".3b" file, otherwise its value is overwritten. "N" determines the number of table entries "M" that follow: "M = N * N * (N+1)" (symmetric triplet, e.g. SiCC) or "M = 2 * N * N * N" (asymmetric triplet, e.g. SiCSi). Therefore "M = 12 * 12 * 13 = 1872" in the above

symmetric example. The parameters "rmin" and "rmax" are also required and determine the minimum and maximum of the inter-particle distances $r_{ij}$ and $r_{ik}$.

Following a blank line, the next M lines of the angular table file list the tabulated values. On each line, the first value is the index from 1 to M, the second value is the distance $r_{ij}$, the third value is the distance $r_{ik}$, the fourth value is the angle $\theta_{ijk}$), the next six values are the force constants $f_{i1}$, $f_{i2}$, $f_{j1}$, $f_{j2}$, $f_{k1}$, and $f_{k2}$, and the last value is the energy e.

Note that one three-body potential file can contain many sections, each with a tabulated potential. LAMMPS reads the file section by section until it finds one that matches the specified *keyword* of appropriate section of the ".3b" file.

At the moment, only the *style linear* is allowed and implemented. After reading in the force table, it is internally stored in LAMMPS as a lookup table. For each triplet configuration occurring in the simulation within the cutoff distance, the next nearest tabulated triplet configuration is looked up. No interpolation is done. This allows for a very efficient force calculation with the stored force constants and energies. Due to the know table structure, the lookup can be done efficiently. It has been tested (*Scherer2*) that with a reasonably small bin size, the accuracy and speed is comparable to that of a Stillinger-Weber potential with tabulated three-body interactions (*pair_style sw/angle/table*) while the table format of this pair style allows for more flexible three-body interactions.

As for the Stillinger-Weber potential, the three-body potential file must contain entries for all the elements listed in the pair_coeff command. It can also contain entries for additional elements not being used in a particular simulation; LAMMPS ignores those entries.

For a single-element simulation, only a single entry is required (e.g. SiSiSi). For a two-element simulation, the file must contain 8 entries (for SiSiSi, SiSiC, SiCSi, SiCC, CSiSi, CSiC, CCSi, CCC), that specify threebody parameters for all permutations of the two elements interacting in three-body configurations. Thus for 3 elements, 27 entries would be required, etc.

As annotated above, the first element in the entry is the center atom in a three-body interaction. Thus an entry for SiCC means a Si atom with 2 C atoms as neighbors. The tabulated three-body forces can in principle be specific to the three elements of the configuration. However, the user must ensure that it makes physically sense. E.g., the tabulated three-body forces for the entries CSiC and CCSi should be the same exchanging $r_{ij}$ with r_{ik}, $f_{j1}$ with $f_{k1}$, and $f_{j2}$ with $f_{k2}$.

Additional input files and reference data can be found at: https://gitlab.mpcdf.mpg.de/votca/votca/-/tree/master/csg-tutorials/ml

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.306.4 Mixing, shift, table, tail correction, restart, rRESPA info

As all interactions are tabulated, no mixing is performed.

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.306.5 Restrictions

This pair style is part of the MANYBODY package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires the *newton* setting to be "on" for pair interactions.

### 4.306.6 Related commands

*pair_coeff*, *pair sw/angle/table*

---

**(Scherer2)** C. Scherer, R. Scheid, D. Andrienko, and T. Bereau, J. Chem. Theor. Comp. 16, 3194-3204 (2020).

## 4.307 pair_style tracker command

### 4.307.1 Syntax

```
pair_style tracker fix_ID N keyword values attribute1 attribute2 ...
```

- fix_ID = ID of associated internal fix to store data
- N = prepare data for output every this many timesteps
- zero or more keywords may be appended
- keyword = *finite* or *time/min* or *type/include*

```
finite value = none
   pair style uses atomic diameters to identify contacts
time/min value = T
   T = minimum number of timesteps of interaction
type/include value = list1 list2
   list1,list2 = separate lists of types (see below)
```

- one or more attributes may be appended

```
possible attributes = id1 id2 time/created time/broken time/total
                      r/min r/ave x y z
```

```
id1, id2 = IDs of the two atoms in each pair interaction
time/created = the timestep that the two atoms began interacting
time/broken = the timestep that the two atoms stopped interacting
time/total = the total number of timesteps the two atoms interacted
r/min = the minimum radial distance between the two atoms during the interaction␣
↪(distance units)
r/ave = the average radial distance between the two atoms during the interaction␣
↪(distance units)
x, y, z = the center of mass position of the two atoms when they stopped␣
↪interacting (distance units)
```

### 4.307.2 Examples

```
pair_style hybrid/overlay tracker myfix 1000 id1 id2 type/include 1 * type/include 2 3,4␣
↪ lj/cut 2.5
pair_coeff 1 1 tracker 2.0

pair_style hybrid/overlay tracker myfix 1000 finite x y z time/min 100 granular
pair_coeff * * tracker

dump 1 all local 1000 dump.local f_myfix[1] f_myfix[2] f_myfix[3]
dump_modify 1 write_header no
```

### 4.307.3 Description

Style *tracker* monitors information about pairwise interactions. It does not calculate any forces on atoms. *Pair hybrid/overlay* can be used to combine this pair style with any other pair style, as shown in the examples above.

At each timestep, if two neighboring atoms move beyond the interaction cutoff, pairwise data is processed and transferred to an internal fix labeled *fix_ID*. This allows the local data to be accessed by other LAMMPS commands. Additional filters can be applied using the *time/min* or *type/include* keywords described below. Note that this is the interaction cutoff defined by this pair style, not the short-range cutoff defined by the pair style that is calculating forces on atoms.

Following any optional keyword/value arguments, a list of one or more attributes is specified. These include the IDs of the two atoms in the pair. The other attributes for the pair of atoms are the duration of time they were "interacting" or at the point in time they started or stopped interacting. In this context, "interacting" means the time window during which the two atoms were closer than the interaction cutoff distance. The attributes for time/* refer to timesteps.

Data is continuously accumulated by the internal fix over intervals of *N* timesteps. At the end of each interval, all of the saved accumulated data is deleted to make room for new data. Individual datum may therefore persist anywhere between *1* to *N* timesteps depending on when they are saved. This data can be accessed using the *fix_ID* and a *dump local* command. To ensure all data is output, the dump frequency should correspond to the same interval of *N* timesteps. A dump frequency of an integer multiple of *N* can be used to regularly output a sample of the accumulated data.

---

The following optional keywords may be used.

If the *finite* keyword is not used, the following coefficients must be defined for each pair of atom types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- cutoff (distance units)

If the *finite* keyword is used, there are no additional coefficients to set for each pair of atom types via the *pair_coeff* command. Interaction cutoffs are instead calculated based on the diameter of finite particles. However you must still use the *pair_coeff* for all atom types. For example the command

```
pair_coeff * *
```

should be used.

The *time/min* keyword sets a minimum amount of time that an interaction must persist to be included. This setting can be used to exclude short-lived interactions.

The *type/include* keyword filters interactions based on the types of the two atoms. Data is only saved for interactions between atoms whose two atom types appear in *list1* and *list2*. Atom type 1 must be in list1 and atom type 2 in list2. Or vice versa.

Each type list consists of a series of type ranges separated by commas. Each range can be specified as a single numeric value, or a wildcard asterisk can be used to specify a range of values. This takes the form "*" or "*n" or "n*" or "m*n".

For example, if M = the number of atom types, then an asterisk with no numeric values means all types from 1 to M. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to M (inclusive). A middle asterisk means all types from m to n (inclusive). Note that the *type/include* keyword can be specified multiple times.

### 4.307.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the cutoff coefficient and cutoff distance for this pair style can be mixed. The cutoff is always mixed via a *geometric* rule. The cutoff is mixed according to the pair_modify mix value. The default mix value is *geometric*. See the "pair_modify" command for details.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

The *pair_modify* shift, table, and tail options are not relevant for this pair style.

The accumulated data is not written to restart files and should be output before a restart file is written to avoid missing data.

The internal fix calculates a local vector or local array depending on the number of input values. The length of the vector or number of rows in the array is the number of recorded, lost interactions. If a single input is specified, a local vector is produced. If two or more inputs are specified, a local array is produced where the number of columns = the number of inputs. The vector or array can be accessed by any command that uses local values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The vector or array will be floating point values that correspond to the specified attribute.

---

### 4.307.5 Restrictions

This pair style is part of the MISC package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style is currently incompatible with granular pair styles that extend beyond the contact (e.g. JKR and DMT).

### 4.307.6 Related commands

### 4.307.7 Default

none

## 4.308 pair_style tri/lj command

Accelerator Variants: *tri/lj/omp*

### 4.308.1 Syntax

```
pair_style tri/lj cutoff
```

cutoff = global cutoff for interactions (distance units)

### 4.308.2 Examples

```
pair_style tri/lj 3.0
pair_coeff * * 1.0 1.0
pair_coeff 1 1 1.0 1.5 2.5
```

### 4.308.3 Description

Style *tri/lj* treats particles which are triangles as a set of small spherical particles that tile the triangle surface as explained below. Interactions between two triangles, each with N1 and N2 spherical particles, are calculated as the pairwise sum of N1*N2 Lennard-Jones interactions. Interactions between a triangle with N spherical particles and a point particle are treated as the pairwise sum of N Lennard-Jones interactions. See the *pair_style lj/cut* doc page for the definition of Lennard-Jones interactions.

The cutoff distance for an interaction between 2 triangles, or between a triangle and a point particle, is calculated from the position of the triangle (its centroid), not between pairs of individual spheres comprising the triangle. Thus an interaction is either calculated in its entirety or not at all.

The set of non-overlapping spherical particles that represent a triangle, for purposes of this pair style, are generated in the following manner. Assume the triangle is of type I, and sigma_II has been specified. We want a set of spheres with centers in the plane of the triangle, none of them larger in diameter than sigma_II, which completely cover the triangle's area, but with minimal overlap and a minimal total number of spheres. This is done in a recursive manner. Place a sphere at the centroid of the original triangle. Calculate what diameter it must have to just cover all 3 corner points of the triangle. If that diameter is equal to or smaller than sigma_II, then include a sphere of the calculated diameter in the set of covering spheres. It the diameter is larger than sigma_II, then split the triangle into 2 triangles by bisecting its longest side. Repeat the process on each sub-triangle, recursing as far as needed to generate a set of covering spheres. When finished, the original criteria are met, and the set of covering spheres should be near minimal in number and overlap, at least for input triangles with a reasonable aspect-ratio.

The LJ interaction between 2 spheres on different triangles of types I,J is computed with an arithmetic mixing of the sigma values of the 2 spheres and using the specified epsilon value for I,J atom types. Note that because the sigma values for triangles spheres is computed using only sigma_II values, specific to the triangles's type, this means that any specified sigma_IJ values (for I != J) are effectively ignored.

For style *tri/lj*, the following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- epsilon (energy units)
- sigma (distance units)
- cutoff (distance units)

The last coefficient is optional. If not specified, the global cutoff is used.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.308.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon and sigma coefficients and cutoff distance for all of this pair style can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write its information to *binary restart files*.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.308.5 Restrictions

This style is part of the ASPHERE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Defining particles to be triangles so they participate in tri/tri or tri/particle interactions requires the use the *atom_style tri* command.

### 4.308.6 Related commands

*pair_coeff*, *pair_style line/lj*

### 4.308.7 Default

none

## 4.309 pair_style uf3 command

Accelerator Variants: *uf3/kk*

### 4.309.1 Syntax

```
pair_style style BodyFlag
```

- style = *uf3* or *uf3/kk*

```
BodyFlag = Indicates whether to calculate only 2-body or 2 and 3-body interactions.␣
↪Possible values: 2 or 3
```

### 4.309.2 Examples

```
pair_style uf3 3
pair_coeff * * Nb.uf3 Nb

pair_style uf3 2
pair_coeff * * NbSn.uf3 Nb Sn

pair_style uf3 3
pair_coeff * * NbSn.uf3 Nb Sn
```

### 4.309.3 Description

Added in version 27June2024.

The *uf3* style computes the *Ultra-Fast Force Fields (UF3)* potential, a machine-learning interatomic potential. In UF3, the total energy of the system is defined via two- and three-body interactions:

$$E = \sum_{i,j} V_2(r_{ij}) + \sum_{i,j,k} V_3(r_{ij}, r_{ik}, r_{jk})$$

$$V_2(r_{ij}) = \sum_{n=0}^{N} c_n B_n(r_{ij})$$

$$V_3(r_{ij}, r_{ik}, r_{jk}) = \sum_{l=0}^{N_l} \sum_{m=0}^{N_m} \sum_{n=0}^{N_n} c_{l,m,n} B_l(r_{ij}) B_m(r_{ik}) B_n(r_{jk})$$

where $V_2(r_{ij})$ and $V_3(r_{ij}, r_{ik}, r_{jk})$ are the two- and three-body interactions, respectively. For the two-body the summation is over all neighbors J and for the three-body the summation is over all neighbors J and K of atom I within a cutoff distance determined from the potential files. $B_n(r_{ij})$ are the cubic b-spline basis, $c_n$ and $c_{l,m,n}$ are the machine-learned interaction parameters and $N$, $N_l$, $N_m$, and $N_n$ denote the number of basis functions per spline or tensor spline dimension.

With *uf3* style only a single pair_coeff command is used to indicate the UF3 LAMMPS potential file containing all the two- and three-body interactions followed by N additional arguments specifying the mapping of UF3 elements to LAMMPS atom types, where N is the number of LAMMPS atom types:

- UF3 LAMMPS potential file
- N elements names = mapping of UF3 elements to atom types

As an example, if a LAMMPS simulation contains 2 atom types (elements 'A' and 'B'), the pair_coeff command will be:

```
pair_style uf3 3
pair_coeff * * AB.uf3 A B
```

The AB.uf3 file should contain all two-body (A-A, A-B, B-B) and three-body (A-A-A, A-A-B, A-B-B, B-A-A, B-A-B, B-B-B).

If a value of "2" is specified in the `pair_style uf3` command, only the two-body potentials are needed. For 3-body interaction the first atom type is the central atom. We recommend using the `generate_uf3_lammps_pots.py` script (found here) for generating the UF3 LAMMPS potential file from the UF3 JSON potentials.

---

UF3 LAMMPS potential file in the *potentials* directory of the LAMMPS distribution have a ".uf3" suffix. The interaction block in UF3 LAMMPS potential file should start with `#UF3 POT` and end with # characters. Following shows the format of a generic 2-body and 3-body potential block in UF3 LAMMPS potential file-

```
#UF3 POT UNITS: units DATE: POT_GEN_DATE AUTHOR: AUTHOR_NAME CITATION: CITE
2B ELEMENT1 ELEMENT2 LEADING_TRIM TRAILING_TRIM
Rij_CUTOFF NUM_OF_KNOTS
BSPLINE_KNOTS
NUM_OF_COEFF
COEFF
#
#UF3 POT UNITS: units DATE: POT_GEN_DATE AUTHOR: AUTHOR_NAME CITATION: CITE
3B ELEMENT1 ELEMENT2 ELEMENT3 LEADING_TRIM TRAILING_TRIM
Rjk_CUTOFF Rik_CUTOFF Rij_CUTOFF NUM_OF_KNOTS_JK NUM_OF_KNOTS_IK NUM_OF_KNOTS_IJ
BSPLINE_KNOTS_FOR_JK
BSPLINE_KNOTS_FOR_IK
BSPLINE_KNOTS_FOR_IJ
SHAPE_OF_COEFF_MATRIX[I][J][K]
```

(continues on next page)

(continued from previous page)

```
COEFF_MATRIX[0][0][K]
COEFF_MATRIX[0][1][K]
COEFF_MATRIX[0][2][K]
.
.
.
COEFF_MATRIX[1][0][K]
COEFF_MATRIX[1][1][K]
COEFF_MATRIX[1][2][K]
.
.
.
#
```

The second line indicates whether the block contains data for 2-body (`2B`) or 3-body (`3B`) interaction. This is followed by element combination interaction, `LEADING_TRIM` and `TRAILING_TRIM` number on the same line. The current implementation is only tested for `LEADING_TRIM=0` and `TRAILING_TRIM=3`. If other values are used LAMMPS is terminated after issuing an error message. The `Rij_CUTOFF` sets the 2-body cutoff for the interaction described by the potential block. `NUM_OF_KNOTS` is the number of knots (or the length of the knot vector) present on the very next line. The `BSPLINE_KNOTS` line should contain all the knots in ascending order. `NUM_OF_COEFF` is the number of coefficients in the `COEFF` line. All the numbers in the BSPLINE_KNOTS and COEFF line should be space-separated. Similar to the 2-body potential block, the third line sets the cutoffs and length of the knots. The cutoff distance between atom-type I and J is `Rij_CUTOFF`, atom-type I and K is `Rik_CUTOFF` and between J and K is `Rjk_CUTOFF`.

**Note**

The current implementation only works for UF3 potentials with cutoff distances for 3-body interactions that follows `2Rij_CUTOFF=2Rik_CUTOFF=Rjk_CUTOFF` relation.

The `BSPLINE_KNOTS_FOR_JK`, `BSPLINE_KNOTS_FOR_IK`, and `BSPLINE_KNOTS_FOR_IJ` lines (note the order) contain the knots in increasing order for atoms J and K, I and K, and atoms I and J respectively. The number of knots is defined by the `NUM_OF_KNOTS_*` characters in the previous line. The shape of the coefficient matrix is defined on the `SHAPE_OF_COEFF_MATRIX[I][J][K]` line followed by the columns of the coefficient matrix, one per line, as shown above. For example, if the coefficient matrix has the shape of 8x8x13, then `SHAPE_OF_COEFF_MATRIX[I][J][K]` will be `8 8 13` followed by 64 (8x8) lines each containing 13 coefficients separated by space.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.309.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, where types I and J correspond to two different element types, mixing is performed by LAMMPS as described above from values in the potential file.

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in potential file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.309.5 Restrictions

The 'uf3' pair style is part of the ML-UF3 package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires the *newton* setting to be "on".

The UF3 LAMMPS potential file provided with LAMMPS (see the potentials directory) are parameterized for metal *units*.

The single() function of 'uf3' pair style only return the 2-body interaction energy.

### 4.309.6 Related commands

*pair_coeff*

### 4.309.7 Default

none

---

**(Xie23)** Xie, S.R., Rupp, M. & Hennig, R.G. Ultra-fast interpretable machine-learning potentials. npj Comput Mater 9, 162 (2023). https://doi.org/10.1038/s41524-023-01092-7

## 4.310 pair_style ufm command

Accelerator Variants: *ufm/gpu*, *ufm/kk*, *ufm/omp*, *ufm/opt*

### 4.310.1 Syntax

```
pair_style ufm cutoff
```

- cutoff = global cutoff for *ufm* interactions (distance units)

### 4.310.2 Examples

```
pair_style ufm 4.0
pair_coeff 1 1 100.0 1.0 2.5
pair_coeff * * 100.0 1.0

pair_style ufm 4.0
pair_coeff * * 10.0 1.0
variable prefactor equal ramp(10,100)
fix 1 all adapt 1 pair ufm epsilon * * v_prefactor
```

### 4.310.3 Description

Style *ufm* computes pairwise interactions using the Uhlenbeck-Ford model (UFM) potential *(Paula Leite2016)* which is given by

$$E = -\varepsilon \ln\left[1 - \exp\left(-r^2/\sigma^2\right)\right] \qquad r < r_c$$
$$\varepsilon = p\, k_B\, T$$

where $r_c$ is the cutoff, $\sigma$ is a distance-scale and $\varepsilon$ is an energy-scale, i.e., a product of Boltzmann constant $k_B$, temperature $T$ and the Uhlenbeck-Ford p-parameter which is responsible to control the softness of the interactions *(Paula Leite2017)*. This model is useful as a reference system for fluid-phase free-energy calculations *(Paula Leite2016)*.

The following coefficients must be defined for each pair of atom types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- $\varepsilon$ (energy units)
- $\sigma$ (distance units)
- cutoff (distance units)

The last coefficient is optional. If not specified, the global *ufm* cutoff is used.

The *fix adapt* command can be used to vary epsilon and sigma for this pair style over the course of a simulation, in which case pair_coeff settings for epsilon and sigma must still be specified, but will be overridden. For example these commands will vary the prefactor epsilon for all pairwise interactions from 10.0 at the beginning to 100.0 at the end of a run:

```
variable prefactor equal ramp(10,100)
fix 1 all adapt 1 pair ufm epsilon * * v_prefactor
```

**Note**

The thermodynamic integration procedure can be performed with this potential using *fix adapt*. This command will rescale the force on each atom by varying a scale variable, which always starts with value 1.0. The syntax is the same described above, however, changing epsilon to scale. A detailed explanation of how to use this command and perform nonequilibrium thermodynamic integration in LAMMPS is given in the paper by *(Freitas)*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.310.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon and sigma coefficients and cutoff distance for this pair style can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

This pair style support the *pair_modify* shift option for the energy of the pair interaction.

The *pair_modify* table and tail are not relevant for this pair style.

This pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.310.5 Restrictions

This pair style is part of the EXTRA-PAIR package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 4.310.6 Related commands

*pair_coeff* , *fix adapt*

### 4.310.7 Default

none

**(Paula Leite2017)** Paula Leite, Santos-Florez, and de Koning, Phys Rev E, 96, 32115 (2017).

**(Paula Leite2016)** Paula Leite , Freitas, Azevedo, and de Koning, J Chem Phys, 126, 044509 (2016).

**(Freitas)** Freitas, Asta, and de Koning, Computational Materials Science, 112, 333 (2016).

## 4.311 pair_style vashishta command

Accelerator Variants: *vashishta/gpu*, *vashishta/omp*, *vashishta/kk*

## 4.312 pair_style vashishta/table command

Accelerator Variants: *vashishta/table/omp*

### 4.312.1 Syntax

```
pair_style style args
```

- style = *vashishta* or *vashishta/table*
- args = list of arguments for a particular style

```
vashishta args = none
vashishta/table args = Ntable cutinner
  Ntable = # of tabulation points
  cutinner = tablulate from cutinner to cutoff
```

### 4.312.2 Examples

```
pair_style vashishta
pair_coeff * * SiC.vashishta Si C

pair_style vashishta/table 100000 0.2
pair_coeff * * SiC.vashishta Si C
```

### 4.312.3 Description

The *vashishta* and *vashishta/table* styles compute the combined 2-body and 3-body family of potentials developed in the group of Priya Vashishta and collaborators. By combining repulsive, screened Coulombic, screened charge-dipole, and dispersion interactions with a bond-angle energy based on the Stillinger-Weber potential, this potential has been used to describe a variety of inorganic compounds, including SiO2 *Vashishta1990*, SiC *Vashishta2007*, and InP *Branicio2009*.

The potential for the energy U of a system of atoms is

$$U = \sum_{i}^{N}\sum_{j>i}^{N} U_{ij}^{(2)}(r_{ij}) + \sum_{i}^{N}\sum_{j\neq i}^{N}\sum_{k>j,k\neq i}^{N} U_{ijk}^{(3)}(r_{ij}, r_{ik}, \theta_{ijk})$$

$$U_{ij}^{(2)}(r) = \frac{H_{ij}}{r^{\eta_{ij}}} + \frac{Z_i Z_j}{r}\exp(-r/\lambda_{1,ij}) - \frac{D_{ij}}{r^4}\exp(-r/\lambda_{4,ij}) - \frac{W_{ij}}{r^6}, r < r_{c,ij}$$

$$U_{ijk}^{(3)}(r_{ij}, r_{ik}, \theta_{ijk}) = B_{ijk}\frac{\left[\cos\theta_{ijk} - \cos\theta_{0ijk}\right]^2}{1 + C_{ijk}\left[\cos\theta_{ijk} - \cos\theta_{0ijk}\right]^2} \times \exp\left(\frac{\gamma_{ij}}{r_{ij} - r_{0,ij}}\right)\exp\left(\frac{\gamma_{ik}}{r_{ik} - r_{0,ik}}\right), r_{ij} < r_{0,ij}, r_{ik} < r_{0,ik}$$

where we follow the notation used in *Branicio2009*. $U^2$ is a two-body term and U3 is a three-body term. The summation over two-body terms is over all neighbors J within a cutoff distance = $r_c$. The twobody terms are shifted and tilted by a linear function so that the energy and force are both zero at $r_c$. The summation over three-body terms is over all neighbors $i$ and $k$ within a cut-off distance = $r_0$, where the exponential screening function becomes zero.

The *vashishta* style computes these formulas analytically. The *vashishta/table* style tabulates the analytic values for *Ntable* points from cutinner to the cutoff of the potential. The points are equally spaced in R^2 space from cutinner^2 to cutoff^2. For the two-body term in the above equation, a linear interpolation for each pairwise distance between adjacent points in the table. In practice the tabulated version can run 3-5x faster than the analytic version with moderate to little loss of accuracy for Ntable values between 10000 and 1000000. It is not recommended to use less than 5000 tabulation points.

Only a single pair_coeff command is used with either style which specifies a Vashishta potential file with parameters for all needed elements. These are mapped to LAMMPS atom types by specifying N additional arguments after the filename in the pair_coeff command, where N is the number of LAMMPS atom types:

- filename
- N element names = mapping of Vashishta elements to atom types

See the *pair_coeff* page for alternate ways to specify the path for the potential file.

As an example, imagine a file SiC.vashishta has parameters for Si and C. If your LAMMPS simulation has 4 atoms types and you want the first 3 to be Si, and the fourth to be C, you would use the following pair_coeff command:

```
pair_coeff * * SiC.vashishta Si Si Si C
```

The first 2 arguments must be * * so as to span all LAMMPS atom types. The first three Si arguments map LAMMPS atom types 1,2,3 to the Si element in the file. The final C argument maps LAMMPS atom type 4 to the C element in the file. If a mapping value is specified as NULL, the mapping is not performed. This can be used when a *vashishta*

potential is used as part of the *hybrid* pair style. The NULL values are placeholders for atom types that will be used with other potentials.

Vashishta files in the *potentials* directory of the LAMMPS distribution have a ".vashishta" suffix. Lines that are not blank or comments (starting with #) define parameters for a triplet of elements. The parameters in a single entry correspond to the two-body and three-body coefficients in the formulae above:

- element 1 (the center atom in a 3-body interaction)
- element 2
- element 3
- $H$ (energy units)
- $\eta$
- $Z_i$ (electron charge units)
- $Z_j$ (electron charge units)
- $\lambda_1$ (distance units)
- $D$ (energy units)
- $\lambda_4$ (distance units)
- $W$ (energy units)
- $r_c$ (distance units)
- $B$ (energy units)
- $\gamma$
- $r_0$ (distance units)
- $C$
- $\cos \theta_0$

The non-annotated parameters are unitless. The Vashishta potential file must contain entries for all the elements listed in the pair_coeff command. It can also contain entries for additional elements not being used in a particular simulation; LAMMPS ignores those entries. For a single-element simulation, only a single entry is required (e.g. SiSiSi). For a two-element simulation, the file must contain 8 entries (for SiSiSi, SiSiC, SiCSi, SiCC, CSiSi, CSiC, CCSi, CCC), that specify parameters for all permutations of the two elements interacting in three-body configurations. Thus for 3 elements, 27 entries would be required, etc.

Depending on the particular version of the Vashishta potential, the values of these parameters may be keyed to the identities of zero, one, two, or three elements. In order to make the input file format unambiguous, general, and simple to code, LAMMPS uses a slightly confusing method for specifying parameters. All parameters are divided into two classes: two-body and three-body. Two-body and three-body parameters are handled differently, as described below. The two-body parameters are $H$, $\eta$, $\lambda_1$, $D$, $\lambda_4$, $W$, $r_c$, $\gamma$, and $r_0$. They appear in the above formulae with two subscripts. The parameters $Z_i$ and $Z_j$ are also classified as two-body parameters, even though they only have 1 subscript. The three-body parameters are $B$, $C$, $\cos \theta_0$. They appear in the above formulae with three subscripts. Two-body and three-body parameters are handled differently, as described below.

The first element in each entry is the center atom in a three-body interaction, while the second and third elements are two neighbor atoms. Three-body parameters for a central atom I and two neighbors J and K are taken from the IJK entry. Note that even though three-body parameters do not depend on the order of J and K, LAMMPS stores three-body parameters for both IJK and IKJ. The user must ensure that these values are equal. Two-body parameters for an atom I interacting with atom J are taken from the IJJ entry, where the second and third elements are the same. Thus the two-body parameters for Si interacting with C come from the SiCC entry. Note that even though two-body parameters (except possibly gamma and r0 in U3) do not depend on the order of the two elements, LAMMPS will get the Si-C

value from the SiCC entry and the C-Si value from the CSiSi entry. The user must ensure that these values are equal. Two-body parameters appearing in entries where the second and third elements are different are stored but never used. It is good practice to enter zero for these values. Note that the three-body function U3 above contains the two-body parameters $\gamma$ and $r_0$. So U3 for a central C atom bonded to an Si atom and a second C atom will take three-body parameters from the CSiC entry, but two-body parameters from the CCC and CSiSi entries.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.312.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, where types I and J correspond to two different element types, mixing is performed by LAMMPS as described above from values in the potential file.

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style does not write its information to *binary restart files*, since it is stored in potential files. Thus, you need to re-specify the pair_style and pair_coeff commands in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.312.5 Restrictions

These pair styles are part of the MANYBODY package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

These pair styles requires the *newton* setting to be "on" for pair interactions.

The Vashishta potential files provided with LAMMPS (see the potentials directory) are parameterized for metal *units*. You can use the Vashishta potential with any LAMMPS units, but you would need to create your own potential file with coefficients listed in the appropriate units if your simulation does not use "metal" units.

### 4.312.6 Related commands

*pair_coeff*

### 4.312.7 Default

none

---

**(Vashishta1990)** P. Vashishta, R. K. Kalia, J. P. Rino, Phys. Rev. B 41, 12197 (1990).

**(Vashishta2007)** P. Vashishta, R. K. Kalia, A. Nakano, J. P. Rino. J. Appl. Phys. 101, 103515 (2007).

**(Branicio2009)** Branicio, Rino, Gan and Tsuzuki, J. Phys Condensed Matter 21 (2009) 095002

## 4.313 pair_style wf/cut command

Accelerator Variants: *wf/cut/omp*, *wf/cut/kk*

### 4.313.1 Syntax

```
pair_style wf/cut cutoff
```

- cutoff = cutoff for wf interactions (distance units)

### 4.313.2 Examples

```
pair_style        wf/cut 2.0
pair_coeff        1 1 1.0 1.0 1 1  2.0
```

### 4.313.3 Description

The *wf/cut* (Wang-Frenkel) style computes LJ-like potentials as described in *Wang2020*. This potential is by construction finite ranged and it vanishes quadratically at the cutoff distance, avoiding truncation, shifting, interpolation and other typical procedures with the LJ potential. The *wf/cut* can be used when a typical short-ranged potential with attraction is required. The potential is given by which is given by:

$$\phi(r) = \varepsilon\alpha \left( \left[\frac{\sigma}{r}\right]^{2\mu} - 1 \right) \left( \left[\frac{r_c}{r}\right]^{2\mu} - 1 \right)^{2\nu}$$

with

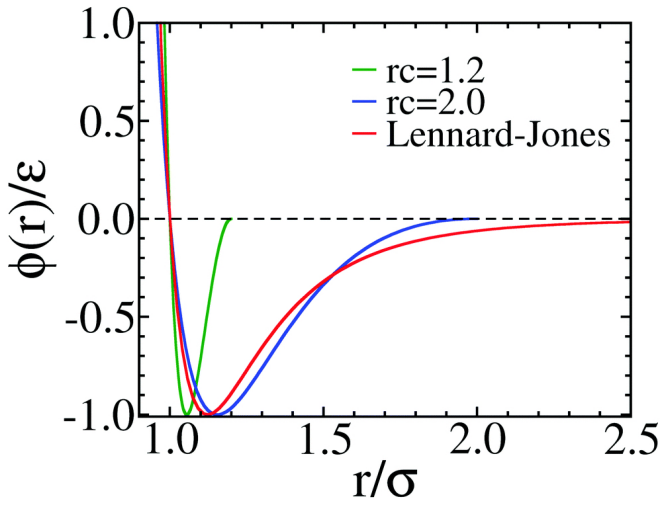
$$\alpha = 2\nu \left(\frac{r_c}{\sigma}\right)^{2\mu} \left[ \frac{1+2\nu}{2\nu\left[(r_c/\sigma)^{2\mu} - 1\right]} \right]^{2\nu+1}$$

and

$$r_{min} = r_c \left[ \frac{1+2\nu}{1+2\nu(r_c/\sigma)^{2\nu}} \right]^{1/2\mu}$$

$r_c$ is the cutoff.

Comparison of the non-truncated Lennard-Jones 12-6 potential (red curve), and the WF potentials with $\mu = 1$ and $\nu = 1$ are shown in the figure below. The blue curve has $r_c = 2.0$ and the green curve has $r_c = 1.2$ and can be used to describe colloidal interactions.



The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $\varepsilon$ (energy units)
- $\sigma$ (distance units)
- $\nu$
- $\mu$
- $r_c$ (distance units)

The last coefficient is optional. If not specified, the global cutoff given in the pair_style command is used. The exponents $\nu$ and $\mu$ are positive integers, usually set to 1. There is usually little to be gained by choosing other values of $\nu$ and $\mu$ (See discussion in *Wang2020*)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

**Mixing, shift, table, tail correction, restart, rRESPA info**:

This pair style does not support the *pair_modify* mixing and table options.

The *pair_modify* tail and shift options are not relevant for this pair style as it goes to zero at the cut-off radius.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style does not support the use of the *inner*, *middle*, and *outer* keywords of the *run_style respa* command.

---

### 4.313.4 Restrictions

This pair style can only be used if LAMMPS was built with the EXTRA-PAIR package. See the *Build package* doc page for more info.

### 4.313.5 Related commands

*pair_coeff*

**Default:** none

---

**(Wang2020)** X. Wang, S. Ramirez-Hinestrosa, J. Dobnikar, and D. Frenkel, Phys. Chem. Chem. Phys. 22, 10624 (2020).

# 4.314 pair_style ylz command

Accelerator Variants: *ylz/omp*, *ylz/kk*

## 4.314.1 Syntax

```
pair_style ylz cutoff
```

- cutoff = global cutoff for interactions (distance units)

## 4.314.2 Examples

```
pair_style   ylz  2.6
pair_coeff   *  *  1.0  1.0  4  3  0.0  2.6
```

## 4.314.3 Description

Added in version 3Nov2022.

The *ylz* (Yuan-Li-Zhang) style computes an anisotropic interaction between pairs of coarse-grained particles considering the relative particle orientations. This potential was originally developed as a particle-based solvent-free model for biological membranes *(Yuan2010a)*. Unlike *pair_style gayberne*, whose orientation dependence is strictly derived from the closest distance between two ellipsoidal rigid bodies, the orientation-dependence of this pair style is mathematically defined such that the particles can self-assemble into one-particle-thick fluid membranes. The potential of this pair style is described by:

$$U(\mathbf{r}_{ij}, \mathbf{n}_i, \mathbf{n}_j) = \begin{cases} u_R(r) + \left[1 - \phi(\hat{\mathbf{r}}_{ij}, \mathbf{n}_i, \mathbf{n}_j)\right] \varepsilon, & r < r_{min} \\ u_A(r)\phi(\hat{\mathbf{r}}_{ij}, \mathbf{n}_i, \mathbf{n}_j), & r_{min} < r < r_c \end{cases}$$

$$\phi(\hat{\mathbf{r}}_{ij}, \mathbf{n}_i, \mathbf{n}_j) = 1 + \left[\mu(a(\hat{\mathbf{r}}_{ij}, \mathbf{n}_i, \mathbf{n}_j) - 1)\right]$$

$$a(\hat{\mathbf{r}}_{ij}, \mathbf{n}_i, \mathbf{n}_j) = (\mathbf{n}_i \times \hat{\mathbf{r}}_{ij}) \cdot (\mathbf{n}_j \times \hat{\mathbf{r}}_{ij}) + \beta(\mathbf{n}_i - \mathbf{n}_j) \cdot \hat{\mathbf{r}}_{ij} - \beta^2$$

$$u_R(r) = \varepsilon \left[ \left(\frac{r_{min}}{r}\right)^4 - 2\left(\frac{r_{min}}{r}\right)^2 \right]$$

$$u_A(r) = -\varepsilon \, cos^{2\zeta}\left[\frac{\pi}{2}\frac{(r - r_{min})}{(r_c - r_{min})}\right]$$

where $\mathbf{r}_i$ and $\mathbf{r}_j$ are the center position vectors of particles i and j, respectively, $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ is the inter-particle distance vector, $r = |\mathbf{r}_{ij}|$ and $\hat{\mathbf{r}}_{ij} = \mathbf{r}_{ij}/r$. The unit vectors $\mathbf{n}_i$ and $\mathbf{n}_j$ represent the axes of symmetry of particles i and j, respectively, $u_R$ and $u_A$ are the repulsive and attractive potentials, $\phi$ is an angular function which depends on the relative orientation between pair particles, $\mu$ is the parameter related to the bending rigidity of the membrane, $\beta$ is the parameter related to the spontaneous curvature, and $\varepsilon$ is the energy unit, respectively. The $\zeta$ controls the slope of the attractive branch and hence the diffusivity of the particles in the in-plane direction of the membrane. $r_c$ is the cutoff radius, $r_{min}$ is the distance which minimizes the potential energy $u_A(r)$ and $r_{min} = 2^{1/6}\sigma$, where $\sigma$ is the length unit.

This pair style is suited for solvent-free coarse-grained simulations of biological systems involving lipid bilayer membranes, such as vesicle shape transformations *(Yuan2010b)*, nanoparticle endocytosis *(Huang)*, modeling of red blood cell membranes *(Fu)*, *(Appshaw)*, and modeling of cell elasticity *(Becton)*.

Use of this pair style requires the NVE, NVT, or NPT fixes with the *asphere* extension (e.g. *fix nve/asphere*) in order to integrate particle rotation. Additionally, *atom_style ellipsoid* should be used since it defines the rotational state of each particle.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- $\varepsilon$ = well depth (energy units)
- $\sigma$ = minimum effective particle radii (distance units)
- $\zeta$ = tuning parameter for the slope of the attractive branch
- $\mu$ = parameter related to bending rigidity
- $\beta$ = parameter related to the spontaneous curvature
- cutoff (distance units)

The last coefficient is optional. If not specified, the global cutoff specified in the pair_style command is used.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.314.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the epsilon and sigma coefficients and cutoff distance for this pair style can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

The *pair_modify* table option is not relevant for this pair style.

This pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.314.5 Restrictions

The *ylz* style is part of the ASPHERE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires that atoms store torque and a quaternion to represent their orientation, as defined by the *atom_style*. It also requires they store a per-atom *shape*. The particles cannot store a per-particle diameter. To avoid being mistakenly considered as point particles, the shape parameters ought to be non-spherical, like [1 0.99 0.99]. Unlike the *resquared* pair style for which the shape directly determines the mathematical expressions of the potential, the shape parameters for this pair style is only involved in the computation of the moment of inertia and thus only influences the rotational dynamics of individual particles.

This pair style requires that **all** atoms are ellipsoids as defined by the *atom_style ellipsoid* command.

### 4.314.6 Related commands

*pair_coeff*, *fix nve/asphere*, *compute temp/asphere*, *pair_style resquared*, *pair_style gayberne*

### 4.314.7 Default

none

---

**(Yuan2010a)** Yuan, Huang, Li, Lykotrafitis, Zhang, Phys. Rev. E, 82, 011905(2010).

**(Yuan2010b)** Yuan, Huang, Zhang, Soft. Matter, 6, 4571(2010).

**(Huang)** Huang, Zhang, Yuan, Gao, Zhang, Nano Lett. 13, 4546(2013).

**(Fu)** Fu, Peng, Yuan, Kfoury, Young, Comput. Phys. Commun, 210, 193-203(2017).

**(Appshaw)** Appshaw, Seddon, Hanna, Soft. Matter,18, 1747(2022).

**(Becton)** Becton, Averett, Wang, Biomech. Model. Mechanobiology, 18, 425-433(2019).

## 4.315 pair_style yukawa command

Accelerator Variants: *yukawa/gpu*, *yukawa/omp*, *yukawa/kk*

### 4.315.1 Syntax

```
pair_style yukawa kappa cutoff
```

- kappa = screening length (inverse distance units)
- cutoff = global cutoff for Yukawa interactions (distance units)

### 4.315.2 Examples

```
pair_style yukawa 2.0 2.5
pair_coeff 1 1 100.0 2.3
pair_coeff * * 100.0
```

### 4.315.3 Description

Style *yukawa* computes pairwise interactions with the formula

$$E = A\frac{e^{-\kappa r}}{r} \qquad r < r_c$$

$r_c$ is the cutoff.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- A (energy*distance units)
- cutoff (distance units)

The last coefficient is optional. If not specified, the global yukawa cutoff is used.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.315.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the A coefficient and cutoff distance for this pair style can be mixed. A is an energy value mixed like a LJ epsilon. The default mix value is *geometric*. See the "pair_modify" command for details.

This pair style supports the *pair_modify* shift option for the energy of the pair interaction.

The *pair_modify* table option is not relevant for this pair style.

This pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.315.5 Restrictions

none

### 4.315.6 Related commands

*pair_coeff*

### 4.315.7 Default

none

## 4.316 pair_style yukawa/colloid command

Accelerator Variants: *yukawa/colloid/gpu*, *yukawa/colloid/kk*, *yukawa/colloid/omp*

### 4.316.1 Syntax

```
pair_style yukawa/colloid kappa cutoff
```

- kappa = screening length (inverse distance units)
- cutoff = global cutoff for colloidal Yukawa interactions (distance units)

### 4.316.2 Examples

```
pair_style yukawa/colloid 2.0 2.5
pair_coeff 1 1 100.0 2.3
pair_coeff * * 100.0
```

### 4.316.3 Description

Style *yukawa/colloid* computes pairwise interactions with the formula

$$E = \frac{A}{\kappa} e^{-\kappa(r-(r_i+r_j))} \qquad r < r_c$$

where $r_i$ and $r_j$ are the radii of the two particles and $r_c$ is the cutoff.

In contrast to *pair_style yukawa*, this functional form arises from the Coulombic interaction between two colloid particles, screened due to the presence of an electrolyte, see the book by *Safran* for a derivation in the context of DLVO theory. *Pair_style yukawa* is a screened Coulombic potential between two point-charges and uses no such approximation.

This potential applies to nearby particle pairs for which the Derjagin approximation holds, meaning $h << r_i + r_j$, where $h$ is the surface-to-surface separation of the two particles.

When used in combination with *pair_style colloid*, the two terms become the so-called DLVO potential, which combines electrostatic repulsion and van der Waals attraction.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- A (energy/distance units)
- cutoff (distance units)

The prefactor A is determined from the relationship between surface charge and surface potential due to the presence of electrolyte. Note that the A for this potential style has different units than the A used in *pair_style yukawa*. For low surface potentials, i.e. less than about 25 mV, A can be written as:

$$A = 2\pi R \varepsilon \varepsilon_0 \kappa \psi^2$$

where

- $R$ = colloid radius (distance units)
- $\varepsilon_0$ = permittivity of free space (charge^2/energy/distance units)
- $\varepsilon$ = relative permittivity of fluid medium (dimensionless)
- $\kappa$ = inverse screening length (1/distance units)
- $\psi$ = surface potential (energy/charge units)

The last coefficient is optional. If not specified, the global yukawa/colloid cutoff is used.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.316.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs I,J and I != J, the A coefficient and cutoff distance for this pair style can be mixed. A is an energy value mixed like a LJ epsilon. The default mix value is *geometric*. See the "pair_modify" command for details.

This pair style supports the *pair_modify* shift option for the energy of the pair interaction.

The *pair_modify* table option is not relevant for this pair style.

This pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

---

### 4.316.5 Restrictions

This style is part of the COLLOID package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This pair style requires that atoms be finite-size spheres with a diameter, as defined by the *atom_style sphere* command.

Per-particle polydispersity is not yet supported by this pair style; per-type polydispersity is allowed. This means all particles of the same type must have the same diameter. Each type can have a different diameter.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 4.316.6 Related commands

*pair_coeff*

### 4.316.7 Default

none

---

**(Safran)** Safran, Statistical Thermodynamics of Surfaces, Interfaces, And Membranes, Westview Press, ISBN: 978-0813340791 (2003).

# 4.317 pair_style zbl command

Accelerator Variants: *zbl/gpu*, *zbl/kk*, *zbl/omp*

## 4.317.1 Syntax

```
pair_style zbl inner outer
```

- inner = distance where switching function begins
- outer = global cutoff for ZBL interaction

## 4.317.2 Examples

```
pair_style zbl 3.0 4.0
pair_coeff * * 73.0 73.0
pair_coeff 1 1 14.0 14.0
```

## 4.317.3 Description

Style *zbl* computes the Ziegler-Biersack-Littmark (ZBL) screened nuclear repulsion for describing high-energy collisions between atoms. *(Ziegler)*. It includes an additional switching function that ramps the energy, force, and curvature smoothly to zero between an inner and outer cutoff. The potential energy due to a pair of atoms at a distance r_ij is given by:

$$E_{ij}^{ZBL} = \frac{1}{4\pi\varepsilon_0}\frac{Z_i Z_j\, e^2}{r_{ij}}\phi(r_{ij}/a) + S(r_{ij})$$

$$a = \frac{0.46850}{Z_i^{0.23} + Z_j^{0.23}}$$

$$\phi(x) = 0.18175e^{-3.19980x} + 0.50986e^{-0.94229x} + 0.28022e^{-0.40290x} + 0.02817e^{-0.20162x}$$

where $e$ is the electron charge, $\varepsilon_0$ is the electrical permittivity of vacuum, and $Z_i$ and $Z_j$ are the nuclear charges of the two atoms. The switching function $S(r)$ is identical to that used by *pair_style lj/gromacs*. Here, the inner and outer cutoff are the same for all pairs of atom types.

The following coefficients must be defined for each pair of atom types via the *pair_coeff* command as in the examples above, or in the LAMMPS data file.

- $Z_i$ (atomic number for first atom type, e.g. 13.0 for aluminum)
- $Z_j$ (ditto for second atom type)

The values of $Z_i$ and $Z_j$ are normally equal to the atomic numbers of the two atom types. Thus, the user may optionally specify only the coefficients for each $i == j$ pair, and rely on the obvious mixing rule for cross interactions (see below). Note that when $i == j$ it is required that $Z_i == Z_j$. When used with *hybrid/overlay* and pairs are assigned to more than one sub-style, the mixing rule is not used and each pair of types interacting with the ZBL sub-style must be included in a pair_coeff command.

**Note**

The numerical values of the exponential decay constants in the screening function depend on the unit of distance. In the above equation they are given for units of Angstroms. LAMMPS will automatically convert these values to the distance unit of the specified LAMMPS *units* setting. The values of Z should always be given as multiples of a proton's charge, e.g. 29.0 for copper.

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 4.317.4 Mixing, shift, table, tail correction, restart, rRESPA info

For atom type pairs *i*,*j* and $i \neq j$, the $Z_i$ and $Z_j$ coefficients can be mixed by taking $Z_a$ and $Z_b$ from the values specified for the two cases where $i == j$ and thus $Z_a = Z_i == Z_j$ and $Z_b = Z_i == Z_j$ for different elements. When used with *hybrid/overlay* and pairs are assigned to more than one sub-style, the mixing rule is not used and each pair of types interacting with the ZBL sub-style must be included in a pair_coeff command. The *pair_modify* mix option has no effect on the mixing behavior

The ZBL pair style does not support the *pair_modify* shift option, since the ZBL interaction is already smoothed to 0.0 at the cutoff.

The *pair_modify* table option is not relevant for this pair style.

This pair style does not support the *pair_modify* tail option for adding long-range tail corrections to energy and pressure, since there are no corrections for a potential that goes to 0.0 at the cutoff.

This pair style does not write information to *binary restart files*, so pair_style and pair_coeff commands must be specified in an input script that reads a restart file.

This pair style can only be used via the *pair* keyword of the *run_style respa* command. It does not support the *inner*, *middle*, *outer* keywords.

### 4.317.5 Restrictions

none

### 4.317.6 Related commands

*pair_coeff*

### 4.317.7 Default

none

**(Ziegler)** J.F. Ziegler, J. P. Biersack and U. Littmark, "The Stopping and Range of Ions in Matter", Volume 1, Pergamon, 1985.

# 4.318 pair_style zero command

## 4.318.1 Syntax

```
pair_style zero cutoff [nocoeff] [full]
```

- zero = style name of this pair style
- cutoff = global cutoff (distance units)
- nocoeff = ignore all pair_coeff parameters (optional)
- full = build full neighbor list (optional)

## 4.318.2 Examples

```
pair_style zero 10.0
pair_style zero 5.0 nocoeff
pair_coeff * *
pair_coeff 1 2*4 3.0
```

## 4.318.3 Description

Define a global or per-type cutoff length for the purpose of building a neighbor list and acquiring ghost atoms, but do not compute any pairwise forces or energies.

This can be useful for fixes or computes which require a neighbor list to enumerate pairs of atoms within some cutoff distance, but when pairwise forces are not otherwise needed. Examples are the *fix bond/create*, *compute rdf*, *compute voronoi/atom* commands.

Note that the *comm_modify cutoff* command can be used to ensure communication of ghost atoms even when a pair style is not defined, but it will not trigger neighbor list generation.

The optional *nocoeff* flag allows to read data files with a PairCoeff section for any pair style. Similarly, any pair_coeff commands will only be checked for the atom type numbers and the rest ignored. In this case, only the global cutoff will be used.

Added in version 3Nov2022.

The optional *full* flag builds a full neighbor list instead of the default half neighbor list.

The following coefficients must be defined for each pair of atoms types via the *pair_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands, or by mixing as described below:

- cutoff (distance units)

This coefficient is optional. If not specified, the global cutoff specified in the pair_style command is used. If the pair_style has been specified with the optional *nocoeff* flag, then a cutoff pair coefficient is ignored.

---

## 4.318.4 Mixing, shift, table, tail correction, restart, rRESPA info

The cutoff distance for this pair style can be mixed. The default mix value is *geometric*. See the "pair_modify" command for details.

This pair style does not support the *pair_modify* shift, table, and tail options.

This pair style writes its information to *binary restart files*, so pair_style and pair_coeff commands do not need to be specified in an input script that reads a restart file.

This pair style supports the use of the *inner*, *middle*, and *outer* keywords of the *run_style respa* command.

---

### 4.318.5 Restrictions

none

### 4.318.6 Related commands

*pair_style none*

### 4.318.7 Default

none

CHAPTER

FIVE

# BOND STYLES

## 5.1 bond_style bpm/rotational command

### 5.1.1 Syntax

```
bond_style bpm/rotational keyword value attribute1 attribute2 ...
```

- optional keyword = *store/local* or *overlay/pair* or *smooth* or *normalize* or *break* or *frame* or *damping*

```
store/local values = fix_ID N attributes ...
   * fix_ID = ID of associated internal fix to store data
   * N = prepare data for output every this many timesteps
   * attributes = zero or more of the below attributes may be appended

     id1, id2 = IDs of two atoms in the bond
     time = the timestep the bond broke
     x, y, z = the center of mass position of the two atoms when the bond broke␣
↪(distance units)
     x/ref, y/ref, z/ref = the initial center of mass position of the two atoms␣
↪(distance units)

overlay/pair value = yes or no
   bonded particles will still interact with pair forces

smooth value = yes or no
   smooths bond forces near the breaking point

normalize value = yes or no
   normalizes normal and shear forces by the reference length

break value = yes or no
   indicates whether bonds break during a run

frame value = average or particle
   what frame is used for calculating the relative displacement and rotation

damping value = derivative or dem
   damping construction
```

### 5.1.2 Examples

```
bond_style bpm/rotational
bond_coeff 1 1.0 0.2 0.02 0.02 0.20 0.04 0.04 0.04 0.1 0.02 0.002 0.002

bond_style bpm/rotational frame particle damping derivative
bond_coeff 1 1.0 0.2 0.02 0.02 0.20 0.04 0.04 0.04 0.1 0.02 0.002 0.002

bond_style bpm/rotational store/local myfix 1000 time id1 id2
dump 1 all local 1000 dump.broken f_myfix[1] f_myfix[2] f_myfix[3]
dump_modify 1 write_header no
```

### 5.1.3 Description

Added in version 4May2022.

The *bpm/rotational* bond style computes forces and torques based on deviations from the initial reference state of the two atoms. The reference state is stored by each bond when it is first computed in the setup of a run. Data is then preserved across run commands and is written to *binary restart files* such that restarting the system will not reset the reference state of a bond.

Forces include a normal and tangential component. The base normal force has a magnitude of

$$f_{\text{radial}} = k_{\text{radial}}(r - r_0)$$

where $k_{\text{radial}}$ is a stiffness and $r$ is the current distance and $r_0$ is the initial distance between the two particles.

A tangential force, proportional to the tangential shear displacement with a stiffness of $k_{\text{shear}}$, is applied perpendicular to the normal direction. This tangential force also induces a torque. In addition, bending and twisting torques, proportional to angular bending and twisting displacements with stiffnesses of $k_{\text{bend}}$ and $k_{\text{twist}}$, respectively, are also applied to particles. Details on the calculations of shear and angular displacements can be found in *(Wang)*, *(Wang and Mora)*, and/or *(Alkuino et al)* depending on the *frame* (discussed below).

Bonds will break under sufficient stress. A breaking criterion is calculated

$$B = \max\left\{0, \frac{f_r}{f_{r,c}} + \frac{|f_s|}{f_{s,c}} + \frac{|\tau_t|}{\tau_{t,c}} + \frac{|\tau_b|}{\tau_{b,c}}\right\}$$

where $|f_s|$ is the magnitude of the shear force and $|\tau_t|$ and $|\tau_b|$ are the magnitudes of the twisting and bending torques, respectively, and $r$, $s$, $t$ and $b$ are shorthand for radial, shear, twist, and bend. The corresponding variables $f_{r,c}$, $f_{s,c}$, $\tau_{t,c}$, and $\tau_{b,c}$ are critical limits to each force or torque. If $B$ ever equals or exceeds one, the bond will break. This is done by setting the bond type to 0 such that forces and torques are no longer computed.

**Note**

The breaking criterion uses non-damped forces and torques for *frame average* and damped forces and torques for *frame particle* to maintain backwards compatibility with previous versions of this bond style.

After computing the base magnitudes of the forces and torques, they can be optionally multiplied by an extra factor $w$ to smoothly interpolate forces and torques to zero as the bond breaks. This term is calculated as $w = (1.0 - B^4)$. This smoothing factor can be added or removed by setting the *smooth* keyword to *yes* or *no*, respectively.

Finally, additional damping forces and torques are applied to the two particles. A force is applied proportional to the difference in the normal velocity of particles using a similar construction to that of dissipative particle dynamics *(Groot and Warren)*:

$$F_D = -\gamma_{\text{radial}} w(\hat{r} \bullet \vec{v})$$

where $\gamma_{\text{radial}}$ is the damping strength, $\hat{r}$ is the radial normal vector, and $\vec{v}$ is the velocity difference between the two particles. Similarly, additional damping forces/torques are applied to other modes. These details depend on the *damping* setting.

Added in version 28Mar2023.

If the *break* keyword is set to *no*, LAMMPS assumes bonds should not break during a simulation run. This will prevent some unnecessary calculation. The recommended bond communication distance no longer depends on bond failure coefficients (which are ignored) but instead corresponds to the typical heuristic maximum strain used by typical non-bpm bond styles. Similar behavior to *break no* can also be attained by setting arbitrarily high values for all four failure coefficients. One cannot use *break no* with *smooth yes*.

Added in version 4Jul2026.

For *damping* style *derivative* (the default), additional forces/torques are applied on shear, twisting, and bending modes. These are simply proportional to the rate of change of the shear, bend, and twist angles, respectively, with prefactors of $\gamma_{\text{shear}}$, $\gamma_{\text{twist}}$, and $\gamma_{\text{bend}}$. Details are described in *(Alkuino et al)*.

For the *dem* style, forces are applied to each atom proportional to the relative differences in sliding velocities with a constant prefactor $\gamma_{\text{slide}}$ *(Wang et al, 2015)* along with the associated torques. The twisting and rolling components of the relative angular velocities of the two atoms are also damped by applying torques with prefactors of $\gamma_{\text{twist}}$ and $\gamma_{\text{roll}}$, respectively. These modes are commonly used in the discrete element method (DEM) as in *pair granular*.

The following coefficients must be defined for each bond type via the *bond_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $k_{\text{radial}}$ (force/distance units)
- $k_{\text{shear}}$ (force/distance units)
- $k_{\text{twist}}$ (force*distance/radians units)
- $k_{\text{bend}}$ (force*distance/radians units)
- $f_{\text{radial},c}$ (force units)
- $f_{\text{shear},c}$ (force units)
- $\tau_{\text{twist},c}$ (force*distance units)
- $\tau_{\text{bend},c}$ (force*distance units)
- $\gamma_{\text{radial}}$ (force/velocity units)
- $\gamma_{\text{shear/slide}}$ (force/velocity units)
- $\gamma_{\text{twist}}$ (force*distance/velocity units)
- $\gamma_{\text{bend/roll}}$ (force*distance/velocity units)

If the *normalize* keyword is set to *yes*, the radial and shear forces will be normalized by $r_0$ such that $k_r$ and $k_s$ must be given in force units.

By default, pair forces are not calculated between bonded particles. Pair forces can alternatively be overlaid on top of bond forces by setting the *overlay/pair* keyword to *yes*. This keyword is only necessary if bonds can break and requires specific *special_bonds* settings described in the restrictions. Further details can be found in the *how to* page on BPMs.

Added in version 4Jul2026.

The *frame* setting determines the reference used to calculate the relative displacement and rotation. The *particle* option uses the frame of one particle as described in *(Wang)* and *(Wang and Mora)*. This determination is based on particle ID in LAMMPS. The *average* option (the default) defines a central frame across the two particles as described in *(Alkuino et al)*. The latter option implies forces do not depend on particle IDs and can be more stable, particularly in simulations of thin or highly distorted structures such as the wire example in /examples/bpm.

> **Note**
>
> The previous implementation (up to LAMMPS version 30Mar2026) can be recovered by setting *frame* to *particle* and *damping* to *dem*, and swapping the third and fourth damping factors.

If the *store/local* keyword is used, an internal fix will track bonds that break during the simulation. Whenever a bond breaks, data is processed and transferred to an internal fix labeled *fix_ID*. This allows the local data to be accessed by other LAMMPS commands. Following this optional keyword, a list of one or more attributes is specified. These include the IDs of the two atoms in the bond. The other attributes for the two atoms include the timestep during which the bond broke and the current/initial center of mass position of the two atoms.

Data is continuously accumulated over intervals of *N* timesteps. At the end of each interval, all of the saved accumulated data is deleted to make room for new data. Individual datum may therefore persist anywhere between *1* and *N* timesteps depending on when they are saved. This data can be accessed using the *fix_ID* and a *dump local* command. To ensure all data is output, the dump frequency should correspond to the same interval of *N* timesteps. A dump frequency of an integer multiple of *N* can be used to regularly output a sample of the accumulated data.

Note that when unbroken bonds are dumped to a file via the *dump local* command, bonds with type 0 (broken bonds) are not included. The *delete_bonds* command can also be used to query the status of broken bonds or permanently delete them, e.g.:

```
delete_bonds all stats
delete_bonds all bond 0 remove
```

### 5.1.4 Restart and other info

This bond style writes the reference state of each bond to *binary restart files*. Loading a restart file will properly resume bonds. However, the reference state is NOT written to data files. Therefore reading a data file will not restore bonds and will cause their reference states to be redefined.

If the *store/local* option is used, an internal fix will calculate a local vector or local array depending on the number of input values. The length of the vector or number of rows in the array is the number of recorded, broken bonds. If a single input is specified, a local vector is produced. If two or more inputs are specified, a local array is produced where the number of columns = the number of inputs. The vector or array can be accessed by any command that uses local values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The vector or array will be floating point values that correspond to the specified attribute.

Any settings with the *store/local* option are not saved to a restart file and must be redefined.

The `single()` function of this bond style returns 0.0 for the energy of a bonded interaction, since energy is not conserved in these dissipative potentials. It also returns only the normal component of the bonded interaction force. However, the single() function also calculates 7 extra bond quantities. The first 4 are data from the reference state of the bond including the initial distance between particles $r_0$ followed by the $x$, $y$, and $z$ components of the initial unit vector pointing to particle I from particle J. The next 3 quantities (5-7) are the $x$, $y$, and $z$ components of the total force, including normal and tangential contributions, acting on particle I.

These extra quantities can be accessed by the *compute bond/local* command, as *b1*, *b2*, ..., *b7*.

### 5.1.5 Restrictions

This bond style is part of the BPM package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

To handle breaking bonds, BPM bond styles have extra requirements for special bonds. If bonds cannot break (*break no*), then one can use any special bond weights. Otherwise, restrictions depend on whether pair forces are overlaid (*pair/overlay yes*). If so, then all weights must be one:

```
special_bonds lj/coul 1 1 1
```

If pair forces are disabled (*pair/overlay no*), the default, then the weights must be

```
special_bonds lj 0 1 1 coul 1 1 1
```

and *newton* must be set to bond off.

The *bpm/rotational* style requires *atom style bpm/sphere*.

### 5.1.6 Related commands

*bond_coeff*, *fix nve/bpm/sphere*

### 5.1.7 Default

The option defaults are *overlay/pair* = *no*, *smooth* = *yes*, *normalize* = *no*, *break* = *yes*, *frame* = *average*, and *damping* = *derivative*

---

**(Wang)** Wang, Acta Geotechnica, 4, p 117-127 (2009).

**(Wang and Mora)** Wang and Mora, Advances in Geocomputing, 119, p 183-228 (2009).

**(Alkuino et al)** Alkuino, Clemmer, Santangelo, and Zhang, arXiv:2603.27279 (2026).

**(Groot and Warren)** Groot and Warren, J Chem Phys, 107, 4423-35 (1997).

**(Wang et al, 2015)** Wang, Y., Alonso-Marroquin, F., & Guo, W. W. (2015). Particuology, 23, 49-55.

## 5.2 bond_style bpm/spring command

### 5.2.1 Syntax

```
bond_style bpm/spring keyword value attribute1 attribute2 ...
```

- optional keyword = *overlay/pair* or *store/local* or *smooth* or *normalize* or *break* or *volume/factor*

```
store/local values = fix_ID N attributes ...
   * fix_ID = ID of associated internal fix to store data
   * N = prepare data for output every this many timesteps
   * attributes = zero or more of the below attributes may be appended

     id1, id2 = IDs of two atoms in the bond
     time = the timestep the bond broke
     x, y, z = the center of mass position of the two atoms when the bond broke␣
↪(distance units)
     x/ref, y/ref, z/ref = the initial center of mass position of the two atoms␣
↪(distance units)

overlay/pair value = yes or no
   bonded particles will still interact with pair forces
```

```
smooth value = yes or no
   smooths bond forces near the breaking point

normalize value = yes or no
   normalizes bond forces by the reference length

break value = yes or no
   indicates whether bonds break during a run

volume/factor value = yes or no
   indicates whether forces include the volumetric contribution
```

### 5.2.2 Examples

```
bond_style bpm/spring
bond_coeff 1 1.0 0.05 0.1

bond_style bpm/spring volume/factor yes
bond_coeff 1 1.0 0.05 0.1 0.5

bond_style bpm/spring myfix 1000 time id1 id2
dump 1 all local 1000 dump.broken f_myfix[1] f_myfix[2] f_myfix[3]
dump_modify 1 write_header no
```

### 5.2.3 Description

Added in version 4May2022.

The *bpm/spring* bond style computes forces based on deviations from the initial reference state of the two atoms. The reference state is stored by each bond when it is first computed in the setup of a run. Data is then preserved across run commands and is written to *binary restart files* such that restarting the system will not reset the reference state of a bond.

This bond style only applies central-body forces which conserve the translational and rotational degrees of freedom of a bonded set of particles based on a model described by Clemmer and Robbins *(Clemmer)*. The force has a magnitude of

$$F = k(r - r_0)w$$

where $k$ is a stiffness, $r$ is the current distance and $r_0$ is the initial distance between the two particles, and $w$ is an optional smoothing factor discussed below. Bonds will break at a strain of $\varepsilon_c$. This is done by setting the bond type to 0 such that forces are no longer computed.

An additional damping force is applied to the bonded particles. This forces is proportional to the difference in the normal velocity of particles using a similar construction as dissipative particle dynamics *(Groot)*:

$$F_D = -\gamma w(\hat{r} \bullet \vec{v})$$

where $\gamma$ is the damping strength, $\hat{r}$ is the radial normal vector, and $\vec{v}$ is the velocity difference between the two particles.

The smoothing factor $w$ can be added or removed by setting the *smooth* keyword to *yes* or *no*, respectively. It is constructed such that forces smoothly go to zero, avoiding discontinuities, as bonds approach the critical strain

$$w = 1.0 - \left(\frac{r - r_0}{r_0 \varepsilon_c}\right)^8.$$

If the *normalize* keyword is set to *yes*, the elastic bond force will be normalized by $r_0$ such that $k$ must be given in force units.

By default, pair forces are not calculated between bonded particles. Pair forces can alternatively be overlaid on top of bond forces by setting the *overlay/pair* keyword to *yes*. This keyword is only necessary if bonds can break and requires specific *special_bonds* settings described in the restrictions. Further details can be found in the *how to* page on BPMs.

Added in version 28Mar2023.

If the *break* keyword is set to *no*, LAMMPS assumes bonds should not break during a simulation run. This will prevent some unnecessary calculation. The recommended bond communication distance no longer depends on the value of $\varepsilon_c$ (which is ignored) but instead corresponds to the typical heuristic maximum strain used by typical non-bpm bond styles. Similar behavior to *break no* can also be attained by setting an arbitrarily high value of $\varepsilon_c$. One cannot use *break no* with *smooth yes*.

Added in version 4Feb2025.

The *volume/factor* keyword toggles whether an additional multibody contribution is added to he force using the formulation in *(Clemmer2)*,

$$\alpha_v \left( \left[ \frac{V_i + V_j}{V_{0,i} + V_{0,j}} \right]^{1/3} - \frac{r_{ij}}{r_{0,ij}} \right)$$

where $\alpha_v$ is a user specified coefficient and $V_i$ and $V_{0,i}$ are estimates of the current and local volume of atom $i$. These volumes are calculated as the sum of current or initial bond lengths cubed. In 2D, the volume is replaced with an area calculated using bond lengths squared and the cube root in the above equation is accordingly replaced with a square root. This approximation assumes bonds are evenly distributed on a spherical surface and neglects constant prefactors which are irrelevant since only the ratio of volumes matters. This term may be used to adjust the Poisson's ratio. See the simulation in the `examples/bpm/poissons_ratio` directory for a demonstration of this effect.

If a bond is broken (or created), $V_{0,i}$ is updated by subtracting (or adding) that bond's contribution.

The following coefficients must be defined for each bond type via the *bond_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $k$ (force/distance units)
- $\varepsilon_c$ (unitless)
- $\gamma$ (force/velocity units)

Additionally, if *volume/factor* is set to *yes*, a fourth coefficient must be provided:

- $a_v$ (force units)

If the *store/local* keyword is used, an internal fix will track bonds that break during the simulation. Whenever a bond breaks, data is processed and transferred to an internal fix labeled *fix_ID*. This allows the local data to be accessed by other LAMMPS commands. Following this optional keyword, a list of one or more attributes is specified. These include the IDs of the two atoms in the bond. The other attributes for the two atoms include the timestep during which the bond broke and the current/initial center of mass position of the two atoms.

Data is continuously accumulated over intervals of $N$ timesteps. At the end of each interval, all of the saved accumulated data is deleted to make room for new data. Individual datum may therefore persist anywhere between *1* to $N$ timesteps depending on when they are saved. This data can be accessed using the *fix_ID* and a *dump local* command. To ensure all data is output, the dump frequency should correspond to the same interval of $N$ timesteps. A dump frequency of an integer multiple of $N$ can be used to regularly output a sample of the accumulated data.

Note that when unbroken bonds are dumped to a file via the *dump local* command, bonds with type 0 (broken bonds) are not included. The *delete_bonds* command can also be used to query the status of broken bonds or permanently delete them, e.g.:

```
delete_bonds all stats
delete_bonds all bond 0 remove
```

---

### 5.2.4 Restart and other info

This bond style writes the reference state of each bond to *binary restart files*. Loading a restart file will properly restore bonds. However, the reference state is NOT written to data files. Therefore reading a data file will not restore bonds and will cause their reference states to be redefined.

If the *store/local* option is used, an internal fix will calculate a local vector or local array depending on the number of input values. The length of the vector or number of rows in the array is the number of recorded, broken bonds. If a single input is specified, a local vector is produced. If two or more inputs are specified, a local array is produced where the number of columns = the number of inputs. The vector or array can be accessed by any command that uses local values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The vector or array will be floating point values that correspond to the specified attribute.

Any settings with the *store/local* option are not saved to a restart file and must be redefined.

The potential energy and the single() function of this bond style return $k(r-r_0)^2/2$ as a proxy of the energy of a bonded interaction, ignoring any volumetric/smoothing factors or dissipative forces. The single() function also calculates an extra bond quantity, the initial distance $r_0$. This extra quantity can be accessed by the *compute bond/local* command as *b1*.

### 5.2.5 Restrictions

This bond style is part of the BPM package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

To handle breaking bonds, BPM bond styles have extra requirements for special bonds. If bonds cannot break (*break no*), then one can use any special bond weights. Otherwise, restrictions depend on whether pair forces are overlaid (*pair/overlay yes*). If so, then all weights must be one:

```
special_bonds lj/coul 1 1 1
```

If pair forces are disabled (*pair/overlay no*), the default, then the weights must be

```
special_bonds lj 0 1 1 coul 1 1 1
```

and *newton* must be set to bond off.

### 5.2.6 Related commands

*bond_coeff*, *pair bpm/spring*

### 5.2.7 Default

The option defaults are *overlay/pair* = *no*, *smooth* = *yes*, *normalize* = *no*, *break* = *yes*, and *volume/factor* = *no*

---

**(Clemmer)** Clemmer and Robbins, Phys. Rev. Lett. (2022).

**(Groot)** Groot and Warren, J Chem Phys, 107, 4423-35 (1997).

**(Clemmer2)** Clemmer, Monti, Lechman, Soft Matter, 20, 1702 (2024).

## 5.3 bond_style bpm/spring/plastic command

### 5.3.1 Syntax

```
bond_style bpm/spring/plastic keyword value attribute1 attribute2 ...
```

- optional keyword = *overlay/pair* or *store/local* or *smooth* or *normalize* or *break*

```
store/local values = fix_ID N attributes ...
   * fix_ID = ID of associated internal fix to store data
   * N = prepare data for output every this many timesteps
   * attributes = zero or more of the below attributes may be appended

     id1, id2 = IDs of two atoms in the bond
     time = the timestep the bond broke
     x, y, z = the center of mass position of the two atoms when the bond broke␣
↪(distance units)
     x/ref, y/ref, z/ref = the initial center of mass position of the two atoms␣
↪(distance units)

overlay/pair value = yes or no
   bonded particles will still interact with pair forces

smooth value = yes or no
   smooths bond forces near the breaking point

normalize value = yes or no
   normalizes bond forces by the reference length

break value = yes or no
   indicates whether bonds break during a run
```

### 5.3.2 Examples

```
bond_style bpm/spring/plastic
bond_coeff 1 1.0 0.05 0.1 0.02

bond_style bpm/spring/plastic myfix 1000 time id1 id2
dump 1 all local 1000 dump.broken f_myfix[1] f_myfix[2] f_myfix[3]
dump_modify 1 write_header no
```

### 5.3.3 Description

Added in version 2Apr2025.

The *bpm/spring/plastic* bond style computes forces based on deviations from the initial reference state of the two atoms and the strain history. The reference length of the bond $r_0$ is stored by each bond when it is first computed in the setup of a run. Initially, the equilibrium length of each bond $r_{eq}$ is set equal to $r_0$ but can evolve. data is then preserved across run commands and is written to *binary restart files* such that restarting the system will not modify either of these quantities.

This bond style only applies central-body forces which conserve the translational and rotational degrees of freedom of a bonded set of particles. The force has a magnitude of

$$F = -k(r_{eq} - r)w$$

where $k$ is a stiffness, $r$ is the current distance between the two particles, and $w$ is an optional smoothing factor discussed below. If the bond stretches beyond a strain of $\varepsilon_p$ in compression or extension, it will plastically activate and $r_{\mathrm{eq}}$ will evolve to ensure $|(r - r_{\mathrm{eq}})/r_{\mathrm{eq}}|$ never exceeds $\varepsilon_p$. Therefore, if a bond is continually loaded in either tension or compression, the force will initially grow elastically before plateauing. See *(Clemmer)* for more details on these mechanics.

Bonds will break at a strain of $\varepsilon_c$. This is done by setting the bond type to 0 such that forces are no longer computed.

An additional damping force is applied to the bonded particles. This forces is proportional to the difference in the normal velocity of particles:

$$F_D = -\gamma w(\hat{r} \bullet \vec{v})$$

where $\gamma$ is the damping strength, $\hat{r}$ is the radial normal vector, and $\vec{v}$ is the velocity difference between the two particles.

The smoothing factor $w$ is constructed such that forces smoothly go to zero, avoiding discontinuities, as bonds approach the critical breaking strain

$$w = 1.0 - \left(\frac{r - r_0}{r_0 \varepsilon_c}\right)^8.$$

The following coefficients must be defined for each bond type via the *bond_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $k$ (force/distance units)
- $\varepsilon_c$ (unitless)
- $\gamma$ (force/velocity units)
- $\varepsilon_p$ (unitless)

See the *bpm/spring doc page* for information on the *smooth*, *normalize*, *break*, *overlay/pair*, and *store/local* keywords.

Note that when unbroken bonds are dumped to a file via the *dump local* command, bonds with type 0 (broken bonds) are not included. The *delete_bonds* command can also be used to query the status of broken bonds or permanently delete them, e.g.:

```
delete_bonds all stats
delete_bonds all bond 0 remove
```

### 5.3.4 Restart and other info

This bond style writes the reference state and plastic history of each bond to *binary restart files*. Loading a restart file will properly restore bonds. However, the reference state is NOT written to data files. Therefore reading a data file will not restore bonds and will cause their reference states to be redefined.

The potential energy and the single() function of this bond style returns zero. The single() function also calculates two extra bond quantities, the initial distance $r_0$ and the current equilibrium length $r_{eq}$. These extra quantities can be accessed by the *compute bond/local* command as *b1* and *b2*, respectively.

### 5.3.5 Restrictions

This bond style is part of the BPM package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

To handle breaking bonds, BPM bond styles have extra requirements for special bonds. If bonds cannot break (*break no*), then one can use any special bond weights. Otherwise, restrictions depend on whether pair forces are overlaid (*pair/overlay yes*). If so, then all weights must be one:

```
special_bonds lj/coul 1 1 1
```

If pair forces are disabled (*pair/overlay no*), the default, then the weights must be

```
special_bonds lj 0 1 1 coul 1 1 1
```

and *newton* must be set to bond off.

### 5.3.6 Related commands

*bond_coeff* , *bond bpm/spring*

### 5.3.7 Default

The option defaults are *overlay/pair* = *no*, *smooth* = *yes*, *normalize* = *no*, and *break* = *yes*

---

**(Clemmer)** Clemmer and Lechman, Powder Technology (2025).

## 5.4 bond_style class2 command

Accelerator Variants: *class2/omp*, *class2/kk*

### 5.4.1 Syntax

```
bond_style class2
```

### 5.4.2 Examples

```
bond_style class2
bond_coeff 1 1.0 100.0 80.0 80.0
```

### 5.4.3 Description

The *class2* bond style uses the potential

$$E = K_2(r - r_0)^2 + K_3(r - r_0)^3 + K_4(r - r_0)^4$$

where $r_0$ is the equilibrium bond distance.

See *(Sun)* for a description of the COMPASS class2 force field.

The following coefficients must be defined for each bond type via the *bond_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $r_0$ (distance)
- $K_2$ (energy/distance^2)
- $K_3$ (energy/distance^3)
- $K_4$ (energy/distance^4)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 5.4.4 Restrictions

This bond style can only be used if LAMMPS was built with the CLASS2 package. See the *Build package* page for more info.

### 5.4.5 Related commands

*bond_coeff*, *delete_bonds*

### 5.4.6 Default

none

---

**(Sun)** Sun, J Phys Chem B 102, 7338-7364 (1998).

## 5.5 bond_style fene command

Accelerator Variants: *fene/intel*, *fene/kk*, *fene/omp*

## 5.6 bond_style fene/nm command

Accelerator Variants: *fene/nm/kk*, *fene/nm/omp*

### 5.6.1 Syntax

```
bond_style fene
bond_style fene/nm
```

### 5.6.2 Examples

```
bond_style fene
bond_coeff 1 30.0 1.5 1.0 1.0

bond_style fene/nm
bond_coeff 1 2.25344 1.5 1.0 1.12246 2 6
```

### 5.6.3 Description

The *fene* bond style uses the potential

$$E = -0.5KR_0^2 \ln\left[1 - \left(\frac{r}{R_0}\right)^2\right] + 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^6\right] + \varepsilon$$

to define a finite extensible nonlinear elastic (FENE) potential *(Kremer)*, used for bead-spring polymer models. The first term is attractive, the second Lennard-Jones term is repulsive. The first term extends to $R_0$, the maximum extent of the bond. The second term is cutoff at $2^{\frac{1}{6}}\sigma$, the minimum of the LJ potential.

The *fene/nm* bond style substitutes the standard LJ potential with the generalized LJ potential in the same form as in pair style *nm/cut*. The bond energy is then given by

$$E = -0.5KR_0^2 \ln\left[1 - \left(\frac{r}{R_0}\right)^2\right] + \frac{E_0}{(n-m)}\left[m\left(\frac{r_0}{r}\right)^n - n\left(\frac{r_0}{r}\right)^m\right]$$

Similar to the *fene* style, the generalized Lennard-Jones is cut off at the potential minimum, $r_0$, to be repulsive only. The following coefficients must be defined for each bond type via the *bond_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy/distance^2)
- $R_0$ (distance)
- $\varepsilon$ (energy)
- $\sigma$ (distance)

For the *fene/nm* style, the following coefficients are used. Please note, that the standard LJ potential and thus the regular FENE potential is recovered for (n=12 m=6) and $r_0 = 2^{\frac{1}{6}}\sigma$.

- $K$ (energy/distance^2)
- $R_0$ (distance)
- $E_0$ (energy)
- $r_0$ (distance)
- $n$ (unitless)
- $m$ (unitless)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 5.6.4 Restrictions

The *fene* bond style can only be used if LAMMPS was built with the MOLECULE package; the *fene/nm* bond style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* page for more info.

Added in version 4Jul2026: The *fene/nm/omp* style

You typically should specify *special_bonds fene* or *special_bonds lj/coul 0 1 1* to use this bond style. LAMMPS will issue a warning it that's not the case.

### 5.6.5 Related commands

*bond_coeff*, *delete_bonds*, *pair style lj/cut*, *pair style nm/cut*.

### 5.6.6 Default

none

---

**(Kremer)** Kremer, Grest, J Chem Phys, 92, 5057 (1990).

## 5.7 bond_style fene/expand command

Accelerator Variants: *fene/expand/kk*, *fene/expand/omp*

### 5.7.1 Syntax

```
bond_style fene/expand
```

### 5.7.2 Examples

```
bond_style fene/expand
bond_coeff 1 30.0 1.5 1.0 1.0 0.5
```

### 5.7.3 Description

The *fene/expand* bond style uses the potential

$$E = -0.5KR_0^2 \ln\left[1 - \left(\frac{(r-\Delta)}{R_0}\right)^2\right] + 4\varepsilon\left[\left(\frac{\sigma}{(r-\Delta)}\right)^{12} - \left(\frac{\sigma}{(r-\Delta)}\right)^{6}\right] + \varepsilon$$

to define a finite extensible nonlinear elastic (FENE) potential *(Kremer)*, used for bead-spring polymer models. The first term is attractive, the second Lennard-Jones term is repulsive.

The *fene/expand* bond style is similar to *fene* except that an extra shift factor of $\Delta$ (positive or negative) is added to $r$ to effectively change the bead size of the bonded atoms. The first term now extends to $R_0 + \Delta$ and the second term is cutoff at $2^{\frac{1}{6}}\sigma + \Delta$.

The following coefficients must be defined for each bond type via the *bond_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy/distance^2)
- $R_0$ (distance)
- $\varepsilon$ (energy)

- $\sigma$ (distance)
- $\Delta$ (distance)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 5.7.4 Restrictions

This bond style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* page for more info.

You typically should specify *special_bonds fene* or *special_bonds lj/coul 0 1 1* to use this bond style. LAMMPS will issue a warning it that's not the case.

### 5.7.5 Related commands

*bond_coeff* , *delete_bonds*

### 5.7.6 Default

none

---

**(Kremer)** Kremer, Grest, J Chem Phys, 92, 5057 (1990).

## 5.8 bond_style gaussian command

Accelerator Variants: *gaussian/kk*, *gaussian/omp*

### 5.8.1 Syntax

```
bond_style gaussian
```

### 5.8.2 Examples

```
bond_style gaussian
bond_coeff 1 300.0 2 0.0128 0.375 3.37 0.0730 0.148 3.63
```

### 5.8.3 Description

The *gaussian* bond style uses the potential:

$$E = -k_B T ln \left( \sum_{i=1}^{n} \frac{A_i}{w_i \sqrt{\pi/2}} exp \left( \frac{-2(r - r_i)^2}{w_i^2} \right) \right)$$

This analytical form is a suitable potential for obtaining mesoscale effective force fields which can reproduce target atomistic distributions *(Milano)*

The following coefficients must be defined for each bond type via the *bond_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $T$ temperature at which the potential was derived
- $n$ (integer >=1)
- $A_1$ (> 0, distance)
- $w_1$ (> 0, distance)
- $r_1$ (>= 0, distance)
- ...
- $A_n$ (> 0, distance)
- $w_n$ (> 0, distance)
- $r_n$ (>= 0, distance)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 5.8.4 Restrictions

This bond style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

Added in version 4Jul2026: The *gaussian/omp* style

### 5.8.5 Related commands

*bond_coeff*

### 5.8.6 Default

none

---

**(Milano)** G. Milano, S. Goudeau, F. Mueller-Plathe, J. Polym. Sci. B Polym. Phys. 43, 871 (2005).

## 5.9 bond_style gromos command

Accelerator Variants: *gromos/kk*, *gromos/omp*

### 5.9.1 Syntax

```
bond_style gromos
```

### 5.9.2 Examples

```
bond_style gromos
bond_coeff 5 80.0 1.2
```

### 5.9.3 Description

The *gromos* bond style uses the potential

$$E = K(r^2 - r_0^2)^2$$

where $r_0$ is the equilibrium bond distance. Note that the usual 1/4 factor is included in $K$.

The following coefficients must be defined for each bond type via the *bond_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy/distance^4)
- $r_0$ (distance)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 5.9.4 Restrictions

This bond style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* page for more info.

### 5.9.5 Related commands

*bond_coeff*, *delete_bonds*

### 5.9.6 Default

none

## 5.10 bond_style harmonic command

Accelerator Variants: *harmonic/intel*, *harmonic/kk*, *harmonic/omp*

### 5.10.1 Syntax

```
bond_style harmonic
```

### 5.10.2 Examples

```
bond_style harmonic
bond_coeff 5 80.0 1.2
```

### 5.10.3 Description

The *harmonic* bond style uses the potential

$$E = K(r - r_0)^2$$

where $r_0$ is the equilibrium bond distance. Note that the usual 1/2 factor is included in $K$.

The following coefficients must be defined for each bond type via the *bond_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy/distance^2)
- $r_0$ (distance)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 5.10.4 Restrictions

This bond style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* page for more info.

### 5.10.5 Related commands

*bond_coeff*, *delete_bonds bond style harmonic/shift*, *bond style harmonic/shift/cut*

### 5.10.6 Default

none

## 5.11 bond_style harmonic/restrain command

Accelerator Variants: *harmonic/restrain/kk*, *harmonic/restrain/omp*

### 5.11.1 Syntax

```
bond_style harmonic/restrain
```

### 5.11.2 Examples

```
bond_style harmonic
bond_coeff 5 80.0
```

### 5.11.3 Description

Added in version 28Mar2023.

The *harmonic/restrain* bond style uses the potential

$$E = K(r - r_{t=0})^2$$

where $r_{t=0}$ is the distance between the bonded atoms at the beginning of the first *run* or *minimize* command after the bond style has been defined (*t=0*). Note that the usual 1/2 factor is included in $K$. This will effectively restrain bonds to their initial length, whatever that is. This is where this bond style differs from *bond style harmonic* where the bond length is set through the per bond type coefficients.

The following coefficient must be defined for each bond type via the *bond_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands

- $K$ (energy/distance^2)

This bond style differs from other options to add harmonic restraints like *fix restrain* or *pair style list* or *fix colvars* in that it requires a bond topology, and thus the defined bonds will trigger exclusion of special neighbors from the neighbor list according to the *special_bonds* settings.

### 5.11.4 Restart info

This bond style supports the *write_restart* and *read_restart* commands. The state of the initial bond lengths is stored with restart files and read back.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 5.11.5 Restrictions

This bond style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* page for more info.

This bond style maintains internal data to determine the original bond lengths $r_{t=0}$. This information will be written to *binary restart files* but **not** to *data files*. Thus, continuing a simulation is *only* possible with *read_restart*. When using the *read_data command*, the reference bond lengths $r_{t=0}$ will be re-initialized from the current geometry.

This bond style cannot be used with *fix shake or fix rattle*, with *fix filter/corotate*, or any *tip4p pair style* since there is no specific equilibrium distance for a given bond type.

### 5.11.6 Related commands

*bond_coeff*, *bond_harmonic*, *fix restrain*, *pair style list*

### 5.11.7 Default

none

## 5.12 bond_style harmonic/shift command

Accelerator Variants: *harmonic/shift/kk*, *harmonic/shift/omp*

### 5.12.1 Syntax

```
bond_style harmonic/shift
```

### 5.12.2 Examples

```
bond_style harmonic/shift
bond_coeff 5 10.0 0.5 1.0
```

### 5.12.3 Description

The *harmonic/shift* bond style is a shifted harmonic bond that uses the potential

$$E = \frac{U_{\min}}{(r_0 - r_c)^2} \left[ (r - r_0)^2 - (r_c - r_0)^2 \right]$$

where $r_0$ is the equilibrium bond distance, and $r_c$ the critical distance. The potential energy has the value $-U_{\min}$ at $r_0$ and zero at $r_c$. This bond style differs from *bond_style harmonic* by the value of the potential energy.

The equivalent spring constant value $K$ for use with *bond_style harmonic* can be computed using $K = U_{\min}/[(r_0 - r_c)^2]$.

The following coefficients must be defined for each bond type via the *bond_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $U_{\min}$ (energy)
- $r_0$ (distance)
- $r_c$ (distance)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages*

page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 5.12.4 Restrictions

This bond style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 5.12.5 Related commands

*bond_coeff* , *delete_bonds*, *bond style harmonic*, *bond style harmonic/shift/cut*

### 5.12.6 Default

none

## 5.13 bond_style harmonic/shift/cut command

Accelerator Variants: *harmonic/shift/cut/kk*, *harmonic/shift/cut/omp*

### 5.13.1 Syntax

```
bond_style harmonic/shift/cut
```

### 5.13.2 Examples

```
bond_style harmonic/shift/cut
bond_coeff 5 10.0 0.5 1.0
```

### 5.13.3 Description

The *harmonic/shift/cut* bond style is a shifted harmonic bond that uses the potential

$$E = \frac{U_{\min}}{(r_0 - r_c)^2} \left[ (r - r_0)^2 - (r_c - r_0)^2 \right]$$

where $r_0$ is the equilibrium bond distance, and $r_c$ the critical distance. The bond potential is zero and thus its force also zero for distances $r > r_c$. The potential energy has the value $-U_{\min}$ at $r_0$ and zero at $r_c$.

The equivalent spring constant value $K$ for use with *bond_style harmonic* for $r <= r_c$, can be computed using $K = U_{\min}/[(r_0 - r_c)^2]$

The following coefficients must be defined for each bond type via the *bond_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $U_{\min}$ (energy)

- $r_0$ (distance)
- $r_c$ (distance)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 5.13.4 Restrictions

This bond style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 5.13.5 Related commands

*bond_coeff* , *delete_bonds*, *bond_harmonic*, *bond_style harmonic/shift*

### 5.13.6 Default

none

## 5.14 bond_style hybrid command

Accelerator Variants: *hybrid/kk*

### 5.14.1 Syntax

```
bond_style hybrid style1 style2 ...
```

- style1,style2 = list of one or more bond styles

### 5.14.2 Examples

```
bond_style hybrid harmonic fene
bond_coeff 1 harmonic 80.0 1.2
bond_coeff 2* fene 30.0 1.5 1.0 1.0
```

### 5.14.3 Description

The *hybrid* style enables the use of multiple bond styles in one simulation. A bond style is assigned to each bond type. For example, bonds in a polymer flow (of bond type 1) could be computed with a *fene* potential and bonds in the wall boundary (of bond type 2) could be computed with a *harmonic* potential. The assignment of bond type to style is made via the *bond_coeff* command or in the data file.

In the bond_coeff commands, the name of a bond style must be added after the bond type, with the remaining coefficients being those appropriate to that style. In the example above, the 2 bond_coeff commands set bonds of bond type 1 to be computed with a *harmonic* potential with coefficients 80.0, 1.2 for $K$, $r_0$. All other bond types (2-N) are computed with a *fene* potential with coefficients 30.0, 1.5, 1.0, 1.0 for $K$, $R_0$, $\epsilon$, $\sigma$.

If bond coefficients are specified in the data file read via the *read_data* command, then the same rule applies. E.g. "harmonic" or "fene" must be added after the bond type, for each line in the "Bond Coeffs" section, e.g.

```
Bond Coeffs

1 harmonic 80.0 1.2
2 fene 30.0 1.5 1.0 1.0
...
```

A bond style of *none* with no additional coefficients can be used in place of a bond style, either in a input script bond_coeff command or in the data file, if you desire to turn off interactions for specific bond types.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 5.14.4 Restrictions

This bond style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* page for more info.

Unlike other bond styles, the hybrid bond style does not store bond coefficient info for individual sub-styles in *binary restart files* or *data files*. Thus when restarting a simulation, you need to re-specify the bond_coeff commands.

### 5.14.5 Related commands

*bond_coeff*, *delete_bonds*

### 5.14.6 Default

none

## 5.15 bond_style lepton command

Accelerator Variants: *lepton/omp*

### 5.15.1 Syntax

```
bond_style style args
```

- style = *lepton*
- args = optional arguments

```
args = auto_offset or no_offset
  auto_offset = offset the potential energy so that the value at r0 is 0.0 (default)
  no_offset = do not offset the potential energy
```

### 5.15.2 Examples

```
bond_style lepton
bond_style lepton no_offset

bond_coeff  1  1.5 "k*r^2; k=250.0"
bond_coeff  2  1.1 "k2*r^2 + k3*r^3 + k4*r^4; k2=300.0; k3=-100.0; k4=50.0"
bond_coeff  3  1.3 "k*r^2; k=350.0"
```

### 5.15.3 Description

Added in version 8Feb2023.

Bond style *lepton* computes bonded interactions between two atoms with a custom function. The potential function must be provided as an expression string using "r" as the distance variable relative to the reference distance $r_0$ which is provided as a bond coefficient. For example *"200.0*r^2"* represents a harmonic potential with a force constant $K$ of 200.0 energy units:

$$U_{bond,i} = K(r_i - r_0)^2 = Kr^2 \qquad r = r_i - r_0$$

Changed in version 7Feb2024.

By default the potential energy U is shifted so that he value U is 0.0 for $r = r_0$. This is equivalent to using the optional keyword *auto_offset*. When using the keyword *no_offset* instead, the potential energy is not shifted.

The Lepton library, that the *lepton* bond style interfaces with, evaluates this expression string at run time to compute the pairwise energy. It also creates an analytical representation of the first derivative of this expression with respect to "r" and then uses that to compute the force between the atom pairs forming bonds as defined by the topology data.

The following coefficients must be defined for each bond type via the *bond_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- Lepton expression (energy units)
- $r_0$ (distance)

The Lepton expression must be either enclosed in quotes or must not contain any whitespace so that LAMMPS recognizes it as a single keyword. More on valid Lepton expressions below. The $r_0$ is the "equilibrium distance". The potential energy function in the Lepton expression is shifted in such a way, that the potential energy is 0 for a bond length $r_i == r_0$.

### 5.15.4 Lepton expression syntax and features

Lepton supports the following operators in expressions:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| + | Add | - | Subtract | * | Multiply | / | Divide | ^ | Power |

The following mathematical functions are available:

| | | | |
|---|---|---|---|
| sqrt(x) | Square root | exp(x) | Exponential |
| log(x) | Natural logarithm | sin(x) | Sine (angle in radians) |
| cos(x) | Cosine (angle in radians) | sec(x) | Secant (angle in radians) |
| csc(x) | Cosecant (angle in radians) | tan(x) | Tangent (angle in radians) |
| cot(x) | Cotangent (angle in radians) | asin(x) | Inverse sine (in radians) |
| acos(x) | Inverse cosine (in radians) | atan(x) | Inverse tangent (in radians) |
| sinh(x) | Hyperbolic sine | cosh(x) | Hyperbolic cosine |
| tanh(x) | Hyperbolic tangent | erf(x) | Error function |
| erfc(x) | Complementary Error function | abs(x) | Absolute value |
| min(x,y) | Minimum of two values | max(x,y) | Maximum of two values |
| delta(x) | delta(x) is 1 for $x = 0$, otherwise 0 | step(x) | step(x) is 0 for $x < 0$, otherwise 1 |

Numbers may be given in either decimal or exponential form. All of the following are valid numbers: *5*, *-3.1*, *1e6*, and *3.12e-2*.

As an extension to the standard Lepton syntax, it is also possible to use LAMMPS *variables* in the format "v_name". Before evaluating the expression, "v_name" will be replaced with the value of the variable "name". This is compatible with all kinds of scalar variables, but not with vectors, arrays, local, or per-atom variables. If necessary, a custom scalar variable needs to be defined that can access the desired (single) item from a non-scalar variable. As an example, the following lines will instruct LAMMPS to ramp the force constant for a harmonic bond from 100.0 to 200.0 during the next run:

```
variable fconst equal ramp(100.0, 200)
bond_style lepton
bond_coeff 1 1.5 "v_fconst * (r^2)"
```

An expression may be followed by definitions for intermediate values that appear in the expression. A semicolon ";" is used as a delimiter between value definitions. For example, the expression:

```
a^2+a*b+b^2; a=a1+a2; b=b1+b2
```

is exactly equivalent to

```
(a1+a2)^2+(a1+a2)*(b1+b2)+(b1+b2)^2
```

The definition of an intermediate value may itself involve other intermediate values. Whitespace and quotation characters ('' and "") are ignored. All uses of a value must appear *before* that value's definition. For efficiency reasons, the expression string is parsed, optimized, and then stored in an internal, pre-parsed representation for evaluation.

Evaluating a Lepton expression is typically between 2.5 and 5 times slower than the corresponding compiled and optimized C++ code. If additional speed or GPU acceleration (via GPU or KOKKOS) is required, the interaction can be represented as a table. Suitable table files can be created either internally using the *pair_write* or *bond_write* command or through the Python scripts in the *tools/tabulate* folder.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages*

page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 5.15.5 Restrictions

This bond style is part of the LEPTON package and only enabled if LAMMPS was built with this package. See the *Build package* page for more info.

### 5.15.6 Related commands

*bond_coeff*, *bond_style table*, *bond_write*, *angle_style lepton*, *dihedral_style lepton*

### 5.15.7 Default

none

## 5.16 bond_style mesocnt command

### 5.16.1 Syntax

```
bond_style mesocnt
```

### 5.16.2 Examples

```
bond_style mesocnt
bond_coeff 1 C 10 10 20.0
bond_coeff 4 custom 800.0 10.0
```

### 5.16.3 Description

Added in version 15Sep2022.

The *mesocnt* bond style is a wrapper for the *harmonic* style, and uses the potential

$$E = K(r - r_0)^2$$

where $r_0$ is the equilibrium bond distance. Note that the usual 1/2 factor is included in $K$. The style implements parameterization presets of $K$ for mesoscopic simulations of carbon nanotubes based on the atomistic simulations of *(Srivastava)*.

Other presets can be readily implemented in the future.

The following coefficients must be defined for each bond type via the *bond_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- preset = *C* or *custom*

- additional parameters depending on preset

Preset *C* is for carbon nanotubes, and the additional parameters are:

- chiral index $n$ (unitless)
- chiral index $m$ (unitless)
- $r_0$ (distance)

Preset *custom* is simply a direct wrapper for the *harmonic* style, and the additional parameters are:

- $K$ (energy/distance^2)
- $r_0$ (distance)

### 5.16.4 Restrictions

This bond style can only be used if LAMMPS was built with the MOLECULE and MESONT packages. See the *Build package* page for more info.

### 5.16.5 Related commands

*bond_coeff*, *delete_bonds*

### 5.16.6 Default

none

---

**(Srivastava)** Zhigilei, Wei and Srivastava, Phys. Rev. B 71, 165417 (2005).

## 5.17 bond_style mm3 command

Accelerator Variants: *mm3/kk*, *mm3/omp*

### 5.17.1 Syntax

```
bond_style mm3
```

### 5.17.2 Examples

```
bond_style mm3
bond_coeff 1 100.0 107.0
```

### 5.17.3 Description

The *mm3* bond style uses the potential that is anharmonic in the bond as defined in *(Allinger)*

$$E = K(r - r_0)^2 \left[1 - 2.55(r - r_0) + \frac{7}{12}2.55^2(r - r_0)^2\right]$$

where $r_0$ is the equilibrium value of the bond, and $K$ is a prefactor. The anharmonic prefactors have units $\text{Å}^{-n}$: $-2.55\text{Å}^{-1}$ and $\frac{7}{12}2.55^2\text{Å}^{-2}$. The code takes care of the necessary unit conversion for these factors internally. Note that the MM3 papers contain an error in Eq (1): $\frac{7}{12}2.55$ should be replaced with $\frac{7}{12}2.55^2$

The following coefficients must be defined for each bond type via the *bond_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy/distance^2)
- $r_0$ (distance)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 5.17.4 Restrictions

This bond style can only be used if LAMMPS was built with the YAFF package. See the *Build package* doc page for more info.

### 5.17.5 Related commands

*bond_coeff*

### 5.17.6 Default

none

---

**(Allinger)** Allinger, Yuh, Lii, JACS, 111(23), 8551-8566 (1989),

## 5.18 bond_style morse command

Accelerator Variants: *morse/kk*, *morse/omp*

### 5.18.1 Syntax

```
bond_style morse
```

### 5.18.2 Examples

```
bond_style morse
bond_coeff 5 1.0 2.0 1.2
```

### 5.18.3 Description

The *morse* bond style uses the potential

$$E = D\left[1 - e^{-\alpha(r-r_0)}\right]^2$$

where $r_0$ is the equilibrium bond distance, $\alpha$ is a stiffness parameter, and $D$ determines the depth of the potential well.

The following coefficients must be defined for each bond type via the *bond_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $D$ (energy)
- $\alpha$ (inverse distance)
- $r_0$ (distance)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 5.18.4 Restrictions

This bond style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* page for more info.

### 5.18.5 Related commands

*bond_coeff*, *delete_bonds*

### 5.18.6 Default

none

## 5.19 bond_style none command

### 5.19.1 Syntax

```
bond_style none
```

### 5.19.2 Examples

```
bond_style none
```

### 5.19.3 Description

Using a bond style of none means bond forces and energies are not computed, even if pairs of bonded atoms were listed in the data file read by the *read_data* command.

See the *bond_style zero* command for a way to calculate bond statistics, but compute no bond interactions.

### 5.19.4 Restrictions

none

### 5.19.5 Related commands

none

*bond_style zero*

### 5.19.6 Default

none

## 5.20 bond_style nonlinear command

Accelerator Variants: *nonlinear/kk*, *nonlinear/omp*

### 5.20.1 Syntax

```
bond_style nonlinear
```

### 5.20.2 Examples

```
bond_style nonlinear
bond_coeff 2 100.0 1.1 1.4
```

### 5.20.3 Description

The *nonlinear* bond style uses the potential

$$E = \frac{\varepsilon(r - r_0)^2}{[\lambda^2 - (r - r_0)^2]}$$

to define an anharmonic spring *(Rector)* of equilibrium length $r_0$ and maximum extension lamda.

The following coefficients must be defined for each bond type via the *bond_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $\varepsilon$ (energy)
- $r_0$ (distance)
- $\lambda$ (distance)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 5.20.4 Restrictions

This bond style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* page for more info.

### 5.20.5 Related commands

*bond_coeff*, *delete_bonds*

### 5.20.6 Default

none

---

**(Rector)** Rector, Van Swol, Henderson, Molecular Physics, 82, 1009 (1994).

## 5.21 bond_style oxdna/fene command

## 5.22 bond_style oxdna2/fene command

## 5.23 bond_style oxdna3/fene command

## 5.24 bond_style oxrna2/fene command

### 5.24.1 Syntax

```
bond_style oxdna/fene

bond_style oxdna2/fene

bond_style oxdna3/fene

bond_style oxrna2/fene
```

### 5.24.2 Examples

```
# LJ units
bond_style oxdna/fene
bond_coeff * 2.0 0.25 0.7525

bond_style oxdna2/fene
bond_coeff * 2.0 0.25 0.7564
```

(continues on next page)

(continued from previous page)

```
bond_style oxdna3/fene
bond_coeff * oxdna3_lj.cgdna

bond_style oxrna2/fene
bond_coeff * 2.0 0.25 0.76107

# Real units
bond_style oxdna/fene
bond_coeff * 11.92337812042065 2.1295 6.409795

bond_style oxdna2/fene
bond_coeff * 11.92337812042065 2.1295 6.4430152

bond_style oxdna3/fene
bond_coeff * oxdna3_real.cgdna

bond_style oxrna2/fene
bond_coeff * 11.92337812042065 2.1295 6.482800913
```

**Note**

The coefficients in the above examples have to be kept fixed and cannot be changed without reparameterizing the entire model. They are provided in forms compatible with both *units lj* and *units real* (see documentation of *units*). These can also be read from a potential file (sole option for oxDNA3) with correct unit style by specifying the name of the file. Several potential files for each unit style are included in the `potentials` directory of the LAMMPS distribution.

### 5.24.3 Description

The *oxdna/fene*, *oxdna2/fene*, *oxdna3/fene* and *oxrna2/fene* bond styles use the potential

$$E = -\frac{\varepsilon}{2} \ln\left[1 - \left(\frac{r - r_0}{\Delta}\right)^2\right]$$

to define a modified finite extensible nonlinear elastic (FENE) potential *(Ouldridge)* to model the connectivity of the phosphate backbone in the oxDNA/oxRNA force field for coarse-grained modelling of DNA/RNA.

The following coefficients must be defined for the bond type via the *bond_coeff* command as given in the above example, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $\varepsilon$ (energy)
- $\Delta$ (distance)
- $r_0$ (distance)

**Note**

The oxDNA bond style has to be used together with the corresponding oxDNA pair styles for excluded volume interaction *oxdna/excv* , stacking *oxdna/stk* , cross-stacking *oxdna/xstk* and coaxial stacking interaction *oxdna/coaxstk* as well as hydrogen-bonding interaction *oxdna/hbond* (see also documentation of *pair_style oxdna/excv*). For the oxDNA2 *(Snodin)* bond style the analogous pair styles *oxdna2/excv* , *oxdna2/stk* , *oxdna2/xstk* , *oxdna2/coaxstk* ,

*oxdna2/hbond* and an additional Debye-Hueckel pair style *oxdna2/dh* have to be defined. The same applies to the oxDNA3 *(Bonato)* and oxRNA2 *(Sulc1)* styles.

**Note**

This bond style has to be used with the *atom_style hybrid bond ellipsoid oxdna* (see documentation of *atom_style*). The *atom_style oxdna* stores the 3'-to-5' polarity of the nucleotide strand, which is set through the bond topology in the data file. The first (second) atom in a bond definition is understood to point towards the 3'-end (5'-end) of the strand.

**Warning**

If data files are produced with *write_data*, then the *newton* command should be set to *newton on*. Otherwise the data files will not have the same 3'-to-5' polarity as the initial data file. This limitation does not apply to binary restart files produced with *write_restart*.

Example input and data files for DNA and RNA duplexes can be found in `examples/PACKAGES/cgdna/examples/lj_units/oxDNA/`, `.../oxDNA2/`, `.../oxDNA3/` and `.../oxRNA2/` or in the corresponding folder for real units. A simple python setup tool which creates single straight or helical DNA strands, DNA/RNA duplexes or arrays of DNA/RNA duplexes can be found in `examples/PACKAGES/cgdna/util/`.

Please cite *(Henrich)* in any publication that uses this implementation. An updated documentation that contains general information on the model, its implementation and performance as well as the structure of the data and input file can be found here.

Please cite also the relevant oxDNA/oxRNA publications. These are *(Ouldridge)* and *(Ouldridge-DPhil)* for oxDNA, *(Snodin)* for oxDNA2, *(Bonato)* for oxDNA3, *(Sulc1)* for oxRNA2 and for sequence-specific hydrogen-bonding and stacking interactions *(Sulc2)*.

---

### 5.24.4 Potential file reading

For each style oxdna, oxdna2, oxdna3 and oxrna2, the first parameter argument can be a filename, and if it is, no further arguments should be supplied. Therefore the following command:

```
bond_style oxdna/fene
bond_coeff * oxdna_lj.cgdna
```

will be interpreted as a request to read the (FENE) potential *(Ouldridge)* parameters from the file with the given name. The file can define multiple potential parameters for both bonded and pair interactions, but for the above bonded interactions there must exist in the file a line of the form:

```
*   fene    epsilon delta r0
```

There are sample potential files for each unit style in the `potentials` directory of the LAMMPS distribution. The potential file unit system must align with the units defined via the *units* command. For conversion between different *LJ* and *real* unit systems for oxDNA, the python tool *lj2real.py* located in the `examples/PACKAGES/cgdna/util/` directory can be used. This tool assumes similar file structure to the examples found in `examples/PACKAGES/cgdna/examples/`.

---

### 5.24.5 Restrictions

This bond style can only be used if LAMMPS was built with the CG-DNA package and the MOLECULE and ASPHERE package. See the *Build package* page for more info.

### 5.24.6 Related commands

*pair_style oxdna/excv*, *pair_style oxdna2/excv*, *pair_style oxdna3/excv*, *pair_style oxrna2/excv*, *bond_coeff*, *atom_style oxdna*, *fix nve/dotc/langevin*

### 5.24.7 Default

none

---

**(Henrich)** O. Henrich, Y.A. Gutierrez-Fosado, T. Curk, T.E. Ouldridge, Eur. Phys. J. E 41, 57 (2018).

**(Ouldridge-DPhil)** T.E. Ouldridge, Coarse-grained modelling of DNA and DNA self-assembly, DPhil. University of Oxford (2011).

**(Ouldridge)** T.E. Ouldridge, A.A. Louis, J.P.K. Doye, J. Chem. Phys. 134, 085101 (2011).

**(Snodin)** B.E. Snodin, F. Randisi, M. Mosayebi, et al., J. Chem. Phys. 142, 234901 (2015).

**(Bonato)** A. Bonato, T.E. Ouldridge, A.A. Louis, J.P.K. Doye, L. Rovigatti, M. Matthies, O.Henrich, in preparation.

**(Sulc1)** P. Sulc, F. Romano, T.E. Ouldridge, et al., J. Chem. Phys. 140, 235102 (2014).

**(Sulc2)** P. Sulc, F. Romano, T.E. Ouldridge, L. Rovigatti, J.P.K. Doye, A.A. Louis, J. Chem. Phys. 137, 135101 (2012).

## 5.25 bond_style quartic command

Accelerator Variants: *quartic/kk*, *quartic/omp*

### 5.25.1 Syntax

```
bond_style quartic
```

### 5.25.2 Examples

```
bond_style quartic
bond_coeff 2 1200 -0.55 0.25 1.3 34.6878
```

### 5.25.3 Description

The *quartic* bond style uses the potential

$$
\begin{aligned}
E &= E_q + E_{LJ} \\
E_q &= K(r - R_c)^2(r - R_c - B_1)(r - R_c - B_2) + U_0 \\
E_{LJ} &= \begin{cases} 4\varepsilon \left[ \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] + \varepsilon & : \quad r < 2^{\frac{1}{6}}, \varepsilon = 1, \sigma = 1 \\ 0 & : \quad r >= 2^{\frac{1}{6}} \end{cases}
\end{aligned}
$$

to define a bond that can be broken as the simulation proceeds (e.g. due to a polymer being stretched). The $\sigma$ and $\varepsilon$ used in the LJ portion of the formula are both set equal to 1.0 by LAMMPS and the LJ portion is cut off at its minimum, i.e. at $r_c = 2^{\frac{1}{6}}$.

The following coefficients must be defined for each bond type via the *bond_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy/distance^4)
- $B_1$ (distance)
- $B_2$ (distance)
- $R_c$ (distance)
- $U_0$ (energy)

This potential was constructed to mimic the FENE bond potential for coarse-grained polymer chains. When monomers with $\sigma = \varepsilon = 1.0$ are used, the following choice of parameters gives a quartic potential that looks nearly like the FENE potential:

$$
\begin{aligned}
K &= 1200 \\
B_1 &= -0.55 \\
B_2 &= 0.25 \\
R_c &= 1.3 \\
U_0 &= 34.6878
\end{aligned}
$$

Different parameters can be specified using the *bond_coeff* command, but you will need to choose them carefully so they form a suitable bond potential.

$R_c$ is the cutoff length at which the bond potential goes smoothly to a local maximum. If a bond length ever becomes $> R_c$, LAMMPS "breaks" the bond, which means two things. First, the bond potential is turned off by setting its type to 0, and is no longer computed. Second, a pairwise interaction between the two atoms is turned on, since they are no longer bonded. See the *Howto* page on broken bonds for more information.

LAMMPS does the second task via a computational sleight-of-hand. It subtracts the pairwise interaction as part of the bond computation. When the bond breaks, the subtraction stops. For this to work, the pairwise interaction must always be computed by the *pair_style* command, whether the bond is broken or not. This means that *special_bonds* must be set to 1,1,1, as indicated as a restriction below.

Note that when bonds are dumped to a file via the *dump local* command, bonds with type 0 are not included. The *delete_bonds* command can also be used to query the status of broken bonds or permanently delete them, e.g.:

```
delete_bonds all stats
delete_bonds all bond 0 remove
```

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 5.25.4 Restrictions

This bond style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* page for more info.

The *quartic* style requires that *special_bonds* parameters be set to 1,1,1. Three- and four-body interactions (angle, dihedral, etc) cannot be used with *quartic* bonds.

### 5.25.5 Related commands

*bond_coeff*, *delete_bonds*

### 5.25.6 Default

none

## 5.26 bond_style quartic/exp command

Accelerator Variants: *quartic/exp/kk*

### 5.26.1 Syntax

```
bond_style quartic/exp
```

### 5.26.2 Examples

```
bond_style quartic/exp
bond_coeff 1 1.54 200.0 -100.0 50.0 0.0 1.0
bond_coeff 2 1.0 500.0 0.0 -200.0 1000.0 0.3
```

### 5.26.3 Description

Added in version 4Jul2026.

The *quartic/exp* bond style uses the potential

$$E = k_2(r - r_0)^2 + k_3(r - r_0)^3 + k_4(r - r_0)^4 + A \exp\left(-\frac{r}{B}\right)$$

where $r_0$ is the equilibrium bond distance. The first three terms form a quartic polynomial in the bond displacement $(r - r_0)$ that can model anharmonic stretching. The optional exponential term $(A \exp(-r/B))$ adds a repulsive or attractive wall at short range. Setting $A = 0$ disables the exponential term entirely.

The following coefficients must be defined for each bond type via the *bond_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $r_0$ (distance)
- $k_2$ (energy/distance^2)
- $k_3$ (energy/distance^3)
- $k_4$ (energy/distance^4)
- $A$ (energy)
- $B$ (distance, must be non-zero if $A \neq 0$)

Note that the $k_2$ coefficient plays the role of the spring constant. The $k_3$ and $k_4$ coefficients introduce anharmonicity. Setting $k_3 = k_4 = 0$ and $A = 0$ reduces this style to *bond_style harmonic* with $K = k_2$ (the factor of 1/2 is *not* included in $k_2$ here).

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 5.26.4 Restrictions

This bond style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* page for more info.

The parameter $B$ must be non-zero whenever $A \neq 0$.

### 5.26.5 Related commands

*bond_coeff*, *delete_bonds*, *bond_style harmonic*, *bond_style nonlinear*

### 5.26.6 Default

none

## 5.27 bond_style rheo/shell command

### 5.27.1 Syntax

```
bond_style rheo/shell keyword value attribute1 attribute2 ...
```

- required keyword = *t/form*
- optional keyword = *store/local*

```
t/form value = formation time for a bond (time units)

store/local values = fix_ID N attributes ...
   * fix_ID = ID of associated internal fix to store data
   * N = prepare data for output every this many timesteps
   * attributes = zero or more of the below attributes may be appended

     id1, id2 = IDs of two atoms in the bond
     time = the timestep the bond broke
     x, y, z = the center of mass position of the two atoms when the bond broke␣
↪(distance units)
     x/ref, y/ref, z/ref = the initial center of mass position of the two atoms␣
↪(distance units)
```

### 5.27.2 Examples

```
bond_style rheo/shell t/form 10.0
bond_coeff 1 1.0 0.05 0.1
```

### 5.27.3 Description

Added in version 29Aug2024.

The *rheo/shell* bond style is designed to work with *fix rheo/oxidation* which creates candidate bonds between eligible surface or near-surface particles. When a bond is first created, it computes no forces and starts a timer. Forces are not computed until the timer reaches the specified bond formation time, *t/form*, and the bond is enabled and applies forces. If the two particles move outside of the maximum bond distance or move into the bulk before the timer reaches *t/form*, the bond automatically deletes itself. This deletion is not recorded as a broken bond in the optional *store/local* fix.

Before bonds are enabled, they are still treated as regular bonds by all other parts of LAMMPS. This means they are written to data files and counted in computes such as *nbond/atom*. To only count enabled bonds, use the *nbond/shell* attribute in *compute rheo/property/atom*.

When enabled, the bond then computes forces based on deviations from the initial reference state of the two atoms much like a BPM style bond (as further discussed in the *BPM howto page*). The reference state is stored by each bond when it is first enabled. Data is then preserved across run commands and is written to *binary restart files* such that restarting the system will not reset the reference state of a bond or the timer.

This bond style is based on a model described in *(Clemmer)*. The force has a magnitude of

$$F = 2k(r - r_0) + \frac{2k}{r_0^2 \varepsilon_c^2}(r - r_0)^3$$

where $k$ is a stiffness, $r$ is the current distance and $r_0$ is the initial distance between the two particles, and $\varepsilon_c$ is maximum strain beyond which a bond breaks. This is done by setting the bond type to 0 such that forces are no longer computed.

A damping force proportional to the difference in the normal velocity of particles is also applied to bonded particles:

$$F_D = -\gamma w(\hat{r} \bullet \vec{v})$$

where $\gamma$ is the damping strength, $\hat{r}$ is the displacement normal vector, and $\vec{v}$ is the velocity difference between the two particles.

The following coefficients must be defined for each bond type via the *bond_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $k$ (force/distance units)
- $\varepsilon_c$ (unitless)
- $\gamma$ (force/velocity units)

Unlike other BPM-style bonds, this bond style does not update special bond settings when bonds are created or deleted. This bond style also does not enforce specific *special_bonds* settings. This behavior is purposeful such *RHEO pair* forces and heat flows are still calculated.

If the *store/local* keyword is used, an internal fix will track bonds that break during the simulation. Whenever a bond breaks, data is processed and transferred to an internal fix labeled *fix_ID*. This allows the local data to be accessed by other LAMMPS commands. Following this optional keyword, a list of one or more attributes is specified. These include the IDs of the two atoms in the bond. The other attributes for the two atoms include the timestep during which the bond broke and the current/initial center of mass position of the two atoms.

Data is continuously accumulated over intervals of $N$ timesteps. At the end of each interval, all of the saved accumulated data is deleted to make room for new data. Individual datum may therefore persist anywhere between *1* to $N$ timesteps depending on when they are saved. This data can be accessed using the *fix_ID* and a *dump local* command. To ensure

all data is output, the dump frequency should correspond to the same interval of $N$ timesteps. A dump frequency of an integer multiple of $N$ can be used to regularly output a sample of the accumulated data.

Note that when unbroken bonds are dumped to a file via the *dump local* command, bonds with type 0 (broken bonds) are not included. The *delete_bonds* command can also be used to query the status of broken bonds or permanently delete them, e.g.:

```
delete_bonds all stats
delete_bonds all bond 0 remove
```

---

### 5.27.4 Restart and other info

This bond style writes the reference state of each bond to *binary restart files*. Loading a restart file will properly restore bonds. However, the reference state is NOT written to data files. Therefore reading a data file will not restore bonds and will cause their reference states to be redefined.

If the *store/local* option is used, an internal fix will calculate a local vector or local array depending on the number of input values. The length of the vector or number of rows in the array is the number of recorded, broken bonds. If a single input is specified, a local vector is produced. If two or more inputs are specified, a local array is produced where the number of columns = the number of inputs. The vector or array can be accessed by any command that uses local values from a compute as input. See the *Howto output* page for an overview of LAMMPS output options.

The vector or array will be floating point values that correspond to the specified attribute.

The single() function of this bond style returns 0.0 for the energy of a bonded interaction, since energy is not conserved in these dissipative potentials. The single() function also calculates two extra bond quantities, the initial distance $r_0$ and a time. These extra quantities can be accessed by the *compute bond/local* command as *b1* and *b2*.

### 5.27.5 Restrictions

This bond style is part of the RHEO package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 5.27.6 Related commands

*bond_coeff*, *fix rheo/oxidation*

### 5.27.7 Default

NA

---

**(Clemmer)** Clemmer, Pierce, O'Connor, Nevins, Jones, Lechman, Tencer, Appl. Math. Model., 130, 310-326 (2024).

## 5.28 bond_style special command

### 5.28.1 Syntax

```
bond_style special
```

### 5.28.2 Examples

```
bond_style special
bond_coeff 0.5 0.5
```

### 5.28.3 Description

The *special* bond style can be used to create conceptual bonds which effectively impose weightings on the pairwise Lennard Jones and/or Coulombic interactions between selected pairs of particles in the system. The form of the pairwise interaction will be whatever is computed by the *pair_style* command defined for the system; this command defines the weightings for its two terms.

This command can thus be useful to apply weightings that cannot be handled by the *special_bonds* command, such as on 1-5 or 1-6 interactions. Or it can be used to add pairwise forces between one or more pairs of atoms that otherwise would not be include in the *pair_style* computation.

The potential for this bond style has the form

$$E = w_{LJ} E_{LJ} + w_{Coul} E_{Coul}$$

The following coefficients must be defined for each bond type via the *bond_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $w_{LJ}$ weight (0.0 to 1.0) on pairwise Lennard-Jones interactions
- $w_{Coul}$ weight (0.0 to 1.0) on pairwise Coulombic interactions

---

Normally this bond style should be used in conjunction with one (or more) other bond styles which compute forces between atoms directly bonded to each other in a molecule. This means the *bond_style hybrid* command should be used with bond_style special as one of its sub-styles.

Note that the same as for any other bond style, pairs of bonded atoms must be enumerated in the data file read by the *read_data* command. Thus if this command is used to weight all 1-5 interactions in the system, all the 1-5 pairs of atoms must be listed in the "Bonds" section of the data file.

This bond style imposes strict requirements on settings made with the *special_bonds* command. These requirements ensure that the new bonds created by this style do not create spurious 1-2, 1-3, or 1-4 interactions within the molecular topology.

Specifically 1-2 interactions must have weights of zero, 1-3 interactions must either have weights of unity or *special_bonds angle yes* must be used, and 1-4 interactions must have weights of unity or *special_bonds dihedral yes* must be used.

If this command is used to create bonded interactions between particles that are further apart than usual (e.g. 1-5 or 1-6 interactions), this style may require an increase in the communication cutoff via the *comm_modify cutoff* command. If LAMMPS cannot find a partner atom in a bond, an error will be issued.

### 5.28.4 Restrictions

This bond style can only be used if LAMMPS was built with the MISC package. See the *Build package* doc page for more info.

This bond style requires the use of a *pair_style* which computes a pairwise additive interaction and provides the ability to compute interactions for individual pairs of atoms. Manybody potentials are not compatible in general, but also some other pair styles are missing the required functionality and thus will cause an error.

This command is not compatible with long-range Coulombic interactions. If a *kspace_style* <*kspace_style*> is declared, an error will be issued.

### 5.28.5 Related commands

*bond_coeff*, *special_bonds*

### 5.28.6 Default

none

## 5.29 bond_style table command

Accelerator Variants: *table/omp*

### 5.29.1 Syntax

```
bond_style table style N
```

- style = *linear* or *spline* = method of interpolation
- N = use N values in table

### 5.29.2 Examples

```
bond_style table linear 1000
bond_coeff 1 file.table ENTRY1
```

### 5.29.3 Description

Style *table* creates interpolation tables of length *N* from bond potential and force values listed in a file(s) as a function of bond length. The files are read by the *bond_coeff* command.

The interpolation tables are created by fitting cubic splines to the file values and interpolating energy and force values at each of *N* distances. During a simulation, these tables are used to interpolate energy and force values as needed. The interpolation is done in one of 2 styles: *linear* or *spline*.

For the *linear* style, the bond length is used to find 2 surrounding table values from which an energy or force is computed by linear interpolation.

For the *spline* style, a cubic spline coefficients are computed and stored at each of the *N* values in the table. The bond length is used to find the appropriate set of coefficients which are used to evaluate a cubic polynomial which computes the energy or force.

The following coefficients must be defined for each bond type via the *bond_coeff* command as in the example above.

- filename
- keyword

The filename specifies a file containing tabulated energy and force values. The keyword specifies a section of the file. The format of this file is described below.

---

Suitable tables for use with this bond style can be created by LAMMPS itself from existing bond styles using the *bond_write* command. This can be useful to have a template file for testing the bond style settings and to build a compatible custom file. Another option to generate tables is the Python code in the `tools/tabulate` folder of the LAMMPS source code distribution.

The format of a tabulated file is as follows (without the parenthesized comments):

```
# Bond potential for harmonic (one or more comment or blank lines)

HAM                           (keyword is the first text on line)
N 101 FP 0 0 EQ 0.5           (N, FP, EQ  parameters)
                              (blank line)
1 0.00 338.0000 1352.0000     (index, bond-length, energy, force)
2 0.01 324.6152 1324.9600
...
101 1.00 338.0000 -1352.0000
```

A section begins with a non-blank line whose first character is not a "#"; blank lines or lines starting with "#" can be used as comments between sections. The first line begins with a keyword which identifies the section. The line can contain additional text, but the initial text must match the argument specified in the *bond_coeff* command. The next line lists (in any order) one or more parameters for the table. Each parameter is a keyword followed by one or more numeric values.

The parameter "N" is required and its value is the number of table entries that follow. Note that this may be different than the *N* specified in the *bond_style table* command. Let Ntable = *N* in the bond_style command, and Nfile = "N" in the tabulated file. What LAMMPS does is a preliminary interpolation by creating splines using the Nfile tabulated values as nodal points. It uses these to interpolate as needed to generate energy and force values at Ntable different points. The resulting tables of length Ntable are then used as described above, when computing energy and force for individual bond lengths. This means that if you want the interpolation tables of length Ntable to match exactly what is in the tabulated file (with effectively no preliminary interpolation), you should set Ntable = Nfile.

The "FP" parameter is optional. If used, it is followed by two values fplo and fphi, which are the derivatives of the force at the innermost and outermost bond lengths. These values are needed by the spline construction routines. If not specified by the "FP" parameter, they are estimated (less accurately) by the first two and last two force values in the table.

The "EQ" parameter is also optional. If used, it is followed by a the equilibrium bond length, which is used, for example, by the *fix shake* command. If not used, the equilibrium bond length is to the distance in the table with the lowest potential energy.

Following a blank line, the next N lines list the tabulated values. On each line, the first value is the index from 1 to N, the second value is the bond length r (in distance units), the third value is the energy (in energy units), and the fourth is the force (in force units). The bond lengths must range from a LO value to a HI value, and increase from one line to the next. If the actual bond length is ever smaller than the LO value or larger than the HI value, then the calculation is aborted with an error, so it is advisable to cover the whole range of possible bond lengths.

Note that one file can contain many sections, each with a tabulated potential. LAMMPS reads the file section by section until it finds one that matches the specified keyword.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 5.29.4 Restart info

This bond style writes the settings for the "bond_style table" command to *binary restart files*, so a bond_style command does not need to specified in an input script that reads a restart file. However, the coefficient information is not stored in the restart file, since it is tabulated in the potential files. Thus, bond_coeff commands do need to be specified in the restart input script.

### 5.29.5 Restrictions

This bond style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* page for more info.

### 5.29.6 Related commands

*bond_coeff* , *delete_bonds*, *bond_write*

### 5.29.7 Default

none

## 5.30 bond_style zero command

### 5.30.1 Syntax

```
bond_style zero keyword
```

- zero or more keywords may be appended
- keyword = *nocoeff*

### 5.30.2 Examples

```
bond_style zero
bond_style zero nocoeff
bond_coeff *
bond_coeff * 2.14
```

### 5.30.3 Description

Using an bond style of zero means bond forces and energies are not computed, but the geometry of bond pairs is still accessible to other commands.

As an example, the *compute bond/local* command can be used to compute distances for the list of pairs of bond atoms listed in the data file read by the *read_data* command. If no bond style is defined, this command cannot be used.

The optional *nocoeff* flag allows to read data files with a BondCoeff section for any bond style. Similarly, any bond_coeff commands will only be checked for the bond type number and the rest ignored.

Note that the *bond_coeff* command must be used for all bond types. If specified, there can be only one value, which is going to be used to assign an equilibrium distance, e.g. for use with *fix shake*.

### 5.30.4 Restrictions

none

### 5.30.5 Related commands

*bond_style none*

### 5.30.6 Default

none

CHAPTER

SIX

# ANGLE STYLES

## 6.1 angle_style amoeba command

### 6.1.1 Syntax

```
angle_style amoeba
```

### 6.1.2 Examples

```
angle_style amoeba
angle_coeff * 75.0 -25.0 1.0 0.3 0.02 0.003
angle_coeff * ba 3.6551 24.895 1.0119 1.5228
angle_coeff * ub -7.6 1.5537
```

### 6.1.3 Description

The *amoeba* angle style uses the potential

$$
\begin{aligned}
E &= E_a + E_{ba} + E_{ub} \\
E_a &= K_2\,(\theta - \theta_0)^2 + K_3\,(\theta - \theta_0)^3 + K_4\,(\theta - \theta_0)^4 + K_5\,(\theta - \theta_0)^5 + K_6\,(\theta - \theta_0)^6 \\
E_{ba} &= N_1(r_{ij} - r_1)(\theta - \theta_0) + N_2(r_{jk} - r_2)(\theta - \theta_0) \\
E_{UB} &= K_{ub}(r_{ik} - r_{ub})^2
\end{aligned}
$$

where $E_a$ is the angle term, $E_{ba}$ is a bond-angle term, $E_{UB}$ is a Urey-Bradley bond term, $\theta_0$ is the equilibrium angle, $r_1$ and $r_2$ are the equilibrium bond lengths, and $r_{ub}$ is the equilibrium Urey-Bradley bond length.

These formulas match how the Tinker MD code performs its angle calculations for the AMOEBA and HIPPO force fields. See the *Howto amoeba* page for more information about the implementation of AMOEBA and HIPPO in LAMMPS.

Note that the $E_a$ and $E_{ba}$ formulas are identical to those used for the *angle_style class2/p6* command, however there is no bond-bond cross term formula for $E_{bb}$. Additionally, there is a $E_{UB}$ term for a Urey-Bradley bond. It is effectively a harmonic bond between the I and K atoms of angle IJK, even though that bond is not enumerated in the "Bonds" section of the data file.

There are also two ways that Tinker computes the angle $\theta$ in the $E_a$ formula. The first is the standard way of treating IJK as an "in-plane" angle. The second is an "out-of-plane" method which Tinker may use if the center atom J in the angle is bonded to one additional atom in addition to I and K. In this case, all 4 atoms are used to compute the $E_a$ formula, resulting in forces on all 4 atoms. In the Tinker PRM file, these 2 options are denoted by *angle* versus *anglep* entries in the "Angle Bending Parameters" section of the PRM force field file. The *pflag* coefficient described below selects between the 2 options.

Coefficients for the $E_a$, $E_{bb}$, and $E_{ub}$ formulas must be defined for each angle type via the *angle_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands.

These are the 8 coefficients for the $E_a$ formula:

- pflag = 0 or 1
- ubflag = 0 or 1
- $\theta_0$ (degrees)
- $K_2$ (energy)
- $K_3$ (energy)
- $K_4$ (energy)
- $K_5$ (energy)
- $K_6$ (energy)

A pflag value of 0 vs 1 selects between the "in-plane" and "out-of-plane" options described above. Ubflag is 1 if there is a Urey-Bradley term associated with this angle type, else it is 0. $\theta_0$ is specified in degrees, but LAMMPS converts it to radians internally; hence the various $K$ values are effectively energy per radian^2 or radian^3 or radian^4 or radian^5 or radian^6.

For the $E_{ba}$ formula, each line in a *angle_coeff* command in the input script lists 5 coefficients, the first of which is "ba" to indicate they are BondAngle coefficients. In a data file, these coefficients should be listed under a "BondAngle Coeffs" heading and you must leave out the "ba", i.e. only list 4 coefficients after the angle type.

- ba
- $N_1$ (energy/distance^2)
- $N_2$ (energy/distance^2)
- $r_1$ (distance)
- $r_2$ (distance)

The $\theta_0$ value in the $E_{ba}$ formula is not specified, since it is the same value from the $E_a$ formula.

For the $E_{ub}$ formula, each line in a *angle_coeff* command in the input script lists 3 coefficients, the first of which is "ub" to indicate they are UreyBradley coefficients. In a data file, these coefficients should be listed under a "UreyBradley Coeffs" heading and you must leave out the "ub", i.e. only list 2 coefficients after the angle type.

- ub
- $K_{ub}$ (energy/distance^2)
- $r_{ub}$ (distance)

---

### 6.1.4 Restrictions

This angle style can only be used if LAMMPS was built with the AMOEBA package. See the *Build package* doc page for more info.

### 6.1.5 Related commands

*angle_coeff*

### 6.1.6 Default

none

## 6.2 angle_style charmm command

Accelerator Variants: *charmm/intel*, *charmm/kk*, *charmm/omp*

### 6.2.1 Syntax

```
angle_style charmm
```

### 6.2.2 Examples

```
angle_style charmm
angle_coeff 1 300.0 107.0 50.0 3.0
```

### 6.2.3 Description

The *charmm* angle style uses the potential

$$E = K(\theta - \theta_0)^2 + K_{ub}(r - r_{ub})^2$$

with an additional Urey_Bradley term based on the distance $r$ between the first and third atoms in the angle. $K$, $\theta_0$, $K_{ub}$, and $R_{ub}$ are coefficients defined for each angle type.

See *(MacKerell)* for a description of the CHARMM force field.

The following coefficients must be defined for each angle type via the *angle_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy)
- $\theta_0$ (degrees)
- $K_{ub}$ (energy/distance^2)
- $r_{ub}$ (distance)

$\theta_0$ is specified in degrees, but LAMMPS converts it to radians internally; hence $K$ is effectively energy per radian^2.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 6.2.4 Restrictions

This angle style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* doc page for more info.

### 6.2.5 Related commands

*angle_coeff* , *pair_style lj/charmm variants*, *dihedral_style charmm*, *dihedral_style charmmfsw*, *fix cmap*

### 6.2.6 Default

none

---

**(MacKerell)** MacKerell, Bashford, Bellott, Dunbrack, Evanseck, Field, Fischer, Gao, Guo, Ha, et al, J Phys Chem, 102, 3586 (1998).

## 6.3 angle_style class2 command

Accelerator Variants: *class2/kk*, *class2/omp*

## 6.4 angle_style class2xe command

Accelerator Variants: *class2xe/kk*, *class2xe/omp*

## 6.5 angle_style class2/p6 command

Accelerator Variants: *class2/p6/kk*, *class2/p6/omp*

### 6.5.1 Syntax

```
angle_style class2
```

### 6.5.2 Examples

```
angle_style class2
angle_coeff * 75.0 25.0 0.3 0.002
angle_coeff 1 bb 10.5872 1.0119 1.5228
angle_coeff * ba 3.6551 24.895 1.0119 1.5228
```

### 6.5.3 Description

The *class2* angle style uses the potential

$$
\begin{aligned}
E &= E_a + E_{bb} + E_{ba} \\
E_a &= K_2(\theta - \theta_0)^2 + K_3(\theta - \theta_0)^3 + K_4(\theta - \theta_0)^4 \\
E_{bb} &= M(r_{ij} - r_1)(r_{jk} - r_2) \\
E_{ba} &= N_1(r_{ij} - r_1)(\theta - \theta_0) + N_2(r_{jk} - r_2)(\theta - \theta_0)
\end{aligned}
$$

where $E_a$ is the angle term, $E_{bb}$ is a bond-bond term, and $E_{ba}$ is a bond-angle term. $\theta_0$ is the equilibrium angle and $r_1$ and $r_2$ are the equilibrium bond lengths.

See *(Sun)* for a description of the COMPASS class2 force field.

Coefficients for the $E_a$, $E_{bb}$, and $E_{ba}$ formulas must be defined for each angle type via the *angle_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands.

These are the 4 coefficients for the $E_a$ formula:

- $\theta_0$ (degrees)
- $K_2$ (energy)
- $K_3$ (energy)
- $K_4$ (energy)

$\theta_0$ is specified in degrees, but LAMMPS converts it to radians internally; hence the various $K$ are effectively energy per radian^2 or radian^3 or radian^4.

For the $E_{bb}$ formula, each line in a *angle_coeff* command in the input script lists 4 coefficients, the first of which is "bb" to indicate they are BondBond coefficients. In a data file, these coefficients should be listed under a "BondBond Coeffs" heading and you must leave out the "bb", i.e. only list 3 coefficients after the angle type.

- bb
- $M$ (energy/distance^2)
- $r_1$ (distance)
- $r_2$ (distance)

For the $E_{ba}$ formula, each line in a *angle_coeff* command in the input script lists 5 coefficients, the first of which is "ba" to indicate they are BondAngle coefficients. In a data file, these coefficients should be listed under a "BondAngle Coeffs" heading and you must leave out the "ba", i.e. only list 4 coefficients after the angle type.

- ba
- $N_1$ (energy/distance)
- $N_2$ (energy/distance)
- $r_1$ (distance)
- $r_2$ (distance)

The $\theta_0$ value in the $E_{ba}$ formula is not specified, since it is the same value from the $E_a$ formula.

**Note**

It is important that the order of the I,J,K atoms in each angle listed in the Angles section of the data file read by the *read_data* command be consistent with the order of the $r_1$ and $r_2$ BondBond and BondAngle coefficients. This is because the terms in the formulas for $E_{bb}$ and $E_{ba}$ will use the I,J atoms to compute $r_{ij}$ and the J,K atoms to compute $r_{jk}$.

---

Added in version 30Mar2026.

The *class2xe* angle style uses the potential

$$
\begin{aligned}
E_a =& K_2(\theta-\theta_0)^2 + K_3(\theta-\theta_0)^3 + K_4(\theta-\theta_0)^4 \\
E_{bb} =& D\left[1-e^{-\alpha(r_{ij}-r_1)}\right]\left[1-e^{-\alpha(r_{jk}-r_2)}\right] \\
E_{ba} =& D_1\left[1-e^{-\alpha_1(r_{ij}-r_1)}\right][\theta-\theta_0] + D_2\left[1-e^{-\alpha_2(r_{jk}-r_2)}\right][\theta-\theta_0]
\end{aligned}
$$

where $E_a$ is the angle term, $E_{bb}$ is a bond-bond term ($D$ is the dissociation energy), and $E_{ba}$ is a bond-angle term ($D_1$ and $D_2$ are the dissociation energies). $\theta_0$ is the equilibrium angle and $r_1$ and $r_2$ are the equilibrium bond lengths.

See *(Kemppainen)* for a description of the ClassII-xe force field and see *Howto bioFF* page for a motivation for the ClassII-xe force field.

**Note**

The *class2xe* angle style only describes the dissociation of a bond stretch. However once a bond is dissociated and stretched beyond the processor communication cutoff distance in parallel, the simulation will crash with atoms missing errors. This is often after the material fractures and thus for post-fracture phenomena the bonded interactions need to be removed for proper parallel communication.

To disconnect the dissociated bond and remove higher order interactions (angles, dihedrals, and impropers) the following LAMMPS commands can be used with the *class2xe* angle style *fix bond/react* or *fix bond/break*. See the *Howto bioFF* page for more details.

Coefficients for the $E_a$, $E_{bb}$, and $E_{ba}$ formulas must be defined for each angle type via the *angle_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands.

These are the 4 coefficients for the $E_a$ formula:

- $\theta_0$ (degrees)
- $K_2$ (energy)
- $K_3$ (energy)
- $K_4$ (energy)

$\theta_0$ is specified in degrees, but LAMMPS converts it to radians internally; hence the various $K$ are effectively energy per radian^2 or radian^3 or radian^4.

For the $E_{bb}$ formula, each line in a *angle_coeff* command in the input script lists 5 coefficients, the first of which is "bb" to indicate they are BondBond coefficients. In a data file, these coefficients should be listed under a "BondBond Coeffs" heading and you must leave out the "bb", i.e. only list 4 coefficients after the angle type.

- bb
- $D$ (energy)
- $\alpha$ (inverse distance)
- $r_1$ (distance)
- $r_2$ (distance)

For the $E_{ba}$ formula, each line in a *angle_coeff* command in the input script lists 7 coefficients, the first of which is "ba" to indicate they are BondAngle coefficients. In a data file, these coefficients should be listed under a "BondAngle Coeffs" heading and you must leave out the "ba", i.e. only list 6 coefficients after the angle type.

- ba
- $D_1$ (energy)
- $D_2$ (energy)
- $\alpha_1$ (inverse distance)
- $\alpha_2$ (inverse distance)
- $r_1$ (distance)
- $r_2$ (distance)

The $\theta_0$ value in the $E_{ba}$ formula is not specified, since it is the same value from the $E_a$ formula.

---

The *class2/p6* angle style uses the *class2* potential expanded to sixth order:

$$E_a = K_2 (\theta - \theta_0)^2 + K_3 (\theta - \theta_0)^3 + K_4 (\theta - \theta_0)^4 + K_5 (\theta - \theta_0)^5 + K_6 (\theta - \theta_0)^6$$

In this expanded term 6 coefficients for the $E_a$ formula need to be set:

- $\theta_0$ (degrees)
- $K_2$ (energy)
- $K_3$ (energy)
- $K_4$ (energy)
- $K_5$ (energy)
- $K_6$ (energy)

$\theta_0$ is specified in degrees, but LAMMPS converts it to radians internally; hence the various $K$ are effectively energy per radian^2 or radian^3 or radian^4 or radian^5 or radian^6.

The bond-bond and bond-angle terms remain unchanged.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 6.5.4 Restrictions

The *class2* and *class2xe* angle styles can only be used if LAMMPS was built with the CLASS2 package. For the *class2/p6* angle style LAMMPS needs to be built with the MOFFF package. See the *Build package* doc page for more info.

### 6.5.5 Related commands

*angle_coeff*

### 6.5.6 Default

none

---

**(Sun)** Sun, J Phys Chem B 102, 7338-7364 (1998).

**(Kemppainen)** Kemppainen, npj Computational Materials 11, 341 (2025).

## 6.6 angle_style cosine command

Accelerator Variants: *cosine/omp*, *cosine/kk*

### 6.6.1 Syntax

```
angle_style cosine
```

### 6.6.2 Examples

```
angle_style cosine
angle_coeff * 75.0
```

### 6.6.3 Description

The *cosine* angle style uses the potential

$$E = K[1 + \cos(\theta)]$$

where $K$ is defined for each angle type.

The following coefficients must be defined for each angle type via the *angle_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 6.6.4 Restrictions

This angle style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* doc page for more info.

### 6.6.5 Related commands

*angle_coeff*

### 6.6.6 Default

none

## 6.7 angle_style cosine/buck6d command

Accelerator Variants: *cosine/buck6d/omp*

### 6.7.1 Syntax

```
angle_style cosine/buck6d
```

### 6.7.2 Examples

```
angle_style cosine/buck6d
angle_coeff 1  cosine/buck6d  1.978350  4  180.000000
```

### 6.7.3 Description

The *cosine/buck6d* angle style uses the potential

$$E = K\,[1 + \cos(n\theta - \theta_0)]$$

where $K$ is the energy constant, $n$ is the periodic multiplicity and $\theta_0$ is the equilibrium angle.

The coefficients must be defined for each angle type via the *angle_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands in the following order:

- $K$ (energy)
- $n$
- $\theta_0$ (degrees)

$\theta_0$ is specified in degrees, but LAMMPS converts it to radians internally.

Additional to the cosine term the *cosine/buck6d* angle style computes the short range (vdW) interaction belonging to the *pair_style buck6d* between the end atoms of the angle. For this reason this angle style only works in combination with the *pair_style buck6d* styles and needs the *special_bonds* 1-3 interactions to be weighted 0.0 to prevent double counting.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 6.7.4 Restrictions

*cosine/buck6d* can only be used in combination with the *pair_style buck6d* style and with a *special_bonds* 0.0 weighting of 1-3 interactions.

This angle style can only be used if LAMMPS was built with the MOFFF package. See the *Build package* doc page for more info.

### 6.7.5 Related commands

*angle_coeff*

### 6.7.6 Default

none

## 6.8 angle_style cosine/delta command

Accelerator Variants: *cosine/delta/kk*, *cosine/delta/omp*

### 6.8.1 Syntax

```
angle_style cosine/delta
```

### 6.8.2 Examples

```
angle_style cosine/delta
angle_coeff 2*4 75.0 100.0
```

### 6.8.3 Description

The *cosine/delta* angle style uses the potential

$$E = K[1 - \cos(\theta - \theta_0)]$$

where $\theta_0$ is the equilibrium value of the angle, and $K$ is a prefactor. Note that the usual 1/2 factor is included in $K$.

The following coefficients must be defined for each angle type via the *angle_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy)
- $\theta_0$ (degrees)

$\theta_0$ is specified in degrees, but LAMMPS converts it to radians internally.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 6.8.4 Restrictions

This angle style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 6.8.5 Related commands

*angle_coeff* , *angle_style cosine/squared*

### 6.8.6 Default

none

## 6.9 angle_style cosine/periodic command

Accelerator Variants: *cosine/periodic/kk*, *cosine/periodic/omp*

### 6.9.1 Syntax

```
angle_style cosine/periodic
```

### 6.9.2 Examples

```
angle_style cosine/periodic
angle_coeff * 75.0 1 6
```

### 6.9.3 Description

The *cosine/periodic* angle style uses the following potential, which may be particularly used for organometallic systems where $n$ = 4 might be used for an octahedral complex and $n$ = 3 might be used for a trigonal center:

$$E = \frac{2.0}{n^2} * C \left[1 - B(-1)^n \cos\left(n\theta\right)\right]$$

where $C$, $B$ and $n$ are coefficients defined for each angle type.

The following coefficients must be defined for each angle type via the *angle_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $C$ (energy)
- $B$ = 1 or -1
- $n$ = 1, 2, 3, 4, 5 or 6 for periodicity

Note that the prefactor $C$ is specified as coefficient and not the overall force constant $K = \frac{2C}{n^2}$. When $B = 1$, it leads to a minimum for the linear geometry. When $B = -1$, it leads to a maximum for the linear geometry.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 6.9.4 Restrictions

This angle style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 6.9.5 Related commands

*angle_coeff*

### 6.9.6 Default

none

## 6.10 angle_style cosine/shift command

Accelerator Variants: *cosine/shift/kk*, *cosine/shift/omp*

### 6.10.1 Syntax

```
angle_style cosine/shift
```

### 6.10.2 Examples

```
angle_style cosine/shift
angle_coeff * 10.0 45.0
```

### 6.10.3 Description

The *cosine/shift* angle style uses the potential

$$E = -\frac{U_{\min}}{2}\left[1 + \cos(\theta - \theta_0)\right]$$

where $\theta_0$ is the equilibrium angle. The potential is bounded between $-U_{\min}$ and zero. In the neighborhood of the minimum $E = -U_{\min} + U_{\min}/4(\theta - \theta_0)^2$ hence the spring constant is $\frac{U_{\min}}{2}$.

The following coefficients must be defined for each angle type via the *angle_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $U_{\min}$ (energy)
- $\theta$ (angle)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 6.10.4 Restrictions

This angle style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 6.10.5 Related commands

*angle_coeff* , *angle_style cosine/shift/exp*

### 6.10.6 Default

none

## 6.11 angle_style cosine/shift/exp command

Accelerator Variants: *cosine/shift/exp/kk*, *cosine/shift/exp/omp*

### 6.11.1 Syntax

```
angle_style cosine/shift/exp
```

### 6.11.2 Examples

```
angle_style cosine/shift/exp
angle_coeff * 10.0 45.0 2.0
```

### 6.11.3 Description

The *cosine/shift/exp* angle style uses the potential

$$E = -U_{\min}\frac{e^{-aU(\theta,\theta_0)}-1}{e^{a}-1} \quad \text{with} \quad U(\theta,\theta_0) = -0.5\,(1+\cos(\theta-\theta_0))$$

where $U_{\min}$, $\theta$, and $a$ are defined for each angle type.

The potential is bounded between $[-U_{\min}, 0]$ and the minimum is located at the angle $\theta_0$. The a parameter can be both positive or negative and is used to control the spring constant at the equilibrium.

The spring constant is given by $k = A\exp(A)U_{\min}/[2(\exp(a)-1)]$. For $a > 3$, $\frac{k}{U_{\min}} = \frac{a}{2}$ to better than 5% relative error. For negative values of the $a$ parameter, the spring constant is essentially zero, and anharmonic terms takes over. The potential is furthermore well behaved in the limit $a \to 0$, where it has been implemented to linear order in $a$ for $a < 0.001$. In this limit the potential reduces to the cosineshifted potential.

The following coefficients must be defined for each angle type via the *angle_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $U_min$ (energy)

- $\theta$ (angle)
- $A$ (real number)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 6.11.4 Restrictions

This angle style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 6.11.5 Related commands

*angle_coeff*, *angle_style cosine/shift*, *dihedral_style cosine/shift/exp*

### 6.11.6 Default

none

## 6.12 angle_style cosine/squared command

Accelerator Variants: *cosine/squared/kk*, *cosine/squared/omp*

### 6.12.1 Syntax

```
angle_style cosine/squared
```

### 6.12.2 Examples

```
angle_style cosine/squared
angle_coeff 2*4 75.0 100.0
```

### 6.12.3 Description

The *cosine/squared* angle style uses the potential

$$E = K[\cos(\theta) - \cos(\theta_0)]^2$$

, which is commonly used in the *DREIDING* force field, where $\theta_0$ is the equilibrium value of the angle, and $K$ is a prefactor. Note that the usual 1/2 factor is included in $K$.

See *(Mayo)* for a description of the DREIDING force field.

The following coefficients must be defined for each angle type via the *angle_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy)
- $\theta_0$ (degrees)

$\theta_0$ is specified in degrees, but LAMMPS converts it to radians internally.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 6.12.4 Restrictions

This angle style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* doc page for more info.

### 6.12.5 Related commands

*angle_coeff*

### 6.12.6 Default

none

---

**(Mayo)** Mayo, Olfason, Goddard III, J Phys Chem, 94, 8897-8909 (1990).

## 6.13 angle_style cosine/squared/restricted command

Accelerator Variants: *cosine/squared/restricted/kk*, *cosine/squared/restricted/omp*

### 6.13.1 Syntax

```
angle_style cosine/squared/restricted
```

### 6.13.2 Examples

```
angle_style cosine/squared/restricted
angle_coeff 2*4 75.0 100.0
```

### 6.13.3 Description

Added in version 17Apr2024.

The *cosine/squared/restricted* angle style uses the potential

$$E = K[\cos(\theta) - \cos(\theta_0)]^2 / \sin^2(\theta)$$

, which is commonly used in the MARTINI force field, where $\theta_0$ is the equilibrium value of the angle, and $K$ is a prefactor. Note that the usual 1/2 factor is included in $K$.

See *(Bulacu)* for a description of the restricted angle for the MARTINI force field.

The following coefficients must be defined for each angle type via the *angle_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy)
- $\theta_0$ (degrees)

$\theta_0$ is specified in degrees, but LAMMPS converts it to radians internally.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 6.13.4 Restrictions

This angle style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 6.13.5 Related commands

*angle_coeff*

### 6.13.6 Default

none

---

**(Bulacu)** Bulacu, Goga, Zhao, Rossi, Monticelli, Periole, Tieleman, Marrink, J Chem Theory Comput, 9, 3282-3292 (2013).

## 6.14 angle_style cross command

Accelerator Variants: *cross/kk*, *cross/omp*

### 6.14.1 Syntax

```
angle_style cross
```

### 6.14.2 Examples

```
angle_style cross
angle_coeff 1 200.0 100.0 100.0 1.25 1.25 107.0
```

### 6.14.3 Description

The *cross* angle style uses a potential that couples the bond stretches of a bend with the angle stretch of that bend:

$$E = K_{SS}\left(r_{12} - r_{12,0}\right)\left(r_{32} - r_{32,0}\right) + K_{BS0}\left(r_{12} - r_{12,0}\right)\left(\theta - \theta_0\right) + K_{BS1}\left(r_{32} - r_{32,0}\right)\left(\theta - \theta_0\right)$$

where $r_{12,0}$ is the rest value of the bond length between atom 1 and 2, $r_{32,0}$ is the rest value of the bond length between atom 3 and 2, and $\theta_0$ is the rest value of the angle. $K_{SS}$ is the force constant of the bond stretch-bond stretch term and $K_{BS0}$ and $K_{BS1}$ are the force constants of the bond stretch-angle stretch terms.

The following coefficients must be defined for each angle type via the *angle_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K_{SS}$ (energy/distance^2)
- $K_{BS0}$ (energy/distance)
- $K_{BS1}$ (energy/distance)
- $r_{12,0}$ (distance)
- $r_{32,0}$ (distance)
- $\theta_0$ (degrees)

$\theta_0$ is specified in degrees, but LAMMPS converts it to radians internally; hence the $K_{BS0}$ and $K_{BS1}$ are effectively energy/distance per radian.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 6.14.4 Restrictions

This angle style can only be used if LAMMPS was built with the YAFF package. See the *Build package* doc page for more info.

### 6.14.5 Related commands

*angle_coeff*

### 6.14.6 Default

none

## 6.15 angle_style dipole command

Accelerator Variants: *dipole/kk*, *dipole/omp*

### 6.15.1 Syntax

```
angle_style dipole
```

### 6.15.2 Examples

```
angle_style dipole
angle_coeff 6 2.1 180.0
```

### 6.15.3 Description

The *dipole* angle style is used to control the orientation of a dipolar atom within a molecule *(Orsi)*. Specifically, the *dipole* angle style restrains the orientation of a point dipole $\mu_j$ (embedded in atom $j$) with respect to a reference (bond) vector $\vec{r}_{ij} = \vec{r}_i - \vec{r}_j$, where $i$ is another atom of the same molecule (typically, $i$ and $j$ are also covalently bonded).

It is convenient to define an angle gamma between the 'free' vector $\vec{\mu}_j$ and the reference (bond) vector $\vec{r}_{ij}$:

$$\cos\gamma = \frac{\vec{\mu}_j \cdot \vec{r}_{ij}}{\mu_j\, r_{ij}}$$

The *dipole* angle style uses the potential:

$$E = K(\cos\gamma - \cos\gamma_0)^2$$

where $K$ is a rigidity constant and gamma0 is an equilibrium (reference) angle.

The torque on the dipole can be obtained by differentiating the potential using the 'chain rule' as in appendix C.3 of *(AllenTildesley)*:

$$\vec{T}_j = \frac{2K(\cos\gamma - \cos\gamma_0)}{\mu_j\, r_{ij}}\, \vec{r}_{ij} \times \vec{\mu}_j$$

Example: if $\gamma_0$ is set to 0 degrees, the torque generated by the potential will tend to align the dipole along the reference direction defined by the (bond) vector $\vec{r}_{ij}$ (in other words, $\vec{\mu}_j$ is restrained to point towards atom $i$).

The dipolar torque $\vec{T}_j$ must be counterbalanced in order to conserve the local angular momentum. This is achieved via an additional force couple generating a torque equivalent to the opposite of $\vec{T}_j$:

$$-\vec{T}_j = \vec{r}_{ij} \times \vec{F}_i$$
$$\vec{F}_j = -\vec{F}_i$$

where $\vec{F}_i$ and $\vec{F}_j$ are applied on atoms $i$ and $j$, respectively.

The following coefficients must be defined for each angle type via the *angle_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy)
- $\gamma_0$ (degrees)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 6.15.4 Restrictions

This angle style can only be used if LAMMPS was built with the DIPOLE package. See the *Build package* doc page for more info.

**Note**

In the "Angles" section of the data file, the atom ID $j$ defining the direction of the dipole vector to restrain must come before the atom ID of the reference atom $i$. A third atom ID $k$ must also be provided to comply with the requirement of a valid angle definition. This atom ID $k$ should be chosen to be that of an atom bonded to atom $i$ to avoid errors with "lost angle atoms" when running in parallel. Since the LAMMPS code checks for valid angle definitions, cannot use the same atom ID of either $i$ or $j$ (this was allowed and recommended with older LAMMPS versions).

The *newton* command for intramolecular interactions must be "on" (which is the default except when using some accelerator packages).

**Note**

This angle style should **NOT** be used with fix shake.

### 6.15.5 Related commands

*angle_coeff* , *angle_hybrid*

### 6.15.6 Default

none

---

**(Orsi)** Orsi & Essex, The ELBA force field for coarse-grain modeling of lipid membranes, PloS ONE 6(12): e28637, 2011.

**(AllenTildesley)** Allen & Tildesley, Computer Simulation of Liquids, Clarendon Press, Oxford, 1987.

## 6.16 angle_style fourier command

Accelerator Variants: *fourier/kk*, *fourier/omp*

### 6.16.1 Syntax

```
angle_style fourier
```

### 6.16.2 Examples

```
angle_style fourier
angle_coeff 75.0 1.0 1.0 1.0
```

### 6.16.3 Description

The *fourier* angle style uses the potential

$$E = K[C_0 + C_1 \cos(\theta) + C_2 \cos(2\theta)]$$

The following coefficients must be defined for each angle type via the *angle_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy)
- $C_0$ (real)
- $C_1$ (real)
- $C_2$ (real)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 6.16.4 Restrictions

This angle style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 6.16.5 Related commands

*angle_coeff*

### 6.16.6 Default

none

## 6.17 angle_style fourier/simple command

Accelerator Variants: *fourier/simple/kk*, *fourier/simple/omp*

### 6.17.1 Syntax

```
angle_style fourier/simple
```

### 6.17.2 Examples

```
angle_style fourier/simple
angle_coeff 100.0 -1.0 1.0
```

### 6.17.3 Description

The *fourier/simple* angle style uses the potential

$$E = K[1.0 + c\cos(n\theta)]$$

The following coefficients must be defined for each angle type via the *angle_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy)
- $c$ (real)
- $n$ (real)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 6.17.4 Restrictions

This angle style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 6.17.5 Related commands

*angle_coeff*

### 6.17.6 Default

none

## 6.18 angle_style gaussian command

Accelerator Variants: *gaussian/kk*, *gaussian/omp*

### 6.18.1 Syntax

```
angle_style gaussian
```

### 6.18.2 Examples

```
angle_style gaussian
angle_coeff 1 300.0 2 0.0128 0.375 80.0 0.0730 0.148 123.0
```

### 6.18.3 Description

The *gaussian* angle style uses the potential:

$$E = -k_B T ln \left( \sum_{i=1}^{n} \frac{A_i}{w_i \sqrt{\pi/2}} exp \left( \frac{-2(\theta - \theta_i)^2}{w_i^2} \right) \right)$$

This analytical form is a suitable potential for obtaining mesoscale effective force fields which can reproduce target atomistic distributions *(Milano)*.

The following coefficients must be defined for each angle type via the *angle_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $T$ temperature at which the potential was derived
- $n$ (integer >=1)
- $A_1$ (> 0, radians)
- $w_1$ (> 0, radians)
- $\theta_1$ (degrees)
- ...
- $A_n$ (> 0, radians)
- $w_n$ (> 0, radians)
- $\theta_n$ (degrees)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 6.18.4 Restrictions

This angle style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 6.18.5 Related commands

*angle_coeff*

### 6.18.6 Default

none

---

**(Milano)** G. Milano, S. Goudeau, F. Mueller-Plathe, J. Polym. Sci. B Polym. Phys. 43, 871 (2005).

## 6.19 angle_style harmonic command

Accelerator Variants: *harmonic/intel*, *harmonic/kk*, *harmonic/omp*

### 6.19.1 Syntax

```
angle_style harmonic
```

### 6.19.2 Examples

```
angle_style harmonic
angle_coeff 1 300.0 107.0
```

### 6.19.3 Description

The *harmonic* angle style uses the potential

$$E = K(\theta - \theta_0)^2$$

where $\theta_0$ is the equilibrium value of the angle, and $K$ is a prefactor. Note that the usual 1/2 factor is included in $K$.

The following coefficients must be defined for each angle type via the *angle_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy)
- $\theta_0$ (degrees)

$\theta_0$ is specified in degrees, but LAMMPS converts it to radians internally; hence $K$ is effectively energy per radian^2.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 6.19.4 Restrictions

This angle style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* doc page for more info.

### 6.19.5 Related commands

*angle_coeff*

### 6.19.6 Default

none

## 6.20 angle_style hybrid command

Accelerator Variants: *hybrid/kk*

### 6.20.1 Syntax

```
angle_style hybrid style1 style2 ...
```

- style1,style2 = list of one or more angle styles

### 6.20.2 Examples

```
angle_style hybrid harmonic cosine
angle_coeff 1 harmonic 80.0 30.0
angle_coeff 2* cosine 50.0
```

### 6.20.3 Description

The *hybrid* style enables the use of multiple angle styles in one simulation. An angle style is assigned to each angle type. For example, angles in a polymer flow (of angle type 1) could be computed with a *harmonic* potential and angles in the wall boundary (of angle type 2) could be computed with a *cosine* potential. The assignment of angle type to style is made via the *angle_coeff* command or in the data file.

In the *angle_coeff* commands, the name of an angle style must be added after the angle type, with the remaining coefficients being those appropriate to that style. In the example above, the 2 angle_coeff commands set angles of angle type 1 to be computed with a *harmonic* potential with coefficients 80.0, 30.0 for $K$, $\theta_0$. All other angle types $(2 - N)$ are computed with a *cosine* potential with coefficient 50.0 for $K$.

If angle coefficients are specified in the data file read via the *read_data* command, then the same rule applies. E.g. "harmonic" or "cosine", must be added after the angle type, for each line in the "Angle Coeffs" section, e.g.

```
Angle Coeffs

1 harmonic 80.0 30.0
2 cosine 50.0
...
```

If *class2* is one of the angle hybrid styles, the same rule holds for specifying additional BondBond (and BondAngle) coefficients either via the input script or in the data file. I.e. *class2* must be added to each line after the angle type. For lines in the BondBond (or BondAngle) section of the data file for angle types that are not *class2*, you must use an angle style of *skip* as a placeholder, e.g.

```
BondBond Coeffs

1 skip
2 class2 3.6512 1.0119 1.0119
...
```

Note that it is not necessary to use the angle style *skip* in the input script, since BondBond (or BondAngle) coefficients need not be specified at all for angle types that are not *class2*.

An angle style of *none* with no additional coefficients can be used in place of an angle style, either in a input script *angle_coeff* command or in the data file, if you desire to turn off interactions for specific angle types.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 6.20.4 Restrictions

This angle style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* doc page for more info.

Unlike other angle styles, the hybrid angle style does not store angle coefficient info for individual sub-styles in *binary restart files* or *data files*. Thus when restarting a simulation, you need to re-specify the angle_coeff commands.

### 6.20.5 Related commands

*angle_coeff*

### 6.20.6 Default

none

## 6.21 angle_style lepton command

Accelerator Variants: *lepton/omp*

### 6.21.1 Syntax

```
angle_style style args
```

- style = *lepton*
- args = optional arguments

```
args = auto_offset or no_offset
  auto_offset = offset the potential energy so that the value at theta0 is 0.0 (default)
  no_offset = do not offset the potential energy
```

### 6.21.2 Examples

```
angle_style lepton
angle_style lepton no_offset

angle_coeff  1  120.0  "k*theta^2; k=250.0"
angle_coeff  2   90.0  "k2*theta^2 + k3*theta^3 + k4*theta^4; k2=300.0; k3=-100.0; k4=50.
↪0"
angle_coeff  3  109.47 "k*theta^2; k=350.0"
```

### 6.21.3 Description

Added in version 8Feb2023.

Angle style *lepton* computes angular interactions between three atoms with a custom potential function. The potential function must be provided as an expression string using "theta" as the angle variable relative to the reference angle $\theta_0$ which is provided as an angle coefficient. For example *"200.0*theta^2"* represents a *harmonic angle* potential with a force constant $K$ of 200.0 energy units:

$$U_{angle,i} = K(\theta_i - \theta_0)^2 = K\theta^2 \qquad \theta = \theta_i - \theta_0$$

Changed in version 7Feb2024.

By default the potential energy U is shifted so that the value U is 0.0 for $theta = theta_0$. This is equivalent to using the optional keyword *auto_offset*. When using the keyword *no_offset* instead, the potential energy is not shifted.

The Lepton library, that the *lepton* angle style interfaces with, evaluates this expression string at run time to compute the pairwise energy. It also creates an analytical representation of the first derivative of this expression with respect to "theta" and then uses that to compute the force between the angle atoms as defined by the topology data.

The following coefficients must be defined for each angle type via the *angle_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- Lepton expression (energy units)
- $\theta_0$ (degrees)

The Lepton expression must be either enclosed in quotes or must not contain any whitespace so that LAMMPS recognizes it as a single keyword. More on valid Lepton expressions below. The $\theta_0$ coefficient is the "equilibrium angle". It is entered in degrees, but internally converted to radians. Thus the expression must assume "theta" is in radians. The potential energy function in the Lepton expression is shifted in such a way, that the potential energy is 0 for a angle $\theta_i == \theta_0$.

---

### 6.21.4 Lepton expression syntax and features

Lepton supports the following operators in expressions:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| + | Add | - | Subtract | * | Multiply | / | Divide | ^ | Power |

The following mathematical functions are available:

| | | | |
|---|---|---|---|
| sqrt(x) | Square root | exp(x) | Exponential |
| log(x) | Natural logarithm | sin(x) | Sine (angle in radians) |
| cos(x) | Cosine (angle in radians) | sec(x) | Secant (angle in radians) |
| csc(x) | Cosecant (angle in radians) | tan(x) | Tangent (angle in radians) |
| cot(x) | Cotangent (angle in radians) | asin(x) | Inverse sine (in radians) |
| acos(x) | Inverse cosine (in radians) | atan(x) | Inverse tangent (in radians) |
| sinh(x) | Hyperbolic sine | cosh(x) | Hyperbolic cosine |
| tanh(x) | Hyperbolic tangent | erf(x) | Error function |
| erfc(x) | Complementary Error function | abs(x) | Absolute value |
| min(x,y) | Minimum of two values | max(x,y) | Maximum of two values |
| delta(x) | delta(x) is 1 for $x = 0$, otherwise 0 | step(x) | step(x) is 0 for $x < 0$, otherwise 1 |

Numbers may be given in either decimal or exponential form. All of the following are valid numbers: *5*, *-3.1*, *1e6*, and *3.12e-2*.

As an extension to the standard Lepton syntax, it is also possible to use LAMMPS *variables* in the format "v_name". Before evaluating the expression, "v_name" will be replaced with the value of the variable "name". This is compatible with all kinds of scalar variables, but not with vectors, arrays, local, or per-atom variables. If necessary, a custom scalar variable needs to be defined that can access the desired (single) item from a non-scalar variable. As an example, the following lines will instruct LAMMPS to ramp the force constant for a harmonic bond from 100.0 to 200.0 during the next run:

```
variable fconst equal ramp(100.0, 200)
bond_style lepton
bond_coeff 1 1.5 "v_fconst * (r^2)"
```

An expression may be followed by definitions for intermediate values that appear in the expression. A semicolon ";" is used as a delimiter between value definitions. For example, the expression:

```
a^2+a*b+b^2; a=a1+a2; b=b1+b2
```

is exactly equivalent to

```
(a1+a2)^2+(a1+a2)*(b1+b2)+(b1+b2)^2
```

The definition of an intermediate value may itself involve other intermediate values. Whitespace and quotation characters (’’ and “”) are ignored. All uses of a value must appear *before* that value’s definition. For efficiency reasons, the expression string is parsed, optimized, and then stored in an internal, pre-parsed representation for evaluation.

Evaluating a Lepton expression is typically between 2.5 and 5 times slower than the corresponding compiled and optimized C++ code. If additional speed or GPU acceleration (via GPU or KOKKOS) is required, the interaction can be represented as a table. Suitable table files can be created either internally using the *pair_write* or *bond_write* command or through the Python scripts in the *tools/tabulate* folder.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 6.21.5 Restrictions

This angle style is part of the LEPTON package and only enabled if LAMMPS was built with this package. See the *Build package* page for more info.

### 6.21.6 Related commands

*angle_coeff* , *angle_style table*, *bond_style lepton*,:doc:*dihedral_style lepton <dihedral_lepton>*

### 6.21.7 Default

none

## 6.22 angle_style mesocnt command

### 6.22.1 Syntax

```
angle_style mesocnt
```

### 6.22.2 Examples

```
angle_style mesocnt
angle_coeff 1 buckling C 10 10 20.0
angle_coeff 4 harmonic C 8 4 10.0
angle_coeff 2 buckling custom 400.0 50.0 5.0
angle_coeff 1 harmonic custom 300.0
```

### 6.22.3 Description

Added in version 15Sep2022.

The *mesocnt* angle style uses the potential

$$E = K_{\mathrm{H}}\Delta\theta^2, \qquad |\Delta\theta| < \Delta\theta_{\mathrm{B}}$$
$$E = K_{\mathrm{H}}\Delta\theta_{\mathrm{B}}^2 + K_{\mathrm{B}}(\Delta\theta - \Delta\theta_{\mathrm{B}}), \qquad |\Delta\theta| \geq \Delta\theta_{\mathrm{B}}$$

where $\Delta\theta = \theta - \pi$ is the bending angle of the nanotube, $K_{\mathrm{H}}$ and $K_{\mathrm{B}}$ are prefactors for the harmonic and linear regime respectively and $\Delta\theta_{\mathrm{B}}$ is the buckling angle. Note that the usual 1/2 factor for the harmonic potential is included in $K_{\mathrm{H}}$.

The style implements parameterization presets of $K_{\mathrm{H}}$, $K_{\mathrm{B}}$ and $\Delta\theta_{\mathrm{B}}$ for mesoscopic simulations of carbon nanotubes based on the atomistic simulations of *(Srivastava)* and buckling considerations of *(Zhigilei)*.

The following coefficients must be defined for each angle type via the *angle_coeff* command as in the examples above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- mode = *buckling* or *harmonic*
- preset = *C* or *custom*
- additional parameters depending on preset

If mode *harmonic* is chosen, the potential is simply harmonic and does not switch to the linear term when the buckling angle is reached. In *buckling* mode, the full piecewise potential is used.

Preset *C* is for carbon nanotubes, and the additional parameters are:

- chiral index $n$ (unitless)
- chiral index $m$ (unitless)
- $r_0$ (distance)

Here, $r_0$ is the equilibrium distance of the bonds included in the angle, see *bond_style mesocnt*.

In harmonic mode with preset *custom*, the additional parameter is:

- $K_{\mathrm{H}}$ (energy)

Hence, this setting is simply a wrapper for *bond_style harmonic* with an equilibrium angle of 180 degrees.

In harmonic mode with preset *custom*, the additional parameters are:

- $K_{\mathrm{H}}$ (energy)
- $K_{\mathrm{B}}$ (energy)
- $\Delta\theta_{\mathrm{B}}$ (degrees)

$\Delta\theta_{\mathrm{B}}$ is specified in degrees, but LAMMPS converts it to radians internally; hence $K_{\mathrm{H}}$ is effectively energy per radian^2 and $K_{\mathrm{B}}$ is energy per radian.

---

In *buckling* mode, this angle style adds the *buckled* property to all atoms in the simulation, which is an integer flag indicating whether the bending angle at a given atom has exceeded $\Delta\theta_{\mathrm{B}}$. It can be accessed as an atomic variable, e.g. for custom dump commands, as *i_buckled*.

**Note**

If the initial state of the simulation contains buckled nanotubes and *pair_style mesocnt* is used, the *i_buckled* atomic variable needs to be initialized before the pair_style is defined by doing a *run 0* command straight after the angle_style command. See below for an example.

If CNTs are already buckled at the start of the simulation, this script will correctly initialize *i_buckled*:

```
angle_style mesocnt
angle_coeff 1 buckling C 10 10 20.0

run 0

pair_style mesocnt 60.0
pair_coeff * * C_10_10.mesocnt 1
```

### 6.22.4 Restrictions

This angle style can only be used if LAMMPS was built with the MOLECULE and MESONT packages. See the *Build package* doc page for more info.

### 6.22.5 Related commands

*angle_coeff*

### 6.22.6 Default

none

---

**(Srivastava)** Zhigilei, Wei, Srivastava, Phys. Rev. B 71, 165417 (2005).

**(Zhigilei)** Volkov and Zhigilei, ACS Nano 4, 6187 (2010).

## 6.23 angle_style mm3 command

Accelerator Variants: *mm3/kk*, *mm3/omp*

### 6.23.1 Syntax

```
angle_style mm3
```

### 6.23.2 Examples

```
angle_style mm3
angle_coeff 1 100.0 107.0
```

### 6.23.3 Description

The *mm3* angle style uses the potential that is anharmonic in the angle as defined in *(Allinger)*

$$E = K(\theta - \theta_0)^2 \left[1 - 0.014(\theta - \theta_0) + 5.6(10)^{-5}(\theta - \theta_0)^2 - 7.0(10)^{-7}(\theta - \theta_0)^3 + 9(10)^{-10}(\theta - \theta_0)^4\right]$$

where $\theta_0$ is the equilibrium value of the angle, and $K$ is a prefactor. The anharmonic prefactors have units $\mathrm{deg}^{-n}$, for example $-0.014\,\mathrm{deg}^{-1}$, $5.6 \cdot 10^{-5}\,\mathrm{deg}^{-2}$, ...

The following coefficients must be defined for each angle type via the *angle_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy)
- $\theta_0$ (degrees)

$\theta_0$ is specified in degrees, but LAMMPS converts it to radians internally; hence $K$ is effectively energy per radian^2.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 6.23.4 Restrictions

This angle style can only be used if LAMMPS was built with the YAFF package. See the *Build package* doc page for more info.

### 6.23.5 Related commands

*angle_coeff*

### 6.23.6 Default

none

## 6.24 angle_style mwlc command

Accelerator Variants: *mwlc/kk*, *mwlc/omp*

### 6.24.1 Syntax

```
angle_style mwlc
```

### 6.24.2 Examples

```
angle_style mwlc
angle_coeff * 25 1 10 1
```

### 6.24.3 Description

Added in version 4Feb2025.

The *mwlc* angle style models a meltable wormlike chain and can be used to model non-linear bending elasticity of polymers, e.g. DNA. *mwlc* uses a potential that is a canonical-ensemble superposition of a non-melted and a melted state *(Farrell)*. The potential is

$$E = -k_B T \log[q + q^m] + E_0,$$

where the non-melted and melted partition functions are

$$q = \exp[-k_1(1 + \cos\theta)/k_B T];$$
$$q^m = \exp[-(\mu + k_2(1 + \cos\theta))/k_B T].$$

$k_1$ is the bending elastic constant of the non-melted state, $k_2$ is the bending elastic constant of the melted state, $\mu$ is the melting energy, and $T$ is the reference temperature. The reference energy,

$$E_0 = -k_B T \log[1 + \exp[-\mu/k_B T]],$$

ensures that E is zero for a fully extended chain.

This potential is a continuous version of the two-state potential introduced by *(Yan)*.

The following coefficients must be defined for each angle type via the *angle_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $k_1$ (energy)
- $k_2$ (energy)
- $\mu$ (energy)
- $T$ (temperature)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 6.24.4 Restrictions

This angle style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 6.24.5 Related commands

*angle_coeff*

### 6.24.6 Default

none

---

**(Farrell)** Farrell, Dobnikar, Podgornik, Curk, Phys Rev Lett, 133, 148101 (2024).

**(Yan)** Yan, Marko, Phys Rev Lett, 93, 108108 (2004).

## 6.25 angle_style none command

### 6.25.1 Syntax

```
angle_style none
```

### 6.25.2 Examples

```
angle_style none
```

### 6.25.3 Description

Using an angle style of none means angle forces and energies are not computed, even if triplets of angle atoms were listed in the data file read by the *read_data* command.

See the *angle_style zero* command for a way to calculate angle statistics, but compute no angle interactions.

### 6.25.4 Restrictions

none

### 6.25.5 Related commands

*angle_style zero*

### 6.25.6 Default

none

## 6.26 angle_style quartic command

Accelerator Variants: *quartic/kk*, *quartic/omp*

### 6.26.1 Syntax

```
angle_style quartic
```

### 6.26.2 Examples

```
angle_style quartic
angle_coeff 1 129.1948 56.8726 -25.9442 -14.2221
```

### 6.26.3 Description

The *quartic* angle style uses the potential

$$E = K_2(\theta - \theta_0)^2 + K_3(\theta - \theta_0)^3 + K_4(\theta - \theta_0)^4$$

where $\theta_0$ is the equilibrium value of the angle, and $K$ is a prefactor. Note that the usual 1/2 factor is included in $K$.

The following coefficients must be defined for each angle type via the *angle_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $\theta_0$ (degrees)
- $K_2$ (energy)
- $K_3$ (energy)
- $K_4$ (energy)

$\theta_0$ is specified in degrees, but LAMMPS converts it to radians internally; hence the various $K$ are effectively energy per radian^2 or radian^3 or radian^4.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 6.26.4 Restrictions

This angle style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 6.26.5 Related commands

*angle_coeff*

### 6.26.6 Default

none

## 6.27 angle_style spica command

Accelerator Variants: *spica/omp*, *spica/kk*

### 6.27.1 Syntax

```
angle_style spica

angle_style spica/omp
```

### 6.27.2 Examples

```
angle_style spica
angle_coeff 1 300.0 107.0
```

### 6.27.3 Description

The *spica* angle style is a combination of the harmonic angle potential,

$$E = K(\theta - \theta_0)^2$$

where $\theta_0$ is the equilibrium value of the angle and $K$ a prefactor, with the *repulsive* part of the non-bonded *lj/spica* pair style between the atoms 1 and 3. This angle potential is intended for coarse grained MD simulations with the SPICA (formerly called SDK) parameterization using the *pair_style lj/spica*. Relative to the pair_style *lj/spica*, however, the energy is shifted by $\varepsilon$, to avoid sudden jumps. Note that the usual 1/2 factor is included in $K$.

The following coefficients must be defined for each angle type via the *angle_coeff* command as in the example above:

- $K$ (energy)
- $\theta_0$ (degrees)

$\theta_0$ is specified in degrees, but LAMMPS converts it to radians internally; hence $K$ is effectively energy per radian^2.

The required *lj/spica* parameters are extracted automatically from the pair_style.

Style *sdk*, the original implementation of style *spica*, is available for backward compatibility.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 6.27.4 Restrictions

This angle style can only be used if LAMMPS was built with the CG-SPICA package. See the *Build package* doc page for more info.

### 6.27.5 Related commands

*angle_coeff*, *angle_style harmonic*, *pair_style lj/spica*, *pair_style lj/spica/coul/long*

### 6.27.6 Default

none

## 6.28 angle_style table command

Accelerator Variants: *table/omp*

### 6.28.1 Syntax

```
angle_style table style N
```

- style = *linear* or *spline* = method of interpolation
- N = use N values in table

### 6.28.2 Examples

```
angle_style table linear 1000
angle_coeff 3 file.table ENTRY1
```

### 6.28.3 Description

Style *table* creates interpolation tables of length *N* from angle potential and derivative values listed in a file(s) as a function of angle The files are read by the *angle_coeff* command.

The interpolation tables are created by fitting cubic splines to the file values and interpolating energy and derivative values at each of *N* angles. During a simulation, these tables are used to interpolate energy and force values on individual atoms as needed. The interpolation is done in one of 2 styles: *linear* or *spline*.

For the *linear* style, the angle is used to find 2 surrounding table values from which an energy or its derivative is computed by linear interpolation.

For the *spline* style, a cubic spline coefficients are computed and stored at each of the *N* values in the table. The angle is used to find the appropriate set of coefficients which are used to evaluate a cubic polynomial which computes the energy or derivative.

The following coefficients must be defined for each angle type via the *angle_coeff* command as in the example above.

- filename
- keyword

The filename specifies a file containing tabulated energy and derivative values. The keyword specifies a section of the file. The format of this file is described below.

---

Suitable tables for use with this angle style can be created by LAMMPS itself from existing angle styles using the *angle_write* command. This can be useful to have a template file for testing the angle style settings and to build a compatible custom file. Another option to generate tables is the Python code in the `tools/tabulate` folder of the LAMMPS source code distribution.

The format of a tabulated file is as follows (without the parenthesized comments):

```
# Angle potential for harmonic (one or more comment or blank lines)

HAM                            (keyword is the first text on line)
```

(continues on next page)

(continued from previous page)

```
N 181 FP 0 0 EQ 90.0          (N, FP, EQ parameters)
                              (blank line)
1 0.0 200.5 2.5               (index, angle, energy, derivative)
2 1.0 198.0 2.5
...
181 180.0 0.0 0.0
```

A section begins with a non-blank line whose first character is not a "#"; blank lines or lines starting with "#" can be used as comments between sections. The first line begins with a keyword which identifies the section. The line can contain additional text, but the initial text must match the argument specified in the *angle_coeff* command. The next line lists (in any order) one or more parameters for the table. Each parameter is a keyword followed by one or more numeric values.

The parameter "N" is required and its value is the number of table entries that follow. Note that this may be different than the *N* specified in the *angle_style table* command. Let Ntable = *N* in the angle_style command, and Nfile = "N" in the tabulated file. What LAMMPS does is a preliminary interpolation by creating splines using the Nfile tabulated values as nodal points. It uses these to interpolate as needed to generate energy and derivative values at Ntable different points. The resulting tables of length Ntable are then used as described above, when computing energy and force for individual angles and their atoms. This means that if you want the interpolation tables of length Ntable to match exactly what is in the tabulated file (with effectively no preliminary interpolation), you should set Ntable = Nfile.

The "FP" parameter is optional. If used, it is followed by two values fplo and fphi, which are the second derivatives at the innermost and outermost angle settings. These values are needed by the spline construction routines. If not specified by the "FP" parameter, they are estimated (less accurately) by the first two and last two derivative values in the table.

The "EQ" parameter is also optional. If used, it is followed by a the equilibrium angle value, which is used, for example, by the *fix shake* command. If not used, the equilibrium angle is set to 180.0.

Following a blank line, the next N lines list the tabulated values. On each line, the first value is the index from 1 to N, the second value is the angle value (in degrees), the third value is the energy (in energy units), and the fourth is -dE/d(theta) (also in energy units). The third term is the energy of the 3-atom configuration for the specified angle. The last term is the derivative of the energy with respect to the angle (in degrees, not radians). Thus the units of the last term are still energy, not force. The angle values must increase from one line to the next. The angle values must also begin with 0.0 and end with 180.0, i.e. span the full range of possible angles.

Note that one file can contain many sections, each with a tabulated potential. LAMMPS reads the file section by section until it finds one that matches the specified keyword.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 6.28.4 Restart, fix_modify, output, run start/stop, minimize info

This angle style writes the settings for the "angle_style table" command to *binary restart files*, so a angle_style command does not need to specified in an input script that reads a restart file. However, the coefficient information is not stored in the restart file, since it is tabulated in the potential files. Thus, angle_coeff commands do need to be specified in the restart input script.

### 6.28.5 Restrictions

This angle style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* doc page for more info.

### 6.28.6 Related commands

*angle_coeff* , *angle_write*

### 6.28.7 Default

none

## 6.29 angle_style zero command

### 6.29.1 Syntax

```
angle_style zero keyword
```

- zero or more keywords may be appended
- keyword = *nocoeff*

### 6.29.2 Examples

```
angle_style zero
angle_style zero nocoeff
angle_coeff *
angle_coeff * 120.0
```

### 6.29.3 Description

Using an angle style of zero means angle forces and energies are not computed, but the geometry of angle triplets is still accessible to other commands.

As an example, the *compute angle/local* command can be used to compute the theta values for the list of triplets of angle atoms listed in the data file read by the *read_data* command. If no angle style is defined, this command cannot be used.

The optional *nocoeff* flag allows to read data files with AngleCoeff section for any angle style. Similarly, any *angle_coeff* commands will only be checked for the angle type number and the rest ignored.

Note that the *angle_coeff* command must be used for all angle types. If specified, there can be only one value, which is going to be used to assign an equilibrium angle, e.g. for use with *fix shake*.

### 6.29.4 Restrictions

none

### 6.29.5 Related commands

*angle_style none*

### 6.29.6 Default

none

CHAPTER

SEVEN

# DIHEDRAL STYLES

## 7.1 dihedral_style charmm command

Accelerator Variants: *charmm/intel*, *charmm/kk*, *charmm/omp*

## 7.2 dihedral_style charmmfsw command

Accelerator Variants: *charmmfsw/kk*, *charmmfsw/omp*

### 7.2.1 Syntax

```
dihedral_style style
```

- style = *charmm* or *charmmfsw*

### 7.2.2 Examples

```
dihedral_style charmm
dihedral_style charmmfsw
dihedral_coeff  1 0.2 1 180 1.0
dihedral_coeff  2 1.8 1   0 1.0
dihedral_coeff  1 3.1 2 180 0.5
```

### 7.2.3 Description

The *charmm* and *charmmfsw* dihedral styles use the potential

$$E = K[1 + \cos(n\phi - d)]$$

See *(MacKerell)* for a description of the CHARMM force field. This dihedral style can also be used for the AMBER force field (see comment on weighting factors below). See *(Cornell)* for a description of the AMBER force field.

**Note**

The newer *charmmfsw* style was released in March 2017. We recommend it be used instead of the older *charmm* style when running a simulation with the CHARMM force field, either with long-range Coulombics or a Coulombic cutoff, via the *pair_style lj/charmmfsw/coul/long* and *pair_style lj/charmmfsw/coul/charmmfsh* commands respectively. Otherwise the older *charmm* style is fine to use. See the discussion below and more details on the *pair_style charmm* doc page.

The following coefficients must be defined for each dihedral type via the *dihedral_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- *K* (energy)
- *n* (integer >= 0)
- *d* (integer value of degrees)
- weighting factor (1.0, 0.5, or 0.0)

The weighting factor is required to correct for double counting pairwise non-bonded Lennard-Jones interactions in cyclic systems or when using the CHARMM dihedral style with non-CHARMM force fields. With the CHARMM dihedral style, interactions between the first and fourth atoms in a dihedral are skipped during the normal non-bonded force computation and instead evaluated as part of the dihedral using special epsilon and sigma values specified with the *pair_coeff* command of pair styles that contain "lj/charmm" (e.g. *pair_style lj/charmm/coul/long*) In 6-membered rings, the same 1-4 interaction would be computed twice (once for the clockwise 1-4 pair in dihedral 1-2-3-4 and once in the counterclockwise dihedral 1-6-5-4) and thus the weighting factor has to be 0.5 in this case. In 4-membered or 5-membered rings, the 1-4 dihedral also is counted as a 1-2 or 1-3 interaction when going around the ring in the opposite direction and thus the weighting factor is 0.0, as the 1-2 and 1-3 exclusions take precedence.

Note that this dihedral weighting factor is unrelated to the scaling factor specified by the *special bonds* command which applies to all 1-4 interactions in the system. For CHARMM force fields, the special_bonds 1-4 interaction scaling factor should be set to 0.0. Since the corresponding 1-4 non-bonded interactions are computed with the dihedral. This means that if any of the weighting factors defined as dihedral coefficients (fourth coeff above) are non-zero, then you must use a pair style with "lj/charmm" and set the special_bonds 1-4 scaling factor to 0.0 (which is the default). Otherwise 1-4 non-bonded interactions in dihedrals will be computed twice.

For simulations using the CHARMM force field with a Coulombic cutoff, the difference between the *charmm* and *charmmfsw* styles is in the computation of the 1-4 non-bond interactions, though only if the distance between the two atoms is within the switching region of the pairwise potential defined by the corresponding CHARMM pair style, i.e. within the outer cutoff specified for the pair style. The *charmmfsw* style should only be used when using the corresponding *pair_style lj/charmmfsw/coul/charmmfsw* or *pair_style lj/charmmfsw/coul/long* commands. Use the *charmm* style with the older *pair_style* commands that have just "charmm" in their style name. See the discussion on the *CHARMM pair_style* page for details.

Note that for AMBER force fields, which use pair styles with "lj/cut", the special_bonds 1-4 scaling factor should be set to the AMBER defaults (1/2 and 5/6) and all the dihedral weighting factors (fourth coeff above) must be set to 0.0. In this case, you can use any pair style you wish, since the dihedral does not need any Lennard-Jones parameter information and will not compute any 1-4 non-bonded interactions. Likewise the *charmm* or *charmmfsw* styles are identical in this case since no 1-4 non-bonded interactions are computed.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 7.2.4 Restrictions

When using run_style *respa*, these dihedral styles must be assigned to the same r-RESPA level as *pair* or *outer*.

When used in combination with CHARMM pair styles, the 1-4 *special_bonds* scaling factors must be set to 0.0. Otherwise non-bonded contributions for these 1-4 pairs will be computed multiple times.

These dihedral styles can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* doc page for more info.

### 7.2.5 Related commands

*dihedral_coeff*, *pair_style lj/charmm variants*, *angle_style charmm*, *fix cmap*

### 7.2.6 Default

none

---

**(Cornell)** Cornell, Cieplak, Bayly, Gould, Merz, Ferguson, Spellmeyer, Fox, Caldwell, Kollman, JACS 117, 5179-5197 (1995).

**(MacKerell)** MacKerell, Bashford, Bellott, Dunbrack, Evanseck, Field, Fischer, Gao, Guo, Ha, et al, J Phys Chem B, 102, 3586 (1998).

## 7.3 dihedral_style class2 command

Accelerator Variants: *class2/omp*, *class2/kk*

## 7.4 dihedral_style class2xe command

Accelerator Variants: *class2xe/kk*, *class2xe/omp*

### 7.4.1 Syntax

```
dihedral_style class2
```

### 7.4.2 Examples

```
dihedral_style class2
dihedral_coeff 1 100 75 100 70 80 60
dihedral_coeff * mbt 3.5945 0.1704 -0.5490 1.5228
dihedral_coeff * ebt 0.3417 0.3264 -0.9036 0.1368 0.0 -0.8080 1.0119 1.1010
dihedral_coeff 2 at 0.0 -0.1850 -0.7963 -2.0220 0.0 -0.3991 110.2453 105.1270
dihedral_coeff * aat -13.5271 110.2453 105.1270
dihedral_coeff * bb13 0.0 1.0119 1.1010
```

### 7.4.3 Description

The *class2* dihedral style uses the potential

$$
\begin{aligned}
E =& E_d + E_{mbt} + E_{ebt} + E_{at} + E_{aat} + E_{bb13} \\
E_d =& \sum_{n=1}^{3} K_n[1 - \cos(n\phi - \phi_n)] \\
E_{mbt} =& (r_{jk} - r_2)[A_1 \cos(\phi) + A_2 \cos(2\phi) + A_3 \cos(3\phi)] \\
E_{ebt} =& (r_{ij} - r_1)[B_1 \cos(\phi) + B_2 \cos(2\phi) + B_3 \cos(3\phi)] + \\
& (r_{kl} - r_3)[C_1 \cos(\phi) + C_2 \cos(2\phi) + C_3 \cos(3\phi)] \\
E_{at} =& (\theta_{ijk} - \theta_1)[D_1 \cos(\phi) + D_2 \cos(2\phi) + D_3 \cos(3\phi)] + \\
& (\theta_{jkl} - \theta_2)[E_1 \cos(\phi) + E_2 \cos(2\phi) + E_3 \cos(3\phi)] \\
E_{aat} =& M(\theta_{ijk} - \theta_1)(\theta_{jkl} - \theta_2) \cos(\phi) \\
E_{bb13} =& N(r_{ij} - r_1)(r_{kl} - r_3)
\end{aligned}
$$

where $E_d$ is the dihedral term, $E_{mbt}$ is a middle-bond-torsion term, $E_{ebt}$ is an end-bond-torsion term, $E_{at}$ is an angle-torsion term, $E_{aat}$ is an angle-angle-torsion term, and $E_{bb13}$ is a bond-bond-13 term.

$\theta_1$ and $\theta_2$ are equilibrium angles and :math:$r_1$, $r_2$, and *r_3* are equilibrium bond lengths.

See *(Sun)* for a description of the COMPASS class2 force field.

Coefficients for the $E_d$, $E_{mbt}$, $E_{ebt}$, $E_{at}$, $E_{aat}$, and $E_{bb13}$ formulas must be defined for each dihedral type via the *dihedral_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands.

These are the 6 coefficients for the $E_d$ formula:

- $K_1$ (energy)
- $\phi_1$ (degrees)
- $K_2$ (energy)
- $\phi_2$ (degrees)
- $K_3$ (energy)
- $phi_3$ (degrees)

For the $E_{mbt}$ formula, each line in a *dihedral_coeff* command in the input script lists 5 coefficients, the first of which is *mbt* to indicate they are MiddleBondTorsion coefficients. In a data file, these coefficients should be listed under a *MiddleBondTorsion Coeffs* heading and you must leave out the *mbt*, i.e. only list 4 coefficients after the dihedral type.

- *mbt*
- $A_1$ (energy/distance)
- $A_2$ (energy/distance)
- $A_3$ (energy/distance)
- $r_2$ (distance)

For the $E_{ebt}$ formula, each line in a *dihedral_coeff* command in the input script lists 9 coefficients, the first of which is *ebt* to indicate they are EndBondTorsion coefficients. In a data file, these coefficients should be listed under a *EndBondTorsion Coeffs* heading and you must leave out the *ebt*, i.e. only list 8 coefficients after the dihedral type.

- *ebt*
- $B_1$ (energy/distance)
- $B_2$ (energy/distance)

- $B_3$ (energy/distance)
- $C_1$ (energy/distance)
- $C_2$ (energy/distance)
- $C_3$ (energy/distance)
- $r_1$ (distance)
- $r_3$ (distance)

For the $E_{at}$ formula, each line in a *dihedral_coeff* command in the input script lists 9 coefficients, the first of which is *at* to indicate they are AngleTorsion coefficients. In a data file, these coefficients should be listed under a *AngleTorsion Coeffs* heading and you must leave out the *at*, i.e. only list 8 coefficients after the dihedral type.

- *at*
- $D_1$ (energy)
- $D_2$ (energy)
- $D_3$ (energy)
- $E_1$ (energy)
- $E_2$ (energy)
- $E_3$ (energy)
- $\theta_1$ (degrees)
- $\theta_2$ (degrees)

$\theta_1$ and $\theta_2$ are specified in degrees, but LAMMPS converts them to radians internally; hence the various $D$ and $E$ are effectively energy per radian.

For the $E_{aat}$ formula, each line in a *dihedral_coeff* command in the input script lists 4 coefficients, the first of which is *aat* to indicate they are AngleAngleTorsion coefficients. In a data file, these coefficients should be listed under a *AngleAngleTorsion Coeffs* heading and you must leave out the *aat*, i.e. only list 3 coefficients after the dihedral type.

- *aat*
- $M$ (energy)
- $\theta_1$ (degrees)
- $\theta_2$ (degrees)

$\theta_1$ and $\theta_2$ are specified in degrees, but LAMMPS converts them to radians internally; hence $M$ is effectively energy per radian^2.

For the $E_{bb13}$ formula, each line in a *dihedral_coeff* command in the input script lists 4 coefficients, the first of which is *bb13* to indicate they are BondBond13 coefficients. In a data file, these coefficients should be listed under a *BondBond13 Coeffs* heading and you must leave out the *bb13*, i.e. only list 3 coefficients after the dihedral type.

- *bb13*
- $N$ (energy/distance^2)
- $r_1$ (distance)
- $r_3$ (distance)

---

Added in version 30Mar2026.

The *class2xe* dihedral style uses the potential

$$
\begin{aligned}
E_{mbt} &= \left[1 - e^{-\alpha_2(r_{jk}-r_2)}\right] \left[A_1 \cos(\phi) + A_2 \cos(2\phi) + A_3 \cos(3\phi)\right] \\
E_{ebt} &= \left[1 - e^{-\alpha_1(r_{ij}-r_1)}\right] \left[B_1 \cos(\phi) + B_2 \cos(2\phi) + B_3 \cos(3\phi)\right] + \\
&\quad \left[1 - e^{-\alpha_3(r_{kl}-r_3)}\right] \left[C_1 \cos(\phi) + C_2 \cos(2\phi) + C_3 \cos(3\phi)\right] \\
E_{bb13} &= D \left[1 - e^{-\alpha(r_{ij}-r_1)}\right] \left[1 - e^{-\alpha(r_{kl}-r_3)}\right]
\end{aligned}
$$

where $E_{mbt}$ is a middle-bond-torsion term, $E_{ebt}$ is an end-bond-torsion term, and $E_{bb13}$ is a bond-bond-13 term ($D$ is the dissociation energy).

$\theta_1$ and $\theta_2$ are equilibrium angles and :math:$r_1$, $r_2$, and *r_3* are equilibrium bond lengths.

See *(Kemppainen)* for a description of the ClassII-xe force field and see *Howto bioFF* page for a motivation for the ClassII-xe force field.

**Note**

The *class2xe* dihedral style only describes the dissociation of a bond stretch. However once a bond is dissociated and stretched beyond the processor communication cutoff distance in parallel, the simulation will crash with atoms missing errors. This is often after the material fractures and thus for post-fracture phenomena the bonded interactions need to be removed for proper parallel communication.

To disconnect the dissociated bond and remove higher order interactions (angles, dihedrals, and impropers) the following LAMMPS commands can be used with the *class2xe* dihedral style *fix bond/react* or *fix bond/break*. See the *Howto bioFF* page for more details.

Coefficients for the $E_{mbt}$, $E_{ebt}$, and $E_{bb13}$ formulas must be defined for each dihedral type via the *dihedral_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands.

For the $E_{mbt}$ formula, each line in a *dihedral_coeff* command in the input script lists 6 coefficients, the first of which is *mbt* to indicate they are MiddleBondTorsion coefficients. In a data file, these coefficients should be listed under a *MiddleBondTorsion Coeffs* heading and you must leave out the *mbt*, i.e. only list 5 coefficients after the dihedral type.

- *mbt*
- $\alpha_2$ (inverse distance)
- $A_1$ (energy/distance)
- $A_2$ (energy/distance)
- $A_3$ (energy/distance)
- $r_2$ (distance)

For the $E_{ebt}$ formula, each line in a *dihedral_coeff* command in the input script lists 11 coefficients, the first of which is *ebt* to indicate they are EndBondTorsion coefficients. In a data file, these coefficients should be listed under a *EndBondTorsion Coeffs* heading and you must leave out the *ebt*, i.e. only list 10 coefficients after the dihedral type.

- *ebt*
- $\alpha_1$ (inverse distance)
- $\alpha_3$ (inverse distance)
- $B_1$ (energy/distance)
- $B_2$ (energy/distance)
- $B_3$ (energy/distance)

- $C_1$ (energy/distance)
- $C_2$ (energy/distance)
- $C_3$ (energy/distance)
- $r_1$ (distance)
- $r_3$ (distance)

For the $E_{bb13}$ formula, each line in a *dihedral_coeff* command in the input script lists 5 coefficients, the first of which is *bb13* to indicate they are BondBond13 coefficients. In a data file, these coefficients should be listed under a *BondBond13 Coeffs* heading and you must leave out the *bb13*, i.e. only list 4 coefficients after the dihedral type.

- *bb13*
- $D$ (energy)
- $\alpha$ (inverse distance)
- $r_1$ (distance)
- $r_3$ (distance)

The dihedral, AngleTorsion and AngleAngleTorsion terms remain unchanged.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 7.4.4 Restrictions

This dihedral style can only be used if LAMMPS was built with the CLASS2 package. See the *Build package* doc page for more info.

### 7.4.5 Related commands

*dihedral_coeff*

### 7.4.6 Default

none

---

**(Sun)** Sun, J Phys Chem B 102, 7338-7364 (1998).

**(Kemppainen)** Kemppainen, npj Computational Materials 11, 341 (2025).

# 7.5 dihedral_style cosine/shift/exp command

Accelerator Variants: *cosine/shift/exp/kk*, *cosine/shift/exp/omp*

## 7.5.1 Syntax

```
dihedral_style cosine/shift/exp
```

## 7.5.2 Examples

```
dihedral_style cosine/shift/exp
dihedral_coeff 1 10.0 45.0 2.0
```

## 7.5.3 Description

The *cosine/shift/exp* dihedral style uses the potential

$$E = -U_{min}\frac{e^{-aU(\theta,\theta_0)}-1}{e^a-1} \quad \text{with} \quad U(\theta,\theta_0) = -0.5\,(1+\cos(\theta-\theta_0))$$

where $U_{min}$, $\theta$, and $a$ are defined for each dihedral type.

The potential is bounded between $[-U_{min} : 0]$ and the minimum is located at the angle $\theta_0$. The a parameter can be both positive or negative and is used to control the spring constant at the equilibrium.

The spring constant is given by $k = ae^a\frac{U_{min}}{2(e^a-1)}$. For $a > 3$ and $\frac{k}{U_{min}} = \frac{a}{2}$ to better than 5% relative error. For negative values of the a parameter, the spring constant is essentially zero, and anharmonic terms takes over. The potential is furthermore well behaved in the limit $a \to 0$, where it has been implemented to linear order in $a$ for $a < 0.001$.

The following coefficients must be defined for each dihedral type via the *dihedral_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $U_{min}$ (energy)
- $\theta$ (angle)
- $a$ (real number)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

## 7.5.4 Restrictions

This dihedral style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* doc page for more info.

### 7.5.5 Related commands

*dihedral_coeff*, *angle_style cosine/shift/exp*

### 7.5.6 Default

none

## 7.6 dihedral_style cosine/squared/restricted command

Accelerator Variants: *cosine/squared/restricted/kk*, *cosine/squared/restricted/omp*

### 7.6.1 Syntax

```
dihedral_style cosine/squared/restricted
```

### 7.6.2 Examples

```
dihedral_style cosine/squared/restricted
dihedral_coeff 1 10.0 120
```

### 7.6.3 Description

Added in version 17Apr2024.

The *cosine/squared/restricted* dihedral style uses the potential

$$E = K[\cos(\phi) - \cos(\phi_0)]^2 / \sin^2(\phi)$$

, which is commonly used in the MARTINI force field.

See *(Bulacu)* for a description of the restricted dihedral for the MARTINI force field.

The following coefficients must be defined for each dihedral type via the *dihedral_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy)
- $\phi_0$ (degrees)

$\phi_0$ is specified in degrees, but LAMMPS converts it to radians internally.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 7.6.4 Restrictions

This dihedral style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 7.6.5 Related commands

*dihedral_coeff*

### 7.6.6 Default

none

---

**(Bulacu)** Bulacu, Goga, Zhao, Rossi, Monticelli, Periole, Tieleman, Marrink, J Chem Theory Comput, 9, 3282-3292 (2013).

## 7.7 dihedral_style fourier command

Accelerator Variants: *fourier/intel*, *fourier/kk*, *fourier/omp*

### 7.7.1 Syntax

```
dihedral_style fourier
```

### 7.7.2 Examples

```
dihedral_style fourier
dihedral_coeff 1 3 -0.846200 3 0.0 7.578800 1 0 0.138000 2 -180.0
```

### 7.7.3 Description

The *fourier* dihedral style uses the potential:

$$E = \sum_{i=1,m} K_i[1.0 + \cos(n_i\phi - d_i)]$$

The following coefficients must be defined for each dihedral type via the *dihedral_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $m$ (integer >=1)
- $K_1$ (energy)
- $n_1$ (integer >= 0)
- $d_1$ (degrees)
- [...]
- $K_m$ (energy)
- $n_m$ (integer >= 0)
- $d_m$ (degrees)

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

### 7.7.4 Restrictions

This dihedral style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 7.7.5 Related commands

*dihedral_coeff*

### 7.7.6 Default

none

## 7.8 dihedral_style harmonic command

Accelerator Variants: *harmonic/intel*, *harmonic/kk*, *harmonic/omp*

### 7.8.1 Syntax

```
dihedral_style harmonic
```

### 7.8.2 Examples

```
dihedral_style harmonic
dihedral_coeff 1 80.0 1 2
```

### 7.8.3 Description

The *harmonic* dihedral style uses the potential

$$E = K[1 + d\cos(n\phi)]$$

The following coefficients must be defined for each dihedral type via the *dihedral_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy)
- $d$ (+1 or -1)
- $n$ (integer >= 0)

**Note**

Here are important points to take note of when defining LAMMPS dihedral coefficients for the harmonic style, so that they are compatible with how harmonic dihedrals are defined by other force fields:

- The LAMMPS convention is that the trans position = 180 degrees, while in some force fields trans = 0 degrees.
- Some force fields reverse the sign convention on $d$.
- Some force fields let $n$ be positive or negative which corresponds to $d = 1$ or $d = -1$ for the harmonic style.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 7.8.4 Restrictions

This dihedral style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* doc page for more info.

### 7.8.5 Related commands

*dihedral_coeff*

### 7.8.6 Default

none

## 7.9 dihedral_style helix command

Accelerator Variants: *helix/kk*, *helix/omp*

### 7.9.1 Syntax

```
dihedral_style helix
```

### 7.9.2 Examples

```
dihedral_style helix
dihedral_coeff 1 80.0 100.0 40.0
```

### 7.9.3 Description

The *helix* dihedral style uses the potential

$$E = A[1 - \cos(\theta)] + B[1 + \cos(3\theta)] + C[1 + \cos(\theta + \frac{\pi}{4})]$$

This coarse-grain dihedral potential is described in *(Guo)*. For dihedral angles in the helical region, the energy function is represented by a standard potential consisting of three minima, one corresponding to the trans (t) state and the other to gauche states (g+ and g-). The paper describes how the $A$, $B$ and, $C$ parameters are chosen so as to balance secondary (largely driven by local interactions) and tertiary structure (driven by long-range interactions).

The following coefficients must be defined for each dihedral type via the *dihedral_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $A$ (energy)
- $B$ (energy)
- $C$ (energy)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 7.9.4 Restrictions

This dihedral style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 7.9.5 Related commands

*dihedral_coeff*

### 7.9.6 Default

none

---

**(Guo)** Guo and Thirumalai, Journal of Molecular Biology, 263, 323-43 (1996).

## 7.10 dihedral_style hybrid command

Accelerator Variants: *hybrid/kk*

### 7.10.1 Syntax

```
dihedral_style hybrid style1 style2 ...
```

- style1,style2 = list of one or more dihedral styles

### 7.10.2 Examples

```
dihedral_style hybrid harmonic helix
dihedral_coeff 1 harmonic 6.0 1 3
dihedral_coeff 2* helix 10 10 10
```

### 7.10.3 Description

The *hybrid* style enables the use of multiple dihedral styles in one simulation. An dihedral style is assigned to each dihedral type. For example, dihedrals in a polymer flow (of dihedral type 1) could be computed with a *harmonic* potential and dihedrals in the wall boundary (of dihedral type 2) could be computed with a *helix* potential. The assignment of dihedral type to style is made via the *dihedral_coeff* command or in the data file.

In the dihedral_coeff commands, the name of a dihedral style must be added after the dihedral type, with the remaining coefficients being those appropriate to that style. In the example above, the 2 dihedral_coeff commands set dihedrals of dihedral type 1 to be computed with a *harmonic* potential with coefficients 6.0, 1, 3 for K, d, n. All other dihedral types (2-N) are computed with a *helix* potential with coefficients 10, 10, 10 for A, B, C.

If dihedral coefficients are specified in the data file read via the *read_data* command, then the same rule applies. E.g. "harmonic" or "helix", must be added after the dihedral type, for each line in the "Dihedral Coeffs" section, e.g.

```
Dihedral Coeffs

1 harmonic 6.0 1 3
2 helix 10 10 10
...
```

If *class2* is one of the dihedral hybrid styles, the same rule holds for specifying additional AngleTorsion (and EndBondTorsion, etc) coefficients either via the input script or in the data file. I.e. *class2* must be added to each line after the dihedral type. For lines in the AngleTorsion (or EndBondTorsion, etc) Coeffs section of the data file for dihedral types that are not *class2*, you must use an dihedral style of *skip* as a placeholder, e.g.

```
AngleTorsion Coeffs

1 skip
2 class2 1.0 1.0 1.0 3.0 3.0 3.0 30.0 50.0
...
```

Note that it is not necessary to use the dihedral style *skip* in the input script, since AngleTorsion (or EndBondTorsion, etc) coefficients need not be specified at all for dihedral types that are not *class2*.

A dihedral style of *none* with no additional coefficients can be used in place of a dihedral style, either in a input script dihedral_coeff command or in the data file, if you desire to turn off interactions for specific dihedral types.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 7.10.4 Restrictions

This dihedral style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* doc page for more info.

Unlike other dihedral styles, the hybrid dihedral style does not store dihedral coefficient info for individual sub-styles in *binary restart files* or *data files*. Thus when restarting a simulation, you need to re-specify the dihedral_coeff commands.

### 7.10.5 Related commands

*dihedral_coeff*

### 7.10.6 Default

none

## 7.11 dihedral_style lepton command

Accelerator Variants: *lepton/omp*

### 7.11.1 Syntax

```
dihedral_style lepton
```

### 7.11.2 Examples

```
dihedral_style lepton

dihedral_coeff  1 "k*(1 + d*cos(n*phi)); k=75.0; d=1; n=2"
dihedral_coeff  2 "45*(1-cos(4*phi))"
dihedral_coeff  2 "k2*cos(phi) + k3*cos(phi)^2; k2=100.0"
dihedral_coeff  3 "k*(phi-phi0)^2; k=85.0; phi0=120.0"
```

### 7.11.3 Description

Added in version 8Feb2023.

Dihedral style *lepton* computes dihedral interactions between four atoms forming a dihedral angle with a custom potential function. The potential function must be provided as an expression string using "phi" as the dihedral angle variable. For example *"200.0*(phi-120.0)^2"* represents a *quadratic dihedral* potential around a 120 degree dihedral angle with a force constant $K$ of 200.0 energy units:

$$U_{dihedral,i} = K(\phi_i - \phi_0)^2$$

The Lepton library, that the *lepton* dihedral style interfaces with, evaluates this expression string at run time to compute the pairwise energy. It also creates an analytical representation of the first derivative of this expression with respect to "phi" and then uses that to compute the force between the dihedral atoms as defined by the topology data.

The potential function expression for each dihedral type is provided via the *dihedral_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands. The expression is in energy units.

The Lepton expression must be either enclosed in quotes or must not contain any whitespace so that LAMMPS recognizes it as a single keyword. More on valid Lepton expressions below. Dihedral angles are internally computed in radians and thus the expression must assume "phi" is in radians.

---

### 7.11.4 Lepton expression syntax and features

Lepton supports the following operators in expressions:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| + | Add | - | Subtract | * | Multiply | / | Divide | ^ | Power |

The following mathematical functions are available:

| | | | |
|---|---|---|---|
| sqrt(x) | Square root | exp(x) | Exponential |
| log(x) | Natural logarithm | sin(x) | Sine (angle in radians) |
| cos(x) | Cosine (angle in radians) | sec(x) | Secant (angle in radians) |
| csc(x) | Cosecant (angle in radians) | tan(x) | Tangent (angle in radians) |
| cot(x) | Cotangent (angle in radians) | asin(x) | Inverse sine (in radians) |
| acos(x) | Inverse cosine (in radians) | atan(x) | Inverse tangent (in radians) |
| sinh(x) | Hyperbolic sine | cosh(x) | Hyperbolic cosine |
| tanh(x) | Hyperbolic tangent | erf(x) | Error function |
| erfc(x) | Complementary Error function | abs(x) | Absolute value |
| min(x,y) | Minimum of two values | max(x,y) | Maximum of two values |
| delta(x) | delta(x) is 1 for $x = 0$, otherwise 0 | step(x) | step(x) is 0 for $x < 0$, otherwise 1 |

Numbers may be given in either decimal or exponential form. All of the following are valid numbers: *5*, *-3.1*, *1e6*, and *3.12e-2*.

As an extension to the standard Lepton syntax, it is also possible to use LAMMPS *variables* in the format "v_name". Before evaluating the expression, "v_name" will be replaced with the value of the variable "name". This is compatible with all kinds of scalar variables, but not with vectors, arrays, local, or per-atom variables. If necessary, a custom scalar variable needs to be defined that can access the desired (single) item from a non-scalar variable. As an example, the following lines will instruct LAMMPS to ramp the force constant for a harmonic bond from 100.0 to 200.0 during the next run:

```
variable fconst equal ramp(100.0, 200)
bond_style lepton
bond_coeff 1 1.5 "v_fconst * (r^2)"
```

An expression may be followed by definitions for intermediate values that appear in the expression. A semicolon ";" is used as a delimiter between value definitions. For example, the expression:

```
a^2+a*b+b^2; a=a1+a2; b=b1+b2
```

is exactly equivalent to

```
(a1+a2)^2+(a1+a2)*(b1+b2)+(b1+b2)^2
```

The definition of an intermediate value may itself involve other intermediate values. Whitespace and quotation characters (’’ and “”) are ignored. All uses of a value must appear *before* that value’s definition. For efficiency reasons, the expression string is parsed, optimized, and then stored in an internal, pre-parsed representation for evaluation.

Evaluating a Lepton expression is typically between 2.5 and 5 times slower than the corresponding compiled and optimized C++ code. If additional speed or GPU acceleration (via GPU or KOKKOS) is required, the interaction can be represented as a table. Suitable table files can be created either internally using the *pair_write* or *bond_write* command or through the Python scripts in the *tools/tabulate* folder.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 7.11.5 Restrictions

This dihedral style is part of the LEPTON package and only enabled if LAMMPS was built with this package. See the *Build package* page for more info.

### 7.11.6 Related commands

*dihedral_coeff*, *dihedral_style table*, *bond_style lepton*, *angle_style lepton*

### 7.11.7 Default

none

## 7.12 dihedral_style multi/harmonic command

Accelerator Variants: *multi/harmonic/kk*, *multi/harmonic/omp*

### 7.12.1 Syntax

```
dihedral_style multi/harmonic
```

### 7.12.2 Examples

```
dihedral_style multi/harmonic
dihedral_coeff 1 20 20 20 20 20
```

### 7.12.3 Description

The *multi/harmonic* dihedral style uses the potential

$$E = \sum_{n=1,5} A_n \cos^{n-1}(\phi)$$

The following coefficients must be defined for each dihedral type via the *dihedral_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $A_1$ (energy)
- $A_2$ (energy)
- $A_3$ (energy)
- $A_4$ (energy)
- $A_5$ (energy)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 7.12.4 Restrictions

This dihedral style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* doc page for more info.

### 7.12.5 Related commands

*dihedral_coeff*

### 7.12.6 Default

none

## 7.13 dihedral_style nharmonic command

Accelerator Variants: *nharmonic/kk*, *nharmonic/omp*

### 7.13.1 Syntax

```
dihedral_style nharmonic
```

### 7.13.2 Examples

```
dihedral_style nharmonic
dihedral_coeff * 3 10.0 20.0 30.0
```

### 7.13.3 Description

The *nharmonic* dihedral style uses the potential:

$$E = \sum_{i=1,n} A_i \cos^{i-1}(\phi)$$

The following coefficients must be defined for each dihedral type via the *dihedral_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $n$ (integer >=1)
- $A_1$ (energy)
- $A_2$ (energy)
- ...
- $A_n$ (energy)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 7.13.4 Restrictions

This dihedral style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 7.13.5 Related commands

*dihedral_coeff*

### 7.13.6 Default

none

## 7.14 dihedral_style none command

### 7.14.1 Syntax

```
dihedral_style none
```

### 7.14.2 Examples

```
dihedral_style none
```

### 7.14.3 Description

Using a dihedral style of none means dihedral forces and energies are not computed, even if quadruplets of dihedral atoms were listed in the data file read by the *read_data* command.

See the *dihedral_style zero* command for a way to calculate dihedral statistics, but compute no dihedral interactions.

### 7.14.4 Restrictions

none

### 7.14.5 Related commands

*dihedral_style zero*

### 7.14.6 Default

none

## 7.15 dihedral_style opls command

Accelerator Variants: *opls/intel*, *opls/kk*, *opls/omp*

### 7.15.1 Syntax

```
dihedral_style opls
```

### 7.15.2 Examples

```
dihedral_style opls
dihedral_coeff 1 1.740 -0.157 0.279 0.00   # CT-CT-CT-CT
dihedral_coeff 2 0.000 0.000 0.366 0.000   # CT-CT-CT-HC
dihedral_coeff 3 0.000 0.000 0.318 0.000   # HC-CT-CT-HC
```

### 7.15.3 Description

The *opls* dihedral style uses the potential

$$E = \frac{1}{2}K_1[1+\cos(\phi)] + \frac{1}{2}K_2[1-\cos(2\phi)] + \frac{1}{2}K_3[1+\cos(3\phi)] + \frac{1}{2}K_4[1-\cos(4\phi)]$$

Note that the usual 1/2 factor is not included in the K values.

This dihedral potential is used in the OPLS force field and is described in *(Watkins)*.

The following coefficients must be defined for each dihedral type via the *dihedral_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K_1$ (energy)
- $K_2$ (energy)
- $K_3$ (energy)
- $K_4$ (energy)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 7.15.4 Restrictions

This dihedral style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* doc page for more info.

### 7.15.5 Related commands

*dihedral_coeff*

### 7.15.6 Default

none

---

**(Watkins)** Watkins and Jorgensen, J Phys Chem A, 105, 4118-4125 (2001).

## 7.16 dihedral_style quadratic command

Accelerator Variants: *quadratic/kk*, *quadratic/omp*

### 7.16.1 Syntax

```
dihedral_style quadratic
```

### 7.16.2 Examples

```
dihedral_style quadratic
dihedral_coeff 100.0 80.0
```

### 7.16.3 Description

The *quadratic* dihedral style uses the potential:

$$E = K(\phi - \phi_0)^2$$

This dihedral potential can be used to keep a dihedral in a predefined value (cis=zero, right-hand convention is used).

The following coefficients must be defined for each dihedral type via the *dihedral_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy)
- $\phi_0$ (degrees)

$\phi_0$ is specified in degrees, but LAMMPS converts it to radians internally; hence $K$ is effectively energy per radian^2.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 7.16.4 Restrictions

This dihedral style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 7.16.5 Related commands

*dihedral_coeff*

### 7.16.6 Default

none

## 7.17 dihedral_style spherical command

Accelerator Variants: *spherical/kk*, *spherical/omp*

### 7.17.1 Syntax

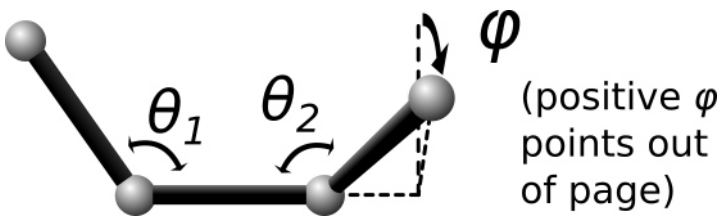
```
dihedral_style spherical
```

### 7.17.2 Examples

```
dihedral_coeff 1 1  286.1  1 124  1    1 90.0 0    1 90.0 0
dihedral_coeff 1 3  69.3   1 93.9 1    1 90   0    1 90   0  &
                    49.1   0 0.00 0    1 74.4 1    0 0.00 0  &
                    25.2   0 0.00 0    0 0.00 0    1 48.1 1
```

### 7.17.3 Description

The *spherical* dihedral style uses the potential:




$$E(\phi, \theta_1, \theta_2) = \sum_{i=1}^{N} C_i \, \Phi_i(\phi) \, \Theta_{1i}(\theta_1) \, \Theta_{2i}(\theta_2)$$

$$\Phi_i(\phi) = u_i - \cos((\phi - a_i)K_i)$$

$$\Theta_{1i}(\theta_1) = v_i - \cos((\theta_1 - b_i)L_i)$$

$$\Theta_{2i}(\theta_2) = w_i - \cos((\theta_2 - c_i)M_i)$$

For this dihedral style, the energy can be any function that combines the 4-body dihedral-angle ($\phi$) and the two 3-body bond-angles ($\theta_1$, $\theta_2$). For this reason, there is usually no need to define 3-body "angle" forces separately for the atoms participating in these interactions. It is probably more efficient to incorporate 3-body angle forces into the dihedral interaction even if it requires adding additional terms to the expansion (as was done in the second example). A careful choice of parameters can prevent singularities that occur with traditional force-fields whenever theta1 or theta2 approach 0 or 180 degrees.

The last example above corresponds to an interaction with a single energy minima located near $\phi$ = 93.9, $\theta_1$ = 74.4, $\theta_2$ = 48.1 degrees, and it remains numerically stable at all angles ($\phi$, $\theta_1$, $\theta_2$). In this example, the coefficients 49.1, and 25.2 can be physically interpreted as the harmonic spring constants for theta1 and theta2 around their minima. The coefficient 69.3 is the harmonic spring constant for phi after division by sin(74.4)*sin(48.1) (the minima positions for theta1 and theta2).

The following coefficients must be defined for each dihedral type via the *dihedral_coeff* command as in the example above, or in the Dihedral Coeffs section of a data file read by the *read_data* command:

- $n$ (integer >= 1)
- $C_1$ (energy)
- $K_1$ (typically an integer)
- $a_1$ (degrees)
- $u_1$ (typically 0.0 or 1.0)
- $L_1$ (typically an integer)
- $b_1$ (degrees, typically 0.0 or 90.0)
- $v_1$ (typically 0.0 or 1.0)
- $M_1$ (typically an integer)
- $c_1$ (degrees, typically 0.0 or 90.0)
- $w_1$ (typically 0.0 or 1.0)

- [...]
- $C_n$ (energy)
- $K_n$ (typically an integer)
- $a_n$ (degrees)
- $u_n$ (typically 0.0 or 1.0)
- $L_n$ (typically an integer)
- $b_n$ (degrees, typically 0.0 or 90.0)
- $v_n$ (typically 0.0 or 1.0)
- $M_n$ (typically an integer)
- $c_n$ (degrees, typically 0.0 or 90.0)
- $w_n$ (typically 0.0 or 1.0)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 7.17.4 Restrictions

This dihedral style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 7.17.5 Related commands

*dihedral_coeff*

### 7.17.6 Default

none

## 7.18 dihedral_style table command

Accelerator Variants: *table/omp*

## 7.19 dihedral_style table/cut command

Accelerator Variants: *table/cut/omp*

### 7.19.1 Syntax

```
dihedral_style style interpolation Ntable
```

- style = *table* or *table/cut*
- interpolation = *linear* or *spline* = method of interpolation
- Ntable = size of the internal lookup table

### 7.19.2 Examples

```
dihedral_style table spline 400
dihedral_style table linear 1000
dihedral_coeff 1 file.table DIH_TABLE1
dihedral_coeff 2 file.table DIH_TABLE2

dihedral_style table/cut spline 400
dihedral_style table/cut linear 1000
dihedral_coeff 1 aat 1.0 177 180 file.table DIH_TABLE1
dihedral_coeff 2 aat 0.5 170 180 file.table DIH_TABLE2
```

### 7.19.3 Description

The *table* and *table/cut* dihedral styles create interpolation tables of length *Ntable* from dihedral potential and derivative values listed in a file(s) as a function of the dihedral angle "phi". The files are read by the *dihedral_coeff* command. For dihedral style *table/cut* additionally an analytic cutoff that is quadratic in the bond-angle (theta) is applied in order to regularize the dihedral interaction.

The interpolation tables are created by fitting cubic splines to the file values and interpolating energy and derivative values at each of *Ntable* dihedral angles. During a simulation, these tables are used to interpolate energy and force values on individual atoms as needed. The interpolation is done in one of 2 styles: *linear* or *spline*.

For the *linear* style, the dihedral angle (phi) is used to find 2 surrounding table values from which an energy or its derivative is computed by linear interpolation.

For the *spline* style, cubic spline coefficients are computed and stored at each of the *Ntable* evenly-spaced values in the interpolated table. For a given dihedral angle (phi), the appropriate coefficients are chosen from this list, and a cubic polynomial is used to compute the energy and the derivative at this angle.

For dihedral style *table* the following coefficients must be defined for each dihedral type via the *dihedral_coeff* command as in the example above.

- filename
- keyword

The filename specifies a file containing tabulated energy and derivative values. The keyword specifies which section of the file to read. The format of this file is the same for both dihedral styles and described below.

For dihedral style *table/cut* the following coefficients must be defined for each dihedral type via the *dihedral_coeff* command as in the example above.

- style (= aat)
- cutoff prefactor (unitless)
- cutoff angle1 (degrees)
- cutoff angle2 (degrees)

- filename
- keyword

The cutoff dihedral style uses a tabulated dihedral interaction with a cutoff function:

$$f(\theta) = K \qquad \theta < \theta_1$$
$$f(\theta) = K\left(1 - \frac{(\theta - \theta_1)^2}{(\theta_2 - \theta_1)^2}\right) \qquad \theta_1 < \theta < \theta_2$$

The cutoff includes a prefactor $K$ to the cutoff function $f(\theta)$. While this value would ordinarily be 1, there may be situations where the value could be different.

The cutoff $\theta_1$ specifies the angle (in degrees) below which the dihedral interaction is unmodified, i.e. the cutoff function is 1.

The cutoff function is applied between $\theta_1$ and $\theta_2$, which is the angle at which the cutoff function drops to zero. The value of zero effectively "turns off" the dihedral interaction.

The filename specifies a file containing tabulated energy and derivative values. The keyword specifies which section of the file to read. The format of this file is the same for both dihedral styles and described below.

---

Suitable tables for use with this dihedral style can be created using the Python code in the `tools/tabulate` folder of the LAMMPS source code distribution.

The format of a tabulated file is as follows (without the parenthesized comments). It can begin with one or more comment or blank lines.

```
# Table of the potential and its negative derivative

DIH_TABLE1                    (keyword is the first text on line)
N 30 DEGREES                  (N, NOF, DEGREES, RADIANS, CHECKU/F)
                              (blank line)
1 -168.0 -1.40351172223 0.0423346818422
2 -156.0 -1.70447981034 0.00811786522531
3 -144.0 -1.62956100432 -0.0184129719987
...
30 180.0 -0.707106781187 0.0719306095245

# Example 2: table of the potential. Forces omitted

DIH_TABLE2
N 30 NOF CHECKU testU.dat CHECKF testF.dat

1 -168.0 -1.40351172223
2 -156.0 -1.70447981034
3 -144.0 -1.62956100432
...
30 180.0 -0.707106781187
```

A section begins with a non-blank line whose first character is not a "#"; blank lines or lines starting with "#" can be used as comments between sections. The first line begins with a keyword which identifies the section. The line can contain additional text, but the initial text must match the argument specified in the *dihedral_coeff* command. The next line lists (in any order) one or more parameters for the table. Each parameter is a keyword followed by one or more numeric values.

Following a blank line, the next N lines list the tabulated values. On each line, the first value is the index from 1 to N, the second value is the angle value, the third value is the energy (in energy units), and the fourth is -dE/d(phi) also in energy units). The third term is the energy of the 4-atom configuration for the specified angle. The fourth term (when present) is the negative derivative of the energy with respect to the angle (in degrees, or radians depending on whether the user selected DEGREES or RADIANS). Thus the units of the last term are still energy, not force. The dihedral angle values must increase from one line to the next.

Dihedral table splines are cyclic. There is no discontinuity at 180 degrees (or at any other angle). Although in the examples above, the angles range from -180 to 180 degrees, in general, the first angle in the list can have any value (positive, zero, or negative). However the *range* of angles represented in the table must be *strictly* less than 360 degrees (2pi radians) to avoid angle overlap. (You may not supply entries in the table for both 180 and -180, for example.) If the user's table covers only a narrow range of dihedral angles, strange numerical behavior can occur in the large remaining gap.

**Parameters:**

The parameter "N" is required and its value is the number of table entries that follow. Note that this may be different than the N specified in the *dihedral_style table* command. Let *Ntable* is the number of table entries requested dihedral_style command, and let *Nfile* be the parameter following "N" in the tabulated file ("30" in the sparse example above). What LAMMPS does is a preliminary interpolation by creating splines using the *Nfile* tabulated values as nodal points. It uses these to interpolate as needed to generate energy and derivative values at *Ntable* different points (which are evenly spaced over a 360 degree range, even if the angles in the file are not). The resulting tables of length *Ntable* are then used as described above, when computing energy and force for individual dihedral angles and their atoms. This means that if you want the interpolation tables of length *Ntable* to match exactly what is in the tabulated file (with effectively nopreliminary interpolation), you should set *Ntable* = *Nfile*. To ensure the nodal points in the user's file are aligned with the interpolated table entries, the angles in the table should be integer multiples of 360/*Ntable* degrees, or 2*PI/*Ntable* radians (depending on your choice of angle units).

The optional "NOF" keyword allows the user to omit the forces (negative energy derivatives) from the table file (normally located in the fourth column). In their place, forces will be calculated automatically by differentiating the potential energy function indicated by the third column of the table (using either linear or spline interpolation).

The optional "DEGREES" keyword allows the user to specify angles in degrees instead of radians (default).

The optional "RADIANS" keyword allows the user to specify angles in radians instead of degrees. (Note: This changes the way the forces are scaled in the fourth column of the data file.)

The optional "CHECKU" keyword is followed by a filename. This allows the user to save all of the *Ntable* different entries in the interpolated energy table to a file to make sure that the interpolated function agrees with the user's expectations. (Note: You can temporarily increase the *Ntable* parameter to a high value for this purpose. "*Ntable*" is explained above.)

The optional "CHECKF" keyword is analogous to the "CHECKU" keyword. It is followed by a filename, and it allows the user to check the interpolated force table. This option is available even if the user selected the "NOF" option.

Note that one file can contain many sections, each with a tabulated potential. LAMMPS reads the file section by section until it finds one that matches the specified keyword.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 7.19.4 Restart, fix_modify, output, run start/stop, minimize info

These dihedral styles write the settings for the "dihedral_style table" or "dihedral_style table/cut" command to *binary restart files*, so a dihedral_style command does not need to specified in an input script that reads a restart file. However, the coefficient information loaded from the table file(s) is not stored in the restart file, since it is tabulated in the potential files. Thus, suitable dihedral_coeff commands do need to be specified in the restart input script after reading the restart file.

### 7.19.5 Restrictions

The *table* dihedral style can only be used if LAMMPS was built with the MOLECULE package. The *table/cut* dihedral style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 7.19.6 Related commands

*dihedral_coeff*

### 7.19.7 Default

none

## 7.20 dihedral_style zero command

### 7.20.1 Syntax

```
dihedral_style zero keyword
```

- zero or more keywords may be appended
- keyword = *nocoeff*

### 7.20.2 Examples

```
dihedral_style zero
dihedral_style zero nocoeff
dihedral_coeff *
```

### 7.20.3 Description

Using a dihedral style of zero means dihedral forces and energies are not computed, but the geometry of dihedral quadruplets is still accessible to other commands.

As an example, the *compute dihedral/local* command can be used to compute the theta values for the list of quadruplets of dihedral atoms listed in the data file read by the *read_data* command. If no dihedral style is defined, this command cannot be used.

The optional *nocoeff* flag allows to read data files with a DihedralCoeff section for any dihedral style. Similarly, any dihedral_coeff commands will only be checked for the dihedral type number and the rest ignored.

Note that the *dihedral_coeff* command must be used for all dihedral types, though no additional values are specified.

### 7.20.4 Restrictions

none

### 7.20.5 Related commands

none

*dihedral_style none*

### 7.20.6 Default

none

CHAPTER

EIGHT

# IMPROPER STYLES

## 8.1 improper_style amoeba command

### 8.1.1 Syntax

```
improper_style amoeba
```

### 8.1.2 Examples

```
improper_style amoeba
improper_coeff 1 49.6
```

### 8.1.3 Description

The *amoeba* improper style uses the potential

$$E = K(\chi)^2$$

where $\chi$ is the improper angle and $K$ is a prefactor. Note that the usual 1/2 factor is included in $K$.

This formula seems like a simplified version of the formula for the *improper_style harmonic* command with $\chi_0 = 0.0$. However the computation of the angle $\chi$ is done differently to match how the Tinker MD code computes its out-of-plane improper for the AMOEBA and HIPPO force fields. See the *Howto amoeba* doc page for more information about the implementation of AMOEBA and HIPPO in LAMMPS.

If the 4 atoms in an improper quadruplet (listed in the data file read by the *read_data* command are ordered I,J,K,L then atoms I,K,L are considered to lie in a plane and atom J is out-of-plane. The angle $\chi_0$ is computed as the Allinger angle which is defined as the angle between the plane of I,K,L, and the vector from atom I to atom J.

The following coefficient must be defined for each improper type via the *improper_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy)

Note that the angle $\chi$ is computed in radians; hence $K$ is effectively energy per radian^2.

---

### 8.1.4 Symmetry convention

For the *amoeba* improper style, the second atom in the quadruplet is the atom of symmetry; all other atoms are considered interchangeable. This convention is relevant for operations that require knowledge of how atoms are ordered, such as automatic assignment of new improper types by *fix bond/react*.

### 8.1.5 Restrictions

This improper style can only be used if LAMMPS was built with the AMOEBA package. See the *Build package* doc page for more info.

### 8.1.6 Related commands

*improper_coeff*, *improper_harmonic*

### 8.1.7 Default

none

## 8.2 improper_style class2 command

Accelerator Variants: *class2/omp*, *class2/kk*

### 8.2.1 Syntax

```
improper_style class2
```

### 8.2.2 Examples

```
improper_style class2
improper_coeff 1 100.0 0
improper_coeff * aa 0.0 0.0 0.0 115.06 130.01 115.06
```

### 8.2.3 Description

The *class2* improper style uses the potential

$$
\begin{aligned}
E =& E_i + E_{aa} \\
E_i =& K[\frac{\chi_{ijkl} + \chi_{kjli} + \chi_{ljik}}{3} - \chi_0]^2 \\
E_{aa} =& M_1(\theta_{ijk} - \theta_1)(\theta_{kjl} - \theta_3) + \\
& M_2(\theta_{ijk} - \theta_1)(\theta_{ijl} - \theta_2) + \\
& M_3(\theta_{ijl} - \theta_2)(\theta_{kjl} - \theta_3)
\end{aligned}
$$

where $E_i$ is the improper term and $E_{aa}$ is an angle-angle term. The 3 $\chi$ terms in $E_i$ are an average over 3 out-of-plane angles.

The 4 atoms in an improper quadruplet (listed in the data file read by the *read_data* command) are ordered I,J,K,L. $\chi_{ijkl}$ refers to the angle between the plane of I,J,K and the plane of J,K,L, and the bond JK lies in both planes. Similarly for $\chi_{kjli}$ and $\chi_{ljik}$. Note that atom J appears in the common bonds (JI, JK, JL) of all 3 X terms. Thus J (the second atom in the quadruplet) is the atom of symmetry in the 3 $\chi$ angles.

The subscripts on the various $\theta$s refer to different combinations of three atoms (I,J,K,L) used to form a particular angle. E.g. $\theta_{ijl}$ is the angle formed by atoms I,J,L with J in the middle. $\theta_1$, $\theta_2$, $\theta_3$ are the equilibrium positions of those angles. Again, atom J (the second atom in the quadruplet) is the atom of symmetry in the theta angles, since it is always the center atom.

Since atom J is the atom of symmetry, normally the bonds J-I, J-K, J-L would exist for an improper to be defined between the 4 atoms, but this is not required.

See *(Sun)* for a description of the COMPASS class2 force field.

Coefficients for the $E_i$ and $E_{aa}$ formulas must be defined for each improper type via the *improper_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands.

These are the 2 coefficients for the $E_i$ formula:

- $K$ (energy)
- $\chi_0$ (degrees)

$\chi_0$ is specified in degrees, but LAMMPS converts it to radians internally; hence $K$ is effectively energy per radian^2.

For the $E_{aa}$ formula, each line in a *improper_coeff* command in the input script lists 7 coefficients, the first of which is *aa* to indicate they are AngleAngle coefficients. In a data file, these coefficients should be listed under a *AngleAngle Coeffs* heading and you must leave out the *aa*, i.e. only list 6 coefficients after the improper type.

- *aa*
- $M_1$ (energy)
- $M_2$ (energy)
- $M_3$ (energy)
- $\theta_1$ (degrees)
- $\theta_2$ (degrees)
- $\theta_3$ (degrees)

The $\theta$ values are specified in degrees, but LAMMPS converts them to radians internally; hence the hence the various $M$ are effectively energy per radian^2.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 8.2.4 Symmetry convention

For the *class2* improper style, the second atom in the quadruplet is the atom of symmetry; all other atoms are considered interchangeable. This convention is relevant for operations that require knowledge of how atoms are ordered, such as automatic assignment of new improper types by *fix bond/react*.

### 8.2.5 Restrictions

This improper style can only be used if LAMMPS was built with the CLASS2 package. See the *Build package* doc page for more info.

### 8.2.6 Related commands

*improper_coeff*

### 8.2.7 Default

none

---

**(Sun)** Sun, J Phys Chem B 102, 7338-7364 (1998).

## 8.3 improper_style cossq command

Accelerator Variants: *cossq/kk*, *cossq/omp*

### 8.3.1 Syntax

```
improper_style cossq
```

### 8.3.2 Examples

```
improper_style cossq
improper_coeff 1 4.0 0.0
```

### 8.3.3 Description

The *cossq* improper style uses the potential

$$E = \frac{1}{2} K \cos^2 \left( \chi - \chi_0 \right)$$

where $\chi$ is the improper angle, $\chi_0$ is its equilibrium value, and $K$ is a prefactor.

If the 4 atoms in an improper quadruplet (listed in the data file read by the *read_data* command) are ordered I,J,K,L then $\chi$ is the angle between the plane of I,J,K and the plane of J,K,L. Alternatively, you can think of atoms J,K,L as being in a plane, and atom I above the plane, and $\chi$ as a measure of how far out-of-plane I is with respect to the other 3 atoms.

Note that defining 4 atoms to interact in this way, does not mean that bonds necessarily exist between I-J, J-K, or K-L, as they would in a linear dihedral. Normally, the bonds I-J, I-K, I-L would exist for an improper to be defined between the 4 atoms.

The following coefficients must be defined for each improper type via the *improper_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy)
- $\chi_0$ (degrees)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 8.3.4 Symmetry convention

For the *cossq* improper style, the first atom in the quadruplet is the atom of symmetry; all other atoms are considered interchangeable. This convention is relevant for operations that require knowledge of how atoms are ordered, such as automatic assignment of new improper types by *fix bond/react*.

### 8.3.5 Restrictions

This improper style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 8.3.6 Related commands

*improper_coeff*

### 8.3.7 Default

none

## 8.4 improper_style cvff command

Accelerator Variants: *cvff/intel*, *cvff/kk*, *cvff/omp*

### 8.4.1 Syntax

```
improper_style cvff
```

### 8.4.2 Examples

```
improper_style cvff
improper_coeff 1 80.0 -1 4
```

### 8.4.3 Description

The *cvff* improper style uses the potential

$$E = K[1 + d\cos(n\phi)]$$

where phi is the improper dihedral angle.

If the 4 atoms in an improper quadruplet (listed in the data file read by the *read_data* command) are ordered I,J,K,L then the improper dihedral angle is between the plane of I,J,K and the plane of J,K,L. Note that because this is effectively a dihedral angle, the formula for this improper style is the same as for *dihedral_style harmonic*.

Note that defining 4 atoms to interact in this way, does not mean that bonds necessarily exist between I-J, J-K, or K-L, as they would in a linear dihedral. Normally, the bonds I-J, I-K, I-L would exist for an improper to be defined between the 4 atoms.

The following coefficients must be defined for each improper type via the *improper_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- *K* (energy)
- *d* (+1 or -1)
- *n* (0,1,2,3,4,6)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 8.4.4 Symmetry convention

For the *cvff* improper style, the first atom in the quadruplet is the atom of symmetry; all other atoms are considered interchangeable. This convention is relevant for operations that require knowledge of how atoms are ordered, such as automatic assignment of new improper types by *fix bond/react*.

### 8.4.5 Restrictions

This improper style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* doc page for more info.

### 8.4.6 Related commands

*improper_coeff*

### 8.4.7 Default

none

## 8.5 improper_style distance command

Accelerator Variants: *distance/kk*, *distance/omp*

### 8.5.1 Syntax

```
improper_style distance
```

### 8.5.2 Examples

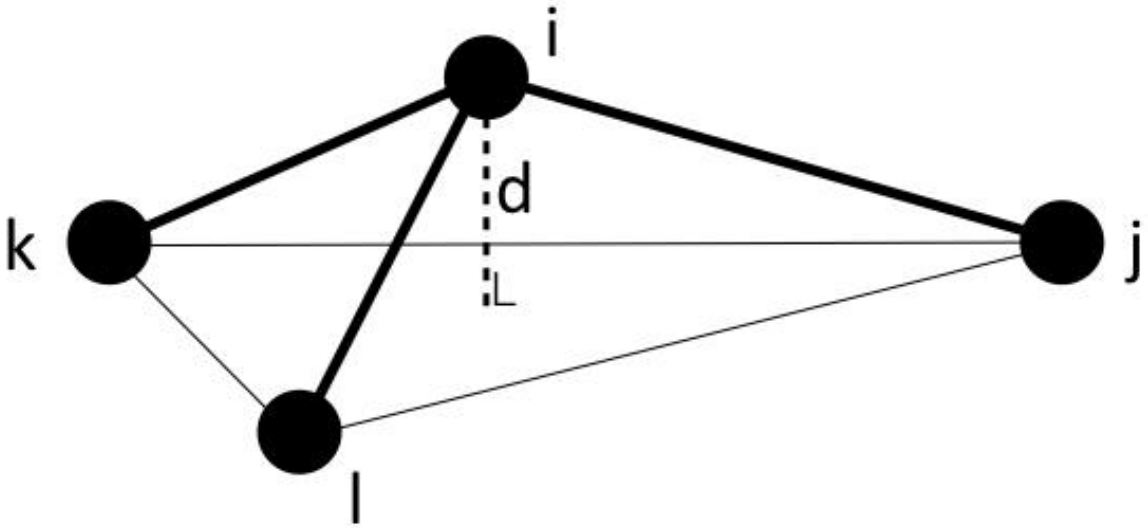
```
improper_style distance
improper_coeff 1 80.0 100.0
```

### 8.5.3 Description

The *distance* improper style uses the potential

$$E = K_2 d^2 + K_4 d^4$$

where $d$ is the distance between the central atom and the plane formed by the other three atoms. If the 4 atoms in an improper quadruplet (listed in the data file read by the *read_data* command) are ordered I,J,K,L then the I-atom is assumed to be the central atom.




Note that defining 4 atoms to interact in this way, does not mean that bonds necessarily exist between I-J, J-K, or K-L, as they would in a linear dihedral. Normally, the bonds I-J, I-K, I-L would exist for an improper to be defined between the 4 atoms.

The following coefficients must be defined for each improper type via the improper_coeff command as in the example above, or in the data file or restart files read by the read_data or read_restart commands:

- $K_2$ (energy/distance^2)
- $K_4$ (energy/distance^4)

---

### 8.5.4 Symmetry convention

For the *distance* improper style, the first atom in the quadruplet is the atom of symmetry; all other atoms are considered interchangeable. This convention is relevant for operations that require knowledge of how atoms are ordered, such as automatic assignment of new improper types by *fix bond/react*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 8.5.5 Restrictions

This improper style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 8.5.6 Related commands

*improper_coeff*

### 8.5.7 Default

none

## 8.6 improper_style distharm command

Accelerator Variants: *distharm/kk*, *distharm/omp*

### 8.6.1 Syntax

```
improper_style distharm
```

### 8.6.2 Examples

```
improper_style distharm
improper_coeff 1 25.0 0.5
```

### 8.6.3 Description

The *distharm* improper style uses the potential

$$E = K(d - d_0)^2$$

where $d$ is the oriented distance between the central atom and the plane formed by the other three atoms. If the 4 atoms in an improper quadruplet (listed in the data file read by the *read_data* command) are ordered I,J,K,L then the L-atom is assumed to be the central atom. Note that this is different from the convention used in the improper_style distance. The distance $d$ is oriented and can take on negative values. This may lead to unwanted behavior if $d_0$ is not equal to zero.

The following coefficients must be defined for each improper type via the improper_coeff command as in the example above, or in the data file or restart files read by the read_data or read_restart commands:

- $K$ (energy/distance^2)
- $d_0$ (distance)

---

### 8.6.4 Symmetry convention

For the *distharm* improper style, the fourth atom in the quadruplet is the atom of symmetry; all other atoms are considered interchangeable. This convention is relevant for operations that require knowledge of how atoms are ordered, such as automatic assignment of new improper types by *fix bond/react*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 8.6.5 Restrictions

This improper style can only be used if LAMMPS was built with the YAFF package. See the *Build package* doc page for more info.

### 8.6.6 Related commands

*improper_coeff*

### 8.6.7 Default

none

## 8.7 improper_style fourier command

Accelerator Variants: *fourier/kk*, *fourier/omp*

### 8.7.1 Syntax

```
improper_style fourier
```

### 8.7.2 Examples

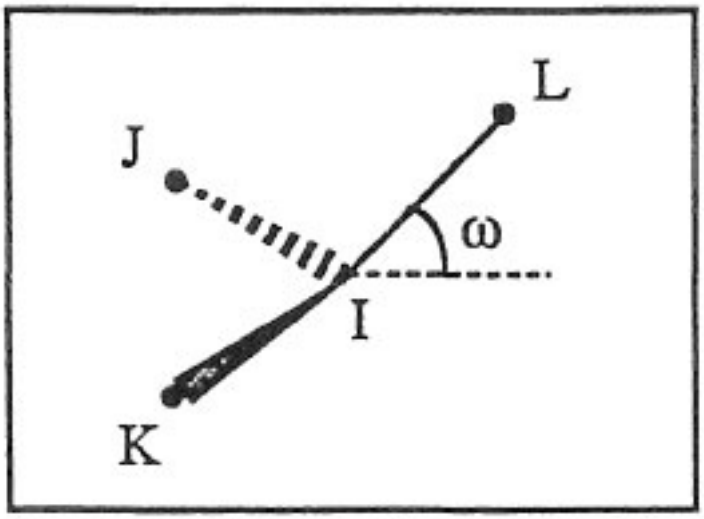
```
improper_style fourier
improper_coeff 1 100.0 0.0 1.0 0.5 1
```

### 8.7.3 Description

The *fourier* improper style uses the following potential:

$$E = K[C_0 + C_1 \cos(\omega) + C_2 \cos(2\omega)]$$

where K is the force constant, C0, C1, C2 are dimensionless coefficients, and omega is the angle between the IL axis and the IJK plane:




If all parameter (see below) is not zero, the all the three possible angles will taken in account.

The following coefficients must be defined for each improper type via the *improper_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy)
- $C_0$ (unitless)
- $C_1$ (unitless)
- $C_2$ (unitless)
- all (0 or 1, optional)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 8.7.4 Symmetry convention

For the *fourier* improper style, the first and fourth atoms in the quadruplet are atoms of symmetry; only the second and third atoms are considered interchangeable. This convention is relevant for operations that require knowledge of how atoms are ordered, such as automatic assignment of new improper types by *fix bond/react*.

### 8.7.5 Restrictions

This angle style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 8.7.6 Related commands

*improper_coeff*

### 8.7.7 Default

none

## 8.8 improper_style harmonic command

Accelerator Variants: *harmonic/intel*, *harmonic/kk*, *harmonic/omp*

### 8.8.1 Syntax

```
improper_style harmonic
```

### 8.8.2 Examples

```
improper_style harmonic
improper_coeff 1 100.0 0
```

### 8.8.3 Description

The *harmonic* improper style uses the potential

$$E = K(\chi - \chi_0)^2$$

where $\chi$ is the improper angle, $\chi_0$ is its equilibrium value, and $K$ is a prefactor. Note that the usual 1/2 factor is included in $K$.

If the 4 atoms in an improper quadruplet (listed in the data file read by the *read_data* command) are ordered I,J,K,L then $\chi$ is the angle between the plane of I,J,K and the plane of J,K,L. Alternatively, you can think of atoms J,K,L as being in a plane, and atom I above the plane, and $\chi$ as a measure of how far out-of-plane I is with respect to the other 3 atoms.

Note that defining 4 atoms to interact in this way, does not mean that bonds necessarily exist between I-J, J-K, or K-L, as they would in a linear dihedral. Normally, the bonds I-J, I-K, I-L would exist for an improper to be defined between the 4 atoms.

The following coefficients must be defined for each improper type via the *improper_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy)
- $\chi_0$ (degrees)

$\chi_0$ is specified in degrees, but LAMMPS converts it to radians internally; hence $K$ is effectively energy per radian^2.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages*

page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 8.8.4 Symmetry convention

For the *harmonic* improper style, the first atom in the quadruplet is the atom of symmetry; all other atoms are considered interchangeable. This convention is relevant for operations that require knowledge of how atoms are ordered, such as automatic assignment of new improper types by *fix bond/react*.

### 8.8.5 Restrictions

This improper style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* doc page for more info.

### 8.8.6 Related commands

*improper_coeff*

### 8.8.7 Default

none

## 8.9 improper_style hybrid command

Accelerator Variants: *hybrid/kk*

### 8.9.1 Syntax

```
improper_style hybrid style1 style2 ...
```

- style1,style2 = list of one or more improper styles

### 8.9.2 Examples

```
improper_style hybrid harmonic cvff
improper_coeff 1 harmonic 120.0 30
improper_coeff 2 cvff 20.0 -1 2
```

### 8.9.3 Description

The *hybrid* style enables the use of multiple improper styles in one simulation. An improper style is assigned to each improper type. For example, impropers in a polymer flow (of improper type 1) could be computed with a *harmonic* potential and impropers in the wall boundary (of improper type 2) could be computed with a *cvff* potential. The assignment of improper type to style is made via the *improper_coeff* command or in the data file.

In the improper_coeff command, the first coefficient sets the improper style and the remaining coefficients are those appropriate to that style. In the example above, the 2 improper_coeff commands would set impropers of improper type 1 to be computed with a *harmonic* potential with coefficients 120.0, 30 for $K$, $\chi_0$. Improper type 2 would be computed with a *cvff* potential with coefficients 20.0, -1, 2 for K, d, and n, respectively.

If improper coefficients are specified in the data file read via the *read_data* command, then the same rule applies. E.g. "harmonic" or "cvff", must be added after the improper type, for each line in the "Improper Coeffs" section, e.g.

```
Improper Coeffs

1 harmonic 120.0 30
2 cvff 20.0 -1 2
...
```

If *class2* is one of the improper hybrid styles, the same rule holds for specifying additional AngleAngle coefficients either via the input script or in the data file. I.e. *class2* must be added to each line after the improper type. For lines in the AngleAngle Coeffs section of the data file for dihedral types that are not *class2*, you must use an improper style of *skip* as a placeholder, e.g.

```
AngleAngle Coeffs

1 skip
2 class2 0.0 0.0 0.0 115.06 130.01 115.06
...
```

Note that it is not necessary to use the improper style *skip* in the input script, since AngleAngle coefficients need not be specified at all for improper types that are not *class2*.

An improper style of *none* can be specified as the second argument to the improper_coeff command, if you desire to turn off certain improper types.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 8.9.4 Restrictions

This improper style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* doc page for more info.

Unlike other improper styles, the hybrid improper style does not store improper coefficient info for individual sub-styles in *binary restart files* or *data files*. Thus when restarting a simulation, you need to re-specify the improper_coeff commands.

### 8.9.5 Related commands

*improper_coeff*

### 8.9.6 Default

none

## 8.10 improper_style inversion/harmonic command

Accelerator Variants: *inversion/harmonic/kk*, *inversion/harmonic/omp*

### 8.10.1 Syntax

```
improper_style inversion/harmonic
```

### 8.10.2 Examples

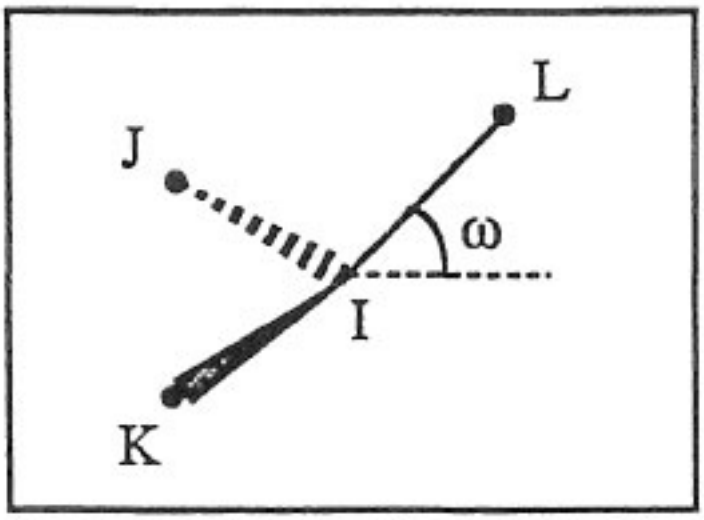
```
improper_style inversion/harmonic
improper_coeff 1 18.776340 0.000000
```

### 8.10.3 Description

The *inversion/harmonic* improper style follows the Wilson-Decius out-of-plane angle definition and uses an harmonic potential:

$$E = K\,(\omega - \omega_0)^2$$

where $K$ is the force constant and $\omega$ is the angle evaluated for all three axis-plane combinations centered around the atom I. For the IL axis and the IJK plane $\omega$ looks as follows:




Note that the *inversion/harmonic* angle term evaluation differs to the *improper_umbrella* due to the cyclic evaluation of all possible angles $\omega$.

The following coefficients must be defined for each improper type via the *improper_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy)
- $\omega_0$ (degrees)

If $\omega_0$ = 0 the potential term has a single minimum for the planar structure. Otherwise it has two minima at +/- $\omega_0$, with a barrier in between.

---

### 8.10.4 Symmetry convention

For the *inversion/harmonic* improper style, the first atom in the quadruplet is the atom of symmetry; all other atoms are considered interchangeable. This convention is relevant for operations that require knowledge of how atoms are ordered, such as automatic assignment of new improper types by *fix bond/react*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 8.10.5 Restrictions

This improper style can only be used if LAMMPS was built with the MOFFF package. See the *Build package* doc page for more info.

### 8.10.6 Related commands

*improper_coeff*

### 8.10.7 Default

none

## 8.11 improper_style none command

### 8.11.1 Syntax

```
improper_style none
```

### 8.11.2 Examples

```
improper_style none
```

### 8.11.3 Description

Using an improper style of none means improper forces and energies are not computed, even if quadruplets of improper atoms were listed in the data file read by the *read_data* command.

See the *improper_style zero* command for a way to calculate improper statistics, but compute no improper interactions.

### 8.11.4 Restrictions

none

### 8.11.5 Related commands

*improper_style zero*

### 8.11.6 Default

none

## 8.12 improper_style ring command

Accelerator Variants: *ring/kk*, *ring/omp*

### 8.12.1 Syntax

```
improper_style ring
```

### 8.12.2 Examples

```
improper_style ring
improper_coeff 1 8000 70.5
```

### 8.12.3 Description

The *ring* improper style uses the potential

$$
\begin{aligned}
E &= \frac{1}{6} K \left(\Delta_{ijl} + \Delta_{ijk} + \Delta_{kjl}\right)^6 \\
\Delta_{ijl} &= \cos\theta_{ijl} - \cos\theta_0 \\
\Delta_{ijk} &= \cos\theta_{ijk} - \cos\theta_0 \\
\Delta_{kjl} &= \cos\theta_{kjl} - \cos\theta_0
\end{aligned}
$$

where $K$ is a prefactor, $\theta$ is the angle formed by the atoms specified by (i,j,k,l) indices and $\theta_0$ its equilibrium value.

If the 4 atoms in an improper quadruplet (listed in the data file read by the *read_data* command) are ordered i,j,k,l then $\theta_{ijl}$ is the angle between atoms i,j and l, $\theta_{ijk}$ is the angle between atoms i,j and k, $\theta_{kjl}$ is the angle between atoms j,k, and l.

The "ring" improper style implements the improper potential introduced by Destree et al., in Equation (9) of *(Destree)*. This potential does not affect small amplitude vibrations but is used in an ad-hoc way to prevent the onset of accidentally large amplitude fluctuations leading to the occurrence of a planar conformation of the three bonds i-j, j-k and j-l, an intermediate conformation toward the chiral inversion of a methine carbon. In the "Impropers" section of data file four atoms: i, j, k and l are specified with i,j and l lying on the backbone of the chain and k specifying the chirality of j.

The following coefficients must be defined for each improper type via the *improper_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy)
- $\theta_0$ (degrees)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 8.12.4 Symmetry convention

For the *ring* improper style, the second atom in the quadruplet is the atom of symmetry; all other atoms are considered interchangeable. This convention is relevant for operations that require knowledge of how atoms are ordered, such as automatic assignment of new improper types by *fix bond/react*.

### 8.12.5 Restrictions

This improper style can only be used if LAMMPS was built with the EXTRA-MOLECULE package. See the *Build package* doc page for more info.

### 8.12.6 Related commands

*improper_coeff*

**(Destree)** M. Destree, F. Laupretre, A. Lyulin, and J.-P. Ryckaert, J Chem Phys, 112, 9632 (2000).

## 8.13 improper_style sqdistharm command

Accelerator Variants: *sqdistharm/kk*, *sqdistharm/omp*

### 8.13.1 Syntax

```
improper_style sqdistharm
```

### 8.13.2 Examples

```
improper_style sqdistharm
improper_coeff 1 50.0 0.1
```

### 8.13.3 Description

The *sqdistharm* improper style uses the potential

$$E = K(d^2 - {d_0}^2)^2$$

where $d$ is the distance between the central atom and the plane formed by the other three atoms. If the 4 atoms in an improper quadruplet (listed in the data file read by the *read_data* command) are ordered I,J,K,L then the L-atom is assumed to be the central atom. Note that this is different from the convention used in the improper_style distance.

The following coefficients must be defined for each improper type via the improper_coeff command as in the example above, or in the data file or restart files read by the read_data or read_restart commands:

- $K$ (energy/distance^4)
- ${d_0}^2$ (distance^2)

Note that ${d_0}^2$ (in units distance^2) has be provided and not $d_0$.

---

### 8.13.4 Symmetry convention

For the *sqdistharm* improper style, the fourth atom in the quadruplet is the atom of symmetry; all other atoms are considered interchangeable. This convention is relevant for operations that require knowledge of how atoms are ordered, such as automatic assignment of new improper types by *fix bond/react*.

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 8.13.5 Restrictions

This improper style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* doc page for more info.

### 8.13.6 Related commands

*improper_coeff*

### 8.13.7 Default

none

## 8.14 improper_style umbrella command

Accelerator Variants: *umbrella/kk*, *umbrella/omp*

### 8.14.1 Syntax

```
improper_style umbrella
```

### 8.14.2 Examples

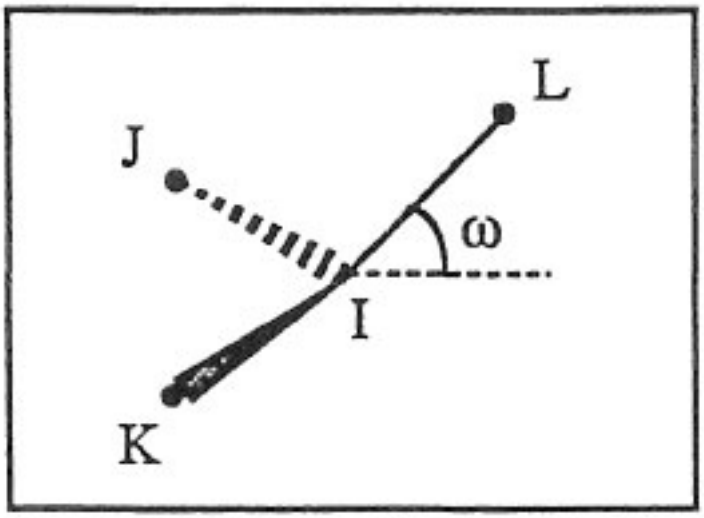
```
improper_style umbrella
improper_coeff 1 100.0 180.0
```

### 8.14.3 Description

The *umbrella* improper style uses the following potential, which is commonly referred to as a classic inversion and used in the *DREIDING* force field:

$$E = \frac{1}{2} K \left( \frac{1}{\sin \omega_0} \right)^2 (\cos \omega - \cos \omega_0)^2 \qquad \omega_0 \neq 0^o$$
$$E = K (1 - cos\omega) \qquad \omega_0 = 0^o$$

where $K$ is the force constant and $\omega$ is the angle between the IL axis and the IJK plane:




If $\omega_0 = 0$ the potential term has a minimum for the planar structure. Otherwise it has two minima at $\omega + / - \omega_0$, with a barrier in between.

See *(Mayo)* for a description of the DREIDING force field.

The following coefficients must be defined for each improper type via the *improper_coeff* command as in the example above, or in the data file or restart files read by the *read_data* or *read_restart* commands:

- $K$ (energy)
- $\omega_0$ (degrees)

---

Styles with a *gpu*, *intel*, *kk*, *omp*, or *opt* suffix are functionally the same as the corresponding style without the suffix. They have been optimized to run faster, depending on your available hardware, as discussed on the *Accelerator packages* page. The accelerated styles take the same arguments and should produce the same results, except for round-off and precision issues.

These accelerated styles are part of the GPU, INTEL, KOKKOS, OPENMP, and OPT packages, respectively. They are only enabled if LAMMPS was built with those packages. See the *Build package* page for more info.

You can specify the accelerated styles explicitly in your input script by including their suffix, or you can use the *-suffix command-line switch* when you invoke LAMMPS, or you can use the *suffix* command in your input script.

See the *Accelerator packages* page for more instructions on how to use the accelerated styles effectively.

---

### 8.14.4 Symmetry convention

For the *umbrella* improper style, the first and fourth atoms in the quadruplet are atoms of symmetry; only the second and third atoms are considered interchangeable. This convention is relevant for operations that require knowledge of how atoms are ordered, such as automatic assignment of new improper types by *fix bond/react*.

### 8.14.5 Restrictions

This improper style can only be used if LAMMPS was built with the MOLECULE package. See the *Build package* doc page for more info.

### 8.14.6 Related commands

*improper_coeff*

### 8.14.7 Default

none

---

**(Mayo)** Mayo, Olfason, Goddard III, J Phys Chem, 94, 8897-8909 (1990),

## 8.15 improper_style zero command

### 8.15.1 Syntax

```
improper_style zero [nocoeff]
```

### 8.15.2 Examples

```
improper_style zero
improper_style zero nocoeff
improper_coeff *
```

### 8.15.3 Description

Using an improper style of zero means improper forces and energies are not computed, but the geometry of improper quadruplets is still accessible to other commands.

As an example, the *compute improper/local* command can be used to compute the chi values for the list of quadruplets of improper atoms listed in the data file read by the *read_data* command. If no improper style is defined, this command cannot be used.

The optional *nocoeff* flag allows to read data files with a ImproperCoeff section for any improper style. Similarly, any improper_coeff commands will only be checked for the improper type number and the rest ignored.

Note that the *improper_coeff* command must be used for all improper types, though no additional values are specified.

### 8.15.4 Restrictions

none

### 8.15.5 Related commands

none

*improper_style none*

### 8.15.6 Default

none

CHAPTER

NINE

# DUMP STYLES

## 9.1 dump command

## 9.2 *dump vtk* command

## 9.3 *dump h5md* command

## 9.4 *dump molfile* command

## 9.5 *dump netcdf* command

## 9.6 *dump image* command

## 9.7 *dump movie* command

## 9.8 *dump atom/adios* command

## 9.9 *dump custom/adios* command

## 9.10 *dump cfg/uef* command

### 9.10.1 Syntax

```
dump ID group-ID style N file attribute1 attribute2 ...
```

- ID = user-assigned name for the dump
- group-ID = ID of the group of atoms to be dumped
- style = *atom* or *atom/adios* or *atom/gz* or *atom/zstd* or *cfg* or *cfg/gz* or *cfg/zstd* or *cfg/uef* or *custom* or *custom/gz* or *custom/zstd* or *custom/adios* or *dcd* or *extxyz* or *grid* or *grid/vtk* or *h5md* or *image* or *local* or *local/gz* or *local/zstd* or *molfile* or *movie* or *netcdf* or *netcdf/mpiio* or *vtk* or *xtc* or *xyz* or *xyz/gz* or *xyz/zstd* or *yaml*
- N = dump on timesteps which are multiples of N
- file = name of file to write dump info to
- attribute1,attribute2,... = list of attributes for a particular style

  *atom* attributes = none
  *atom/adios* attributes = none, discussed on *dump atom/adios* page
  *atom/gz* attributes = none

*atom/zstd* attributes = none
*cfg* attributes = same as *custom* attributes, see below
*cfg/gz* attributes = same as *custom* attributes, see below
*cfg/zstd* attributes = same as *custom* attributes, see below
*cfg/uef* attributes = same as *custom* attributes, discussed on *dump cfg/uef* page
*custom*, *custom/gz*, *custom/zstd* attributes = see below
*custom/adios* attributes = same as *custom* attributes, discussed on *dump custom/adios* page
*dcd* attributes = none
*extxyz* attributes = none
*h5md* attributes = discussed on *dump h5md* page
*grid* attributes = see below
*grid/vtk* attributes = see below
*image* attributes = discussed on *dump image* page
*local*, *local/gz*, *local/zstd* attributes = see below
*molfile* attributes = discussed on *dump molfile* page
*movie* attributes = discussed on *dump image* page
*netcdf* attributes = discussed on *dump netcdf* page
*netcdf/mpiio* attributes = discussed on *dump netcdf* page
*vtk* attributes = same as *custom* attributes, see below, also *dump vtk* page
*xtc* attributes = none
*xyz* attributes = none
*xyz/gz* attributes = none
*xyz/zstd* attributes = none
*yaml* attributes = same as *custom* attributes, see below

- *custom* or *custom/gz* or *custom/zstd* or *cfg* or *cfg/gz* or *cfg/zstd* or *cfg/uef* or *netcdf* or *netcdf/mpiio* or *yaml* attributes:

```
possible attributes = id, mol, proc, procp1, type, element, mass,
                      x, y, z, xs, ys, zs, xu, yu, zu,
                      xsu, ysu, zsu, ix, iy, iz,
                      vx, vy, vz, fx, fy, fz,
                      q, mux, muy, muz, mu,
                      radius, diameter, omegax, omegay, omegaz,
                      angmomx, angmomy, angmomz, tqx, tqy, tqz,
                      c_ID, c_ID[I], f_ID, f_ID[I], v_name,
                      i_name, d_name, i2_name[I], d2_name[I]
```

id = atom ID
mol = molecule ID
proc = ID of processor that owns atom
procp1 = ID+1 of processor that owns atom
type = atom type
typelabel = atom *type label*
element = name of atom element, as defined by *dump_modify* command
mass = atom mass
x,y,z = unscaled atom coordinates
xs,ys,zs = scaled atom coordinates
xu,yu,zu = unwrapped atom coordinates
xsu,ysu,zsu = scaled unwrapped atom coordinates
ix,iy,iz = box image that the atom is in
vx,vy,vz = atom velocities
fx,fy,fz = forces on atoms
q = atom charge

```
mux,muy,muz = orientation of dipole moment of atom
mu = magnitude of dipole moment of atom
radius,diameter = radius, diameter of spherical particle
omegax,omegay,omegaz = angular velocity of spherical particle
angmomx,angmomy,angmomz = angular momentum of aspherical particle
tqx,tqy,tqz = torque on finite-size particles
c_ID = per-atom vector calculated by a compute with ID
c_ID[I] = Ith column of per-atom array calculated by a compute with ID, I can␣
↪include wildcard (see below)
f_ID = per-atom vector calculated by a fix with ID
f_ID[I] = Ith column of per-atom array calculated by a fix with ID, I can include␣
↪wildcard (see below)
v_name = per-atom vector calculated by an atom-style variable with name
i_name = custom integer vector with name
d_name = custom floating point vector with name
i2_name[I] = Ith column of custom integer array with name, I can include wildcard␣
↪(see below)
d2_name[I] = Ith column of custom floating point vector with name, I can include␣
↪wildcard (see below)
```

- *local* or *local/gz* or *local/zstd* attributes:

```
possible attributes = index, c_ID, c_ID[I], f_ID, f_ID[I]
  index = enumeration of local values
  c_ID = local vector calculated by a compute with ID
  c_ID[I] = Ith column of local array calculated by a compute with ID, I can␣
↪include wildcard (see below)
  f_ID = local vector calculated by a fix with ID
  f_ID[I] = Ith column of local array calculated by a fix with ID, I can include␣
↪wildcard (see below)
```

- *grid* or *grid/vtk* attributes:

```
possible attributes = c_ID:gname:dname, c_ID:gname:dname[I], f_ID:gname:dname, f_
↪ID:gname:dname[I]
  gname = name of grid defined by compute or fix
  dname = name of data field defined by compute or fix
  c_ID = per-grid vector calculated by a compute with ID
  c_ID[I] = Ith column of per-grid array calculated by a compute with ID, I can␣
↪include wildcard (see below)
  f_ID = per-grid vector calculated by a fix with ID
  f_ID[I] = Ith column of per-grid array calculated by a fix with ID, I can include␣
↪wildcard (see below)
```

### 9.10.2 Examples

```
dump myDump all atom 100 dump.lammpstrj
dump myDump all atom/gz 100 dump.atom.gz
dump myDump all atom/zstd 100 dump.atom.zst
dump 2 subgroup atom 50 dump.run.bin
dump 4a all custom 100 dump.myforce.* id type x y vx fx
dump 4a all custom 100 dump.myvel.lammpsbin id type x y z vx vy vz
dump 4b flow custom 100 dump.%.myforce id type c_myF[3] v_ke
dump 4b flow custom 100 dump.%.myforce id type c_myF[*] v_ke
```

(continues on next page)

(continued from previous page)

```
dump 2 inner cfg 10 dump.snap.*.cfg mass type xs ys zs vx vy vz
dump snap all cfg 100 dump.config.*.cfg mass type xs ys zs id type c_Stress[2]
dump 1 all xtc 1000 file.xtc
```

### 9.10.3 Description

Dump a snapshot of quantities to one or more files once every *N* timesteps in one of several styles. The timesteps on which dump output is written can also be controlled by a variable. See the *dump_modify every* command.

Almost all the styles output per-atom data, i.e. one or more values per atom. The exceptions are as follows. The *local* styles output one or more values per bond (angle, dihedral, improper) or per pair of interacting atoms (force or neighbor interactions). The *grid* styles output one or more values per grid cell, which are produced by other commands which overlay the simulation domain with a regular grid. See the *Howto grid* doc page for details. The *image* style renders a JPG, PNG, or PPM image file of the system for each snapshot, while the *movie* style combines and compresses the series of images into a movie file; both styles are discussed in detail on the *dump image* page.

Only information for atoms in the specified group is dumped. The *dump_modify thresh and region and refresh* commands can also alter what atoms are included. Not all styles support these options; see details on the *dump_modify* doc page.

As described below, the filename determines the kind of output: text or binary or compressed, one big file or one per timestep, one file for all the processors or multiple smaller files.

**Note**

Because periodic boundary conditions are enforced only on timesteps when neighbor lists are rebuilt, the coordinates of an atom written to a dump file may be slightly outside the simulation box. Re-neighbor timesteps will not typically coincide with the timesteps dump snapshots are written. See the *dump_modify pbc* command if you wish to force coordinates to be strictly inside the simulation box.

**Note**

Unless the *dump_modify sort* option is invoked, the lines of atom or grid information written to dump files (typically one line per atom or grid cell) will be in an indeterminate order for each snapshot. This is even true when running on a single processor, if the *atom_modify sort* option is on, which it is by default. In this case atoms are re-ordered periodically during a simulation, due to spatial sorting. It is also true when running in parallel, because data for a single snapshot is collected from multiple processors, each of which owns a subset of the atoms.

**Warning**

Without either including atom IDs or using the *dump_modify sort* option, it is impossible for visualization programs (e.g. OVITO or VMD) or analysis tools to assign data in different frames consistently to the same atom. This can lead to incorrect visualizations or results. LAMMPS will print a warning in such cases.

For the *atom*, *custom*, *cfg*, *grid*, and *local* styles, sorting is off by default. For the *dcd*, *extxyz*, *grid/vtk*, *xtc*, *xyz*, and *molfile* styles, sorting by atom ID or grid ID is on by default. See the *dump_modify* page for details.

The *style* keyword determines what kind of data is written to the dump file(s) and in what format.

Note that *atom*, *custom*, *dcd*, *extxyz*, *xtc*, *xyz*, and *yaml* style dump files can be read directly by VMD, a popular tool for visualizing and analyzing trajectories from atomic and molecular systems. For reading *netcdf* style dump files, the

netcdf plugin needs to be recompiled from source using a NetCDF version compatible with the one used by LAMMPS. The bundled plugin binary uses a very old version of NetCDF that is not compatible with LAMMPS.

Likewise the OVITO visualization package, popular for materials modeling, can read the *atom*, *custom*, *extxyz*, *local*, *xtc*, *cfg*, *netcdf*, and *xyz* style atom dump files directly. With version 3.8 and above, OVITO can also read and visualize *grid* style dump files with grid cell data, including iso-surface images of the grid cell values.

Note that settings made via the *dump_modify* command can also alter the format of individual values and content of the dump file itself. This includes the precision of values output to text-based dump files which is controlled by the *dump_modify format* command and its options.

---

Format of native LAMMPS format dump files:

The *atom*, *custom*, *grid*, and *local* styles create files in a simple LAMMPS-specific text format that is mostly self-explanatory when viewing a dump file. Many post-processing tools either included with LAMMPS or third-party tools can read this format, as does the *rerun* command. See tools described on the *Tools* doc page for examples, including Pizza.py.

For all these styles, the dimensions of the simulation box are included in each snapshot. The simulation box in LAMMPS can be defined in one of 3 ways: orthogonal, restricted triclinic, and general triclinic. See the *Howto triclinic* doc page for a detailed description of all 3 options.

For an orthogonal simulation box the box information is formatted as:

```
ITEM: BOX BOUNDS xx yy zz
xlo xhi
ylo yhi
zlo zhi
```

where xlo,xhi are the maximum extents of the simulation box in the *x*-dimension, and similarly for *y* and *z*. The "xx yy zz" terms are six characters that encode the style of boundary for each of the six simulation box boundaries (xlo,xhi; ylo,yhi; and zlo,zhi). Each of the six characters is one of *p* (periodic), *f* (fixed), *s* (shrink wrap), or *m* (shrink wrapped with a minimum value). See the *boundary* command for details.

For a restricted triclinic simulation box, an orthogonal bounding box which encloses the restricted triclinic simulation box is output, along with the three tilt factors (*xy*, *xz*, *yz*) of the triclinic box, formatted as follows:

```
ITEM: BOX BOUNDS xy xz yz xx yy zz
xlo_bound xhi_bound xy
ylo_bound yhi_bound xz
zlo_bound zhi_bound yz
```

The presence of the text "xy xz yz" in the ITEM line indicates that the three tilt factors will be included on each of the three following lines. This bounding box is convenient for many visualization programs. The meaning of the six character flags for "xx yy zz" is the same as above.

Note that the first two numbers on each line are now xlo_bound instead of xlo, etc. because they represent a bounding box. See the *Howto triclinic* page for a geometric description of triclinic boxes, as defined by LAMMPS, simple formulas for how the six bounding box extents (xlo_bound, xhi_bound, etc.) are calculated from the triclinic parameters, and how to transform those parameters to and from other commonly used triclinic representations.

For a general triclinic simulation box, see the "General triclinic" section below for a description of the ITEM: BOX BOUNDS format as well as how per-atom coordinates and per-atom vector quantities are output.

The *atom* and *custom* styles output a "ITEM: NUMBER OF ATOMS" line with the count of atoms in the snapshot. Likewise they output an "ITEM: ATOMS" line which includes column descriptors for the per-atom lines that follow. For example, the descriptors would be "id type xs ys zs" for the default *atom* style, and would be the atom attributes you specify in the dump command for the *custom* style. Each subsequent line will list the data for a single atom.

For style *atom*, atom coordinates are written to the file, along with the atom ID and atom type. By default, atom coords are written in a scaled format (from 0 to 1). That is, an *x* value of 0.25 means the atom is at a location 1/4 of the distance from *xlo* to *xhi* of the box boundaries. The format can be changed to unscaled coords via the *dump_modify* settings. Image flags can also be added for each atom via dump_modify.

Style *custom* allows you to specify a list of atom attributes to be written to the dump file for each atom. Possible attributes are listed above and will appear in the order specified. You cannot specify a quantity that is not defined for a particular simulation—such as *q* for atom style *bond*, since that atom style does not assign charges. Dumps occur at the very end of a timestep, so atom attributes will include effects due to fixes that are applied during the timestep. An explanation of the possible dump custom attributes is given below.

Added in version 22Dec2022.

For style *grid* the dimension of the simulation domain and size of the Nx by Ny by Nz grid that overlays the simulation domain are also output with each snapshot:

```
ITEM: DIMENSION
dim
ITEM: GRID SIZE
nx ny nz
```

The value dim will be 2 or 3 for 2d or 3d simulations. It is included so that post-processing tools like OVITO, which can visualize grid-based quantities know how to draw each grid cell. The grid size will match the input script parameters for grid(s) created by the computes or fixes which are referenced by the the dump command. For 2d simulations (and grids), nz will always be 1.

There will also be an "ITEM: GRID DATA" line which includes column descriptors for the per grid cell data. Each subsequent line (Nx * Ny * Nz lines) will list the data for a single grid cell. If grid cell IDs are included in the output via the *compute property/grid* command, then the IDs will range from 1 to N = Nx*Ny*Nz. The ordering of IDs is with the x index varying fastest, then the y index, and the z index varying slowest.

For style *local*, local output generated by *computes* and *fixes* is used to generate lines of output that is written to the dump file. This local data is typically calculated by each processor based on the atoms it owns, but there may be zero or more entities per atom (e.g., a list of bond distances). An explanation of the possible dump local attributes is given below. Note that by using input from the *compute property/local* command with dump local, it is possible to generate information on bonds, angles, etc. that can be cut and pasted directly into a data file read by the *read_data* command.

---

Dump files in other popular formats:

**Note**

This section only discusses file formats relevant to this doc page. The top of this page has links to other dump commands (with their own pages) which write files in additional popular formats.

Style *cfg* has the same command syntax as style *custom* and writes extended CFG format files, as used by the AtomEye visualization package. Since the extended CFG format uses a single snapshot of the system per file, a wildcard "*" must be included in the filename, as discussed below. The list of atom attributes for style *cfg* must begin with either "mass type xs ys zs" or "mass type xsu ysu zsu" since these quantities are needed to write the CFG files in the appropriate format (though the "mass" and "type" fields do not appear explicitly in the file). Any remaining attributes will be stored as "auxiliary properties" in the CFG files. Note that you will typically want to use the *dump_modify element* command with CFG-formatted files, to associate element names with atom types, so that AtomEye can render atoms appropriately. When unwrapped coordinates *xsu*, *ysu*, and *zsu* are requested, the nominal AtomEye periodic cell dimensions are expanded by a large factor UNWRAPEXPAND = 10.0, which ensures atoms that are displayed correctly for up to UNWRAPEXPAND/2 periodic boundary crossings in any direction. Beyond this, AtomEye will rewrap the unwrapped

coordinates. The expansion causes the atoms to be drawn farther away from the viewer, but it is easy to zoom the atoms closer, and the interatomic distances are unaffected.

The *dcd* style writes DCD files, a standard atomic trajectory format used by the CHARMM, NAMD, and XPlor molecular dynamics packages. DCD files are binary and thus may not be portable to different machines. The number of atoms per snapshot cannot change with the *dcd* style. The *unwrap* option of the *dump_modify* command allows DCD coordinates to be written "unwrapped" by the image flags for each atom. Unwrapped means that if the atom has passed through a periodic boundary one or more times, the value is printed for what the coordinate would be if it had not been wrapped back into the periodic box. Note that these coordinates may thus be far outside the box size stored with the snapshot.

The *xtc* style writes XTC files, a compressed trajectory format used by the GROMACS molecular dynamics package, and described here. The precision used in XTC files can be adjusted via the *dump_modify* command. The default value of 1000 means that coordinates are stored to 1/1000 nanometer accuracy. XTC files are portable binary files written in the NFS XDR data format, so that any machine which supports XDR should be able to read them. The number of atoms per snapshot cannot change with the *xtc* style. The *unwrap* option of the *dump_modify* command allows XTC coordinates to be written "unwrapped" by the image flags for each atom. Unwrapped means that if the atom has passed through a periodic boundary one or more times, the value is printed for what the coordinate would be if it had not been wrapped back into the periodic box. Note that these coordinates may thus be far outside the box size stored with the snapshot.

The *xyz* style writes XYZ files, which is a simple text-based coordinate format that many codes can read. Specifically it has a line with the number of atoms, then a comment line that is usually ignored followed by one line per atom with the atom type and the *x*-, *y*-, and *z*-coordinate of that atom. You can use the *dump_modify element* option to change the output from using the (numerical) atom type to an element name (or some other label). This option will help many visualization programs to guess bonds and colors. You can use the *dump_modify types labels* option to replace numeric atom types with *type labels*.

Added in version 2Apr2025.

The *extxyz* style writes XYZ files compatible with the Extended XYZ (or ExtXYZ) format as defined as defined in the libAtoms specification. Specifically, the following information will be dumped:

- timestep
- time, which can be disabled with *dump_modify time no*
- simulation box lattice and pbc conditions
- atomic forces, which can be disabled with *dump_modify forces no*
- atomic velocities, which can be disabled with *dump_modify vel no*
- atomic masses, if enabled with *dump_modify mass yes*

Dump style *extxyz* requires either that a *type label map for atoms types* is defined or *dump_modify element* is used to set up an atom type number to atom name mapping.

Added in version 22Dec2022.

The *grid/vtk* style writes VTK files for grid data on a regular rectilinear grid. Its content is conceptually similar to that of the text file produced by the *grid* style, except that it in an XML-based format which visualization programs which support the VTK format can read, e.g. the ParaView tool. For this style, there can only be 1 or 3 per grid cell attributes specified. If it is a single value, it is a scalar quantity. If 3 values are specified it is encoded in the VTK file as a vector quantity (for each grid cell). The filename for this style must include a "*" wildcard character to produce one file per snapshot; see details below.

Added in version 4May2022.

Dump style *yaml* has the same command syntax as style *custom* and writes YAML format files that can be easily parsed by a variety of data processing tools and programming languages. Each timestep will be written as a YAML "document" (i.e., starts with "—" and ends with "..."). The style supports writing one file per timestep through the

"*" wildcard but not multi-processor outputs with the "%" token in the filename. In addition to per-atom data, *thermo* data can be included in the *yaml* style dump file using the *dump_modify thermo yes*. The data included in the dump file uses the "thermo" tag and is otherwise identical to data specified by the *thermo_style* command.

Below is an example for a YAML format dump created by the following commands.

```
dump out all yaml 100 dump.yaml id type x y z vx vy vz ix iy iz
dump_modify out time yes units yes thermo yes format 1 %5d format "% 10.6e"
```

The tags "time", "units", and "thermo" are optional and enabled by the dump_modify command. The list under the "box" tag has three lines for orthogonal boxes and four lines for triclinic boxes, where the first three are the box boundaries and the fourth the three tilt factors (*xy*, *xz*, *yz*). The "thermo" data follows the format of the *yaml* thermo style. The "keywords" tag lists the per-atom properties contained in the "data" columns, which contain a list with one line per atom. The keywords may be renamed using the dump_modify command same as for the *custom* dump style.

```
---
creator: LAMMPS
timestep: 0
units: lj
time: 0
natoms: 4000
boundary: [ p, p, p, p, p, p, ]
thermo:
  - keywords: [ Step, Temp, E_pair, E_mol, TotEng, Press, ]
  - data: [ 0, 0, -27093.472213010766, 0, 0, 0, ]
box:
  - [ 0, 16.795961913825074 ]
  - [ 0, 16.795961913825074 ]
  - [ 0, 16.795961913825074 ]
  - [ 0, 0, 0 ]
keywords: [ id, type, x, y, z, vx, vy, vz, ix, iy, iz,  ]
data:
  - [     1 , 1 ,  0.000000e+00 ,  0.000000e+00 ,  0.000000e+00 ,  -1.841579e-01 , -9.
↪710036e-01 , -2.934617e+00 , 0 , 0 , 0, ]
  - [     2 , 1 ,  8.397981e-01 ,  8.397981e-01 ,  0.000000e+00 ,  -1.799591e+00 ,  2.
↪127197e+00 ,  2.298572e+00 , 0 , 0 , 0, ]
  - [     3 , 1 ,  8.397981e-01 ,  0.000000e+00 ,  8.397981e-01 ,  -1.807682e+00 , -9.
↪585130e-01 ,  1.605884e+00 , 0 , 0 , 0, ]

  [...]
...
---
timestep: 100
units: lj
time: 0.5

  [...]

...
```

Frequency of dump output:

Dumps are performed on timesteps that are a multiple of *N* (including timestep 0) and on the last timestep of a minimization if the minimization converges. Note that this means a dump will not be performed on the initial timestep after

the dump command is invoked, if the current timestep is not a multiple of *N*. This behavior can be changed via the *dump_modify first* command, which can also be useful if the dump command is invoked after a minimization ended on an arbitrary timestep.

The value of *N* can be changed between runs by using the *dump_modify every* command (not allowed for *dcd* style). The *dump_modify every* command also allows a variable to be used to determine the sequence of timesteps on which dump files are written. In this mode a dump on the first timestep of a run will also not be written unless the *dump_modify first* command is used.

If you instead want to dump snapshots based on simulation time (in time units of the *units command* command), the *dump_modify every/time* command can be used. This can be useful when the timestep size varies during a simulation run, e.g. by use of the *fix dt/reset* command.

---

Dump filenames:

The specified dump filename determines how the dump file(s) is written. The default is to write one large text file, which is opened when the dump command is invoked and closed when an *undump* command is used or when LAMMPS exits. For the *dcd* and *xtc* styles, this is a single large binary file.

Many of the styles allow dump filenames to contain either or both of two wildcard characters. If a "*" character appears in the filename, then one file per snapshot is written and the "*" character is replaced with the timestep value. For example, tmp.dump.* becomes tmp.dump.0, tmp.dump.10000, tmp.dump.20000, etc. This option is not available for the *dcd* and *xtc* styles. Note that the *dump_modify pad* command can be used to ensure all timestep numbers are the same length (e.g., 00010), which can make it easier to read a series of dump files in order with some post-processing tools.

If a "%" character appears in the filename, then each of P processors writes a portion of the dump file, and the "%" character is replaced with the processor ID from 0 to $P-1$. For example, tmp.dump.% becomes tmp.dump.0, tmp.dump.1, ... tmp.dump.:math:*P-1*, etc. This creates smaller files and can be a fast mode of output on parallel machines that support parallel I/O for output. This option is **not** available for the *dcd*, *extxyz*, *xtc*, *xyz*, *grid/vtk*, and *yaml* styles.

By default, *P* is the the number of processors, meaning one file per processor, but *P* can be set to a smaller value via the *nfile* or *fileper* keywords of the *dump_modify* command. These options can be the most efficient way of writing out dump files when running on large numbers of processors.

Note that using the "*" and "%" characters together can produce a large number of small dump files!

Deprecated since version 21Nov2023.

The MPIIO package and the the corresponding "/mpiio" dump styles, except for the unrelated "netcdf/mpiio" style were removed from LAMMPS.

---

Compression of dump file data:

If the specified filename ends with ".bin" or ".lammpsbin", the dump file (or files, if "*" or "%" is also used) is written in binary format. A binary dump file will be about the same size as a text version, but will typically write out much faster. Of course, when post-processing, you will need to convert it back to text format (see the *binary2txt tool*) or write your own code to read the binary file. The format of the binary file can be understood by looking at the `tools/binary2txt.cpp` file. This option is only available for the *atom* and *custom* styles.

If LAMMPS has been compiled with the *corresponding setting* and if the filename ends with ".gz" or some other *supported compression format suffix*, the dump file (or files, if "*" or "%" is also used) is written in compressed format. A compressed dump file will be about 3× smaller than the text version, but will also take longer to write. This option is not available for the *dcd* and *xtc* styles.

Note that styles that end with *gz* are identical in command syntax to the corresponding styles without "gz", however, they generate compressed files using the zlib library. Thus the filename suffix ".gz" is mandatory. This is an alternative approach to writing compressed files via a pipe (see above), as done by the regular dump styles, which may be required

on HPC clusters where the interface to the high-speed network disallows using the fork() library call (which is needed for a pipe). For the remainder of this page, you should thus consider the *atom* and *atom/gz* styles (etc.) to be interchangeable, with the exception of the required filename suffix.

Similarly, styles that end with *zstd* are identical to the gz styles, but use the Zstd compression library instead and require a ".zst" suffix. See the *dump_modify* page for details on how to control the compression level in both variants.

---

General triclinic simulation box output for the *atom* and *custom* styles:

As mentioned above, the simulation box can be defined as a general triclinic box, which means that 3 arbitrary box edge vectors **A**, **B**, **C** can be specified. See the *Howto triclinic* doc page for a detailed description of general triclinic boxes.

This option is provided as a convenience for users who may be converting data from solid-state crystallographic representations or from DFT codes for input to LAMMPS. However, as explained on the *Howto_triclinic* doc page, internally, LAMMPS only uses restricted triclinic simulation boxes. This means the box and per-atom information (e.g. coordinates, velocities) LAMMPS stores are converted (rotated) from general to restricted triclinic form when the system is created.

For dump output, if the *dump_modify triclinic/general* command is used, the box description and per-atom coordinates and other per-atom vectors will be converted (rotated) from restricted to general form when each dump file snapshots is output. This option can only be used if the simulation box was initially created as general triclinic. If the option is not used, and the simulation box is general triclinic, then the dump file snapshots will reflect the internal restricted triclinic geometry.

The dump_modify triclinic/general option affects 3 aspects of the dump file output.

First, the format for the BOX BOUNDS is as follows

```
ITEM: BOX BOUNDS abc origin
ax ay az originx
bx by bz originy
cx cy cz originz
```

where the **A** edge vector of the box is (ax,ay,az) and similarly for **B** and **C**. The origin of all 3 edge vectors is (originx, originy, originz).

Second, the coordinates of each atom are converted (rotated) so that the atom is inside (or near) the general triclinic box defined by the **A**, **B**, **C** edge vectors. For style *atom*, this only alters output for unscaled atom coords, via the *dump_modify scaled no* setting. For style *custom*, this alters output for either unscaled or unwrapped output of atom coords, via the *x,y,z* or *xu,yu,zu* attributes. For output of scaled atom coords by both styles, there is no difference between restricted and general triclinic values.

Third, the output for any attribute of the *custom* style which represents a per-atom vector quantity will be converted (rotated) to be oriented consistent with the general triclinic box and its orientation relative to the standard xyz coordinate axes.

This applies to the following *custom* style attributes:

- vx,vy,vz = atom velocities
- fx,fy,fz = forces on atoms
- mux,muy,muz = orientation of dipole moment of atom
- omegax,omegay,omegaz = angular velocity of spherical particle
- angmomx,angmomy,angmomz = angular momentum of aspherical particle
- tqx,tqy,tqz = torque on finite-size particles

For example, if the velocity of an atom in a restricted triclinic box is along the x-axis, then it will be output for a general triclinic box as a vector along the **A** edge vector of the box.

**Note**

For style *custom*, the *dump_modify thresh* command may access per-atom attributes either directly or indirectly through a compute or variable. If the attribute is an atom coordinate or one of the vectors mentioned above, its value will *NOT* be a general triclinic (rotated) value. Rather it will be a restricted triclinic value.

---

Arguments for different styles:

The sections below describe per-atom, local, and per grid cell attributes which can be used as arguments to the various styles.

Note that in the discussion below, for styles which can reference values from a compute or fix or custom atom property, like the *custom*, *cfg*, *grid* or *local* styles, the bracketed index *i* can be specified using a wildcard asterisk with the index to effectively specify multiple values. This takes the form "*" or "*n" or "m*" or "m*n". If *N* is the number of columns in the array, then an asterisk with no numeric values means all column indices from 1 to *N*. A leading asterisk means all indices from 1 to n (inclusive). A trailing asterisk means all indices from m to *N* (inclusive). A middle asterisk means all indices from m to n (inclusive).

Using a wildcard is the same as if the individual columns of the array had been listed one by one. For example, these two dump commands are equivalent, since the *compute stress/atom* command creates a per-atom array with six columns:

```
compute myPress all stress/atom NULL
dump 2 all custom 100 tmp.dump id myPress[*]
dump 2 all custom 100 tmp.dump id myPress[1] myPress[2] myPress[3] &
                                  myPress[4] myPress[5] myPress[6]
```

---

Per-atom attributes used as arguments to the *custom* and *cfg* styles:

The *id*, *mol*, *proc*, *procp1*, *type*, *typelabel*, *element*, *mass*, *vx*, *vy*, *vz*, *fx*, *fy*, *fz*, *q* attributes are self-explanatory.

*Id* is the atom ID. *Mol* is the molecule ID, included in the data file for molecular systems. *Proc* is the ID of the processor (0 to $N_{\text{procs}} - 1$) that currently owns the atom. *Procp1* is the proc ID+1, which can be convenient in place of a *type* attribute (1 to $N_{\text{types}}$) for coloring atoms in a visualization program. *Type* is the atom type (1 to $N_{\text{types}}$). *Typelabel* is the atom *type label*. *Element* is typically the chemical name of an element, which you must assign to each type via the *dump_modify element* command. More generally, it can be any string you wish to associated with an atom type. *Mass* is the atom mass. The quantities *vx*, *vy*, *vz*, *fx*, *fy*, *fz*, and *q* are components of atom velocity and force and atomic charge.

There are several options for outputting atom coordinates. The *x*, *y*, and *z* attributes write atom coordinates "unscaled", in the appropriate distance *units* (Å, $\sigma$, etc.). Use *xs*, *ys*, and *zs* if you want the coordinates "scaled" to the box size so that each value is 0.0 to 1.0. If the simulation box is triclinic (tilted), then all atom coords will still be between 0.0 and 1.0. The actual unscaled $(x, y, z)$ coordinate is $x_s a + y_s b + z_s c$, where $(a, b, c)$ are the non-orthogonal vectors of the simulation box edges, as discussed on the *Howto triclinic* page.

Use *xu*, *yu*, and *zu* if you want the coordinates "unwrapped" by the image flags for each atom. Unwrapped means that if the atom has passed through a periodic boundary one or more times, the value is printed for what the coordinate would be if it had not been wrapped back into the periodic box. Note that using *xu*, *yu*, and *zu* means that the coordinate values may be far outside the box bounds printed with the snapshot. Using *xsu*, *ysu*, and *zsu* is similar to using *xu*, *yu*, and *zu*, except that the unwrapped coordinates are scaled by the box size. Atoms that have passed through a periodic boundary will have the corresponding coordinate increased or decreased by 1.0.

The image flags can be printed directly using the *ix*, *iy*, and *iz* attributes. For periodic dimensions, they specify which image of the simulation box the atom is considered to be in. An image of 0 means it is inside the box as defined. A value of 2 means add 2 box lengths to get the true value. A value of −1 means subtract 1 box length to get the true value. LAMMPS updates these flags as atoms cross periodic boundaries during the simulation.

The *mux*, *muy*, and *muz* attributes are specific to dipolar systems defined with an atom style of *dipole*. They give the orientation of the atom's point dipole moment. The *mu* attribute gives the magnitude of the atom's dipole moment.

The *radius* and *diameter* attributes are specific to spherical particles that have a finite size, such as those defined with an atom style of *sphere*.

The *omegax*, *omegay*, and *omegaz* attributes are specific to finite-size spherical particles that have an angular velocity. Only certain atom styles, such as *sphere*, define this quantity.

The *angmomx*, *angmomy*, and *angmomz* attributes are specific to finite-size aspherical particles that have an angular momentum. Only the *ellipsoid* atom style defines this quantity.

The *tqx*, *tqy*, and *tqz* attributes are for finite-size particles that can sustain a rotational torque due to interactions with other particles.

The *c_ID* and *c_ID[I]* attributes allow per-atom vectors or arrays calculated by a *compute* to be output. The ID in the attribute should be replaced by the actual ID of the compute that has been defined previously in the input script. See the *compute* command for details. There are computes for calculating the per-atom energy, stress, centro-symmetry parameter, and coordination number of individual atoms.

Note that computes which calculate global or local quantities, as opposed to per-atom quantities, cannot be output in a dump custom command. Instead, global quantities can be output by the *thermo_style custom* command, and local quantities can be output by the dump local command.

If *c_ID* is used as a attribute, then the per-atom vector calculated by the compute is printed. If *c_ID[i]* is used, then *i* must be in the range from 1 to *M*, which will print the *i*th column of the per-atom array with *M* columns calculated by the compute. See the discussion above for how *i* can be specified with a wildcard asterisk to effectively specify multiple values.

The *f_ID* and *f_ID[I]* attributes allow vector or array per-atom quantities calculated by a *fix* to be output. The ID in the attribute should be replaced by the actual ID of the fix that has been defined previously in the input script. The *fix ave/atom* command is one that calculates per-atom quantities. Since it can time-average per-atom quantities produced by any *compute*, *fix*, or atom-style *variable*, this allows those time-averaged results to be written to a dump file.

If *f_ID* is used as a attribute, then the per-atom vector calculated by the fix is printed. If *f_ID[i]* is used, then *i* must be in the range from 1 to *M*, which will print the *i*th column of the per-atom array with *M* columns calculated by the fix. See the discussion above for how *i* can be specified with a wildcard asterisk to effectively specify multiple values.

The *v_name* attribute allows per-atom vectors calculated by a *variable* to be output. The name in the attribute should be replaced by the actual name of the variable that has been defined previously in the input script. Only an atom-style variable can be referenced, since it is the only style that generates per-atom values. Variables of style *atom* can reference individual atom attributes, per-atom attributes, thermodynamic keywords, or invoke other computes, fixes, or variables when they are evaluated, so this is a very general means of creating quantities to output to a dump file.

The *i_name*, *d_name*, *i2_name*, *d2_name* attributes refer to custom per-atom integer and floating-point vectors or arrays that have been added via the *fix property/atom* command. When that command is used specific names are given to each attribute which are the "name" portion of these keywords. For arrays *i2_name* and *d2_name*, the column of the array must also be included following the name in brackets (e.g., d2_xyz[i], i2_mySpin[i], where *i* is in the range from 1 to *M*, where *M* is the number of columns in the custom array). See the discussion above for how *i* can be specified with a wildcard asterisk to effectively specify multiple values.

See the *Modify* page for information on how to add new compute and fix styles to LAMMPS to calculate per-atom quantities which could then be output into dump files.

---

Attributes used as arguments to the *local* style:

The *index* attribute can be used to generate an index number from 1 to N for each line written into the dump file, where N is the total number of local datums from all processors, or lines of output that will appear in the snapshot. Note that because data from different processors depend on what atoms they currently own, and atoms migrate between processor, there is no guarantee that the same index will be used for the same info (e.g. a particular bond) in successive snapshots.

The *c_ID* and *c_ID[I]* attributes allow local vectors or arrays calculated by a *compute* to be output. The ID in the attribute should be replaced by the actual ID of the compute that has been defined previously in the input script. See the *compute* command for details. There are computes for calculating local information such as indices, types, and energies for bonds and angles.

Note that computes which calculate global or per-atom quantities, as opposed to local quantities, cannot be output in a dump local command. Instead, global quantities can be output by the *thermo_style custom* command, and per-atom quantities can be output by the dump custom command.

If *c_ID* is used as a attribute, then the local vector calculated by the compute is printed. If *c_ID[I]* is used, then I must be in the range from 1-M, which will print the Ith column of the local array with M columns calculated by the compute. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

The *f_ID* and *f_ID[I]* attributes allow local vectors or arrays calculated by a *fix* to be output. The ID in the attribute should be replaced by the actual ID of the fix that has been defined previously in the input script.

If *f_ID* is used as a attribute, then the local vector calculated by the fix is printed. If *f_ID[I]* is used, then I must be in the range from 1-M, which will print the Ith column of the local with M columns calculated by the fix. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

Here is an example of how to dump bond info for a system, including the distance and energy of each bond:

```
compute 1 all property/local batom1 batom2 btype
compute 2 all bond/local dist engpot
dump 1 all local 1000 tmp.dump index c_1[1] c_1[2] c_1[3] c_2[1] c_2[2]
```

---

Attributes used as arguments to the *grid* and *grid/vtk* styles:

The attributes that begin with *c_ID* and *f_ID* both take colon-separated fields *gname* and *dname*. These refer to a grid name and data field name which is defined by the compute or fix. Note that a compute or fix can define one or more grids (of different sizes) and one or more data fields for each of those grids. The sizes of all grids output in a single dump grid command must be the same.

The *c_ID:gname:dname* and *c_ID:gname:dname[I]* attributes allow per-grid vectors or arrays calculated by a *compute* to be output. The ID in the attribute should be replaced by the actual ID of the compute that has been defined previously in the input script.

If *c_ID:gname:dname* is used as a attribute, then the per-grid vector calculated by the compute is printed. If *c_ID:gname:dname[I]* is used, then I must be in the range from 1-M, which will print the Ith column of the per-grid array with M columns calculated by the compute. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

The *f_ID:gname:dname* and *f_ID:gname:dname[I]* attributes allow per-grid vectors or arrays calculated by a *fix* to be output. The ID in the attribute should be replaced by the actual ID of the fix that has been defined previously in the input script.

If *f_ID:gname:dname* is used as a attribute, then the per-grid vector calculated by the fix is printed. If *f_ID:gname:dname[I]* is used, then I must be in the range from 1-M, which will print the Ith column of the per-grid with M columns calculated by the fix. See the discussion above for how I can be specified with a wildcard asterisk to effectively specify multiple values.

---

### 9.10.4 Restrictions

To write compressed dump files, you must either compile LAMMPS with the `-DLAMMPS_GZIP` option or use the styles from the COMPRESS package. See the *Build settings* page for details.

To create images or movies, you must install the GRAPHICS package. See the *Build extras* page for details.

While a dump command is active (i.e., has not been stopped by using the *undump command*), no commands may be used that will change the timestep (e.g., *reset_timestep*). LAMMPS will terminate with an error otherwise.

The *atom/gz*, *cfg/gz*, *custom/gz*, and *xyz/gz* styles are part of the COMPRESS package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The *dcd*, *extxyz*, *xtc*, and *yaml* styles are part of the EXTRA-DUMP package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

### 9.10.5 Related commands

*dump atom/adios*, *dump custom/adios*, *dump cfg/uef*, *dump h5md*, *dump image*, *dump molfile*, *dump netcdf*, *dump netcdf/mpiio*, *dump_modify*, *undump*, *write_dump*

### 9.10.6 Default

The defaults for the *image* and *movie* styles are listed on the *dump image* page.

## 9.11 dump atom/adios command

## 9.12 dump custom/adios command

### 9.12.1 Syntax

```
dump ID group-ID atom/adios N file.bp
dump ID group-ID custom/adios N file.bp args
```

- ID = user-assigned name for the dump
- group-ID = ID of the group of atoms to be imaged
- adios = style of dump command (other styles *atom* or *cfg* or *dcd* or *xtc* or *xyz* or *local* or *custom* are discussed on the *dump* doc page)
- N = dump every this many timesteps
- file.bp = name of file/stream to write to
- args = same options as in *dump custom* command

### 9.12.2 Examples

```
dump adios1 all atom/adios   100 atoms.bp
dump 4a     all custom/adios 100 dump_adios.bp id v_p x y z
dump 2 subgroup custom/adios 100 dump_adios.bp mass type xs ys zs vx vy vz
```

### 9.12.3 Description

Dump a snapshot of atom coordinates every *N* timesteps in the ADIOS-based "BP" file format, or using different I/O solutions in ADIOS, to a stream that can be read on-line by another program. ADIOS-BP files are binary, portable, and self-describing.

> **Note**
>
> To be able to use ADIOS, a file `adios2_config.xml` with specific configuration settings is expected in the current working directory. If the file is not present, LAMMPS will try to create a minimal default file. Please refer to the ADIOS documentation for details on how to adjust this file for optimal performance and desired features.

**Use from write_dump:**

It is possible to use these dump styles with the *write_dump* command. In this case, the sub-intervals must not be set at all. The write_dump command can be used to create a new file at each individual dump.

```
dump 4     all atom/adios 100 dump.bp
write_dump all atom/adios singledump.bp
```

---

### 9.12.4 Restrictions

The number of atoms per snapshot **can** change with the adios style. When using the ADIOS tool 'bpls' to list the content of a .bp file, bpls will print __ for the size of the output table indicating that its size is changing every step.

The *atom/adios* and *custom/adios* dump styles are part of the ADIOS package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

---

### 9.12.5 Related commands

*dump*, *dump_modify*, *undump*

## 9.13 dump cfg/uef command

### 9.13.1 Syntax

```
dump ID group-ID cfg/uef N file mass type xs ys zs args
```

- ID = user-assigned name for the dump
- group-ID = ID of the group of atoms to be dumped
- N = dump every this many timesteps
- file = name of file to write dump info to

  `args = same as args for dump custom`

### 9.13.2 Examples

```
dump 1 all cfg/uef 10 dump.*.cfg mass type xs ys zs
dump 2 all cfg/uef 100 dump.*.cfg mass type xs ys zs id c_stress
```

### 9.13.3 Description

This command is used to dump atomic coordinates in the reference frame of the applied flow field when *fix nvt/uef* or *fix npt/uef* is used. Only the atomic coordinates and frame-invariant scalar quantities will be in the flow frame. If velocities are selected as output, for example, they will not be in the same reference frame as the atomic positions.

### 9.13.4 Restrictions

This fix is part of the UEF package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

This command can only be used when *fix nvt/uef* or *fix npt/uef* is active.

### 9.13.5 Related commands

*dump*, *fix nvt/uef*

### 9.13.6 Default

none

## 9.14 dump h5md command

### 9.14.1 Syntax

```
dump ID group-ID h5md N file.h5 args
```

- ID = user-assigned name for the dump
- group-ID = ID of the group of atoms to be imaged
- *h5md* = style of dump command (other styles *atom* or *cfg* or *dcd* or *xtc* or *xyz* or *local* or *custom* are discussed on the *dump* doc page)
- N = dump every this many timesteps
- file.h5 = name of file to write to
- args = *position* options or *image* or *velocity* options or *force* options or *species* options or *file_from* ID or *box* value or *create_group* value or *author* value = list of data elements to dump, with their dump "sub-intervals"

  ```
  position options
  image
  velocity options
  force options
  species options
  file_from ID = do not open a new file, re-use the already opened file from dump ID
  box value = yes or no
  create_group value = yes or no
  author value = quoted string
  ```

Note that at least one element must be specified and that *image* may only be present if *position* is specified first.

For the elements *position*, *velocity*, *force* and *species*, a sub-interval may be specified to write the data only every N_element iterations of the dump (i.e. every N*N_element time steps). This is specified by this option directly following the element declaration:

```
options = every N_element
```

### 9.14.2 Examples

```
dump h5md1 all h5md 100 dump_h5md.h5 position image
dump h5md1 all h5md 100 dump_h5md.h5 position velocity every 10
dump h5md1 all h5md 100 dump_h5md.h5 velocity author "John Doe"
```

### 9.14.3 Description

Dump a snapshot of atom coordinates every N timesteps in the HDF5 based H5MD file format *(de Buyl)*. HDF5 files are binary, portable and self-describing. This dump style will write only one file, on the root node.

Several dumps may write to the same file, by using file_from and referring to a previously defined dump. Several groups may also be stored within the same file by defining several dumps. A dump that refers (via *file_from*) to an already open dump ID and that concerns another particle group must specify *create_group yes*.

Each data element is written every N*N_element steps. For *image*, no sub-interval is needed as it must be present at the same interval as *position*. *image* must be given after *position* in any case. The box information (edges in each dimension) is stored at the same interval than the *position* element, if present. Else it is stored every N steps.

**Note**

Because periodic boundary conditions are enforced only on timesteps when neighbor lists are rebuilt, the coordinates of an atom written to a dump file may be slightly outside the simulation box.

**Use from write_dump:**

It is possible to use this dump style with the *write_dump* command. In this case, the sub-intervals must not be set at all. The write_dump command can be used either to create a new file or to add current data to an existing dump file by using the *file_from* keyword.

Typically, the *species* data is fixed. The following two commands store the position data every 100 timesteps, with the image data, and store once the species data in the same file.

```
dump h5md1 all h5md 100 dump.h5 position image
write_dump all h5md dump.h5 file_from h5md1 species
```

---

### 9.14.4 Restrictions

The number of atoms per snapshot cannot change with the h5md style. The position data is stored wrapped (box boundaries not enforced, see note above). Only orthogonal domains are currently supported. This is a limitation of the present dump h5md command and not of H5MD itself.

The *h5md* dump style is part of the H5MD package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info. It also requires (i) building the ch5md library provided with LAMMPS (See the *Build package* page for more info.) and (ii) having the HDF5 library installed (C bindings are sufficient) on your system. The library ch5md is compiled with the h5cc wrapper provided by the HDF5 library.

---

### 9.14.5 Related commands

*dump*, *dump_modify*, *undump*

---

**(de Buyl)** de Buyl, Colberg and Hofling, H5MD: A structured, efficient, and portable file format for molecular data, Comp. Phys. Comm. 185(6), 1546-1553 (2014) - [arXiv:1308.6382].

## 9.15 dump image command

## 9.16 dump movie command

(see below for *dump_modify options* specific to dump image/movie)

### 9.16.1 Syntax

```
dump ID group-ID style N file color diameter keyword value ...
```

- ID = user-assigned name for the dump
- group-ID = ID of the group of atoms to be imaged
- style = *image* or *movie* = style of dump command (other styles such as *atom* or *cfg* or *dcd* or *xtc* or *xyz* or *local* or *custom* are discussed on the *dump* doc page)
- N = dump every this many timesteps
- file = name of file to write image to
- color = atom attribute that determines color of each atom
- diameter = atom attribute that determines size of each atom
- zero or more keyword/value pairs may be appended
- keyword = *atom* or *adiam* or *autobond* or *bond* or *grid* or *line* or *tri* or *ellipsoid* or *body* or *compute* or *fix* or *size* or *view* or *center* or *up* or *zoom* or *box* or *axes* or *region* or *subbox* or *shiny* or *fsaa* or *ssao*

```
atom = yes or no = do or do not draw atoms
adiam size = numeric value for atom diameter (distance units)
autobond values = cutoff width = bond cutoff and width of bonds
bond values = color width = color and width of bonds
  color = atom or type or none or c_ID or c_ID[I]
    c_ID = per-bond vector computed by a local compute with ID
    c_ID[I] = Ith column of per-bond array computed by a local compute with ID
  width = number or atom or type or none
    number = numeric value for bond width (distance units)
grid = per-grid value to use when coloring each grid cell
  per-grid value = c_ID:gname:dname, c_ID:gname:dname[I], f_ID:gname:dname, f_
↪ID:gname:dname[I]
    gname = name of grid defined by compute or fix
    dname = name of data field defined by compute or fix
    c_ID = per-grid vector calculated by a compute with ID
    c_ID[I] = Ith column of per-grid array calculated by a compute with ID
    f_ID = per-grid vector calculated by a fix with ID
    f_ID[I] = Ith column of per-grid array calculated by a fix with ID
line = color width
  color = type or index or atom
  width = numeric value for line width (distance units)
tri = color tflag width
  color = type or index or atom
  tflag = 1 for just triangle, 2 for just tri edges, 3 for both
  width = numeric value for triangle edge width (distance units)
ellipsoid = color eflag level width
  color = type or index or atom
  eflag = 1 for triangles, 2 for wireframe
```

```
  level = mesh refinement level, value between 1 (low resolution) and 6 (ultra high␣
↪resolution)
  width = diameter of wireframe edges (distance units) (ignored for triangles)
body = color bflag1 bflag2
  color = type or index or atom
  bflag1,bflag2 = 2 numeric flags to affect how bodies are drawn
compute = computeID color cflag1 cflag2
  computeID = ID of computes that generates objects to draw
  color = type or element or const
  cflag1,cflag2 = 2 numeric flags to affect how compute objects are drawn
fix = fixID color fflag1 fflag2
  fixID = ID of fix that generates objects to draw
  color = type or element or const
  fflag1,fflag2 = 2 numeric flags to affect how fix objects are drawn
size values = width height = size of images
  width = width of image in # of pixels
  height = height of image in # of pixels
view values = theta phi = view of simulation box
  theta = view angle from +z axis (degrees)
  phi = azimuthal view angle (degrees)
  theta or phi can be a variable (see below)
center values = flag Cx Cy Cz = center point of image
  flag = s for static, d for dynamic
  Cx,Cy,Cz = center point of image as fraction of box dimension (0.5 = center of␣
↪box)
  Cx,Cy,Cz can be variables (see below)
up values = Ux Uy Uz = direction that is "up" in image
  Ux,Uy,Uz = components of up vector
  Ux,Uy,Uz can be variables (see below)
zoom value = zfactor = size that simulation box appears in image
  zfactor = scale image size by factor > 1 to enlarge, factor < 1 to shrink
  zfactor can be a variable (see below)
box values = yes/no diam = draw outline of simulation box
  yes/no = do or do not draw simulation box lines
  diam = diameter of box lines as fraction of shortest box length
axes values = axes length diam = draw xyz axes
  axes = yes or no or center or lowerleft or lowerright or upperleft or upperright␣
↪= do or do not draw xyz axes arrows and select location
  length = length of axes lines as fraction of respective box lengths
  diam = diameter of axes lines as fraction of shortest box length
region values = region-ID color drawstyle [opacity (optional) npoints (optional)␣
↪diameter (optional)] [hull_points npoints (optional)]
  region-ID = ID of the region to render
  color = color name for region graphics
  drawstyle = filled or transparent or frame or points
    filled = render region as a filled object, with optional open faces
    transparent = same as filled but has selectable opacity
    frame = render region as a wireframe (like box or subbox)
    points = fill region with spheres at random locations
  opacity  = level of opacity (from 0.0 to 1.0, only for drawstyle transparent)
  npoints  = number of attempted points (only for drawstyle points)
  diameter = diameter of wireframe or points (only for drawstyles frame and points)
  hull_points npoints = set number of points for creating a Delaunay triangulation␣
↪(optional)
```

```
subbox values = lines diam = draw outline of processor subdomains
  lines = yes or no = do or do not draw subdomain lines
  diam = diameter of subdomain lines as fraction of shortest box length
shiny value = sfactor = shinyness of spheres and cylinders
  sfactor = shinyness of spheres and cylinders from 0.0 to 1.0
fsaa arg = yes/no
  yes/no = do or do not apply anti-aliasing
ssao value = shading seed dfactor = SSAO depth shading
  shading = yes or no = turn depth shading on/off
  seed = random # seed (positive integer)
  dfactor = strength of shading from 0.0 to 1.0
```

## 9.17 dump_modify options for dump image/movie

### 9.17.1 Syntax

```
dump_modify dump-ID keyword values ...
```

- these keywords apply only to the *image* and *movie* styles and are documented on this page
- keyword = *acolor* or *adiam* or *amap* or *gmap* or *bmap* or *atrans* or *backcolor* or *backcolor2* or *bcolor* or *bdiam* or *btrans* or *bitrate* or *boxcolor* or *color* or *lights* or *loadcolors* or *savecolors* or *framerate* or *axestrans* or *boxtrans* or *subboxtrans* or *ccolor* or *ctrans* or *fcolor* or *ftrans*
- see the *dump modify* doc page for more general keywords

```
acolor args = type color
  type = atom type (numeric or type label) or range of numeric types (see below)
  color = name of color or color1/color2/...
adiam args = type diam
  type = atom type (numeric or type label) or range of numeric types (see below)
  diam = diameter of atoms of that type (distance units)
amap args = lo hi style delta N entry1 entry2 ... entryN
  lo = number or min = lower bound of range of color map
  hi = number or max = upper bound of range of color map
  style = 2 letters = c or d or s plus a or f
    c for continuous
    d for discrete
    s for sequential
    a for absolute
    f for fractional
  delta = binsize (only used for style s, otherwise ignored)
    binsize = range is divided into bins of this width
  N = # of subsequent entries
  entry = value color (for continuous style)
    value = number or min or max = single value within range
    color = name of color used for that value
  entry = lo hi color (for discrete style)
    lo/hi = number or min or max = lower/upper bound of subset of range
    color = name of color used for that subset of values
  entry = color (for sequential style)
    color = name of color used for a bin of values
atrans args = type transparency
  type = atom type (numeric or type label) or range of numeric types (see below)
```

```
  transparency = transparency of atoms of that type (value between 0 (invisible)␣
↪and 1 (fully opaque))
backcolor arg = color
  color = name of color for background
backcolor2 arg = color
  color = name of second color for vertical background gradient. "none" to disable␣
↪gradient
bcolor args = type color
  type = bond type (numeric or type label) or range of numeric types (see below)
  color = name of color or color1/color2/...
bdiam args = type diam
  type = bond type (numeric or type label) or range of numeric types (see below)
  diam = diameter of bonds of that type (distance units)
btrans args = type transparency
  type = bond type (numeric or type label) or range of numeric types (see below)
  transparency = transparency of bonds of that type (value between 0 (invisible)␣
↪and 1 (fully opaque))
axestrans arg = transparency
  transparency = transparency for axes lines (value between 0 (invisible) and 1␣
↪(fully opaque))
boxtrans arg = transparency
  transparency = transparency for simulation box lines (value between 0 (invisible)␣
↪and 1 (fully opaque))
boxcolor arg = color
  color = name of color for simulation box lines and processor subdomain lines
subboxtrans arg = transparency
  transparency = transparency for simulation subbox lines (value between 0␣
↪(invisible) and 1 (fully opaque))
color args = name R G B or name hex
  name = name of color
  R,G,B = red/green/blue numeric values from 0.0 to 1.0
  hex = 24-bit RGB color in hexadecimal
lights args = ambient key fill back
  ambient key fill back = set light intensity value from 0.0 to 1.0
loadcolors arg = filename
  filename = load color definitions, per-type colors, and lights from JSON format␣
↪file
savecolors arg = filename
  filename = save per-type colors and lights to JSON format file
ccolor args = computeID color
  computeID = ID of the compute
  color = name of color for image objects provided by this compute when using␣
↪"const" color style
ctrans args = computeID transparency
  computeID = ID of the compute
  transparency = transparency for image objects provided by this compute
fcolor args = fixID color
  fixID = ID of the fix
  color = name of color for image objects provided by this fix when using "const"␣
↪color style
ftrans args = fixID transparency
  fixID = ID of the fix
  transparency = transparency for image objects provided by this fix
bitrate arg = rate
```

```
  rate = target bitrate for movie in kbps
framerate arg = fps
  fps = frames per second for movie
gmap args = identical to amap args
bmap args = identical to amap args
```

### 9.17.2 Examples

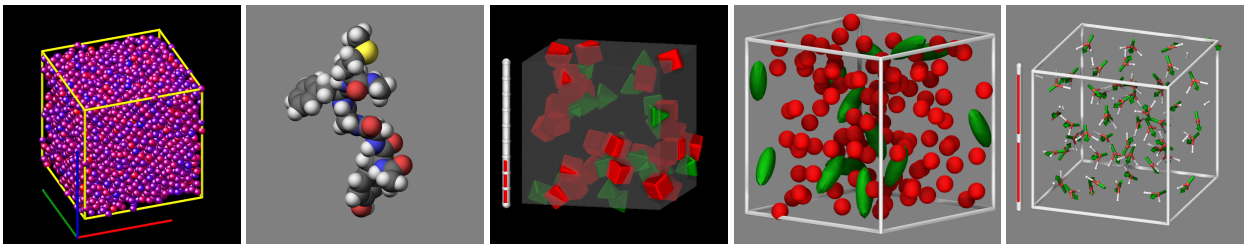
```
dump d0 all image 100 dump.*.jpg type type
dump d1 mobile image 500 snap.*.png element element ssao yes 4539 0.6
dump d2 all image 200 img-*.ppm type type zoom 2.5 adiam 1.5 size 1280 720
dump m0 all movie 1000 movie.mpg type type size 640 480
dump m1 all movie 1000 movie.avi type type size 640 480
dump m2 all movie 100 movie.m4v type type zoom 1.8 adiam v_value size 1280 720

dump_modify 1 amap min max cf 0.0 3 min green 0.5 yellow max blue boxcolor red

labelmap atom 1 C 2 H 3 O 4 N
dump_modify 1 acolor C gray acolor H white acolor O red acolor N blue
dump_modify 1 color gray80 0.8 0.8 0.8 color gray20 0x333333
```

### 9.17.3 Description

Dump a high-quality rendered image of the atom configuration every *N* timesteps and save the images either as a sequence of JPEG, PNG, TGA, or PPM files, or as a single movie file. The options for this command as well as the *dump_modify* command control what is included in the image or movie and how it appears. A series of such images can easily be manually converted into an animated movie of your simulation or the process can be automated without writing the intermediate files using the dump movie style; see further details below. Other dump styles store snapshots of numerical data associated with atoms in various formats, as discussed on the *dump* doc page.

Note that a set of images or a movie can be made after a simulation has been run, using the *rerun* command to read snapshots from an existing dump file, and using these dump commands in the rerun script to generate the images/movie.

Here are five sample images, rendered as JPEG or PNG files.



A detailed discussion of advanced graphics settings and workflows with examples is provided in the *Visualize LAMMPS snapshots* howto.

Added in version 11Feb2026: support for writing compressed TGA files

Only atoms in the specified group are rendered in the image. The *dump_modify region and thresh* commands can also alter what atoms are included in the image. The filename suffix determines whether a JPEG, PNG, TGA, or PPM file is created with the *image* dump style. If the suffix is ".jpg" or ".jpeg", then a JPEG format file is created, if the suffix is ".png", then a PNG format file is created, if the suffix is ".tga", then a compressed 24-bit RGB TGA or TARGA format file is created, else a PPM (aka NETPBM) format file is created. The JPEG, PNG, and TGA files are binary; PPM has a text mode header followed by binary data. JPEG images have lossy compression, PNG and TGA have lossless compression, and PPM files are uncompressed but can be compressed with a supported compression program, if LAMMPS has been compiled with *compression support* and a supported suffix is used.

Similarly, the format of the resulting movie is chosen with the *movie* dump style. This is handled by the underlying FFmpeg converter and thus details have to be looked up in the FFmpeg documentation. Typical examples are: .avi, .mpg, .m4v, .mp4, .mkv, .flv, .mov, .gif Additional settings of the movie compression like *bitrate* and *framerate* can be set using the dump_modify command as described below.

To write out JPEG and PNG format files, you must build LAMMPS with support for the corresponding JPEG or PNG library. To convert images into movies, LAMMPS has to be compiled with the `-DLAMMPS_FFMPEG` flag. See the *Build settings* page for details.

**Note**

Because periodic boundary conditions are enforced only on timesteps when neighbor lists are rebuilt, the coordinates of an atom in the image may be slightly outside the simulation box.

---

Dumps are performed on timesteps that are a multiple of *N* (including timestep 0) and on the last timestep of a minimization if the minimization converges. Note that this means a dump will not be performed on the initial timestep after the dump command is invoked, if the current timestep is not a multiple of *N*. This behavior can be changed via the *dump_modify first* command, which can be useful if the dump command is invoked after a minimization ended on an arbitrary timestep. *N* can be changed between runs by using the *dump_modify every* command.

Dump *image* filenames must contain a wildcard character "*" so that one image file per snapshot is written. The "*" character is replaced with the timestep value. For example, tmp.dump.*.jpg becomes tmp.dump.0.jpg, tmp.dump.10000.jpg, tmp.dump.20000.jpg, etc. Note that the *dump_modify pad* command can be used to ensure all timestep numbers are the same length (e.g., 00010), which can make it easier to convert a series of images into a movie in the correct ordering.

Dump *movie* filenames on the other hand, must not have any wildcard character since only one file combining all images into a single movie will be written by the movie encoder.

---

The *color* and *diameter* settings determine the color and size of atoms rendered in the image. They can be any atom attribute defined for the *dump custom* command, including *type* and *element*. This includes per-atom quantities calculated by a *compute*, *fix*, or *variable*, which are prefixed by "c_", "f_", or "v_", respectively. Note that the *diameter* setting can be overridden with a numeric value applied to all atoms by the optional *adiam* keyword.

Changed in version 4Jul2026: Extended list of colors from 6 to 16

If *type* is specified for the *color* setting, then the color of each atom is determined by its atom type. By default the mapping of atom types to colors is: red, forestgreen, blue, gold, cyan, magenta, silver, orange, lime, gray, darkred, darkgreen, darkblue, darkcyan, darkmagenta, and darkgray for the first 16 atom types and repeats itself after that. This mapping can be changed by the "dump_modify acolor" command, as described below.

If *type* is specified for the *diameter* setting then the diameter of each atom is determined by its atom type. By default all types have diameter 1.0. This mapping can be changed by the "dump_modify adiam" command, as described below.

If *element* is specified for the *color* and/or *diameter* setting, then the color and/or diameter of each atom is determined by which element it is, which in turn is specified by the element-to-type mapping specified by the "dump_modify element" command, as described below. By default the element for every atom type is set to C (carbon). Every element has a color and diameter associated with it, which is the same as the colors and sizes used by the AtomEye visualization package.

If other atom attributes are used for the *color* or *diameter* settings, they are interpreted in the following way.

If "vx", for example, is used as the *color* setting, then the color of the atom will depend on the x-component of its velocity. The association of a per-atom value with a specific color is determined by a "color map", which can be specified via the dump_modify amap command, as described below. The basic idea is that the atom-attribute will be

within a range of values, and every value within the range is mapped to a specific color. Depending on how the color map is defined, that mapping can take place via interpolation so that a value of -3.2 is halfway between "red" and "blue", or discretely so that the value of -3.2 is "orange".

If "vx", for example, is used as the *diameter* setting, then the atom will be rendered using the x-component of its velocity as the diameter. If the per-atom value <= 0.0, then the atom will not be drawn. Note that finite-size spherical particles, as defined by *atom_style sphere* define a per-particle radius or diameter, which can be used as the *diameter* setting.

---

The various keywords listed above control how the image is rendered. As listed below, all of the keywords have defaults, most of which you will likely not need to change. As described below, the dump modify command also has options specific to the dump image style, particularly for assigning colors to atoms, bonds, and other image features.

---

The *atom* keyword allow you to turn off the drawing of all atoms, if the specified value is *no*. Note that this will not turn off the drawing of particles that are represented as lines, triangles, or bodies, as discussed below. These particles can be drawn separately if the *line*, *tri*, *ellipsoid*, or *body* keywords are used.

The *adiam* keyword allows you to override the *diameter* setting to set a single numeric *size*. All atoms will be drawn with that diameter, e.g. 1.5, which is in whatever distance *units* the input script defines, e.g. Angstroms.

---

Added in version 10Sep2025.

The *autobond* keyword enables drawing bonds for systems where bonds are implicit, e.g. for potentials like *AIREBO* or *ReaxFF*. The first argument is the bond cutoff, i.e. bonds are drawn for pairs of atoms that are closer than this cutoff; the second argument is the bond diameter. The implicit bonds are found by searching the pair-wise neighbor list for pairs of atoms that are closer than the bond cutoff. The color of the bond is derived from the color of the atoms forming the implicit bond. For *unit styles metal and real* an additional condition is applied: if the mass of both atoms of a pair within the bond cutoff is lower than 3 atomic mass units, a bond is **not** drawn; this prohibits displaying unwanted hydrogen-hydrogen bonds for alkyl or alcohol groups or for water with typical cutoffs suitable for displaying covalent bonds. For ReaxFF it is also possible to visualize bonds as they are computed through using *fix reaxff/bonds* with the *fix* keyword (see below).

---

The *bond* keyword allows to you to alter how bonds are drawn. A bond is only drawn if both atoms in the bond are being drawn due to being in the specified group and due to other selection criteria (e.g. region, threshold settings of the *dump_modify* command). By default, bonds are drawn if they are defined in the input data file as read by the *read_data* command. Using *none* for both the bond *color* and *width* value will turn off the drawing of all bonds.

If *atom* is specified for the bond *color* value, then each bond is drawn in 2 halves, with the color of each half being the color of the atom at that end of the bond.

Changed in version 4Jul2026: Extended list of default colors from 6 to 16

If *type* is specified for the *color* value, then the color of each bond is determined by its bond type. By default the mapping of bond types to colors is: red, forestgreen, blue, gold, cyan, magenta, silver, orange, lime, gray, darkred, darkgreen, darkblue, darkcyan, darkmagenta, and darkgray for the first 16 bond types and repeats itself after that. This mapping can be changed by the "dump_modify bcolor" command, as described below.

Added in version 4Jul2026.

If a compute reference *c_ID* or *c_ID[1]* is specified for the *color* value, then each bond is colored by a per-bond value taken from that compute, mapped to a color through a color map (set with the "dump_modify bmap" command, described below) in the same way per-atom values are mapped via *amap*. The referenced compute must produce *local* per-bond data, for example *compute bond/local* with the *dist* (bond length) or *engpot* (bond energy) attribute. Use *c_ID*

when the compute produces a per-bond vector (a single attribute) and *c_ID[I]* to select column *I* when it produces a per-bond array (multiple attributes). The compute must generate one value for each bond that is drawn, in the same order, so it should compute over the same set of bonds as is being visualized (typically the *all* group). If the number of values produced does not match the number of bonds drawn, LAMMPS stops with an error.

The bond *width* value can be a numeric value or *atom* or *type* (or *none* as indicated above).

If a numeric value is specified, then all bonds will be drawn as cylinders with that diameter, e.g. 1.0, which is in whatever distance *units* the input script defines, e.g. Angstroms.

If *atom* is specified for the *width* value, then each bond will be drawn with a width corresponding to the minimum diameter of the two atoms in the bond.

If *type* is specified for the *width* value then the diameter of each bond is determined by its bond type. By default all types have diameter 0.5. This mapping can be changed by the "dump_modify bdiam" command, as described below.

---

The *line* keyword can be used when *atom_style line* is used to define particles as line segments, and will draw them as lines. If this keyword is not used, such particles will be drawn as spheres, the same as if they were regular atoms.

Changed in version 30Mar2026: added *index* and *atom* color styles

There are currently three supported settings for the *color* value: *type*, *index*, or *atom*. With the *type* setting the line segment particles will be colored according to the atom type of the particle. With the *index* setting, colors from the list of available per-atom type colors are assigned to the line particles in a non-deterministic round-robin fashion. With the *atom* setting, the color follows the coloring selected for coloring atoms (including using color maps). If more different colors than atom types are desired, the *number of atom types* must be *increased* correspondingly when using either the *create_box* or the *read_data* command.

The line *width* can only be a numeric value, which specifies that all lines will be drawn as cylinders with that diameter, e.g. 1.0, which is in whatever distance *units* the input script defines, e.g. Angstroms.

---

The *tri* keyword can be used when *atom_style tri* is used to define particles as triangles, and will draw them as triangles or edges (3 lines) or both, depending on the setting for *tflag*. If edges are drawn, the *width* setting determines the diameters of the line segments. If this keyword is not used, triangle particles will be drawn as spheres, the same as if they were regular atoms.

Changed in version 30Mar2026: added *index* and *atom* color styles

There are currently three supported settings for the *color* value: *type*, *index*, or *atom*. With the *type* setting the triangles will be colored according to the atom type of the particle. With the *index* setting, colors from the list of available per-atom type colors are assigned to the triangulated particles in a non-deterministic round-robin fashion. With the *atom* setting, the color follows the coloring selected for coloring atoms (including using color maps). If more different colors than atom types are desired, the *number of atom types* must be *increased* correspondingly when using either the *create_box* or the *read_data* command.

---

Added in version 11Feb2026.

Changed in version 30Mar2026: Now uses rounded triangles

The *ellipsoid* keyword can be used when *atom_style ellipsoid* is used to define particles as ellipsoids, and will draw them as a mesh of rounded triangles or edges, depending on the setting for *eflag*(1 for rounded triangles, 2 for edges, other values are accepted for backward compatibility but select rounded triangles). If edges are drawn, the *width* setting determines the diameters of the line segments. If this keyword is not used, ellipsoid particles will be drawn as spheres, the same as if they were regular atoms.

Changed in version 30Mar2026: added *index* and *atom* color styles

There are currently three supported settings for the *color* value: *type*, *index*, or *atom*. With the *type* setting the ellipsoids will be colored according to the atom type of the particle. With the *index* setting, colors from the list of available per-atom type colors are assigned to the ellipsoid particles in a non-deterministic round-robin fashion. With the *atom* setting, the color follows the coloring selected for coloring atoms (including using color maps). If more different colors than atom types are desired, the *number of atom types* must be *increased* correspondingly when using either the *create_box* or the *read_data* command.

Changed in version 30Mar2026: changed initial geometry to icosahedron and use rounded triangles

The *level* setting determines the number of triangles in the mesh of triangles and thus the resolution of the representation of the ellipsoid. At level 1 the ellipsoid is represented by an icosahedron that is stretched according to the ellipsoid's shape parameters. For each higher level, a refinement iteration is performed where any of the triangles are replaced by four triangles and their edges are shifted to be on the surface of the ellipsoid. The maximum allowed refinement level is 6 (corresponding to 12288 triangles per ellipsoid).

**Image quality versus rendering speed**

Since the rendered ellipsoids are constructed from iteratively refined triangle meshes as explained above, the image quality increases with each refinement level, but so does the computational effort to render the image. This becomes more pronounced when FSAA or SSAO or both are enabled.

---

The *body* keyword can be used when *atom_style body* is used to define body particles with internal state (e.g. sub-particles), and will drawn them in a manner specific to the body style. If this keyword is not used, such particles will be drawn as spheres, the same as if they were regular atoms.

The *Howto body* page describes the body styles LAMMPS currently supports, and provides more details as to the kind of body particles they represent and how they are drawn by this dump image command. For all the body styles, individual atoms can be either a body particle or a usual point (non-body) particle. Non-body particles will be drawn the same way they would be as a regular atom. The *bflag1* and *bflag2* settings are numerical values which are passed to the body style to affect how the drawing of a body particle is done. See the *Howto body* page for a description of what these parameters mean for each body style.

Changed in version 11Feb2026: added *index* color style

Changed in version 30Mar2026: added *atom* color style

There are currently three supported settings for the *color* value: *type*, *index*, or *atom*. With the *atom* setting, the color follows the coloring selected for coloring atoms (including using color maps). With the *type* setting the body particles will be colored according to the atom type of the particle. With the *index* setting, colors from the list of available per-atom type colors are assigned to the body particles in a non-deterministic round-robin fashion. If more different colors than atom types are desired, the *number of atom types* must be *increased* correspondingly when using either the *create_box* or the *read_data* command.

---

Changed in version 11Feb2026: Support for computes and several fix styles added and more flexible color selection

The *compute* keyword can be used with a *compute style* that produces information about objects to be drawn. Similarly, the *fix* keyword can be used with a *fix style* that produces information about objects to be drawn. The compute or fix keywords may be used multiple times to include visualizations of graphics objects from multiple computes and fixes. The *compute* keyword is followed by the *compute ID* of the compute, the color style setting and two numerical values *cflag1* and *cflag2*. The *fix* keyword is followed by the *fix ID* of the fix, the color style setting and two numerical values *fflag1* and *fflag2*.

The color style may be either *type*, *element*, or *const*. The first two will use the same color as assigned to the corresponding atom type and thus it depends on the implementation of the compute or fix which atom type it associates with

any object. Often this will be atom type 1. For the *const* type a constant color will be used that can be changed with a *dump_modify ccolor* or *dump_modify fcolor* command (see below). By default the constant color will be "white".

The *cflag1* and *cflag2* or *fflag1* and *fflag2* settings are numerical values which are used by *dump image* to adjust how the drawing of the objects communicated by the fix is done. See the documentation of the individual computes and fixes for a description of what these parameters mean for the graphics objects provided by those fixes.

More details and some examples for including graphics objects from compute and fix commands are in the *Visualize LAMMPS snapshots* howto.

---

Added in version 10Sep2025.

Changed in version 11Feb2026: draw style *transparent* was added

Changed in version 4Jul2026: draw triangulated hull from random points for region style *intersect* or *union*

The *region* keyword can be used to create a graphical representation of a *region*. This can be helpful in debugging the location and extent of regions, especially when those have parameters controlled by variables. The sequence of arguments to the *region* are: the region-ID, the color for drawing the region, the draw style, and possible additional arguments as required by the draw style.

Four draw styles of representing a region are available: *filled*, *transparent*, *frame*, and *points*. With draw style *filled* the surface of the region is triangulated and drawn. For region styles that support open faces, surfaces for such open faces are skipped. The style *transparent* is like *filled* but takes an additional parameter in the range of 0.0 to 1.0 that defines the opacity and thus allows to see what is inside the region for values < 1. Draw style *frame* represents the region with a mesh of "wires". The diameter of these "wires" are set with the following argument. Unlike with the *filled* style and similar to the *transparent* style, you can see what is *inside* the region with this draw style. The fourth draw style, *points*, generates a random point cloud inside the simulation box and draws only those points that are within the region. This uses the same test than what is used to determine if an atom is inside the region but ignores any open faces (which would match *all* positions as "inside"). When using draw styles *filled*, *transparent*, or *frame* with unions or intersections of multiple regions an enclosing hull is first created from a point cloud that is generated the same way as in the *points* draw style. The number of points used for the hull approximation (default is 100000) can be set by the optional *hull_points* keyword.

Recommended transparency values are 0.25, 0.5, or 0.75 when used in combination with *fsaa on*.

---

The *size* keyword sets the width and height of the created images, i.e. the number of pixels in each direction.

---

The *view*, *center*, *up*, and *zoom* values determine how 3d simulation space is mapped to the 2d plane of the image. Basically they control how the simulation box appears in the image.

All of the *view*, *center*, *up*, and *zoom* values can be specified as numeric quantities, whose meaning is explained below. Any of them can also be specified as an *equal-style variable*, by using v_name as the value, where "name" is the variable name. In this case the variable will be evaluated on the timestep each image is created to create a new value. If the equal-style variable is time-dependent, this is a means of changing the way the simulation box appears from image to image, effectively doing a pan or fly-by view of your simulation.

The *view* keyword determines the viewpoint from which the simulation box is viewed, looking towards the *center* point. The *theta* value is the vertical angle from the +z axis, and must be an angle from 0 to 180 degrees. The *phi* value is an azimuthal angle around the z axis and can be positive or negative. A value of 0.0 is a view along the +x axis, towards the *center* point. If *theta* or *phi* are specified via variables, then the variable values should be in degrees.

The *center* keyword determines the point in simulation space that will be at the center of the image. *Cx*, *Cy*, and *Cz* are specified as fractions of the box dimensions, so that (0.5,0.5,0.5) is the center of the simulation box. These values do not have to be between 0.0 and 1.0, if you want the simulation box to be offset from the center of the image. Note,

however, that if you choose strange values for *Cx*, *Cy*, or *Cz* you may get a blank image. Internally, *Cx*, *Cy*, and *Cz* are converted into a point in simulation space. If *flag* is set to "s" for static, then this conversion is done once, at the time the dump command is issued. If *flag* is set to "d" for dynamic then the conversion is performed every time a new image is created. If the box size or shape is changing, this will adjust the center point in simulation space.

The *up* keyword determines what direction in simulation space will be "up" in the image. Internally it is stored as a vector that is in the plane perpendicular to the view vector implied by the *theta* and *phi* values, and which is also in the plane defined by the view vector and user-specified up vector. Thus this internal vector is computed from the user-specified *up* vector as

```
up_internal = view cross (up cross view)
```

This means the only restriction on the specified *up* vector is that it cannot be parallel to the *view* vector, implied by the *theta* and *phi* values.

The *zoom* keyword scales the size of the simulation box as it appears in the image. The default *zfactor* value of 1 should display an image mostly filled by the atoms in the simulation box. A *zfactor* > 1 will make the simulation box larger; a *zfactor* < 1 will make it smaller. *Zfactor* must be a value > 0.0.

---

The *box* keyword determines if and how the simulation box boundaries are rendered as thin cylinders in the image. If *no* is set, then the box boundaries are not drawn and the *diam* setting is ignored. If *yes* is set, the 12 edges of the box are drawn, with a diameter that is a fraction of the shortest box length in x,y,z (for 3d) or x,y (for 2d). The color of the box boundaries can be set with the "dump_modify boxcolor" command.

Changed in version 11Feb2026.

The *axes* keyword determines if and how the coordinate axes are rendered in the image as arrows with the letters 'X', 'Y', and 'Z' to indicate the direction. If *no* is set, then the axes are not drawn and the *length* and *diam* settings are ignored. If *yes* or *lowerleft* is set, 3 arrows are drawn to represent the x,y,z axes in colors red, green, and blue, respectively. The origin of these arrows will be offset from the lower left corner of the box by 10%. If *center* is set the origin of the arrows will be in the center of the box. If *lowerright* is set, the origin of the arrows will be offset by 20% of the lower right corner of the box. If *upperleft* or *upperright* are set the origin of the arrows will be placed similar to the lower corner arrows, but offset by 20% from the top. The *length* setting determines how long the cylinders will be as a fraction of the respective box lengths. The *diam* setting determines their thickness as a fraction of the shortest box length in x,y,z (for 3d) or x,y (for 2d).

The *subbox* keyword determines if and how processor subdomain boundaries are rendered as thin cylinders in the image. If *no* is set (default), then the subdomain boundaries are not drawn and the *diam* setting is ignored. If *yes* is set, the 12 edges of each processor subdomain are drawn, with a diameter that is a fraction of the shortest box length in x,y,z (for 3d) or x,y (for 2d). The color of the subdomain boundaries can be set with the "dump_modify boxcolor" command.

---

The *shiny* keyword determines how shiny the objects rendered in the image will appear. The *sfactor* value must be a value 0.0 <= *sfactor* <= 1.0, where *sfactor* = 1 is a highly reflective surface and *sfactor* = 0 is a rough non-shiny surface.

Added in version 21Nov2023.

The *fsaa* keyword can be used with the dump image command to improve the image quality by enabling full scene anti-aliasing. Internally the image is rendered at twice the width and height and then scaled down by computing the average of each 2x2 block of pixels to produce a single pixel in the final image at the original size. This produces images with smoother, less ragged edges. The application of this algorithm can increase the cost of computing the image by about 3x or more.

The *ssao* keyword turns on/off a screen space ambient occlusion (SSAO) model for depth shading. If *yes* is set, then atoms further away from the viewer are darkened via a randomized process, which is perceived as depth. The strength of the effect can be scaled by the *dfactor* parameter. If *no* is set, no depth shading is performed. The calculation of

this effect can increase the cost of computing the image substantially by 5x or more, especially with larger images. When used in combination with the *fsaa* keyword the computational cost of depth shading is particularly large. In case LAMMPS has been *compiled with OpenMP support*, the SSAO processing is distributed across multiple threads.

---

### 9.17.4 Dump_modify keywords for dump image and dump movie

The following dump_modify keywords apply only to the dump image and dump movie styles. Any keyword that works with dump image also works with dump movie, since the movie is simply a collection of images. Some of the keywords only affect the dump movie style. The descriptions give details.

---

The *acolor* keyword can be used with the dump image command, when its atom color setting is *type*, to set the color that atoms of each type will be drawn in the image.

The specified *type* should be a type label or integer from 1 to Ntypes = the number of atom types. For numeric types, a wildcard asterisk can be used in place of or in conjunction with the *type* argument to specify a range of atom types. This takes the form "*" or "*n" or "n*" or "m*n". If N = the number of atom types, then an asterisk with no numeric values means all types from 1 to N. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from n to N (inclusive). A middle asterisk means all types from m to n (inclusive).

The specified *color* can be a single color which is any of the 140 pre-defined colors (see below) or a color name defined by the "dump_modify color" command, as described below. Or it can be two or more colors separated by a "/" character, e.g. red/green/blue. In the former case, that color is assigned to all the specified atom types. In the latter case, the list of colors are assigned in a round-robin fashion to each of the specified atom types.

---

The *adiam* keyword can be used with the dump image command, when its atom diameter setting is *type*, to set the size that atoms of each type will be drawn in the image. The specified *type* should be a type label or integer from 1 to Ntypes. As with the *acolor* keyword, a wildcard asterisk can be used as part of the *type* argument to specify a range of numeric atom types. The specified *diam* is the size in whatever distance *units* the input script is using, e.g. Angstroms.

---

The *amap* keyword can be used with the dump image command, with its *atom* keyword, when its atom setting is an atom-attribute, to setup a color map. The color map is used to assign a specific RGB (red/green/blue) color value to an individual atom when it is drawn, based on the atom's attribute, which is a numeric value, e.g. its x-component of velocity if the atom-attribute "vx" was specified.

The basic idea of a color map is that the atom-attribute will be within a range of values, and that range is associated with a series of colors (e.g. red, blue, green). An atom's specific value (vx = -3.2) can then mapped to the series of colors (e.g. halfway between red and blue), and a specific color is determined via an interpolation procedure. There are some example command lines and resulting images at the end of this paragraph.

There are many possible options for the color map, enabled by the *amap* keyword. Here are the details.

The *lo* and *hi* settings determine the range of values allowed for the atom attribute. If numeric values are used for *lo* and/or *hi*, then values that are lower/higher than that value are set to the value. I.e. the range is static. If *lo* is specified as *min* or *hi* as *max* then the range is dynamic, and the lower and/or upper bound will be calculated each time an image is drawn, based on the set of atoms being visualized.

The *style* setting is two letters, such as "ca". The first letter is either "c" for continuous, "d" for discrete, or "s" for sequential. The second letter is either "a" for absolute, or "f" for fractional.

A continuous color map is one in which the color changes continuously from value to value within the range. A discrete color map is one in which discrete colors are assigned to sub-ranges of values within the range. A sequential color map is one in which discrete colors are assigned to a sequence of sub-ranges of values covering the entire range.

An absolute color map is one in which the values to which colors are assigned are specified explicitly as values within the range. A fractional color map is one in which the values to which colors are assigned are specified as a fractional portion of the range. For example if the range is from -10.0 to 10.0, and the color red is to be assigned to atoms with a value of 5.0, then for an absolute color map the number 5.0 would be used. But for a fractional map, the number 0.75 would be used since 5.0 is 3/4 of the way from -10.0 to 10.0.

The *delta* setting must be specified for all styles, but is only used for the sequential style; otherwise the value is ignored. It specifies the bin size to use within the range for assigning consecutive colors to. For example, if the range is from $-10.0$ to 10.0 and a *delta* of 1.0 is used, then 20 colors will be assigned to the range. The first will be from $-10.0 \leq$ color1 $< -9.0$, then second from $-9.0 \leq color2 < -8.0$, etc.

The *N* setting is how many entries follow. The format of the entries depends on whether the color map style is continuous, discrete or sequential. In all cases the *color* setting can be any of the 140 pre-defined colors (see below) or a color name defined by the dump_modify color option.

For continuous color maps, each entry has a *value* and a *color*. The *value* is either a number within the range of values or *min* or *max*. The *value* of the first entry must be *min* and the *value* of the last entry must be *max*. Any entries in between must have increasing values. Note that numeric values can be specified either as absolute numbers or as fractions (0.0 to 1.0) of the range, depending on the "a" or "f" in the style setting for the color map.

Here is how the entries are used to determine the color of an individual atom, given the value $X$ of its atom attribute. $X$ will fall between 2 of the entry values. The color of the atom is linearly interpolated (in each of the RGB values) between the 2 colors associated with those entries. For example, if $X = -5.0$ and the two surrounding entries are "red" at $-10.0$ and "blue" at 0.0, then the atom's color will be halfway between "red" and "blue", which happens to be "purple".

For discrete color maps, each entry has a *lo* and *hi* value and a *color*. The *lo* and *hi* settings are either numbers within the range of values or *lo* can be *min* or *hi* can be *max*. The *lo* and *hi* settings of the last entry must be *min* and *max*. Other entries can have any *lo* and *hi* values and the sub-ranges of different values can overlap. Note that numeric *lo* and *hi* values can be specified either as absolute numbers or as fractions (0.0 to 1.0) of the range, depending on the "a" or "f" in the style setting for the color map.

Here is how the entries are used to determine the color of an individual atom, given the value X of its atom attribute. The entries are scanned from first to last. The first time that *lo* <= X <= *hi*, X is assigned the color associated with that entry. You can think of the last entry as assigning a default color (since it will always be matched by X), and the earlier entries as colors that override the default. Also note that no interpolation of a color RGB is done. All atoms will be drawn with one of the colors in the list of entries.

For sequential color maps, each entry has only a *color*. Here is how the entries are used to determine the color of an individual atom, given the value X of its atom attribute. The range is partitioned into N bins of width *binsize*. Thus X will fall in a specific bin from 1 to N, say the Mth bin. If it falls on a boundary between 2 bins, it is considered to be in the higher of the 2 bins. Each bin is assigned a color from the E entries. If E < N, then the colors are repeated. For example if 2 entries with colors red and green are specified, then the odd numbered bins will be red and the even bins green. The color of the atom is the color of its bin. Note that the sequential color map is really a shorthand way of defining a discrete color map without having to specify where all the bin boundaries are.

Here is an example for using a sequential color map to color all the atoms in individual molecules with a different color. See below for how this can be used in the `examples/pour/in.pour.2d.molecule` input script.

```
variable    colors string "red green blue yellow white purple pink orange lime gray"
variable    mol2 atom mol%10
dump        2 all image 250 image.*.png v_mol2 type region slab black frame 0.25 &
                            zoom 3.5 adiam 1.4 size 1200 600 fsaa yes shiny 0.2
dump_modify 2 pad 5 amap 0 10 sa 1 10 ${colors} backcolor darkgray boxcolor silver
```

In this case, 10 colors are defined, and molecule IDs are mapped to one of the colors, even if there are 1000s of molecules.

Here is an example for coloring the atoms in the "melt" example by their velocity with a custom continuous color map and using *fix graphics/labels* to generate a colormap legend:

```
# compute atom velocity
variable vel atom sqrt(vx*vx+vy*vy+vz*vz)

# overlay the top of the image with a horizontal color scale legend
fix obj all graphics/labels 100 colorscale "viz" "Atom Velocity (sigma/tau)" 300.0 560.0␣
↪0.0 size 24 &
            transcolor none framecolor darkgray backcolor darkgray length 800

# output images and set atom color by the value of the variable "vel"
dump viz all image 100 melt-*.png v_vel type size 600 600 zoom 1.4 shiny 0.2 view 85 -5 &
                       fsaa yes box yes 0.025 center s 0.5 0.5 0.6 fix obj const 1 0
dump_modify viz pad 6 boxcolor lightskyblue backcolor darkgray backcolor2 silver adiam *␣
↪1.2

# customize the color map using a continuous map with fractions
dump_modify viz amap 0.0 8 cf 0.0 6 min red 0.2 orange 0.4 green 0.6 darkcyan 0.8 blue␣
↪max purple
```

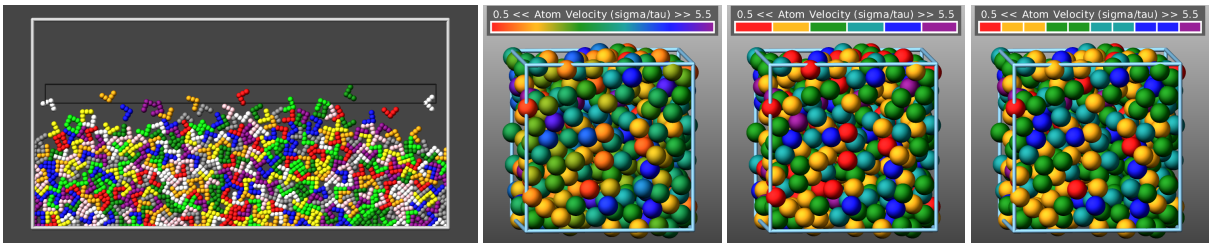
This is an altered *dump_modify* command line to generate a sequential color map:

```
dump_modify viz amap 0.5 5.5 sf 0.167 6 red orange green darkcyan blue purple
```

And another altered *dump_modify* command line to generate a discrete color map using absolute values:

```
dump_modify viz amap 0.5 5.5 da 0.0 6 min 1.0 red 1.0 2.0 orange 2.0 3.0 green 3.0 4.0␣
↪darkcyan
```

Here are images of the examples from above.




---

The *backcolor* sets the background color of the images. The color name can be any of the 140 pre-defined colors (see below) or a color name defined by the dump_modify color option.

Added in version 11Feb2026.

The *backcolor2* sets a second background color of the images to create a vertical background gradient. The regular background color is the color at the bottom and *backcolor2* sets the background color at the top. The color in between is a linear interpolation between those two colors. The color name can be any of the 140 pre-defined colors (see below) or a color name defined by the dump_modify color option. Using a color name of "none" will disable the background gradient feature (this is the default).

---

The *bcolor* keyword can be used with the dump image command, with its *bond* keyword, when its color setting is *type*, to set the color that bonds of each type will be drawn in the image.

The specified *type* should be a type label or integer from 1 to *N*, where *N* is the number of bond types. For numeric types, a wildcard asterisk can be used in place of or in conjunction with the *type* argument to specify a range of bond types. This takes the form "*" or "*n" or "m*" or "m*n". If *N* is the number of bond types, then an asterisk with no numerical values means all types from 1 to *N*. A leading asterisk means all types from 1 to n (inclusive). A trailing asterisk means all types from m to *N* (inclusive). A middle asterisk means all types from m to n (inclusive).

The specified *color* can be a single color which is any of the 140 pre-defined colors (see below) or a color name defined by the dump_modify color option. Or it can be two or more colors separated by a "/" character (e.g., red/green/blue). In the former case, that color is assigned to all the specified bond types. In the latter case, the list of colors are assigned in a round-robin fashion to each of the specified bond types.

---

The *bdiam* keyword can be used with the dump image command, with its *bond* keyword, when its *diam* setting is *type*, to set the diameter that bonds of each type will be drawn in the image. The specified *type* should be a type label or integer from 1 to Nbondtypes. As with the *bcolor* keyword, a wildcard asterisk can be used as part of the *type* argument to specify a range of numeric bond types. The specified *diam* is the size in whatever distance *units* you are using (e.g., Angstroms).

---

The *bitrate* keyword can be used with the *dump movie* command to define the size of the resulting movie file and its quality via setting how many kbits per second are to be used for the movie file. Higher bitrates require less compression and will result in higher quality movies. The quality is also determined by the compression format and encoder. The default setting is 2000 kbit/s, which will result in average quality with older compression formats.

**Note**

Not all movie file formats supported by dump movie allow the bitrate to be set. If not, the setting is silently ignored.

---

The *boxcolor* keyword sets the color of the simulation box drawn around the atoms in each image as well as the color of processor subdomain boundaries. See the "dump image box" command for how to specify that a box be drawn via the *box* keyword, and the subdomain boundaries via the *subbox* keyword. The color name can be any of the 140 pre-defined colors (see below) or a color name defined by the dump_modify color option.

---

Changed in version 30Mar2026: add support for entering colors in hexadecimal

The *color* keyword allows defining new named colors or changing the definition of the 140-predefined colors (see below). Three red/green/blue RGB values are associated with each color name. The color name can then be used with any other *dump_modify* keyword that takes a color name as a value. The RGB values should be either specified as three floating point values between 0.0 and 1.0 inclusive or as a single 24-bit hexadecimal number. The following two commands are equivalent.

```
dump_modify 1 color mygray 0.431 0.498 0.502
dump_modify 1 color mygray 0x6e7f80
```

---

**Transparency settings for atoms bonds and standard visualization objects**

Added in version 11Feb2026.

Various graphical objects in *dump image* output can be rendered in a transparent fashion using the so-called screen-door transparency method. This means that only a subset of pixels for a graphical object are written to the image. This can be controlled with various *dump_modify* settings: *atrans* for atoms, *btrans* for bonds, *axestrans* for axes lines, *boxtrans*

for the simulation box, and *subboxtrans* for the subdomain box lines. The transparency value must be between 0.0 (invisible) and 1.0 (fully opaque). The default setting for all is 1.0.

Recommended transparency values are 0.25, 0.5, or 0.75 when used in combination with *fsaa on*.

---

Added in version 11Feb2026.

The *fcolor* keyword sets the color of any image objects created by a fix when using the color style "const". The first argument is the fix ID used with the *dump image fix* command and the second argument is the color name. The color name can be any of the 140 pre-defined colors (see below) or a color name defined by the *dump_modify color* option.

The *ftrans* keyword sets the transparency of any image objects created by a fix when using the color style "const". The first argument is the fix ID used with the *dump image fix* command and the second argument is the transparency value. The transparency value must be between 0.0 (invisible) and 1.0 (fully opaque). The default setting is 1.0.

Recommended transparency values are 0.25, 0.5, or 0.75 when used in combination with *fsaa on*.

---

The *framerate* keyword can be used with the *dump movie* command to define the duration of the resulting movie file. Movie files written by the dump *movie* command have a default frame rate of 24 frames per second and the images generated will be converted at that rate. Thus a sequence of 1000 dump images will result in a movie of about 42 seconds. To make a movie run longer you can either generate images more frequently or lower the frame rate. To speed a movie up, you can do the inverse. Using a frame rate higher than 24 is not recommended, as it will result in simply dropping the rendered images. It is more efficient to dump images less frequently.

---

The *gmap* keyword can be used with the dump image command, with its *grid* keyword, to setup a color map. The color map is used to assign a specific RGB (red/green/blue) color value to an individual grid cell when it is drawn, based on the grid cell value, which is a numeric quantity specified with the *grid* keyword.

The arguments for the *gmap* keyword are identical to those for the *amap* keyword (for atom coloring) described above.

---

Added in version 4Jul2026.

The *bmap* keyword can be used with the dump image command, with its *bond* keyword, when the bond *color* value is a compute reference (*c_ID* or *c_ID[I]*), to setup a color map. The color map is used to assign a specific RGB (red/green/blue) color value to an individual bond when it is drawn, based on the per-bond value returned by the referenced compute.

The arguments for the *bmap* keyword are identical to those for the *amap* keyword (for atom coloring) described above.

---

Added in version 4Jul2026.

The *lights* keyword can be used to set the relative intensities of the four light sources used to illuminate the scene: *ambient*, *key*, *fill*, and *back*. Each value must be between 0.0 and 1.0.

```
dump_modify 1 lights 0.3 0.7 0.4 0.2
```

The *ambient* light provides base-level illumination from all directions. The *key* light is the primary light source and creates the main highlights. The *fill* light is a secondary light source that softens shadows created by the key light. The *back* light illuminates the scene from behind the camera to provide depth.

---

Added in version 4Jul2026.

The *loadcolors* and *savecolors* keywords can be used to read or write the current per-atom-type color assignments and their definitions from or to a JSON format file. Also, the current *lights* settings are read and applied or stored. These files can be read, modified interactively, and written by LAMMPS-GUI. This provides a convenient way to have custom color definitions and custom color to type assignments. When the system has more atom types than colors, colors are named "type#" with "#" being the number of the atom type so that the color names are unique. Per-element colors currently cannot be saved or customized.

Below is a simple example for a colors file for a system with two atom types using the default color and light settings:

```
{
     "application": "LAMMPS",
     "format": "colors",
     "revision": 1,
     "title": "per-type colors for dump image",
     "schema": "https://download.lammps.org/json/color-schema.json",
     "colors": [
        {
             "name": "red",
             "red": 1.0,
             "green": 0.0,
             "blue": 0.0
         },
         {
             "name": "forestgreen",
             "red": 0.133,
             "green": 0.545,
             "blue": 0.133
         }
     ],
     "lights": {
         "ambient": 0.0,
         "key": 0.9,
         "fill": 0.45,
         "back": 0.9
     }
}
```

## 9.17.5 Restrictions

The *dump image* and *dump movie* commands are part of the GRAPHICS package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

To write JPEG or PNG format images, support for the corresponding graphics libraries must have been compiled and linked into LAMMPS. Please see the *instructions for building LAMMPS with the GRAPHICS package* for more information on how to do that.

To write *movie* dumps, you must use the -DLAMMPS_FFMPEG switch when building LAMMPS and have the FFmpeg executable available on the machine where LAMMPS is being run. Typically its name is lowercase (i.e., "ffmpeg").

Note that since FFmpeg is run as an external program via a pipe, LAMMPS has limited control over its execution and no knowledge about errors and warnings printed by it. Those warnings and error messages will be printed to the screen only. Due to the way image data are communicated to FFmpeg, it will often print the message

```
pipe:: Input/output error
```

which can be safely ignored. Other warnings and errors have to be addressed according to the FFmpeg documentation. One known issue is that certain movie file formats (e.g., MPEG level 1 and 2 format streams) have video bandwidth limits that can be crossed when rendering too large of image sizes. Typical warnings look like this:

```
[mpeg @ 0x98b5e0] packet too large, ignoring buffer limits to mux it
[mpeg @ 0x98b5e0] buffer underflow st=0 bufi=281407 size=285018
[mpeg @ 0x98b5e0] buffer underflow st=0 bufi=283448 size=285018
```

In this case it is recommended either to reduce the size of the image or to encode in a different format that is also supported by your copy of FFmpeg and which does not have this limitation (e.g., .avi, .mkv, mp4).

### 9.17.6 Related commands

*dump*, *dump_modify*, *undump*, *fix graphics/arrows*, *fix graphics/isosurface*, *fix graphics/labels*, *fix graphics/objects*, *fix graphics/periodic*

### 9.17.7 Default

The defaults for the dump image and dump movie keywords are as follows:

- adiam = not specified (use diameter setting)
- atom = yes
- bond = none none (if no bonds in system)
- bond = atom 0.5 (if bonds in system)
- size = 512 512
- view = 60 30 (for 3d)
- view = 0 0 (for 2d)
- center = s 0.5 0.5 0.5
- up = 0 0 1 (for 3d)
- up = 0 1 0 (for 2d)
- zoom = 1.0
- box = yes 0.02
- axes = no 0.0 0.0
- subbox = no 0.0
- shiny = 1.0
- ssao = no
- fsaa = no

---

The defaults for the dump_modify keywords specific to dump image and dump movie are as follows:

- acolor = * red/forestgreen/blue/gold/cyan/magenta/silver/orange/lime/gray/darkred/darkgreen/darkblue/darkcyan/darkmagenta/dar
- adiam = * 1.0
- amap = min max cf 0.0 2 min blue max red

- atrans = 1.0
- backcolor = black
- backcolor2 = none
- bcolor = * red/forestgreen/blue/gold/cyan/magenta/silver/orange/lime/gray/darkred/darkgreen/darkblue/darkcyan/darkmagenta/dar
- bdiam = * 0.5
- btrans = 1.0
- boxcolor = gold
- axestrans = 1.0
- boxtrans = 1.0
- subboxtrans = 1.0
- color = 140 color names are pre-defined as listed below
- lights = 0.0 0.9 0.45 0.9
- bitrate = 2000
- framerate = 24
- gmap = min max cf 0.0 2 min blue max red

Default color sequence:

---

These are the standard 109 element names that LAMMPS pre-defines for use with the dump image and dump_modify commands.

- 1-10 = “H”, “He”, “Li”, “Be”, “B”, “C”, “N”, “O”, “F”, “Ne”
- 11-20 = “Na”, “Mg”, “Al”, “Si”, “P”, “S”, “Cl”, “Ar”, “K”, “Ca”
- 21-30 = “Sc”, “Ti”, “V”, “Cr”, “Mn”, “Fe”, “Co”, “Ni”, “Cu”, “Zn”
- 31-40 = “Ga”, “Ge”, “As”, “Se”, “Br”, “Kr”, “Rb”, “Sr”, “Y”, “Zr”
- 41-50 = “Nb”, “Mo”, “Tc”, “Ru”, “Rh”, “Pd”, “Ag”, “Cd”, “In”, “Sn”
- 51-60 = “Sb”, “Te”, “I”, “Xe”, “Cs”, “Ba”, “La”, “Ce”, “Pr”, “Nd”
- 61-70 = “Pm”, “Sm”, “Eu”, “Gd”, “Tb”, “Dy”, “Ho”, “Er”, “Tm”, “Yb”
- 71-80 = “Lu”, “Hf”, “Ta”, “W”, “Re”, “Os”, “Ir”, “Pt”, “Au”, “Hg”
- 81-90 = “Tl”, “Pb”, “Bi”, “Po”, “At”, “Rn”, “Fr”, “Ra”, “Ac”, “Th”
- 91-100 = “Pa”, “U”, “Np”, “Pu”, “Am”, “Cm”, “Bk”, “Cf”, “Es”, “Fm”
- 101-109 = “Md”, “No”, “Lr”, “Rf”, “Db”, “Sg”, “Bh”, “Hs”, “Mt”

---

These are the 140 colors that LAMMPS pre-defines for use with the dump image and dump_modify commands. Additional colors can be defined with the dump_modify color command. The 3 numbers listed for each name are the RGB (red/green/blue) values.

Table 1: Pre-defined colors

| | | |
|---|---|---|
| aliceblue: 0.941, 0.973, 1.000 | antiquewhite: 0.980, 0.922, 0.843 | aqua: 0.000, 1.000, 1.000 |
| aquamarine: 0.498, 1.000, 0.831 | azure: 0.941, 1.000, 1.000 | beige: 0.961, 0.961, 0.863 |
| bisque: 1.000, 0.894, 0.769 | black: 0.000, 0.000, 0.000 | blanchedalmond: 1.000, 1.000, 0.804 |
| blue: 0.000, 0.000, 1.000 | blueviolet: 0.541, 0.169, 0.886 | brown: 0.647, 0.165, 0.165 |
| burlywood: 0.871, 0.722, 0.529 | cadetblue: 0.373, 0.620, 0.627 | chartreuse: 0.498, 1.000, 0.000 |
| chocolate: 0.824, 0.412, 0.118 | coral: 1.000, 0.498, 0.314 | cornflowerblue: 0.392, 0.584, 0.929 |
| cornsilk: 1.000, 0.973, 0.863 | crimson: 0.863, 0.078, 0.235 | cyan: 0.000, 1.000, 1.000 |
| darkblue: 0.000, 0.000, 0.545 | darkcyan: 0.000, 0.545, 0.545 | darkgoldenrod: 0.722, 0.525, 0.043 |
| darkgray: 0.271, 0.271, 0.271 | darkgreen: 0.000, 0.392, 0.000 | darkkhaki: 0.741, 0.718, 0.420 |
| darkmagenta: 0.545, 0.000, 0.545 | darkolivegreen: 0.333, 0.420, 0.184 | darkorange: 0.545, 0.271, 0.000 |
| darkorchid: 0.600, 0.196, 0.800 | darkred: 0.545, 0.000, 0.000 | darksalmon: 0.914, 0.588, 0.478 |
| darkseagreen: 0.561, 0.737, 0.561 | darkslateblue: 0.282, 0.239, 0.545 | darkslategray: 0.184, 0.310, 0.310 |
| darkturquoise: 0.000, 0.808, 0.820 | darkviolet: 0.580, 0.000, 0.827 | deeppink: 1.000, 0.078, 0.576 |
| deepskyblue: 0.000, 0.749, 1.000 | dimgray: 0.412, 0.412, 0.412 | dodgerblue: 0.118, 0.565, 1.000 |
| firebrick: 0.698, 0.133, 0.133 | floralwhite: 1.000, 0.980, 0.941 | forestgreen: 0.133, 0.545, 0.133 |
| fuchsia: 1.000, 0.000, 1.000 | gainsboro: 0.863, 0.863, 0.863 | ghostwhite: 0.973, 0.973, 1.000 |
| gold: 1.000, 0.843, 0.000 | goldenrod: 0.855, 0.647, 0.125 | gray: 0.502, 0.502, 0.502 |
| green: 0.000, 1.000, 0.000 | greenyellow: 0.678, 1.000, 0.184 | honeydew: 0.941, 1.000, 0.941 |
| hotpink: 1.000, 0.412, 0.706 | indianred: 0.804, 0.361, 0.361 | indigo: 0.294, 0.000, 0.510 |
| ivory: 1.000, 0.941, 0.941 | khaki: 0.941, 0.902, 0.549 | lavender: 0.902, 0.902, 0.980 |
| lavenderblush: 1.000, 0.941, 0.961 | lawngreen: 0.486, 0.988, 0.000 | lemonchiffon: 1.000, 0.980, 0.804 |
| lightblue: 0.678, 0.847, 0.902 | lightcoral: 0.941, 0.502, 0.502 | lightcyan: 0.878, 1.000, 1.000 |
| lightgoldenrodyellow: 0.980, 0.980, 0.824 | lightgreen: 0.565, 0.933, 0.565 | lightgrey: 0.827, 0.827, 0.827 |
| lightpink: 1.000, 0.714, 0.757 | lightsalmon: 1.000, 0.627, 0.478 | lightseagreen: 0.125, 0.698, 0.667 |
| lightskyblue: 0.529, 0.808, 0.980 | lightslategray: 0.467, 0.533, 0.600 | lightsteelblue: 0.690, 0.769, 0.871 |
| lightyellow: 1.000, 1.000, 0.878 | lime: 0.000, 1.000, 0.000 | limegreen: 0.196, 0.804, 0.196 |
| linen: 0.980, 0.941, 0.902 | magenta: 1.000, 0.000, 1.000 | maroon: 0.502, 0.000, 0.000 |
| mediumaquamarine: 0.400, 0.804, 0.667 | mediumblue: 0.000, 0.000, 0.804 | mediumorchid: 0.729, 0.333, 0.827 |

continues on next page

Table 1 – continued from previous page

| | | |
|---|---|---|
| mediumpurple: 0.576, 0.439, 0.859 | mediumseagreen: 0.235, 0.702, 0.443 | mediumslateblue: 0.482, 0.408, 0.933 |
| mediumspringgreen: 0.000, 0.980, 0.604 | mediumturquoise: 0.282, 0.820, 0.800 | mediumvioletred: 0.780, 0.082, 0.522 |
| midnightblue: 0.098, 0.098, 0.439 | mintcream: 0.961, 1.000, 0.980 | mistyrose: 1.000, 0.894, 0.882 |
| moccasin: 1.000, 0.894, 0.710 | navajowhite: 1.000, 0.871, 0.678 | navy: 0.000, 0.000, 0.502 |
| oldlace: 0.992, 0.961, 0.902 | olive: 0.502, 0.502, 0.000 | olivedrab: 0.420, 0.557, 0.137 |
| orange: 1.000, 0.502, 0.000 | orangered: 1.000, 0.251, 0.000 | orchid: 0.855, 0.439, 0.839 |
| palegoldenrod: 0.933, 0.910, 0.667 | palegreen: 0.596, 0.984, 0.596 | paleturquoise: 0.686, 0.933, 0.933 |
| palevioletred: 0.859, 0.439, 0.576 | papayawhip: 1.000, 0.937, 0.835 | peachpuff: 1.000, 0.937, 0.835 |
| peru: 0.804, 0.522, 0.247 | pink: 1.000, 0.753, 0.796 | plum: 0.867, 0.627, 0.867 |
| powderblue: 0.690, 0.878, 0.902 | purple: 0.502, 0.000, 0.502 | red: 1.000, 0.000, 0.000 |
| rosybrown: 0.737, 0.561, 0.561 | royalblue: 0.255, 0.412, 0.882 | saddlebrown: 0.545, 0.271, 0.075 |
| salmon: 0.980, 0.502, 0.447 | sandybrown: 0.957, 0.643, 0.376 | seagreen: 0.180, 0.545, 0.341 |
| seashell: 1.000, 0.961, 0.933 | sienna: 0.627, 0.322, 0.176 | silver: 0.753, 0.753, 0.753 |
| skyblue: 0.529, 0.808, 0.922 | slateblue: 0.416, 0.353, 0.804 | slategray: 0.439, 0.502, 0.565 |
| snow: 1.000, 0.980, 0.980 | springgreen: 0.000, 1.000, 0.498 | steelblue: 0.275, 0.510, 0.706 |
| tan: 0.824, 0.706, 0.549 | teal: 0.000, 0.502, 0.502 | thistle: 0.847, 0.749, 0.847 |
| tomato: 0.992, 0.388, 0.278 | turquoise: 0.251, 0.878, 0.816 | violet: 0.933, 0.510, 0.933 |
| wheat: 0.961, 0.871, 0.702 | white: 1.000, 1.000, 1.000 | whitesmoke: 0.961, 0.961, 0.961 |
| yellow: 1.000, 1.000, 0.000 | yellowgreen: 0.604, 0.804, 0.196 | |

## 9.18 dump_modify command

## 9.19 *dump_modify* command for image/movie options

### 9.19.1 Syntax

```
dump_modify dump-ID keyword values ...
```

- dump-ID = ID of dump to modify
- one or more keyword/value pairs may be appended
- these keywords apply to various dump styles
- keyword = *append* or *at* or *balance* or *buffer* or *colname* or *delay* or *element* or *every* or *every/time* or *fileper* or *first* or *flush* or *format* or *header* or *image* or *label* or *maxfiles* or *nfile* or *pad* or *pbc* or *precision* or *region* or *refresh* or *scale* or *sfactor* or *skip* or *sort* or *tfactor* or *thermo* or *thresh* or *time* or *triclinic/general* or *types* or *units* or *unwrap*

  *append* arg = *yes* or *no*
  *at* arg = N

```
  N = index of frame written upon first dump
balance arg = yes or no
buffer arg = yes or no
colname values =  ID string, or default
  string = new column header name
  ID = integer from 1 to N, or integer from -1 to -N, where N = # of quantities␣
↪being output
      or a custom dump keyword or reference to compute, fix, property or variable.
delay arg = Dstep
  Dstep = delay output until this timestep
element args = E1 E2 ... EN, where N = # of atom types
  E1,...,EN = element name (e.g., C or Fe or Ga)
every arg = N
  N = dump on timesteps which are a multiple of N
  N can be a variable (see below)
every/time arg = Delta
  Delta = dump once every Delta interval of simulation time (time units)
  Delta can be a variable (see below)
fileper arg = Np
  Np = write one file for every this many processors
first arg = yes or no
flush arg = yes or no
format args = line string, int string, float string, ID string, or none
  string = C-style format string
  ID = integer from 1 to N, or integer from -1 to -N, where N = # of quantities␣
↪being output
      or a custom dump keyword or reference to compute, fix, property or variable.
header arg = yes or no
  yes to write the header
  no to not write the header
image arg = yes or no
label arg = string
  string = character string (e.g., BONDS) to use in header of dump local file
maxfiles arg = Fmax
  Fmax = keep only the most recent Fmax snapshots (one snapshot per file)
nfile arg = Nf
  Nf = write this many files, one from each of Nf processors
pad arg = Nchar = # of characters to convert timestep to
pbc arg = yes or no = remap atoms via periodic boundary conditions
precision arg = power-of-10 value from 10 to 1000000
region arg = region-ID or "none"
refresh arg = c_ID = compute ID that supports a refresh operation
scale arg = yes or no
sfactor arg = coordinate scaling factor (> 0.0)
skip arg = v_name
  v_name = variable with name which evaluates to non-zero (skip) or 0
sort arg = off or id or N or -N
   off = no sorting of per-atom lines within a snapshot
   id = sort per-atom lines by atom ID
   N = sort per-atom lines in ascending order by the Nth column
   -N = sort per-atom lines in descending order by the Nth column
tfactor arg = time scaling factor (> 0.0)
thermo arg = yes or no
thresh args = attribute operator value
```

```
  attribute = same attributes (x,fy,etotal,sxx,etc) used by dump custom style
  operator = "<" or "<=" or ">" or ">=" or "==" or "!=" or "|^"
  value = numeric value to compare to, or LAST
  these 3 args can be replaced by the word "none" to turn off thresholding
time arg = yes or no
triclinic/general arg = yes or no
types value = numeric or labels
units arg = yes or no
unwrap arg = yes or no
```

- these keywords apply only to the *image* and *movie* *styles*
- keyword = *acolor* or *adiam* or *amap* or *backcolor* or *bcolor* or *bdiam* or *boxcolor* or *color* or *bitrate* or *framerate*

  see the *dump image* doc page for details

- these keywords apply only to the extxyz dump style
- keyword = *forces* or *mass* or *vel*

  ```
  forces arg = yes or no
  mass arg = yes or no
  vel arg = yes or no
  ```

- these keywords apply only to the */gz* and */zstd* dump styles
- keyword = *compression_level*

  ```
  compression_level args = level
    level = integer specifying the compression level that should be used (see below␣
  ↪for supported levels)
  ```

- these keywords apply only to the */zstd* dump styles
- keyword = *checksum*

  *checksum* args = *yes* or *no* (add checksum at end of zst file)

- these keywords apply only to the vtk* dump style
- keyword = *binary*

  *binary* args = *yes* or *no* (select between binary and text mode VTK files)

## 9.19.2 Examples

```
dump_modify 1 format line "%d %d %20.15g %g %g" scale yes
dump_modify 1 format float %20.15g scale yes
dump_modify myDump image yes scale no flush yes
dump_modify 1 region mySphere thresh x < 0.0 thresh fx >= 3.2
dump_modify xtcdump precision 10000 sfactor 0.1
dump_modify 1 every 1000 nfile 20
dump_modify 1 every v_myVar
```

## 9.19.3 Description

Modify the parameters of a previously defined dump command. Not all parameters are relevant to all dump styles.

Unless otherwise noted, the following keywords apply to all the various dump styles, including the *dump image* and *dump movie* styles.

The *append* keyword applies to all dump styles except *cfg* and *xtc* and *dcd*. It also applies only to text output files, not to binary or compressed or image/movie files. If specified as *yes*, then dump snapshots are appended to the end of an existing dump file. If specified as *no*, then a new dump file will be created which will overwrite an existing file with the same name.

---

The *at* keyword only applies to the *netcdf* dump style. It can only be used if the *append yes* keyword is also used. The *N* argument is the index of which frame to append to. A negative value can be specified for *N*, which means a frame counted from the end of the file. The *at* keyword can only be used if the dump_modify command is before the first command that causes dump snapshots to be output (e.g., a *run* or *minimize* command). Once the dump file has been opened, this keyword has no further effect.

---

The *buffer* keyword applies only to dump styles *atom*, *cfg*, *custom*, *local*, and *xyz*. It also applies only to text output files, not to binary or compressed files. If specified as *yes*, which is the default, then each processor writes its output into an internal text buffer, which is then sent to the processor(s) which perform file writes, and written by those processors(s) as one large chunk of text. If specified as *no*, each processor sends its per-atom data in binary format to the processor(s) which perform file wirtes, and those processor(s) format and write it line by line into the output file.

The buffering mode is typically faster since each processor does the relatively expensive task of formatting the output for its own atoms. However it requires about twice the memory (per processor) for the extra buffering.

---

Added in version 4May2022.

The *colname* keyword can be used to change the default header keyword for dump styles: *atom*, *custom*, *cfg*, and *local* and their compressed, ADIOS variants. The setting for *ID string* replaces the default text with the provided string. *ID* can be a positive integer when it represents the column number counting from the left, a negative integer when it represents the column number from the right (i.e. -1 is the last column/keyword), or a custom dump keyword (or compute, fix, property, or variable reference) and then it replaces the string for that specific keyword. For *atom* dump styles only the keywords "id", "type", "x", "y", "z", "ix", "iy", "iz" can be accessed via string regardless of whether scaled or unwrapped coordinates were enabled or disabled, and it always assumes 8 columns for indexing regardless of whether image flags are enabled or not. For dump style *cfg* only changes to the "auxiliary" keywords (6th or later keyword) will become visible.

The *colname* keyword can be used multiple times. If multiple *colname* settings refer to the same keyword, the last setting has precedence. A setting of *default* clears all previous settings, reverting all values to their default names. Using the *scale* or *image* keyword will also reset all header keywords to their default values.

---

The *delay* keyword applies to all dump styles. No snapshots will be output until the specified *Dstep* timestep or later. Specifying *Dstep* < 0 is the same as turning off the delay setting. This is a way to turn off unwanted output early in a simulation, for example, during an equilibration phase.

---

The *element* keyword applies only to the dump *cfg*, *xyz*, and *image* styles. It associates element names (e.g., H, C, Fe) with LAMMPS atom types. See the list of element names at the bottom of this page.

In the case of dump *cfg*, this allows the AtomEye visualization package to read the dump file and render atoms with the appropriate size and color.

In the case of dump *image*, the output images will follow the same AtomEye convention. An element name is specified for each atom type (1 to Ntype) in the simulation. The same element name can be given to multiple atom types.

In the case of *xyz* format dumps, there are no restrictions to what label can be used as an element name. Any white-space separated text will be accepted.

The *every* keyword can be used with any dump style except the *dcd* and *xtc* styles. It specifies that the output of dump snapshots will now be performed on timesteps which are a multiple of a new *N* value, This overrides the dump frequency originally specified by the *dump* command.

The *every* keyword can be specified in one of two ways. It can be a numeric value in which case it must be > 0. Or it can be an *equal-style variable*, which should be specified as v_name, where name is the variable name.

In this case, the variable is evaluated at the beginning of a run to determine the next timestep at which a dump snapshot will be written out. On that timestep the variable will be evaluated again to determine the next timestep, etc. Thus the variable should return timestep values. See the stagger() and logfreq() and stride() math functions for *equal-style variables*, as examples of useful functions to use in this context. Other similar math functions could easily be added as options for *equal-style variables*. Also see the next() function, which allows use of a file-style variable which reads successive values from a file, each time the variable is evaluated. Used with the *every* keyword, if the file contains a list of ascending timesteps, you can output snapshots whenever you wish.

Note that when using the variable option with the *every* keyword, you need to use the *first* option if you want an initial snapshot written to the dump file. The *every* keyword cannot be used with the dump *dcd* style.

For example, the following commands will write snapshots at timesteps 0,10,20,30,100,200,300,1000,2000,etc:

```
variable        s equal logfreq(10,3,10)
dump            1 all atom 100 tmp.dump
dump_modify     1 every v_s first yes
```

The following commands would write snapshots at the timesteps listed in file tmp.times:

```
variable        f file tmp.times
variable        s equal next(f)
dump            1 all atom 100 tmp.dump
dump_modify     1 every v_s
```

**Note**

When using a file-style variable with the *every* keyword, the file of timesteps must list a first timestep that is beyond the current timestep (e.g., it cannot be 0). And it must list one or more timesteps beyond the length of the run you perform. This is because the dump command will generate an error if the next timestep it reads from the file is not a value greater than the current timestep. Thus if you wanted output on steps 0,15,100 of a 100-timestep run, the file should contain the values 15,100,101 and you should also use the dump_modify first command. Any final value > 100 could be used in place of 101.

---

Added in version 7Jan2022.

The *every/time* keyword can be used with any dump style except the *dcd* and *xtc* styles. It changes the frequency of dump snapshots from being based on the current timestep to being determined by elapsed simulation time, i.e. in time units of the *units* command, and specifies *Delta* for the interval between snapshots. This can be useful when the timestep size varies during a simulation run, e.g. by use of the *fix dt/reset* command. The default is to perform output on timesteps which a multiples of specified timestep value *N*; see the *every* keyword.

The *every/time* keyword can be used with any dump style except the *dcd* and *xtc* styles. It does two things. It specifies that the interval between dump snapshots will be set in simulation time (i.e. in time units of the *units* command). This can be useful when the timestep size varies during a simulation run (e.g., by use of the *fix dt/reset* command). The default is to specify the interval in timesteps; see the *every* keyword. The *every/time* command also sets the interval value.

**Note**

If you wish dump styles *atom*, *custom*, *local*, or *xyz* to include the simulation time as a field in the header portion of each snapshot, you also need to use the dump_modify *time* keyword with a setting of *yes*. See its documentation below.

Note that since snapshots are output on simulation steps, each snapshot will be written on the first timestep whose associated simulation time is >= the exact snapshot time value.

As with the *every* option, the *Delta* value can be specified in one of two ways. It can be a numeric value in which case it must be > 0.0. Or it can be an *equal-style variable*, which should be specified as v_name, where name is the variable name.

In this case, the variable is evaluated at the beginning of a run to determine the next simulation time at which a dump snapshot will be written out. On that timestep the variable will be evaluated again to determine the next simulation time, etc. Thus the variable should return values in time units. Note the current timestep or simulation time can be used in an *equal-style variables* since they are both thermodynamic keywords. Also see the next() function, which allows use of a file-style variable which reads successive values from a file, each time the variable is evaluated. Used with the *every/time* keyword, if the file contains a list of ascending simulation times, you can output snapshots whenever you wish.

Note that when using the variable option with the *every/time* keyword, you need to use the *first* option if you want an initial snapshot written to the dump file. The *every/time* keyword cannot be used with the dump *dcd* style.

For example, the following commands will write snapshots at successive simulation times which grow by a factor of 1.5 with each interval. The dt value used in the variable is to avoid a zero result when the initial simulation time is 0.0.

```
variable        increase equal 1.5*(time+dt)
dump            1 all atom 100 tmp.dump
dump_modify     1 every/time v_increase first yes
```

The following commands would write snapshots at the times listed in file tmp.times:

```
variable        f file tmp.times
variable        s equal next(f)
dump            1 all atom 100 tmp.dump
dump_modify     1 every/time v_s
```

**Note**

When using a file-style variable with the *every/time* keyword, the file of timesteps must list a first time that is beyond the time associated with the current timestep (e.g., it cannot be 0.0). And it must list one or more times beyond the length of the run you perform. This is because the dump command will generate an error if the next time it reads from the file is not a value greater than the current time. Thus if you wanted output at times 0,15,100 of a run of length 100 in simulation time, the file should contain the values 15,100,101 and you should also use the dump_modify first command. Any final value > 100 could be used in place of 101.

---

The *first* keyword determines whether a dump snapshot is written on the very first timestep after the dump command is invoked. This will always occur if the current timestep is a multiple of $N$, the frequency specified in the *dump* command or *dump_modify every* command, including timestep 0. It will also always occur if the current simulation time is a multiple of *Delta*, the time interval specified in the *dump_modify every/time* command.

But if this is not the case, a dump snapshot will only be written if the setting of this keyword is *yes*. If it is *no*, which is the default, then it will not be written.

Note that if the argument to the *dump_modify every* *dump_modify every/time* commands is a variable and not a numeric value, then specifying *first yes* is the only way to write a dump snapshot on the first timestep after the dump command is invoked.

---

The *flush* keyword determines whether a flush operation is invoked after a dump snapshot is written to the dump file. A flush ensures the output in that file is current (no buffering by the OS), even if LAMMPS halts before the simulation completes. Flushes cannot be performed with dump style *xtc*.

---

The *format* keyword can be used to change the default numeric format output by the text-based dump styles: *atom*, *local*, *custom*, *cfg*, and *xyz* styles. Only the *line* or *none* options can be used with the *atom* and *xyz* styles.

All the specified format strings are C-style formats, such as used by the C/C++ printf() command. The *line* keyword takes a single argument which is the format string for an entire line of output for each atom (do not include a trailing "n"), with *N* fields, which you must enclose in quotes if there is more than one field. The *int* and *float* keywords take a single format argument and are applied to all integer or floating-point quantities output. The setting for *M string* also takes a single format argument which is used for the *M*th value output in each line (e.g., the fifth column is output in high precision by "format 5 %20.15g").

**Note**

When using the *line* keyword for the *cfg* style, the first two fields (atom ID and type) are not actually written into the CFG file, however you must include formats for them in the format string.

The *format* keyword can be used multiple times. The precedence is that for each value in a line of output, the *M* format (if specified) is used, else the *int* or *float* setting (if specified) is used, else the *line* setting (if specified) for that value is used, else the default setting is used. A setting of *none* clears all previous settings, reverting all values to their default format.

**Note**

Atom and molecule IDs are stored internally as 4-byte or 8-byte signed integers, depending on how LAMMPS was compiled. When specifying the *format int* option you can use a "%d"-style format identifier in the format string and LAMMPS will convert this to the corresponding 8-byte form if it is needed when outputting those values. However, when specifying the *line* option or *format M string* option for those values, you should specify a format string appropriate for an 8-byte signed integer (e.g., one with "%ld") if LAMMPS was compiled with the -DLAMMPS_BIGBIG option for 8-byte IDs.

**Note**

Any value written to a text-based dump file that is a per-atom quantity calculated by a *compute* or *fix* is stored internally as a floating-point value. If the value is actually an integer and you wish it to appear in the text dump file as a (large) integer, then you need to use an appropriate format. For example, these commands:

```
compute     1 all property/local batom1 batom2
dump        1 all local 100 tmp.bonds index c_1[1] c_1[2]
dump_modify 1 format line "%d %0.0f %0.0f"
```

will output the two atom IDs for atoms in each bond as integers. If the dump_modify command were omitted, they would appear as floating-point values, assuming they were large integers (more than six digits). The "index" keyword should use the "%d" format since it is not generated by a compute or fix, and is stored internally as an integer.

---

The *fileper* keyword is documented below with the *nfile* keyword.

---

The *header* keyword toggles whether the dump file will include a header. Excluding a header will reduce the size of the dump file for data produced by *pair tracker* or *bpm bond styles* which may not require the information typically written to the header.

---

The *image* keyword applies only to the dump *atom* style. If the image value is *yes*, three flags are appended to each atom's coords which are the absolute box image of the atom in each dimension. For example, an $x$ image flag of $-2$ with a normalized coord of 0.5 means the atom is in the center of the box, but has passed through the box boundary twice and is really two box lengths to the left of its current coordinate. Note that for dump style *custom* these various values can be printed in the dump file by using the appropriate atom attributes in the dump command itself. Using this keyword will reset all custom header names set with *dump_modify colname* to their respective default values.

---

The *label* keyword applies only to the dump *local* style. When it writes local information, such as bond or angle topology to a dump file, it will use the specified *label* to format the header. By default this includes two lines:

```
ITEM: NUMBER OF ENTRIES
ITEM: ENTRIES ...
```

The word "ENTRIES" will be replaced with the string specified (e.g., BONDS or ANGLES).

---

The *maxfiles* keyword can only be used when a '*' wildcard is included in the dump file name (i.e., when writing a new file(s) for each snapshot). The specified *Fmax* is how many snapshots will be kept. Once this number is reached, the file(s) containing the oldest snapshot is deleted before a new dump file is written. If the specified Fmax $\leq 0$, then all files are retained.

This can be useful for debugging, especially if you do not know on what timestep something bad will happen (e.g., when LAMMPS will exit with an error). You can dump every time step and limit the number of dump files produced, even if you run for thousands of steps.

---

The *nfile* or *fileper* keywords can be used in conjunction with the "%" wildcard character in the specified dump file name, for all dump styles except the *dcd*, *image*, *movie*, *xtc*, and *xyz* styles (for which "%" is not allowed). As explained on the *dump* command doc page, the "%" character causes the dump file to be written in pieces, one piece for each of $P$ processors. By default, $P$ is the number of processors the simulation is running on. The *nfile* or *fileper* keyword can be used to set $P$ to a smaller value, which can be more efficient when running on a large number of processors.

The *nfile* keyword sets $P$ to the specified $N_f$ value. For example, if $N_f = 4$, and the simulation is running on 100 processors, four files will be written by processors 0, 25, 50, and 75. Each will collect information from itself and the next 24 processors and write it to a dump file.

For the *fileper* keyword, the specified value of $N_p$ means write one file for every $N_p$ processors. For example, if $N_p = 4$, every fourth processor (0, 4, 8, 12, etc.) will collect information from itself and the next three processors and write it to a dump file.

---

The *pad* keyword only applies when the dump filename is specified with a wildcard "*" character which becomes the timestep. If *pad* is 0, which is the default, the timestep is converted into a string of unpadded length (e.g., 100 or 12000 or 2000000). When *pad* is specified with *Nchar* $>$ 0, the string is padded with leading zeroes so they are all the same length = *Nchar*. For example, pad 7 would yield 0000100, 0012000, 2000000. This can be useful so that post-processing programs can easily read the files in ascending timestep order.

---

The *pbc* keyword applies to all the dump styles. As explained on the *dump* doc page, atom coordinates in a dump file may be slightly outside the simulation box. This is because periodic boundary conditions are enforced only on timesteps when neighbor lists are rebuilt, which will not typically coincide with the timesteps dump snapshots are written. If the setting of this keyword is set to *yes*, then all atoms will be remapped to the periodic box before the snapshot is written, then restored to their original position. If it is set to *no* they will not be. The *no* setting is the default because it requires no extra computation.

---

The *precision* keyword only applies to the dump *xtc* style. A specified value of *N* means that coordinates are stored to $1/N$ nanometer accuracy (e.g., for $N$ = 1000, the coordinates are written to $1/1000$ nanometer accuracy).

---

The *refresh* keyword only applies to the dump *custom*, *cfg*, *image*, and *movie* styles. It allows an "incremental" dump file to be written, by refreshing a compute that is used as a threshold for determining which atoms are included in a dump snapshot. The specified *c_ID* gives the ID of the compute. It is prefixed by "c_" to indicate a compute, which is the only current option. At some point, other options may be added (e.g., fixes or variables).

**Note**

This keyword can only be specified once for a dump. Refreshes of multiple computes cannot yet be performed.

The definition and motivation of an incremental dump file is as follows. Instead of outputting all atoms at each snapshot (with some associated values), you may only wish to output the subset of atoms with a value that has changed in some way compared to the value the last time that atom was output. In some scenarios this can result in a dramatically smaller dump file. If desired, by post-processing the sequence of snapshots, the values for all atoms at all timesteps can be inferred.

A concrete example is a simulation of atom diffusion in a solid, represented as atoms on a lattice. Diffusive hops are rare. Imagine that when a hop occurs an atom moves more than a distance *Dhop*. For any snapshot we only want to output atoms that have hopped since the last snapshot. This can be accomplished with something the following commands:

```
variable        Dhop equal 0.6
variable        check atom "c_dsp[4] > v_Dhop"
compute         dsp all displace/atom refresh check
dump            1 all custom 20 tmp.dump id type x y z
dump_modify     1 append yes thresh c_dsp[4] > ${Dhop} refresh c_dsp
```

The *compute displace/atom* command calculates the displacement of each atom from its reference position. The "4" index is the scalar displacement; 1, 2, and 3 are the *xyz* components of the displacement. The *dump_modify thresh* command will cause only atoms that have displaced more than 0.6 Å to be output on a given snapshot (assuming metal units). However, note that when an atom is output, we also need to update the reference position for that atom to its new coordinates. So that it will not be output in every snapshot thereafter. That reference position is stored by *compute displace/atom*. So the dump_modify *refresh* option triggers a call to compute displace/atom at the end of every dump to perform that update. The *refresh check* option shown as part of the *compute displace/atom* command enables the compute to respond to the call from the dump command, and update the appropriate reference positions. This is done

be defining an *atom-style variable*, *check* in this example, which calculates a Boolean value (0 or 1) for each atom, based on the same criterion used by dump_modify thresh.

See the *compute displace/atom* command for more details, including an example of how to produce output that includes an initial snapshot with the reference position of all atoms.

Note that only computes with a *refresh* option will work with dump_modify refresh. See individual compute doc pages for details. Currently, only compute displace/atom supports this option. Others may be added at some point. If you use a compute that does not support refresh operations, LAMMPS will not complain; dump_modify refresh will simply do nothing.

---

The *region* keyword only applies to the dump *custom*, *cfg*, *image*, and *movie* styles. If specified, only atoms in the region will be written to the dump file or included in the image/movie. Only one region can be applied as a filter (the last one specified). See the *region* command for more details. Note that a region can be defined as the "inside" or "outside" of a geometric shape, and it can be the "union" or "intersection" of a series of simpler regions.

---

The *scale* keyword applies only to the dump *atom* style. A scale value of *yes* means atom coords are written in normalized units from 0.0 to 1.0 in each box dimension. If the simulation box is triclinic (tilted), then all atom coords will still be between 0.0 and 1.0. A value of *no* means they are written in absolute distance units (e.g., Å or $\sigma$). Using this keyword will reset all custom header names set with *dump_modify colname* to their respective default values.

---

The *sfactor* and *tfactor* keywords only apply to the dump *xtc* style. They allow customization of the unit conversion factors used when writing to XTC files. By default, they are initialized for whatever *units* style is being used, to write out coordinates in nanometers and time in picoseconds. For example, for *real* units, LAMMPS defines *sfactor* = 0.1 and *tfactor* = 0.001, since the Å and fs used by *real* units are 0.1 nm and 0.001 ps, respectively. If you are using a units system with distance and time units far from nm and ps, you may wish to write XTC files with different units, since the compression algorithm used in XTC files is most effective when the typical magnitude of position data is between 10.0 and 0.1.

---

Added in version 15Sep2022.

The *skip* keyword can be used with all dump styles. It allows a dump snapshot to be skipped (not written to the dump file), if a condition is met. The condition is computed by an *equal-style variable*, which should be specified as v_name, where name is the variable name. If the variable evaluation returns a non-zero value, then the dump snapshot is skipped. If it returns zero, the dump proceeds as usual. Note that *equal-style variable* can contain Boolean operators which effectively evaluate as a true (non-zero) or false (zero) result.

The *skip* keyword can be useful for debugging purposes, e.g. to dump only on a particular timestep. Or to limit output to conditions of interest, e.g. only when the force on some atom exceeds a threshold value.

---

The *sort* keyword determines whether lines of per-atom output in a snapshot are sorted or not. A sort value of *off* means they will typically be written in indeterminate order, either in serial or parallel. This is the case even in serial if the *atom_modify sort* option is turned on, which it is by default, to improve performance. A sort value of *id* means sort the output by atom ID. A sort value of $N$ or $-N$ means sort the output by the value in the $N$th column of per-atom info in either ascending or descending order.

The dump *local* style cannot be sorted by atom ID, since there are typically multiple lines of output per atom. Some dump styles, such as *dcd* and *xtc*, require sorting by atom ID to format the output file correctly. If multiple processors are writing the dump file, via the "%" wildcard in the dump filename and the *nfile* or *fileper* keywords are set to non-default values (i.e., the number of dump file pieces is not equal to the number of procs), then sorting cannot be performed.

In a parallel run, the per-processor dump file pieces can have significant imbalance in number of lines of per-atom info. The *balance* keyword determines whether the number of lines in each processor snapshot are balanced to be nearly the same. A balance value of *no* means no balancing will be done, while *yes* means balancing will be performed. This balancing preserves dump sorting order. For a serial run, this option is ignored since the output is already balanced.

**Note**

Unless it is required by the dump style, sorting dump file output requires extra overhead in terms of CPU and communication cost, as well as memory, versus unsorted output.

---

The *thermo* keyword only applies the dump styles *netcdf* and *yaml*. It triggers writing of *thermo* information to the dump file alongside per-atom data. The values included in the dump file are cached values from the last thermo output and include the exact same the values as specified by the *thermo_style* command. Because these are cached values, they are only up-to-date when dump output is on a timestep that also has thermo output. Dump style *yaml* will skip thermo output on incompatible steps.

---

The *thresh* keyword only applies to the dump *custom*, *cfg*, *image*, and *movie* styles. Multiple thresholds can be specified. Specifying *none* turns off all threshold criteria. If thresholds are specified, only atoms whose attributes meet all the threshold criteria are written to the dump file or included in the image. The possible attributes that can be tested for are the same as those that can be specified in the *dump custom* command, with the exception of the *element* attribute, since it is not a numeric value. Note that a different attributes can be used than those output by the *dump custom* command. For example, you can output the coordinates and stress of atoms whose energy is above some threshold.

If an atom-style variable is used as the attribute, then it can produce continuous numeric values or effective Boolean 0/1 values, which may be useful for the comparison operator. Boolean values can be generated by variable formulas that use comparison or Boolean math operators or special functions like gmask() and rmask() and grmask(). See the *variable* command page for details.

The specified value must be a simple numeric value or the word LAST. If LAST is used, it refers to the value of the attribute the last time the dump command was invoked to produce a snapshot. This is a way to only dump atoms whose attribute has changed (or not changed). Three examples follow.

```
dump_modify ... thresh ix != LAST
```

This will dump atoms which have crossed the periodic $x$ boundary of the simulation box since the last dump. (Note that atoms that crossed once and then crossed back between the two dump timesteps would not be included.)

```
region foo sphere 10 20 10 15
variable inregion atom rmask(foo)
dump_modify ... thresh v_inregion |^ LAST
```

This will dump atoms which crossed the boundary of the spherical region since the last dump.

```
variable charge atom "(q > 0.5) || (q < -0.5)"
dump_modify ... thresh v_charge |^ LAST
```

This will dump atoms whose charge has changed from an absolute value less than $\frac{1}{2}$ to greater than $\frac{1}{2}$ (or vice versa) since the last dump (e.g., due to reactions and subsequent charge equilibration in a reactive force field).

The choice of operators listed above are the usual comparison operators. The XOR operation (exclusive or) is also included as "|^". In this context, XOR means that if either the attribute or value is 0.0 and the other is non-zero, then the result is "true" and the threshold criterion is met. Otherwise it is not met.

**Note**

For style *custom*, the *triclinic/general* keyword can alter dump output for general triclinic simulation boxes and their atoms. See the *dump* command for details of how this changes the format of dump file snapshots. The thresh keyword may access per-atom attributes either directly or indirectly through a compute or variable. If the attribute is an atom coordinate or a per-atom vector (such as velocity, force, or dipole moment), its value will *NOT* be a general triclinic (rotated) value. Rather it will be a restricted triclinic value.

---

The *time* keyword only applies to the dump *atom*, *custom*, *local*, and *xyz* styles (and their COMPRESS package versions *atom/gz*, *custom/gz* and *local/gz*). For the first three styles, if set to *yes*, each frame will will contain two extra lines before the "ITEM: TIMESTEP" entry:

```
ITEM: TIME
<elapsed time>
```

For the *xyz* style, the simulation time is included on the same line as the timestep value.

This will output the current elapsed simulation time in current time units equivalent to the *thermo keyword time*. This is to simplify post-processing of trajectories using a variable time step (e.g., when using *fix dt/reset*). The default setting is *no*.

---

The *types* keyword applies only to the dump xyz style. If this keyword is used with a value of *numeric*, then numeric atom types are printed in the xyz file (default). If the value *labels* is specified, then *type labels* are printed for atom types.

---

The *triclinic/general* keyword only applies to the dump *atom* and *custom* styles. It can only be used with a value of *yes* if the simulation box was created as a general triclinic box. See the *Howto_triclinic* doc page for a detailed explanation of orthogonal, restricted triclinic, and general triclinic simulation boxes.

If this keyword is used with a value of *yes*, the box information at the beginning of each snapshot will include information about the 3 arbitrary edge vectors **A**, **B**, **C** that define the general triclinic box as well as their origin. The format is described on the *dump* doc page.

The coordinates of each atom will likewise be output as values in (or near) the general triclinic box. Likewise, per-atom vector quantities such as velocity, omega, dipole moment, etc will have orientations consistent with the general triclinic box, meaning they will be rotated relative to the standard xyz coordinate axes. See the *dump* doc page for a full list of which dump attributes this affects.

---

The *units* keyword only applies to the dump *atom*, *custom*, and *local* styles (and their COMPRESS package versions *atom/gz*, *custom/gz* and *local/gz*). If set to *yes*, each individual dump file will contain two extra lines at the very beginning with:

```
ITEM: UNITS
<units style>
```

This will output the current selected *units* style to the dump file and thus allows visualization and post-processing tools to determine the choice of units of the data in the dump file. The default setting is *no*.

---

The *unwrap* keyword only applies to the dump *dcd* and *xtc* styles. If set to *yes*, coordinates will be written "unwrapped" by the image flags for each atom. Unwrapped means that if the atom has passed through a periodic boundary one or

more times, the value is printed for what the coordinate would be if it had not been wrapped back into the periodic box. Note that these coordinates may thus be far outside the box size stored with the snapshot.

---

The *COMPRESS package* offers both GZ and Zstd compression variants of styles atom, custom, local, cfg, and xyz. When using these styles the compression level can be controlled by the `compression_level` keyword. File names with these styles have to end in either `.gz` or `.zst`.

GZ supports compression levels from −1 (default), 0 (no compression), and 1 to 9, 9 being the best compression. The COMPRESS `/gz` styles use 9 as default compression level.

Zstd offers a wider range of compression levels, including negative levels that sacrifice compression for performance. 0 is the default, positive levels are 1 to 22, with 22 being the most expensive compression. Zstd promises higher compression/decompression speeds for similar compression ratios. For more details see *https://facebook.github.io/zstd/*.

In addition, Zstd compressed files can include a checksum of the entire contents. The Zstd enabled dump styles enable this feature by default and it can be disabled with the `checksum` keyword.

---

The *VTK package* offers writing dump files in VTK file formats that can be read by a variety of visualization tools based on the VTK library. These VTK files follow naming conventions that collide with the LAMMPS convention to append ".bin" to a file name in order to switch to a binary output. Thus for *vtk style dumps* the dump_modify command supports the keyword *binary* which selects between generating text mode and binary style VTK files.

---

### 9.19.4 Restrictions

Not all *dump_modify* options can be applied to all dump styles. Details are in the discussions of the individual options.

### 9.19.5 Related commands

*dump*, *dump image*, *undump*

### 9.19.6 Default

The option defaults are

- append = no
- balance = no
- buffer = yes for dump styles *atom*, *custom*, *loca*, and *xyz*
- element = "C" for every atom type
- every = whatever it was set to via the *dump* command
- fileper = # of processors
- first = no
- flush = yes
- forces = yes
- format = %d and %g for each integer or floating point value
- image = no
- label = ENTRIES

- mass = no
- maxfiles = -1
- nfile = 1
- pad = 0
- pbc = no
- precision = 1000
- region = none
- scale = yes
- sort = off for dump styles *atom*, *custom*, *cfg*, and *local*
- sort = id for dump styles *dcd*, *xtc*, and *xyz*
- thresh = none
- time = no
- triclinic/general = no
- types = numeric
- units = no
- unwrap = no
- vel = yes
- compression_level = 9 (gz variants)
- compression_level = 0 (zstd variants)
- checksum = yes (zstd variants)

## 9.20 dump molfile command

### 9.20.1 Syntax

```
dump ID group-ID molfile N file format path
```

- ID = user-assigned name for the dump
- group-ID = ID of the group of atoms to be imaged
- molfile = style of dump command (other styles *atom* or *cfg* or *dcd* or *xtc* or *xyz* or *local* or *custom* are discussed on the *dump* doc page)
- N = dump every this many timesteps
- file = name of file to write to
- format = file format to be used
- path = file path with plugins (optional)

### 9.20.2 Examples

```
dump mf1 all molfile 10 melt1.xml hoomd
dump mf2 all molfile 10 melt2-*.pdb pdb .
dump mf3 all molfile 50 melt3.xyz xyz .:/home/akohlmey/vmd/plugins/LINUX/molfile
```

### 9.20.3 Description

Dump a snapshot of atom coordinates and selected additional quantities to one or more files every N timesteps in one of several formats. Only information for atoms in the specified group is dumped. This specific dump style uses molfile plugins that are bundled with the VMD molecular visualization and analysis program.

Unless the filename contains a * character, the output will be written to one single file with the specified format. Otherwise there will be one file per snapshot and the * will be replaced by the time step number when the snapshot is written.

**Note**

Because periodic boundary conditions are enforced only on timesteps when neighbor lists are rebuilt, the coordinates of an atom written to a dump file may be slightly outside the simulation box.

The molfile plugin API has a few restrictions that have to be honored by this dump style: the number of atoms must not change, the atoms must be sorted, outside of the coordinates no change in atom properties (like type, mass, charge) will be recorded.

---

The *format* keyword determines what format is used to write out the dump. For this to work, LAMMPS must be able to find and load a compatible molfile plugin that supports this format. Settings made via the *dump_modify* command can alter per atom properties like element names.

The *path* keyword determines which in directories. This is a "path" like other search paths, i.e. it can contain multiple directories separated by a colon (or semicolon on Windows). This keyword is optional and default to ".", the current directory.

The *unwrap* option of the *dump_modify* command allows coordinates to be written "unwrapped" by the image flags for each atom. Unwrapped means that if the atom has passed through a periodic boundary one or more times, the value is printed for what the coordinate would be if it had not been wrapped back into the periodic box. Note that these coordinates may thus be far outside the box size stored with the snapshot.

---

Dumps are performed on timesteps that are a multiple of N (including timestep 0) and on the last timestep of a minimization if the minimization converges. Note that this means a dump will not be performed on the initial timestep after the dump command is invoked, if the current timestep is not a multiple of N. This behavior can be changed via the *dump_modify first* command, which can be useful if the dump command is invoked after a minimization ended on an arbitrary timestep. N can be changed between runs by using the *dump_modify every* command. The *dump_modify every* command also allows a variable to be used to determine the sequence of timesteps on which dump files are written.

---

### 9.20.4 Restrictions

The *molfile* dump style is part of the MOLFILE package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

Molfile plugins provide a consistent programming interface to read and write file formats commonly used in molecular simulations. The MOLFILE package only provides the interface code, not the plugins. These can be obtained from a VMD installation which has to match the platform that you are using to compile LAMMPS for. By adding plugins to VMD, support for new file formats can be added to LAMMPS (or VMD or other programs that use them) without having to re-compile the application itself. The plugins are installed in the directory: <VMDHOME>/plugins/<VMDARCH>/molfile

**Note**

while the programming interface (API) to the plugins is backward compatible, the binary interface (ABI) has been changing over time, so it is necessary to compile this package with the plugin header files from VMD that match the binary plugins. These header files in the directory: <VMDHOME>/plugins/include For convenience, the package ships with a set of header files that are compatible with VMD 1.9 and 1.9.1 (June 2012)

### 9.20.5 Related commands

*dump*, *dump_modify*, *undump*

### 9.20.6 Default

The default path is ".". All other properties have to be specified.

## 9.21 dump netcdf command

## 9.22 dump netcdf/mpiio command

### 9.22.1 Syntax

```
dump ID group-ID netcdf N file args
dump ID group-ID netcdf/mpiio N file args
```

- ID = user-assigned name for the dump
- group-ID = ID of the group of atoms to be imaged
- *netcdf* or *netcdf/mpiio* = style of dump command (other styles *atom* or *cfg* or *dcd* or *xtc* or *xyz* or *local* or *custom* are discussed on the *dump* doc page)
- N = dump every this many timesteps
- file = name of file to write dump info to
- args = list of atom attributes, same as for *dump_style custom*

### 9.22.2 Examples

```
dump 1 all netcdf 100 traj.nc type x y z vx vy vz
dump_modify 1 append yes at -1 thermo yes
dump 1 all netcdf/mpiio 1000 traj.nc id type x y z
dump 1 all netcdf 1000 traj.*.nc id type x y z
```

### 9.22.3 Description

Dump a snapshot of atom coordinates every N timesteps in AMBER-style NetCDF file format. NetCDF files are binary, portable and self-describing. This dump style will write only one file on the root node. The dump style *netcdf* uses the standard NetCDF library. All data is collected on one processor and then written to the dump file. Dump style *netcdf/mpiio* uses the parallel NetCDF library and MPI-IO to write to the dump file in parallel; it has better performance on a larger number of processors. Note that style *netcdf* outputs all atoms sorted by atom tag while style *netcdf/mpiio* outputs atoms in order of their MPI rank.

NetCDF files can be directly visualized via the following tools:

- Ovito (https://www.ovito.org/). Ovito supports the AMBER convention and all extensions of this dump style.
- VMD (https://www.ks.uiuc.edu/Research/vmd/).

In addition to per-atom data, *thermo* data can be included in the dump file. The data included in the dump file is identical to the data specified by *thermo_style*.

---

### 9.22.4 Restrictions

The *netcdf* and *netcdf/mpiio* dump styles are part of the NETCDF package. They are only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

The *netcdf* and *netcdf/mpiio* dump styles currently cannot dump string properties or properties from variables.

---

### 9.22.5 Related commands

*dump*, *dump_modify*, *undump*

## 9.23 dump vtk command

### 9.23.1 Syntax

```
dump ID group-ID vtk N file args
```

- ID = user-assigned name for the dump
- group-ID = ID of the group of atoms to be dumped
- vtk = style of dump command (other styles such as *atom* or *cfg* or *dcd* or *xtc* or *xyz* or *local* or *custom* are discussed on the *dump* doc page)
- N = dump every this many timesteps
- file = name of file to write dump info to
- args = same as arguments for *dump_style custom*

### 9.23.2 Examples

```
dump dmpvtk all vtk 100 dump*.myforce.vtk id type vx fx
dump dmpvtp flow vtk 100 dump*.%.displace.vtp id type c_myD[1] c_myD[2] c_myD[3] v_ke
```

### 9.23.3 Description

Dump a snapshot of atom quantities to one or more files every *N* timesteps in a format readable by the VTK visualization toolkit or other visualization tools that use it, such as ParaView. The time steps on which dump output is written can also be controlled by a variable; see the *dump_modify every* command for details.

This dump style is similar to *dump_style custom* but uses the VTK library to write data to VTK simple legacy or XML format, depending on the filename extension specified for the dump file. This can be either **.vtk* for the legacy format or **.vtp* and **.vtu*, respectively, for XML format; see the VTK homepage for a detailed description of these formats. Since this naming convention conflicts with the way binary output is usually specified (see below), the *dump_modify binary* command allows setting of a binary option for this dump style explicitly.

Only information for atoms in the specified group is dumped. The *dump_modify thresh and region* commands can also alter what atoms are included; see details below.

As described below, special characters ("*", "%") in the filename determine the kind of output.

**Warning**

Because periodic boundary conditions are enforced only on timesteps when neighbor lists are rebuilt, the coordinates of an atom written to a dump file may be slightly outside the simulation box.

**Warning**

Unless the *dump_modify sort* option is invoked, the lines of atom information written to dump files will be in an indeterminate order for each snapshot. This is even true when running on a single processor, if the *atom_modify sort* option is on, which it is by default. In this case atoms are re-ordered periodically during a simulation, due to spatial sorting. It is also true when running in parallel, because data for a single snapshot is collected from multiple processors, each of which owns a subset of the atoms.

For the *vtk* style, sorting is off by default. See the *dump_modify* page for details.

---

The dimensions of the simulation box are written to a separate file for each snapshot (either in legacy VTK or XML format depending on the format of the main dump file) with the suffix *_boundingBox* appended to the given dump filename.

For an orthogonal simulation box this information is saved as a rectilinear grid (legacy .vtk or .vtr XML format).

Triclinic simulation boxes (non-orthogonal) are saved as hexahedrons in either legacy .vtk or .vtu XML format.

Style *vtk* allows you to specify a list of atom attributes to be written to the dump file for each atom. The list of possible attributes is the same as for the *dump_style custom* command; see its documentation page for a listing and an explanation of each attribute.

**Note**

Since position data is required to write VTK files the atom attributes "x y z" do not have to be specified explicitly; they will be included in the dump file regardless. Also, in contrast to the *custom* style, the specified *vtk* attributes are rearranged to ensure correct ordering of vector components (except for computes and fixes - these have to be given in the right order) and duplicate entries are removed.

The VTK format uses a single snapshot of the system per file, thus a wildcard "*" must be included in the filename, as discussed below. Otherwise the dump files will get overwritten with the new snapshot each time.

Dumps are performed on timesteps that are a multiple of N (including timestep 0) and on the last timestep of a minimization if the minimization converges. Note that this means a dump will not be performed on the initial timestep after the dump command is invoked, if the current timestep is not a multiple of N. This behavior can be changed via the *dump_modify first* command, which can also be useful if the dump command is invoked after a minimization ended on an arbitrary timestep. N can be changed between runs by using the *dump_modify every* command. The *dump_modify every* command also allows a variable to be used to determine the sequence of timesteps on which dump files are written. In this mode a dump on the first timestep of a run will also not be written unless the *dump_modify first* command is used.

Dump filenames can contain two wildcard characters. If a "*" character appears in the filename, then one file per snapshot is written and the "*" character is replaced with the timestep value. For example, tmp.dump*.vtk becomes tmp.dump0.vtk, tmp.dump10000.vtk, tmp.dump20000.vtk, etc. Note that the *dump_modify pad* command can be used to ensure all timestep numbers are the same length (e.g. 00010), which can make it easier to read a series of dump files in order with some post-processing tools.

If a "%" character appears in the filename, then each of P processors writes a portion of the dump file, and the "%" character is replaced with the processor ID from 0 to P-1 preceded by an underscore character. For example, tmp.dump%.vtp becomes tmp.dump_0.vtp, tmp.dump_1.vtp, ... tmp.dump_P-1.vtp, etc. This creates smaller files and can be a fast mode of output on parallel machines that support parallel I/O for output.

By default, P = the number of processors meaning one file per processor, but P can be set to a smaller value via the *nfile* or *fileper* keywords of the *dump_modify* command. These options can be the most efficient way of writing out dump files when running on large numbers of processors.

For the legacy VTK format "%" is ignored and P = 1, i.e., only processor 0 does write files.

Note that using the "*" and "%" characters together can produce a large number of small dump files!

If *dump_modify binary* is used, the dump file (or files, if "*" or "%" is also used) is written in binary format. A binary dump file will be about the same size as a text version, but will typically write out much faster.

### 9.23.4 Restrictions

The *vtk* style does not support writing of gzipped dump files.

The *vtk* dump style is part of the VTK package. It is only enabled if LAMMPS was built with that package. See the *Build package* page for more info.

To use this dump style, you also must link to the VTK library. See the info in lib/vtk/README and ensure the Makefile.lammps file in that directory is appropriate for your machine.

The *vtk* dump style supports neither buffering or custom format strings.

### 9.23.5 Related commands

*dump*, *dump image*, *dump_modify*, *undump*

### 9.23.6 Default

By default, files are written in ASCII format. If the file extension is not one of .vtk, .vtp or .vtu, the legacy VTK file format is used.

CHAPTER

# TEN

# BIBLIOGRAPHY


**(Abascal1)**
Abascal, Sanz, Fernandez, Vega, J Chem Phys, 122, 234511 (2005)

**(Abascal2)**
Abascal, J Chem Phys, 123, 234505 (2005)

**(Ackland)**
Ackland, Jones, Phys Rev B, 73, 054104 (2006).

**(Ackland1)**
Ackland, Condensed Matter (2005).

**(Ackland2)**
Ackland, Mendelev, Srolovitz, Han and Barashev, Journal of Physics: Condensed Matter, 16, S2629 (2004).

**(Addington)**
Addington, Long, Gubbins, J Chem Phys, 149, 084109 (2018).

**(Adhikari et al.)**
Adhikari, R., Stratford, K., Cates, M. E., and Wagner, A. J., Fluctuating lattice Boltzmann, Europhys. Lett. 71 (2005) 473-479.

**(Afshar)**
Afshar, F. Schmid, A. Pishevar, S. Worley, Comput Phys Comm, 184, 1119-1128 (2013).

**(Agnolin and Roux 2007)**
Agnolin, I. & Roux, J-N. (2007). Internal states of model isotropic granular packings. I. Assembling process, geometry, and contact networks. Phys. Rev. E, 76, 061302.

**(Ahrens-Iwers2022)**
Ahrens-Iwers *et al.*, J. Chem. Phys. 157, 084801 (2022).

**(Ahrens-Iwers)**
Ahrens-Iwers and Meissner, J. Chem. Phys. 155, 104104 (2021).

**(Aktulga)**
Aktulga, Fogarty, Pandit, Grama, Parallel Computing, 38, 245-259 (2012).

**(Albe)**
J. Nord, K. Albe, P. Erhart, and K. Nordlund, J. Phys.: Condens. Matter, 15, 5649(2003).

**(Albe1)**
K. Albe, K. Nordlund, J. Nord, and A. Kuronen, Phys. Rev. B, 66, 035205 (2002).

**(Allen)**
Allen and Germano, Mol Phys 104, 3225-3235 (2006).

**(AllenTildesley)**
Allen and Tildesley, Computer Simulation of Liquids, Oxford University Press (1987)

**(Allinger)**
Allinger, Yuh, Lii, JACS, 111(23), 8551-8566 (1989),

**(Andersen)**
H. Andersen, J of Comp Phys, 52, 24-34 (1983).

**(Appshaw)**
Appshaw, Seddon, Hanna, Soft. Matter,18, 1747(2022).

**(Avendano)**
C. Avendano, T. Lafitte, A. Galindo, C. S. Adjiman, G. Jackson, E. Muller, J Phys Chem B, 115, 11154 (2011).

**(Axilrod)**
Axilrod and Teller, J Chem Phys, 11, 299 (1943); Muto, Nippon Sugaku-Buturigakkwaishi 17, 629 (1943).

**(Babadi)**
Babadi, Ejtehadi, Everaers, J Comp Phys, 219, 770-779 (2006).

**(Babadi2)**
Babadi and Ejtehadi, EPL, 77 (2007) 23002.

**(Baczewski)**
A.D. Baczewski and S.D. Bond, J. Chem. Phys. 139, 044107 (2013).

**(Bal)**
K. M Bal and E. C. Neyts, J. Chem. Phys. 141, 204104 (2014).

**(Ball)**
Ball and Melrose, Physica A, 247, 444-472 (1997).

**(Ballenegger)**
Ballenegger, Arnold, Cerda, J Chem Phys, 131, 094107 (2009).

**(Barrat)**
Barrat and Rodney, J. Stat. Phys, 144, 679 (2011).

**(Barrett)**
Barrett, Tschopp, El Kadiri, Scripta Mat. 66, p.666 (2012).

**(Barros)**
Barros, Sinkovits, Luijten, J. Chem. Phys, 140, 064903 (2014)

**(Bartok)**
Bartok, Payne, Risi, Csanyi, Phys Rev Lett, 104, 136403 (2010).

**(Bartok2010)**
Bartok, Payne, Risi, Csanyi, Phys Rev Lett, 104, 136403 (2010).

**(Bartok2013)**
Bartok, Kondor, Csanyi, Phys Rev B, 87, 184115 (2013).

**(Bartok_PhD)**
A Bartok-Partay, PhD Thesis, University of Cambridge, (2010).

**(Baskes)**
Baskes, Phys Rev B, 46, 2727-2742 (1992).

**(Baskes2)**
Baskes, Phys Rev B, 75, 094113 (2007).

**(Beck)**
Beck, Molecular Physics, 14, 311 (1968).

**(Becton)**
Becton, Averett, Wang, Biomech. Model. Mechanobiology, 18, 425-433(2019).

**(Behler and Parrinello 2007)**
Behler, J.; Parrinello, M. Phys. Rev. Lett. 2007, 98 (14), 146401.

**(Bennet)**
Bennet, J Comput Phys, 22, 245 (1976)

**(Berardi)**
Berardi, Fava, Zannoni, Chem Phys Lett, 297, 8-14 (1998). Berardi, Muccioli, Zannoni, J Chem Phys, 128, 024905 (2008).

**(Berendsen)**
Berendsen, Postma, van Gunsteren, DiNola, Haak, J Chem Phys, 81, 3684 (1984).

**(Berendsen2)**
Berendsen, Grigera, Straatsma, J Phys Chem, 91, 6269-6271 (1987).

**(Bessarab)**
Bessarab, Uzdin, Jonsson, Comp Phys Comm, 196, 335-347 (2015).

**(Beutler)**
Beutler, Mark, van Schaik, Gerber, van Gunsteren, Chem Phys Lett, 222, 529 (1994).

**(Bird)**
G. A. Bird, "Molecular Gas Dynamics and the Direct Simulation of Gas Flows" (1994).

**(Bitzek)**
Bitzek, Koskinen, Gahler, Moseler, Gumbsch, Phys Rev Lett, 97, 170201 (2006).

**(Bolintineanu1)**
Bolintineanu, Lechman, Plimpton, Grest, Phys Rev E, 86, 066703 (2012).

**(Bolintineanu2)**
Bolintineanu, Grest, Lechman, Pierce, Plimpton, Schunk, Comp Particle Mechanics, 1, 321-356 (2014).

**(Bomont)**
Bomont, Bretonnet, J. Chem. Phys. 124, 054504 (2006).

**(Bond)**
Bond and Leimkuhler, SIAM J Sci Comput, 30, p 134 (2007).

**(Boone)**
Boone, Babaei, Wilmer, J Chem Theory Comput, 15, 5579–5587 (2019).

**(BoreschKarplus)**
Boresch and Karplus, J Phys Chem A, 103, 103 (1999).

**(Botu1)**
V. Botu and R. Ramprasad, Int. J. Quant. Chem., 115(16), 1074 (2015).

**(Botu2)**
V. Botu and R. Ramprasad, Phys. Rev. B, 92(9), 094306 (2015).

**(Botu3)**
V. Botu, R. Batra, J. Chapman and R. Ramprasad, https://arxiv.org/abs/1610.02098 (2016).

**(Branicio2009)**
Branicio, Rino, Gan and Tsuzuki, J. Phys Condensed Matter 21 (2009) 095002

**(Brennan)**
Brennan, J Chem Phys Lett, 5, 2144-2149 (2014).

**(Brenner)**
Brenner, Shenderova, Harrison, Stuart, Ni, Sinnott, J Physics: Condensed Matter, 14, 783-802 (2002).

**(Brilliantov)**
Brilliantov, Spahn, Hertzsch, Poschel, Phys Rev E, 53, p 5382-5392 (1996).

**(Brooks)**
Brooks, Brooks, MacKerell Jr., J Comput Chem, 30, 1545 (2009).

**(Brooks)**
Brooks, et al, J Comput Chem, 30, 1545 (2009).

**(Brown)**
Brown et al. International Tables for Crystallography Volume C: Mathematical and Chemical Tables, 554-95 (2004).

**(Buck)**
Buck, Bouguet-Bonnet, Pastor, MacKerell Jr., Biophys J, 90, L36 (2006).

**(Bulacu)**
Bulacu, Goga, Zhao, Rossi, Monticelli, Periole, Tieleman, Marrink, J Chem Theory Comput, 9, 3282-3292

**(Bussi)**

G. Bussi, T. Zykova-Timan, M. Parrinello, J Chem Phys, 130, 074101 (2009).

**(Bussi1)**
Bussi, Donadio and Parrinello, J. Chem. Phys. 126, 014101(2007)

**(Bussi2)**
Bussi and Parrinello, Phys. Rev. E 75, 056707 (2007)

**(COMB_1)**
J. Yu, S. B. Sinnott, S. R. Phillpot, Phys Rev B, 75, 085311 (2007),

**(COMB_2)**
T.-R. Shan, B. D. Devine, T. W. Kemper, S. B. Sinnott, and S. R. Phillpot, Phys. Rev. B 81, 125328 (2010)

**(COMB3)**
T. Liang, T.-R. Shan, Y.-T. Cheng, B. D. Devine, M. Noordhoek, Y. Li, Z. Lu, S. R. Phillpot, and S. B. Sinnott, Mat. Sci. & Eng: R 74, 255-279 (2013).

**(Calhoun)**
A. Calhoun, M. Pavese, G. Voth, Chem Phys Letters, 262, 415 (1996).

**(Campana)**
C. Campana and M. H. Muser, Phys. Rev. B [74], 075420 (2006).

**(Cao1)**
J. Cao and B. Berne, J Chem Phys, 99, 2902 (1993).

**(Cao2)**
J. Cao and G. Voth, J Chem Phys, 100, 5093 (1994).

**(Caro)**
A Caro, DA Crowson, M Caro; Phys Rev Lett, 95, 075702 (2005)

**(CasP)**
CasP webpage: https://lief.if.ufrgs.br/pub/CasP/

**(Cawkwell2012)**
A. M. N. Niklasson, M. J. Cawkwell, Phys. Rev. B, 86 (17), 174308 (2012).

**(Cercignani)**
C. Cercignani and M. Lampis. Trans. Theory Stat. Phys. 1, 2, 101 (1971).

**(Cerda)**
Cerda, Ballenegger, Lenz, Holm, J Chem Phys 129, 234104 (2008)

**(Ceriotti)**
M. Ceriotti, M. Parrinello, T. Markland, D. Manolopoulos, J. Chem. Phys. 133, 124104 (2010).

**(Ceriotti1)**
Ceriotti, Bussi and Parrinello, J Chem Theory Comput 6, 1170-80 (2010)

**(Ceriotti2)**
Ceriotti, Bussi and Parrinello, Phys Rev Lett 103, 030603 (2009)

**(Cerutti)**
Cerutti, Duke, Darden, Lybrand, Journal of Chemical Theory and Computation 5, 2322 (2009)

**(Chen)**
J Chen, D Tzou and J Beraun, Int. J. Heat Mass Transfer, 49, 307-316 (2006).

**(Chenoweth_2008)**
Chenoweth, van Duin and Goddard, Journal of Physical Chemistry A, 112, 1040-1053 (2008).

**(Clarke)**
Clarke and Smith, J Chem Phys, 84, 2290 (1986).

**(Clavier)**
G. Clavier, N. Desbiens, E. Bourasseau, V. Lachet, N. Brusselle-Dupend and B. Rousseau, Mol Sim, 43, 1413 (2017).

**(Clemmer)**
Clemmer and Robbins, Phys. Rev. Lett. (2022).

**(Clemmer1)**
Clemmer, Monti, Lechman, Soft Matter, 20, 1702 (2024).

**(Clemmer2)**
Clemmer, Pierce, O'Connor, Nevins, Jones, Lechman, Tencer, Appl. Math. Model., 130, 310-326 (2024).

**(Coleman)**
Coleman, Spearot, Capolungo, MSMSE, 21, 055020 (2013).

**(Colliex)**
Colliex et al. International Tables for Crystallography Volume C: Mathematical and Chemical Tables, 249-429 (2004).

**(Cooke)**
"Cooke, Kremer and Deserno, Phys. Rev. E, 72, 011506 (2005)"

**(Cornell)**
Cornell, Cieplak, Bayly, Gould, Merz, Ferguson, Spellmeyer, Fox, Caldwell, Kollman, JACS 117, 5179-5197 (1995).

**(Cundall, 1987)**
Cundall, P. A. Distinct Element Models of Rock and Soil

**(Curk1)**
T. Curk, J. Yuan, and E. Luijten, JCP 156 (2022).

**(Curk2)**

T. Curk and E. Luijten, PRL 126 (2021)

**(Cusentino)**
Cusentino, Wood, Thompson, J Phys Chem A, 124, 5456, (2020)

**(Daivis and Todd)**
Daivis and Todd, J Chem Phys, 124, 194103 (2006).

**(Daivis and Todd)**
Daivis and Todd, Nonequilibrium Molecular Dynamics (book), Cambridge University Press, https://doi.org/10.1017/9781139017848, (2017).

**(Dammak)**
Dammak, Chalopin, Laroche, Hayoun, and Greffet, Phys Rev Lett, 103, 190601 (2009).

**(Dandurand)**
Dandurand, Vandierendonck, de Supinski, 39th IEEE IPDPS, June 3-7, (2025).

**(Darden)**
Darden, York, Pedersen, J Chem Phys, 98, 10089 (1993).

**(Davidchack)**
R.L Davidchack, T.E. Ouldridge, and M.V. Tretyakov. J. Chem. Phys. 142, 144114 (2015).

**(Daw1)**
Daw, Baskes, Phys Rev Lett, 50, 1285 (1983). Daw, Baskes, Phys Rev B, 29, 6443 (1984).

**(Daw2)**
M. S. Daw, and M. I. Baskes, Phys. Rev. B, 29, 6443 (1984).

**(de Buyl)**
de Buyl, Colberg and Hofling, Comp. Phys. Comm. 185(6), 1546-1553 (2014) -

**(Deissenbeck)**
Deissenbeck *et al.*, Phys. Rev. Letters 126, 136803 (2021).

**(de Koning)**
de Koning and Antonelli, Phys Rev E, 53, 465 (1996).

**(DeVane)**
Shinoda, DeVane, Klein, Soft Matter, 4, 2453-2462 (2008).

**(Deserno)**
Deserno and Holm, J Chem Phys, 109, 7694 (1998).

**(Destree)**
M. Destree, F. Laupretre, A. Lyulin, and J.-P. Ryckaert, J Chem Phys, 112, 9632 (2000).

**(Dickel)**
Dickel, Francis, and Barrett, Computational Materials Science 171 (2020): 109157.

**(Dietz)**
Dietz and Hoy, J. Chem Phys, 156, 014103 (2022).

**(Dobson)**
Dobson, J Chem Phys, 141, 184103 (2014).

**(Drautz)**
Drautz, Phys Rev B, 99, 014104 (2019).

**(Duffy)**
D M Duffy and A M Rutherford, J. Phys.: Condens. Matter, 19, 016207-016218 (2007).

**(Dufils)**
Dufils *et al.*, Phys. Rev. Letters 123, 195501 (2019).

**(Dullweber)**
Dullweber, Leimkuhler and McLachlan, J Chem Phys, 107, 5840 (1997).

**(Dunn1)**
Dunn and Noid, J Chem Phys, 143, 243148 (2015).

**(Dunn2)**
Dunn, Lebold, DeLyser, Rudzinski, and Noid, J. Phys. Chem. B, 122, 3363 (2018).

**(Dunweg)**
Dunweg and Paul, Int J of Modern Physics C, 2, 817-27 (1991).

**(EcheverriRestrepo)**
Echeverri Restrepo, Andric, Comput Mater Sci, 218, 111978 (2023).

**(EDIP)**
J F Justo et al, Phys Rev B 58, 2539 (1998).

**(Eike)**
Eike and Maginn, Journal of Chemical Physics, 124, 164503 (2006).

**(Elliott)**
Elliott, Tadmor and Bernstein, https://openkim.org/kim-api/ (2011) doi: https://doi.org/10.25950/FF8F563A

**(Ellis)**
Ellis, Fiedler, Popoola, Modine, Stephens, Thompson, Cangi, Rajamanickam, Phys Rev B, 104, 035120, (2021)

**(Emmrich)**
Emmrich, Weckner, Commun. Math. Sci., 5, 851-864 (2007),

**(Elstner)**
M. Elstner, D. Poresag, G. Jungnickel, J. Elsner, M. Haugk, T. Frauenheim, S. Suhai, and G. Seifert, Phys. Rev. B, 58, 7260 (1998).

**(Erdmann)**
U. Erdmann , W. Ebeling, L. Schimansky-Geier, and F. Schweitzer, Eur. Phys. J. B 15, 105-113, 2000.

**(Eshelby)**
J.D. Eshelby, Philos. Trans. Royal Soc. London A, Math. Phys. Sci., Vol. 244, No. 877 (1951) pp. 87-112; J. Elasticity, Vol. 5, Nos. 3-4 (1975) pp. 321-335]

**(Espanol and Revenga)**
Espanol, Revenga, Physical Review E, 67, 026705 (2003).

**(Espanol1997)**
Espanol, Europhys Lett, 40(6): 631-636 (1997). DOI:10.1209/epl/i1997-00515-8

**(Evans and Morriss)**
Evans and Morriss, Phys Rev A, 30, 1528 (1984).

**(Evans)**
Evans and Morriss, Phys. Rev. Lett. 56, 2172 (1986).

**(Everaers)**
Everaers and Ejtehadi, Phys Rev E, 67, 041710 (2003).

**(Faken)**
Faken, Jonsson, Comput Mater Sci, 2, 279 (1994).

**(Falk)**
Falk and Langer PRE, 57, 7192 (1998).

**(Fath)**
Fath, Hochbruck, Singh, J Comp Phys, 333, 180-198 (2017).

**(Feng1)**

26. Feng, …, and W. Ouyang, J. Phys. Chem. C. 127(18), 8704-8713 (2023).

**(Feng2)**

26. Feng, …, and W. Ouyang, Langmuir 39(50), 18198-18207 (2023).

**(Fennell)**
C. J. Fennell, J. D. Gezelter, J Chem Phys, 124, 234104 (2006).

**(Feynman)**
R. Feynman and A. Hibbs, Chapter 7, Quantum Mechanics and Path Integrals, McGraw-Hill, New York (1965).

**(Fichthorn)**
Fichthorn, Balankura, Qi, CrystEngComm, 18(29), 5410-5417 (2016).

**(Fily)**
Y. Fily and M.C. Marchetti, Phys. Rev. Lett. 108, 235702, 2012. Default

**(Fincham)**
Fincham, Mackrodt and Mitchell, J Phys Condensed Matter, 6, 393-404 (1994).

**(Finnis1)**
Finnis, Sinclair, Philosophical Magazine A, 50, 45 (1984).

**(Finnis2)**
M. W. Finnis, A. T. Paxton, M. Methfessel, and M. van Schilfgarde, Phys. Rev. Lett., 81, 5149 (1998).

**(Fiorin)**
Fiorin, Klein, Henin, Mol. Phys., DOI:10.1080/00268976.2013.813594

**(Fox)**
Fox, O'Keefe, Tabbernor, Acta Crystallogr. A, 45, 786-93 (1989).

**(Fraige)**
F. Y. Fraige, P. A. Langston, A. J. Matchett, J. Dodds, Particuology, 6, 455 (2008).

**(Freitas)**
Freitas, Asta, and de Koning, Computational Materials Science, 112, 333 (2016).

**(Frenkel)**
Frenkel and Smit, Understanding Molecular Simulation, Academic Press, London, 2002.

**(Fu)**
Fu, Peng, Yuan, Kfoury, Young, Comput. Phys. Commun, 210, 193-203(2017).

**(Gao)**
Gao and Weber, Nuclear Instruments and Methods in Physics Research B 191 (2012) 504.

**(Gingrich)**
Gingrich, *MSc thesis* <https://gingrich.chem.northwestern.edu/papers/ThesiswCorrections.pdf>` (2010).

**(Gissinger2017)**
Gissinger, Jensen and Wise, Polymer, 128, 211-217 (2017).

**(Gissinger2020)**
Gissinger, Jensen and Wise, Macromolecules, 53, 22, 9953-9961 (2020).

**(Gissinger)**
Jacob R. Gissinger, Scott R. Zavada, Joseph G. Smith, Josh Kemppainen, Ivan Gallegos, Gregory M. Odegard, Emilie J. Siochi, and Kristopher E. Wise, Carbon, 202, 336-347 (2023).

**(Gissinger2024)**

J. R. Gissinger, I. Nikiforov, Y. Afshar, B. Waters, M. Choi, D. S. Karls, A. Stukowski, W. Im, H. Heinz, A. Kohlmeyer, and E. B. Tadmor, J Phys Chem B, 128, 3282-3297 (2024).

**(Gloor)**
Gloor, J Chem Phys, 123, 134703 (2005)

**(Glosli)**
Glosli, unpublished, 2005. Streitz, Glosli, Patel, Chan, Yates, de Supinski, Sexton and Gunnels, Journal of Physics: Conference Series, 46, 254 (2006).

**(Goff)**
Goff, Zhang, Negre, Rohskopf, Niklasson, Journal of Chemical Theory and Computation 19, no. 13 (2023).

**(Goldman1)**
Goldman, Reed and Fried, J. Chem. Phys. 131, 204103 (2009)

**(Goldman2)**
Goldman, Srinivasan, Hamel, Fried, Gaus, and Elstner, J. Phys. Chem. C, 117, 7885 (2013).

**(Grime)**
Grime and Voth, to appear in J Chem Theory & Computation (2014).

**(Grimme)**
Grimme, J Comput Chem, 27(15), 1787-1799 (2006).

**(Gronbech-Jensen1)**
Gronbech Jensen and Gronbech-Jensen, Mol Phys, 117, 2511 (2019)

**(Gronbech-Jensen2)**
Gronbech-Jensen and Farago, Mol Phys, 111, 983 (2013)

**(Gronbech-Jensen3)**
Hayre, and Farago, Comp Phys Comm, 185, 524 (2014)

**(Groot)**
Groot and Warren, J Chem Phys, 107: 4423-4435 (1997). DOI:10.1063/1.474784

**(Guenole)**
Guenole, Noehring, Vaid, Houlle, Xie, Prakash, Bitzek, Comput Mater Sci, 175, 109584 (2020).

**(Gullet)**
Gullet, Wagner, Slepoy, SANDIA Report 2003-8782 (2003). DOI:10.2172/918395

**(Guo)**
Guo and Thirumalai, Journal of Molecular Biology, 263, 323-43 (1996).

**(Gupta)**
Gupta ,Phys Rev. B, 23, 6265-6270 (1981).

**(Hardy)**
David Hardy thesis: Multilevel Summation for the Fast Evaluation of Forces for the Simulation of Biomolecules, University of Illinois at Urbana-Champaign, (2006).

**(Hardy2)**
Hardy, Stone, Schulten, Parallel Computing, 35, 164-177 (2009).

**(Hecht)**
Hecht, Harting, Ihle, Herrmann, Phys Rev E, 72, 011408 (2005).

**(Henkelman1)**
Henkelman and Jonsson, J Chem Phys, 113, 9978-9985 (2000).

**(Henkelman2)**
Henkelman, Uberuaga, Jonsson, J Chem Phys, 113, 9901-9904 (2000).

**(Henkes)**
Henkes, S, Fily, Y., and Marchetti, M. C. Phys. Rev. E, 84, 040301(R), 2011.

**(Henrich)**
O. Henrich, Y. A. Gutierrez-Fosado, T. Curk, T. E. Ouldridge, Eur. Phys. J. E 41, 57 (2018).

**(Herman)**
M. F. Herman, E. J. Bruskin, B. J. Berne, J Chem Phys, 76, 5150 (1982).

**(Hess)**
Hess, B. The Journal of Chemical Physics 2002, 116 (1), 209-217.

**(Heyes)**
Heyes, Phys Rev B, 49, 755 (1994).

**(Hijazi)**
M. Hijazi, D. M. Wilkins, M. Ceriotti, J. Chem. Phys. 148, 184109 (2018)

**(Hockney)**
Hockney and Eastwood, Computer Simulation Using Particles, Adam Hilger, NY (1989).

**(Holian)**
Holian and Ravelo, Phys Rev B, 51, 11275 (1995).

**(Hone)**
T. Hone, P. Rossky, G. Voth, J Chem Phys, 124, 154103 (2006).

**(Hoover)**
Hoover, Phys Rev A, 31, 1695 (1985).

**(Huang)**
Huang, Zhang, Yuan, Gao, Zhang, Nano Lett. 13, 4546(2013).

**(Huang2014)**
X. Huang, "Exploring critical-state behavior using DEM", Doctoral dissertation, Imperial College. (2014). https://doi.org/10.25560/25316

**(Hu)**
Hu, and Adams J. Comp. Physics, 213, 844-861 (2006).

**(Hu)**
Hu, J. Chem. Theory Comput. 10, 5254 (2014).

**(Hummer)**
Hummer, Gronbech-Jensen, Neumann, J Chem Phys, 109, 2791 (1998)

**(Hunt)**
Hunt, Mol Simul, 42, 347 (2016).

**(Ikeshoji)**
Ikeshoji and Hafskjold, Molecular Physics, 81, 251-261 (1994).

**(Ikeshoji2)**
Ikeshoji, Hafskjold, Furuholt, Mol Sim, 29, 101-109, (2003).

**(Ilie)**
Ilie, Briels, den Otter, Journal of Chemical Physics, 142, 114103 (2015).

**(In 't Veld)**
In 't Veld, Ismail, Grest, J Chem Phys (accepted) (2007).

**(IPI)**
*https://ipi-code.org/ <https://ipi-code.org/>*

**(IPI-CPC)**
Ceriotti, More and Manolopoulos, Comp Phys Comm, 185, 1019-1026 (2014).

**(Isele-Holder)**
Isele-Holder, Mitchell, Ismail, J Chem Phys, 137, 174107 (2012).

**(Isele-Holder2)**
Isele-Holder, Mitchell, Hammond, Kohlmeyer, Ismail, J Chem Theory Comput 9, 5412 (2013).

**(Ismail)**
Ismail, Tsige, In 't Veld, Grest, Molecular Physics (accepted) (2007).

**(Ivanov)**
Ivanov, Uzdin, Jonsson. arXiv preprint arXiv:1904.02669, (2019).

**(Izrailev)**
Izrailev, Stepaniants, Isralewitz, Kosztin, Lu, Molnar, Wriggers, Schulten. Computational Molecular Dynamics: Challenges, Methods, Ideas, volume 4 of Lecture Notes in Computational Science and

**(Izvekov)**
Izvekov, Voth, J Chem Phys 123, 134105 (2005).

**(Jadhao)**
Jadhao, Solis, Olvera de la Cruz, J Chem Phys, 138, 054119 (2013)

**(Janssens)**
Janssens, Olmsted, Holm, Foiles, Plimpton, Derlet, Nature Materials, 5, 124-127 (2006).

**(Jaramillo-Botero)**
Jaramillo-Botero, Su, Qi, Goddard, Large-scale, Long-term Non-adiabatic Electron Molecular Dynamics for Describing Material Properties and Phenomena in Extreme Environments, J Comp

**(Jarzynski)**
Jarzynski, Phys. Rev. Lett. 78, 2690 (1997)

**(Jiang)**
Jiang, Hardy, Phillips, MacKerell, Schulten, and Roux, J Phys Chem Lett, 2, 87-92 (2011).

**(Jiang1)**
Jiang, Hardy, Phillips, MacKerell, Schulten, and Roux, J Phys Chem Lett, 2, 87-92 (2011).

**(Jiang2)**
J.-W. Jiang, Nanotechnology 26, 315706 (2015).

**(Jiang3)**
J.-W. Jiang, Acta Mech. Solida. Sin 32, 17 (2019).

**(Johnson et al, 1971)**
Johnson, K. L., Kendall, K., & Roberts, A. D. (1971). Surface energy and the contact of elastic solids. Proc. R. Soc. Lond. A, 324(1558), 301-313.

**(Jones)**
Jones, RE; Templeton, JA; Wagner, GJ; Olmsted, D; Modine, JA, "Electron transport enhanced molecular dynamics for metals and semi-metals." International Journal for Numerical Methods in Engineering (2010), 83:940.

**(Jonsson)**
Jonsson, Mills and Jacobsen, in Classical and Quantum Dynamics in Condensed Phase Simulations, edited by Berne, Ciccotti, and Coker World Scientific, Singapore, 1998, p 385.

**(Jorgensen)**
Jorgensen, Chandrasekhar, Madura, Impey, Klein, J Chem Phys, 79, 926 (1983).

**(Jusufi)**
Jusufi, Hynninen, and Panagiotopoulos, J Phys Chem B, 112, 13783 (2008).

**(Kamberaj)**
Kamberaj, Low, Neal, J Chem Phys, 122, 224114 (2005).

**(Katsura)**
H. Katsura, N. Nagaosa, A.V. Balatsky. Phys. Rev. Lett., 95(5), 057205. (2005)

**(Kelchner)**
Kelchner, Plimpton, Hamilton, Phys Rev B, 58, 11085 (1998).

**(Khrapak)**
Khrapak, Chaudhuri, and Morfill, J Chem Phys, 134, 054120 (2011).

**(Kim)**
Kim, Keyes, Straub, J Chem. Phys, 132, 224107 (2010).

**(Klapp)**
Klapp, Schoen, J Chem Phys, 117, 8050 (2002).

**(Kolafa)**
Kolafa and Perram, Molecular Simulation, 9, 351 (1992).

**(Kolmogorov)**
A. N. Kolmogorov, V. H. Crespi, Phys. Rev. B 71, 235415 (2005).

**(Kong)**
L.T. Kong, G. Bartels, C. Campana, C. Denniston, and Martin H. Muser, Computer Physics Communications [180](6):1004-1010 (2009).

**(Kong2011)**
L.T. Kong, Computer Physics Communications [182](10):2201-2207, (2011).

**(Kremer)**
Kremer, Grest, J Chem Phys, 92, 5057 (1990).

**(Kuhn and Bagi, 2005)**
Kuhn, M. R., & Bagi, K. (2004). Contact rolling and deformation in granular media. International journal of solids and structures, 41(21), 5793-5820.

**(Kumagai)**
T. Kumagai, S. Izumi, S. Hara, S. Sakai, Comp. Mat. Science, 39, 457 (2007).

**(Kumar)**
Kumar and Higdon, Phys Rev E, 82, 051401 (2010).

**(Kumar)**
Kumar and Skinner, J. Phys. Chem. B, 112, 8311 (2008)

**(Lafourcade)**
Lafourcade, Maillet, Denoual, Duval, Allera, Goryaeva, and Marinica, Comp. Mat. Science, 230, 112534 (2023)

**(Lamoureux and Roux)**
G. Lamoureux, B. Roux, J. Chem. Phys 119, 3025 (2003)

**(Lamoureux)**
Lamoureux and Roux, J Chem Phys, 119, 3025-3039 (2003).

**(Landsgesell)**

J. Landsgesell, P. Hebbeker, O. Rud, R. Lunkad, P. Kosovan, and C. Holm, Macromolecules 53, 3007-3020 (2020).

**(Larentzos1)**
J.P. Larentzos, J.K. Brennan, J.D. Moore, M. Lisal and W.D. Mattson, Comput. Phys. Commun., 185, 1987-1998 (2014).

**(Larentzos2)**
J.P. Larentzos, J.K. Brennan, J.D. Moore, and W.D. Mattson, ARL-TR-6863, U.S. Army Research Laboratory, Aberdeen Proving Ground, MD (2014).

**(Larsen)**
Larsen, Schmidt, Schiotz, Modelling Simul Mater Sci Eng, 24, 055007 (2016).

**(Lebedeva1)**
I. V. Lebedeva, A. A. Knizhnik, A. M. Popov, Y. E. Lozovik, B. V. Potapkin, Phys. Rev. B, 84, 245437 (2011)

**(Lebedeva2)**

I. V. Lebedeva, A. A. Knizhnik, A. M. Popov, Y. E. Lozovik, B. V. Potapkin, Physica E: 44, 949-954 (2012)

**(Lechman)**
Lechman, et al, in preparation (2010).

**(Lee)**
Lee, Baskes, Phys. Rev. B, 62, 8564-8567 (2000).

**(Lee2)**
Lee, Baskes, Kim, Cho. Phys. Rev. B, 64, 184102 (2001).

**(Lee2020)**
C.W. Lee, et al. (2020) Physical Review B, 102(2), 024107.

**(Lenart)**
Lenart , Jusufi, and Panagiotopoulos, J Chem Phys, 126, 044509 (2007).

**(Lenosky)**
Lenosky, Sadigh, Alonso, Bulatov, de la Rubia, Kim, Voter, Kress, Modelling Simulation Materials Science Engineering, 8, 825 (2000).

**(Leven1)**
I. Leven, I. Azuri, L. Kronik and O. Hod, J. Chem. Phys. 140, 104106 (2014).

**(Leven2)**
I. Leven et al, J. Chem.Theory Comput. 12, 2896-905 (2016).

**(Li2013_POF)**
Li, Hu, Wang, Ma, Zhou, Phys Fluids, 25: 072103 (2013). DOI:10.1063/1.4812366.

**(Li2014_JCP)**
Li, Tang, Lei, Caswell, Karniadakis, J Comput Phys, 265: 113-127 (2014). DOI:10.1016/j.jcp.2014.02.003.

**(Li2015_CC)**
Li, Tang, Li, Karniadakis, Chem Commun, 51: 11038-11040 (2015). DOI:10.1039/C5CC01684C.

**(Li2015_JCP)**
Li, Yazdani, Tartakovsky, Karniadakis, J Chem Phys, 143: 014101 (2015). DOI:10.1063/1.4923254.

**(Liang)**
Liang, Phillpot, Sinnott Phys. Rev. B79 245110, (2009), Erratum: Phys. Rev. B85 199903(E), (2012)

**(Lisal)**
M. Lisal, J.K. Brennan, J. Bonet Avalos, J. Chem. Phys., 135, 204105 (2011).

**(Liu1)**
L. Liu, Y. Liu, S. V. Zybin, H. Sun and W. A. Goddard, Journal of Physical Chemistry A, 115, 11016-11022 (2011).

**(Liu2)**
Liu, Bryantsev, Diallo, Goddard III, J. Am. Chem. Soc 131 (8) 2798 (2009)

**(Los and Fasolino)**
J. H. Los and A. Fasolino, Phys. Rev. B 68, 024107 (2003).

**(Los2017)**
J. H. Los et al. "Extended Tersoff potential for boron nitride: Energetics and elastic properties of pristine and defective h-BN", Phys. Rev. B 96 (184108), 2017.

**(Luding, 2008)**
Luding, S. (2008). Cohesive, frictional powders: contact models for tension. Granular matter, 10(4), 235.

**(Lysogorskiy)**
Lysogorskiy, van der Oord, Bochkarev, Menon, Rinaldi, Hammerschmidt, Mrovec, Thompson, Csanyi, Ortner, Drautz, npj Comp Mat, 7, 97 (2021).

**(Lysogorskiy21)**
Lysogorskiy, van der Oord, Bochkarev, Menon, Rinaldi, Hammerschmidt, Mrovec, Thompson, Csanyi, Ortner, Drautz, npj Comp Mat, 7, 97 (2021).

**(Lysogorskiy23)**
Lysogorskiy, Bochkarev, Mrovec, Drautz, Phys Rev Mater, 7, 043801 (2023) / arXiv:2212.08716 (2022).

**(Maaravi)**
T. Maaravi et al, J. Phys. Chem. C 121, 22826-22835 (2017).

**(MacKerell)**
MacKerell, Bashford, Bellott, Dunbrack, Evanseck, Field, Fischer, Gao, Guo, Ha, et al, J Phys Chem B, 102, 3586 (1998).

**(Mackay and Denniston)**
Mackay, F. E., and Denniston, C., Coupling MD particles to a lattice-Boltzmann fluid through the use of conservative forces, J. Comput. Phys. 237 (2013) 289-298.

**(Mackay et al.)**
Mackay, F. E., Ollila, S.T.T., and Denniston, C., Hydrodynamic Forces Implemented into LAMMPS through a lattice-Boltzmann fluid, Computer Physics Communications 184 (2013) 2021-2031.

**(Magda)**
Magda, Tirrell, Davis, J Chem Phys, 83, 1888-1901 (1985); erratum in JCP 84, 2901 (1986).

**(Maginn)**
Kelkar, Rafferty, Maginn, Siepmann, Fluid Phase Equilibria, 260, 218-231 (2007).

**(Mahoney)**
Mahoney, Jorgensen, J Chem Phys 112, 8910 (2000)

**(Malolepsza)**
Malolepsza, Secor, Keyes, J Phys Chem B 119 (42), 13379-13384 (2015).

**(Mandadapu)**
Mandadapu, KK; Templeton, JA; Lee, JW, "Polarization as a field variable from molecular dynamics simulations." Journal of Chemical Physics (2013), 139:054115. Please refer to the standard finite element (FE) texts, e.g. T.J.R Hughes " The finite element method ", Dover 2003, for the basics of FE simulation.

**(Mandelli_1)**

D. Mandelli, W. Ouyang, M. Urbakh, and O. Hod, ACS Nano 13(7), 7603-7609 (2019).

**(Maras)**
Maras, Trushin, Stukowski, Ala-Nissila, Jonsson, Comp Phys Comm, 205, 13-21 (2016).

**(Marrink)**
Marrink, de Vries, Mark, J Phys Chem B, 108, 750-760 (2004).

**(Marshall, 2009)**
Marshall, J. S. (2009). Discrete-element modeling of particulate aerosol flows. Journal of Computational Physics, 228(5), 1541-1561.

**(Martyna1992)**
Martyna, Klein, Tuckerman, J Chem Phys, 97, 2635 (1992); Martyna, Tuckerman, Tobias, Klein, Mol Phys, 87, 1117.

**(Martyna1994)**
Martyna, Tobias and Klein, J Chem Phys, 101, 4177 (1994).

**(Martyna2)**
G. Martyna, A. Hughes, M. Tuckerman, J. Chem. Phys. 110, 3275 (1999).

**(Mason)**
J. K. Mason, Acta Cryst A65, 259 (2009).

**(Mattice)**
Mattice, Suter, Conformational Theory of Large Molecules, Wiley, New York, 1994.

**(Maxwell)**
J.C. Maxwell, Philos. Tans. Royal Soc. London, 157: 49-88 (1867).

**(Mayergoyz)**
I.D. Mayergoyz, G. Bertotti, C. Serpico (2009). Elsevier (2009)

**(Mayo)**
Mayo, Olfason, Goddard III, J Phys Chem, 94, 8897-8909 (1990).

**(Mees)**
M. J. Mees, G. Pourtois, E. C. Neyts, B. J. Thijsse, and A. Stesmans, Phys. Rev. B 85, 134301 (2012).

**(Mei)**
Mei, Davenport, Fernando, Phys Rev B, 43 4653 (1991)

**(Melchor)**
Gonzalez-Melchor, Mayoral, Velazquez, and Alejandre, J Chem Phys, 125, 224107 (2006).

**(Meloni)**
Meloni, Rosati and Colombo, J Chem Phys, 126, 121102 (2007).

**(Meremianin)**
Meremianin, J. Phys. A, 39, 3099 (2006).

**(Mezei)**
Mezei, J Chem Phys, 86, 7084 (1987)

**(Mickel)**
W. Mickel, S. C. Kapfer, G. E. Schroeder-Turkand, K. Mecke, J. Chem. Phys. 138, 044501 (2013).

**(Mie)**
G. Mie, Ann Phys, 316, 657 (1903).

**(Milano)**
G. Milano, S. Goudeau, F. Mueller-Plathe, J. Polym. Sci. B Polym. Phys. 43, 871 (2005).

**(Miller1)**
T. F. Miller III, M. Eleftheriou, P. Pattnaik, A. Ndirango, G. J. Martyna, J. Chem. Phys., 116, 8649-8659 (2002).

**(Miller2)**
Miller, Tadmor, Gibson, Bernstein and Pavia, J Chem Phys, 144, 184107 (2016).

**(Minary)**
Minary, Martyna, and Tuckerman, J Chem Phys, 18, 2510 (2003).

**(Mindlin and Deresiewicz, 1953)**
Mindlin, R.D., & Deresiewicz, H (1953). Elastic Spheres in Contact under Varying Oblique Force. J. Appl. Mech., ASME 20, 327-344.

**(Mindlin, 1949)**
Mindlin, R. D. (1949). Compliance of elastic bodies in contact. J. Appl. Mech., ASME 16, 259-268.

**(Miron)**
R. A. Miron and K. A. Fichthorn, J Chem Phys, 119, 6210 (2003).

**(Mishin)**
Mishin, Mehl, and Papaconstantopoulos, Acta Mater, 53, 4029 (2005).

**(Mitchell and Fincham)**
Mitchell, Fincham, J Phys Condensed Matter, 5, 1031-1038 (1993).

**(Mitchell2011)**
Mitchell. A non-local, ordinary-state-based viscoelasticity model for peridynamics. Sandia National Lab Report, 8064:1-28 (2011).

**(Mitchell2011a)**
Mitchell. A Nonlocal, Ordinary, State-Based Plasticity Model for Peridynamics. Sandia National Lab Report, 3166:1-34 (2011).

**(Miyazaki)**
Miyazaki, Okazaki, Shinoda, J Chem Theory Comput, 16, 782-793 (2020).

**(Mniszewski)**
S. M. Mniszewski, M. J. Cawkwell, M. E. Wall, J. Mohd-Yusof, N. Bock, T. C. Germann, and A. M. N. Niklasson, J. Chem. Theory Comput., 11, 4644 (2015).

**(Monaghan)**
Monaghan and Gingold, Journal of Computational Physics, 52, 374-389 (1983).

**(Monti)**
Monti, Clemmer, Srivastava, Silbert, Grest, and Lechman, Phys. Rev. E, (2022).

**(Moore)**
Moore, J Chem Phys, 144, 104501 (2016).

**(Mori)**
Y. Mori, Y. Okamoto, J. Phys. Soc. Jpn., 7, 074003 (2010).

**(Moriarty1)**
Moriarty, Physical Review B, 38, 3199 (1988).

**(Moriarty2)**
Moriarty, Physical Review B, 42, 1609 (1990). Moriarty, Physical Review B 49, 12431 (1994).

**(Moriarty3)**
Moriarty, Benedict, Glosli, Hood, Orlikowski, Patel, Soderlind, Streitz, Tang, and Yang, Journal of Materials Research, 21, 563 (2006).

**(Morris)**
Morris, Fox, Zhu, J Comp Physics, 136, 214-226 (1997).

**(Moustafa)**
Sabry G. Moustafa, Andrew J. Schultz, and David A. Kofke, Phys. Rev. E [92], 043303 (2015)

**(Muller-Plathe1)**
Muller-Plathe, J Chem Phys, 106, 6082 (1997).

**(Muller-Plathe2)**
Muller-Plathe, Phys Rev E, 59, 4894-4898 (1999).

**(Murdick)**
D.A. Murdick, X.W. Zhou, H.N.G. Wadley, D. Nguyen-Manh, R. Drautz, and D.G. Pettifor, Phys. Rev. B, 73, 45206 (2006).

**(Murty)**
M.V.R. Murty, H.A. Atwater, Phys Rev B, 51, 4889 (1995).

**(Nakano)**
A. Nakano, Computer Physics Communications, 104, 59-69 (1997).

**(Neelov)**
Neelov, Holm, J Chem Phys 132, 234103 (2010)

**(Nelson)**
Nelson, Halperin, Phys Rev B, 19, 2457 (1979).

**(Nettleton)**
Nettleton and Green, J Chem Phys, 29, 6 (1958).

**(Neyts)**
E. C. Neyts and A. Bogaerts, Theor. Chem. Acc. 132, 1320 (2013).

**(Nguyen2023)**
Nguyen, Physical Review B, 107(14), 144103, (2023).

**(Nguyen2024)**
Nguyen, Journal of Computational Physics, 113102, (2024).

**(Nguyen and Rohskopf)**
Nguyen and Rohskopf, Journal of Computational Physics, 480, 112030, (2023).

**(Nguyen and Sema)**
Nguyen and Sema, https://arxiv.org/abs/2405.00306, (2024).

**(NguyenTD)**
Nguyen, Li, Bagchi, Solis, Olvera de la Cruz, Comput Phys Commun 241, 80-19 (2019)

**(Nicholson and Rutledge)**
Nicholson and Rutledge, J Chem Phys, 145, 244903 (2016).

**(Niklasson2002)**
A. M. N. Niklasson, Phys. Rev. B, 66, 155115 (2002).

**(Niklasson2008)**
A. M. N. Niklasson, Phys. Rev. Lett., 100, 123004 (2008).

**(Niklasson2014)**
A. M. N. Niklasson and M. Cawkwell, J. Chem. Phys., 141, 164123, (2014).

**(Niklasson2017)**
A. M. N. Niklasson, J. Chem. Phys., 147, 054103 (2017).

**(Nitol)**
Nitol, Dickel, and Barrett, Computational Materials Science 188 (2021): 110207.

**(Noid)**
Noid, Chu, Ayton, Krishna, Izvekov, Voth, Das, Andersen, J Chem Phys 128, 134105 (2008).

**(Nordlund95)**
Nordlund, Kai. Computational materials science 3.4 (1995): 448-456.

**(Nordlund98)**
Nordlund, Kai, et al. Physical Review B 57.13 (1998): 7556.

**(Norman)**
G E Norman, S V Starikov, V V Stegailov et al., Contrib. Plasma Phys., 53, 129-139 (2013).

**(Noskov)**
Noskov, Lamoureux and Roux, J Phys Chem B, 109, 6705 (2005).

**(Nurdin)**
Nurdin and Schotte Phys Rev E, 61(4), 3579 (2000)

**(O'Connor)**
O'Connor et al., J. Chem. Phys. 142, 024903 (2015).

**(O'Hearn)**
O'Hearn, Alperen, Aktulga, SIAM J. Sci. Comput., 42(1), C1–C22 (2020).

**(Okabe)**
T. Okabe, M. Kawata, Y. Okamoto, M. Masuhiro, Chem. Phys. Lett., 335, 435-439 (2001).

**(Ollila et al.)**
Ollila, S.T.T., Denniston, C., Karttunen, M., and Ala-Nissila, T., Fluctuating lattice-Boltzmann model for complex fluids, J. Chem. Phys. 134 (2011) 064902.

**(Omelyan)**
Omelyan, Mryglod, and Folk. Phys. Rev. Lett. 86(5), 898. (2001).

**(OPLS-AA96)** Jorgensen, Maxwell, Tirado-Rives, J Am Chem Soc, 118(45), 11225-11236 (1996).

**(Oppelstrup)**
Oppelstrup, unpublished, 2015. Oppelstrup and Moriarty, to be published.

**(Orsi)**
Orsi & Essex, The ELBA force field for coarse-grain modeling of lipid membranes, PloS ONE 6(12): e28637, 2011.

**(Otis R. Walton)**
Walton, O.R., Personal Communication

**(Ouldridge)**
T.E. Ouldridge, A.A. Louis, J.P.K. Doye, J. Chem. Phys. 134, 085101 (2011).

**(Ouldridge-DPhil)**
T.E. Ouldridge, Coarse-grained modelling of DNA and DNA self-assembly, DPhil. University of Oxford (2011).

**(Ouyang1)**
W. Ouyang, D. Mandelli, M. Urbakh and O. Hod, Nano Lett. 18, 6009-6016 (2018).

**(Ouyang2)**
W. Ouyang et al., J. Chem. Theory Comput. 16(1), 666-676 (2020).

**(Ouyang_1)**

W. Ouyang et al., J. Chem. Theory Comput. 16(1), 666-676 (2020).

**(Ouyang6)**

W. Ouyang, O. Hod, and R. Guerra, J. Chem. Theory Comput. 17, 7215 (2021).

**(Ouyang7)**

W. Ouyang, et al., J. Chem. Theory Comput. 17, 7237 (2021).

**(Palkar)**
Palkar V, Kuksenok O, J. Phys. Chem. B, 126 (1), 336-346, 2022

**(Paquay)**
Paquay and Kusters, Biophys. J., 110, 6, (2016). preprint available at arXiv:1411.3019.

**(Park)**
Park, Schulten, J. Chem. Phys. 120 (13), 5946 (2004)

**(Parks)**
Parks, Lehoucq, Plimpton, Silling, Comp Phys Comm, 179(11), 777-783 (2008).

**(Parrinello)**
Parrinello and Rahman, J Appl Phys, 52, 7182 (1981).

**(Paula Leite2016)**
Paula Leite , Freitas, Azevedo, and de Koning, J Chem Phys, 126, 044509 (2016).

**(Paula Leite2017)**
Paula Leite, Santos-Florez, and de Koning, Phys Rev E, 96, 32115 (2017).

**(Pavlov1)**
D Pavlov, V Galigerov, D Kolotinskii, V Nikolskiy, V Stegailov, International Journal of High Performance Computing Applications, 38, 34-49 (2024).

**(Pavlov2)**
Pavlov, Galigerov, Kolotinskii, Nikolskiy, Stegailov, "GPU-based Molecular Dynamics of Fluid Flows: Reaching for Turbulence", Int. J. High Perf. Comp. Appl., (2024)

**(Pearlman)**
Pearlman, J Chem Phys, 98, 1487 (1994)

**(Pedersen)**
Pedersen, J. Chem. Phys., 139, 104102 (2013).

**(Pedone)**

A. Pedone, G. Malavasi, M. C. Menziani, A. N. Cormack, and U. Segre, J. Phys. Chem. B, 110, 11780 (2006)

**(Peng)**
Peng, Ren, Dudarev, Whelan, Acta Crystallogr. A, 52, 257-76 (1996).

**(Perram)**
Perram and Rasmussen, Phys Rev E, 54, 6565-6572 (1996).

**(Petersen)**
Petersen, J Chem Phys, 103, 3668 (1995).

**(Petersen)**
Petersen, Lechman, Plimpton, Grest, in' t Veld, Schunk, J Chem Phys, 132, 174106 (2010).

**(Pettifor_1)**
D.G. Pettifor and I.I. Oleinik, Phys. Rev. B, 59, 8487 (1999).

**(Pettifor_2)**
D.G. Pettifor and I.I. Oleinik, Phys. Rev. Lett., 84, 4124 (2000).

**(Pettifor_3)**
D.G. Pettifor and I.I. Oleinik, Phys. Rev. B, 65, 172103 (2002).

**(PFC)**
PFC Particle Flow Code 6.0 Documentation. Itasca Consulting Group.

**(Phillips)**
C. L. Phillips, J. A. Anderson, S. C. Glotzer, Comput Phys Comm, 230, 7191-7201 (2011).

**(Piaggi)**
Piaggi and Parrinello, J Chem Phys, 147, 114112 (2017).

**(Pisarev)**
V V Pisarev and S V Starikov, J. Phys.: Condens. Matter, 26, 475401 (2014).

**(Plimpton)**
Plimpton and Knight, JPDC, 147, 184-195 (2021).

**(PLUMED)**
G.A. Tribello, M. Bonomi, D. Branduardi, C. Camilloni and G. Bussi, Comp. Phys. Comm 185, 604 (2014)

**(Pollock)**
Pollock and Glosli, Comp Phys Comm, 95, 93 (1996).

**(Ponder)**
Ponder, Wu, Ren, Pande, Chodera, Schnieders, Haque, Mobley, Lambrecht, DiStasio Jr, M. Head-Gordon, Clark, Johnson, T. Head-Gordon, J Phys Chem B, 114, 2549-2564 (2010).

**(Popov1)**
A.M. Popov, I. V. Lebedeva, A. A. Knizhnik, Y. E. Lozovik and B. V. Potapkin, Chem. Phys. Lett. 536, 82-86 (2012).

**(Price1)**
Price and Brooks, J Chem Phys, 121, 10096 (2004).

**(Price2)**
Price, Stone and Alderton, Mol Phys, 52, 987 (1984).

**(QEq/Fire)**
T.-R. Shan, A. P. Thompson, S. J. Plimpton, in preparation

**(Qi)**
Qi and Reed, J. Phys. Chem. A 116, 10451 (2012).

**(Ramirez)**
J. Ramirez, S.K. Sukumaran, B. Vorselaars and A.E. Likhtman, J. Chem. Phys. 133, 154103 (2010).

**(Rappe)**
Rappe and Goddard III, Journal of Physical Chemistry, 95, 3358-3363 (1991).

**(Ravelo)**
Ravelo, Holian, Germann and Lomdahl, Phys Rev B, 70, 014103 (2004).

**(ReaxFF)**
A. C. T. van Duin, S. Dasgupta, F. Lorant, W. A. Goddard III, J Physical Chemistry, 105, 9396-9049 (2001)

**(Rector)**
Rector, Van Swol, Henderson, Molecular Physics, 82, 1009 (1994).

**(Ree)**
Ree, Journal of Chemical Physics, 73, 5401 (1980).

**(Reed)**
Reed, Fried, and Joannopoulos, Phys. Rev. Lett., 90, 235503 (2003).

**(Reed2)**
Reed, J. Phys. Chem. C, 116, 2205 (2012).

**(Rick)**
S. W. Rick, S. J. Stuart, B. J. Berne, J Chem Phys 101, 16141 (1994).

**(Rick2)**

S. W. Rick, S. J. Stuart, B. J. Berne, J Chem Phys 101, 6141

**(Roberts)**

R. Roberts (2019) "Evenly Distributing Points in a Triangle." Extreme Learning. https://extremelearning.com.au/evenly-distributing-points-in-a-triangle/

**(Rohart)**
Rohart and Thiaville, Physical Review B, 88(18), 184422. (2013).

**(Rosenberger)**
Rosenberger, Sanyal, Shell and van der Vegt, Journal of Chemical Physics, 2019, 151 (4), 044111.

**(Rubensson)**
E. H. Rubensson, A. M. N. Niklasson, SIAM J. Sci. Comput. 36 (2), 147-170, (2014).

**(Rutherford)**
A M Rutherford and D M Duffy, J. Phys.: Condens. Matter, 19, 496201-496210 (2007).

**(Ryckaert)**
J.-P. Ryckaert, G. Ciccotti and H. J. C. Berendsen, J of Comp Phys, 23, 327-341 (1977).

**(SMTB-Q_1)**
N. Salles, O. Politano, E. Amzallag, R. Tetot, Comput. Mater. Sci. 111 (2016) 181-189

**(SMTB-Q_2)**
E. Maras, N. Salles, R. Tetot, T. Ala-Nissila, H. Jonsson, J. Phys. Chem. C 2015, 119, 10391-10399

**(SMTB-Q_3)**
R. Tetot, N. Salles, S. Landron, E. Amzallag, Surface Science 616, 19-8722 28 (2013)

**(SRIM)**
SRIM webpage: http://www.srim.org/

**(SW)**
F. H. Stillinger, and T. A. Weber, Phys. Rev. B, 31, 5262 (1985).

**(SWM4-NDP)**
Lamoureux, Harder, Vorobyov, Roux, MacKerell, Chem Phys Let, 418, 245-249 (2006)

**(Sadigh)**
B Sadigh, P Erhart, A Stukowski, A Caro, E Martinez, and L Zepeda-Ruiz, Phys. Rev. B, 85, 184203 (2012).

**(Sadigh1)**

B. Sadigh, P. Erhart, A. Stukowski, A. Caro, E. Martinez, and L. Zepeda-Ruiz, Phys. Rev. B **85**, 184203 (2012)

**(Sadigh2)**

B. Sadigh and P. Erhart, Phys. Rev. B **86**, 134204 (2012)

**(Safran)**
Safran, Statistical Thermodynamics of Surfaces, Interfaces, And Membranes, Westview Press, ISBN: 978-0813340791 (2003).

**(Salanne)**
Salanne, Rotenberg, Jahn, Vuilleumier, Simon, Christian and Madden, Theor Chem Acc, 131, 1143 (2012).

**(Salerno)**
Salerno, Bernstein, J Chem Theory Comput, –, —- (2018).

**(Sanyal1)**
Sanyal and Shell, Journal of Chemical Physics, 2016, 145 (3), 034109.

**(Sanyal2)**
Sanyal and Shell, Journal of Physical Chemistry B, 122 (21), 5678-5693.

**(Scalfi)**
Scalfi *et al.*, J. Chem. Phys., 153, 174704 (2020).

**(Schelling)**
Patrick K. Schelling, Comp. Mat. Science, 44, 274 (2008).

**(Scherer1)**

C. Scherer and D. Andrienko, Phys. Chem. Chem. Phys. 20, 22387-22394 (2018).

**(Scherer2)**

C. Scherer, R. Scheid, D. Andrienko, and T. Bereau, J. Chem. Theor. Comp. 16, 3194-3204 (2020).

**(Schlitter1)**
Schlitter, Swegat, Mulders, "Distance-type reaction coordinates for modelling activated processes", J Molecular Modeling, 7, 171-177 (2001).

**(Schlitter2)**
Schlitter and Klahn, "The free energy of a reaction coordinate at multiple constraints: a concise formulation", Molecular Physics, 101, 3439-3443 (2003).

**(Schmid)**
S. Bureekaew, S. Amirjalayer, M. Tafipolsky, C. Spickermann, T.K. Roy and R. Schmid, Phys. Status Solidi B, 6, 1128 (2013).

**(Schneider)**
Schneider and Stoll, Phys Rev B, 17, 1302 (1978).

**(Schratt & Mohles)**
Schratt, Mohles. Comp. Mat. Sci. 182 (2020) 109774 ————-

**(Schroeder)**
Schroeder and Steinhauser, J Chem Phys, 133, 154511 (2010).

**(Seleson 2010)**
Seleson, Parks, Int J Mult Comp Eng 9(6), pp. 689-706, 2011.

**(Semaev)**
Semaev, Cryptography and Lattices, 181 (2001).

**(Seo)**
Seo, Shinoda, J Chem Theory Comput, 15, 762-774 (2019).

**(Sheppard)**
Sheppard, Terrell, Henkelman, J Chem Phys, 128, 134106 (2008). See ref 1 in this paper for original reference to Qmin in Jonsson, Mills, Jacobsen.

**(Shi)**
Shi, Xia, Zhang, Best, Wu, Ponder, Ren, J Chem Theory Comp, 9, 4046, 2013.

**(Shinoda)**
Shinoda, DeVane, Klein, Mol Sim, 33, 27 (2007).

**(Shinoda)**
Shinoda, Shiga, and Mikami, Phys Rev B, 69, 134103 (2004).

**(Shire)**
Shire, Hanley and Stratford, Comp. Part. Mech., (2020).

**(Sides)**
Sides, Grest, Stevens, Plimpton, J Polymer Science B, 42, 199-208 (2004).

**(Siepmann)**
Siepmann and Sprik, J. Chem. Phys. 102, 511 (1995).

**(Silbert)**
Silbert, Ertas, Grest, Halsey, Levine, Plimpton, Phys Rev E, 64, p 051302 (2001).

**(Silbert, 2001)**
Silbert, L. E., Ertas, D., Grest, G. S., Halsey, T. C., Levine, D., & Plimpton, S. J. (2001). Granular flow down an inclined plane: Bagnold scaling and rheology. Physical Review E,

**(Silling 2000)**
Silling, J Mech Phys Solids, 48, 175-209 (2000).

**(Silling 2005)**
Silling Askari, Computer and Structures, 83, 1526-1535 (2005).

**(Silling 2007)**
Silling, Epton, Weckner, Xu, Askari, J Elasticity, 88, 151-184 (2007).

**(Singh)**
Singh and Warner, Acta Mater, 58, 5797-5805 (2010),

**(Singraber, Behler and Dellago 2019)**
Singraber, A.; Behler, J.; Dellago, C. J., Chem. Theory Comput. 2019, 15 (3), 1827-1840

**(Singraber et al 2019)**
Singraber, A.; Morawietz, T.; Behler, J.; Dellago, C., J. Chem. Theory Comput. 2019, 15 (5), 3075-3092.

**(Sirk1)**
Sirk TW, Sliozberg YR, Brennan JK, Lisal M, Andzelm JW, J Chem Phys, 136 (13) 134903, 2012.

**(Sirk2)**
Sirk, Moore, Brown, J Chem Phys, 138, 064505 (2013).

**(Skomski)**
Skomski, R. (2008). Simple models of magnetism. Oxford University Press.

**(Snodin)**
B.E. Snodin, F. Randisi, M. Mosayebi, et al., J. Chem. Phys. 142, 234901 (2015).

**(Son)**
Son, McDaniel, Cui and Yethiraj, J Phys Chem Lett, 10, 7523 (2019).

**(Srivastava)**
Zhigilei, Wei, Srivastava, Phys. Rev. B 71, 165417 (2005).

**(Steinbach)**
Steinbach, Brooks, J Comput Chem, 15, 667 (1994).

**(Steinhardt)**
P. Steinhardt, D. Nelson, and M. Ronchetti, Phys. Rev. B 28, 784 (1983).

**(Steward)**
Stewart, Spearot, Modelling Simul. Mater. Sci. Eng. 21, 045003, (2013).

**(Stewart2018)**
J.A. Stewart, et al. (2018) Journal of Applied Physics, 123(16), 165902.

**(Stiles)**
Stiles , Hubbard, and Kayser, J Chem Phys, 77, 6189 (1982).

**(Stillinger)**
Stillinger, Weber, Phys. Rev. B 31, 5262 (1985).

**(Stoddard)**
Stoddard and Ford, Phys Rev A, 8, 1504 (1973).

**(Streitz)**
F. H. Streitz, J. W. Mintmire, Phys Rev B, 50, 11996-12003 (1994).

**(Strong)**
Strong and Eaves, J. Phys. Chem. B 121, 189 (2017).

**(Stuart)**
Stuart, Tutein, Harrison, J Chem Phys, 112, 6472-6486 (2000).

**(Stukowski)**
Stukowski, Sadigh, Erhart, Caro; Modeling Simulation Materials Science & Engineering, 7, 075005 (2009).

**(Su)**
Su and Goddard, Excited Electron Dynamics Modeling of Warm Dense Matter, Phys Rev Lett, 99:185003 (2007).

**(Sulc1)**
P. Sulc, F. Romano, T. E. Ouldridge, et al., J. Chem. Phys. 140, 235102 (2014).

**(Sulc2)**
P. Sulc, F. Romano, T.E. Ouldridge, L. Rovigatti, J.P.K. Doye, A.A. Louis, J. Chem. Phys. 137, 135101 (2012).

**(Sun)**
Sun, J. Phys. Chem. B, 102, 7338-7364 (1998).

**(Surblys2019)**
Surblys, Matsubara, Kikugawa, Ohara, Phys Rev E, 99, 051301(R) (2019).

**(Surblys2021)**
Surblys, Matsubara, Kikugawa, Ohara, J Appl Phys 130, 215104 (2021).

**(Sutmann)**
Sutmann, Arnold, Fahrenberger, et. al., Physical review / E 88(6), 063308 (2013)

**(Sutmann) G. Sutmann. ScaFaCoS - a Scalable library of Fast Coulomb Solvers for particle Systems.**
In Bajaj, Zavattieri, Koslowski, Siegmund, Proceedings of the Society of Engineering Science 51st Annual Technical Meeting. 2014.

**(Swinburne)**
Swinburne and Marinica, Physical Review Letters, 120, 1 (2018)

**(Tadmor)**
Tadmor, Elliott, Sethna, Miller and Becker, JOM, 63, 17 (2011). doi: https://doi.org/10.1007/s11837-011-0102-6

**(Tainter 2011)**
Tainter, Pieniazek, Lin, and Skinner, J. Chem. Phys., 134, 184501 (2011)

**(Tainter 2015)**
Tainter, Shi, and Skinner, 11, 2268 (2015)

**(Tang and Toennies)**
J Chem Phys, 80, 3726 (1984).

**(Tee)**
Tee and Searles, J. Chem. Phys. 156, 184101 (2022).

**(Templeton2010)**
Templeton, JA; Jones, RE; Wagner, GJ, "Application of a field-based method to spatially varying thermal transport problems in molecular dynamics." Modelling and Simulation in Materials Science and Engineering (2010), 18:085007.

**(Templeton2011)**
Templeton, JA; Jones, RE; Lee, JW; Zimmerman, JA; Wong, BM, "A long-range electric field solver for molecular dynamics based on atomistic-to-continuum modeling." Journal of Chemical Theory and Computation (2011), 7:1736.

**(tenWolde)**
P. R. ten Wolde, M. J. Ruiz-Montero, D. Frenkel, J. Chem. Phys. 104, 9932 (1996).

**(Tersoff_1)**
J. Tersoff, Phys Rev B, 37, 6991 (1988).

**(Tersoff_2)**
J. Tersoff, Phys Rev B, 38, 9902 (1988).

**(Tersoff_3)**
J. Tersoff, Phys Rev B, 39, 5566 (1989); errata (PRB 41, 3248)

**(Theodorou)**
Theodorou, Suter, Macromolecules, 18, 1206 (1985).

**(Thole)**
Chem Phys, 59, 341 (1981).

**(Thompson1)**
Thompson, Plimpton, Mattson, J Chem Phys, 131, 154107 (2009).

**(Thompson2)**
Thompson, Swiler, Trott, Foiles, Tucker, J Comp Phys, 285, 316 (2015).

**(Thornton et al, 2013)**
Thornton, C., Cummins, S. J., & Cleary, P. W. (2013). An investigation of the comparative behavior of alternative contact force models during inelastic collisions. Powder

**(Thornton, 1991)**
Thornton, C. (1991). Interparticle sliding in the presence of adhesion. J. Phys. D: Appl. Phys. 24 1942

**(To)**
Q.D. To, V.H. Vu, G. Lauriat, and C. Leonard. J. Math. Phys. 56, 103101 (2015).

**(Todd)**
B. D. Todd, Denis J. Evans, and Peter J. Daivis: "Pressure tensor for inhomogeneous fluids", Phys. Rev. E 52, 1627 (1995).

**(Toukmaji)**
Toukmaji, Sagui, Board, and Darden, J Chem Phys, 113, 10913 (2000).

**(Toxvaerd)**
Toxvaerd, Dyre, J Chem Phys, 134, 081102 (2011).

**(Tranchida)**
Tranchida, Plimpton, Thibaudeau and Thompson, Journal of Computational Physics, 372, 406-425, (2018).

**(Tribello)**
G.A. Tribello, M. Bonomi, D. Branduardi, C. Camilloni and G. Bussi, Comp. Phys. Comm 185, 604 (2014)

**(Tsuji et al, 1992)**
Tsuji, Y., Tanaka, T., & Ishida, T. (1992). Lagrangian numerical simulation of plug flow of cohesionless particles in a horizontal pipe. Powder technology, 71(3),

**(Tsuzuki)**
Tsuzuki, Branicio, Rino, Comput Phys Comm, 177, 518 (2007).

**(Tuckerman1)**
M. Tuckerman and B. Berne, J Chem Phys, 99, 2796 (1993).

**(Tuckerman2)**
Tuckerman, Alejandre, Lopez-Rendon, Jochim, and Martyna, J Phys A: Math Gen, 39, 5629 (2006).

**(Tuckerman3)**
Tuckerman, Berne and Martyna, J Chem Phys, 97, p 1990 (1992).

**(Tuckerman4)**
Tuckerman, Mundy, Balasubramanian, Klein, J Chem Phys, 106, 5615 (1997).

**(Tyagi)**
Tyagi, Suzen, Sega, Barbosa, Kantorovich, Holm, J Chem Phys, 132, 154112 (2010)

**(Ulomek)**
Ulomek, Brien, Foiles, Mohles, Modelling Simul. Mater. Sci. Eng. 23 (2015) 025007

**(Vaiwala)**
Vaiwala, Jadhav, and Thaokar, J Chem Phys, 146, 124904 (2017).

**(Valone)**
Valone, Baskes, Martin, Phys. Rev. B, 73, 214209 (2006).

**(vanWijk)**
M. M. van Wijk, A. Schuring, M. I. Katsnelson, and A. Fasolino, Physical Review Letters, 113, 135504 (2014)

**(Van Workum)**
K. Van Workum et al., J. Chem. Phys. 125 144506 (2006)

**(Vargas and McCarthy 2001)**
Vargas, W.L. and McCarthy, J.J. (2001).

**(Varshalovich)**
Varshalovich, Moskalev, Khersonskii, Quantum Theory of Angular Momentum, World Scientific, Singapore (1987).

**(Vashishta1990)**
P. Vashishta, R. K. Kalia, J. P. Rino, Phys. Rev. B 41, 12197 (1990).

**(Vashishta2007)**
P. Vashishta, R. K. Kalia, A. Nakano, J. P. Rino. J. Appl. Phys. 101, 103515 (2007).

**(Veld)**
In ‘t Veld, Ismail, Grest, J Chem Phys, 127, 144711 (2007).

**(Verstraelen)**
Verstraelen, Ayers, Speybroeck, Waroquier, J. Chem. Phys. 138, 074108 (2013).

**(Volkov1)**
Volkov and Zhigilei, J Phys Chem C, 114, 5513 (2010).

**(Volkov2)**
Volkov, Simov and Zhigilei, APS Meeting Abstracts, Q31.013 (2008).

**(Voter1998)**
Voter, Phys Rev B, 57, 13985 (1998).

**(Voter2000)**
Sorensen and Voter, J Chem Phys, 112, 9599 (2000)

**(Voter2002)**
Voter, Montalenti, Germann, Annual Review of Materials Research 32, 321 (2002).

**(Voter2013)**
S. Y. Kim, D. Perez, A. F. Voter, J Chem Phys, 139, 144110 (2013).

**(Wagner)**
Wagner, GJ; Jones, RE; Templeton, JA; Parks, MA, "An atomistic-to-continuum coupling method for heat transfer in solids." Special Issue of Computer Methods and Applied Mechanics (2008) 197:3351.

**(Wang et al, 2015)**
Wang, Y., Alonso-Marroquin, F., & Guo, W. W. (2015). Rolling and sliding in 3-D discrete element models. Particuology, 23, 49-55.

**(Wang2020)**

X. Wang, S. Ramirez-Hinestrosa, J. Dobnikar, and D. Frenkel, Phys. Chem. Chem. Phys. 22, 10624 (2020).

**(Wang1)**
J. Wang, H. S. Yu, P. A. Langston, F. Y. Fraige, Granular Matter, 13, 1 (2011).

**(Wang2)**
J. Wang, and A. Rockett, Phys. Rev. B, 43, 12571 (1991).

**(Wang3)**
Wang and Holm, J Chem Phys, 115, 6277 (2001).

**(Wang4)**
Wang, Van Hove, Ross, Baskes, J. Chem. Phys., 121, 5410 (2004).

**(Ward)**
D.K. Ward, X.W. Zhou, B.M. Wong, F.P. Doty, and J.A. Zimmerman, Phys. Rev. B, 85,115206 (2012).

**(Warren)**
Warren, Phys Rev E, 68, 066702 (2003).

**(Watkins)**
Watkins and Jorgensen, J Phys Chem A, 105, 4118-4125 (2001).

**(Weeks)**
Weeks, Chandler and Andersen, J. Chem. Phys., 54, 5237 (1971)

**(WeinanE)**
E, Ren, Vanden-Eijnden, Phys Rev B, 66, 052301 (2002).

**(Wen)**
M. Wen, S. Carr, S. Fang, E. Kaxiras, and E. B. Tadmor, Phys. Rev. B, 98, 235404 (2018)

**(Wennberg)**
Wennberg, Murtola, Hess, Lindahl, J Chem Theory Comput, 9, 3527 (2013).

**(Wicaksono1)**
Wicaksono, Sinclair, Militzer, Computational Materials Science, 117, 397-405 (2016).

**(Wicaksono2)**
Wicaksono, figshare, https://doi.org/10.6084/m9.figshare.1488628.v1 (2015).

**(Winkler)**
Winkler, Wysocki, and Gompper, Soft Matter, 11, 6680 (2015).

**(Wirnsberger)**
Wirnsberger, Frenkel, and Dellago, J Chem Phys, 143, 124104 (2015).

**(Wolf)**
D. Wolf, P. Keblinski, S. R. Phillpot, J. Eggebrecht, J Chem Phys, 110, 8254 (1999).

**(Wolff)**
Wolff and Rudd, Comp Phys Comm, 120, 200-32 (1999).

**(Wood)**
Wood and Thompson, J Chem Phys, 148, 241721, (2018)

**(Xie23)**
Xie, S.R., Rupp, M. & Hennig, R.G. Ultra-fast interpretable machine-learning potentials. npj Comput Mater 9, 162 (2023). https://doi.org/10.1038/s41524-023-01092-7

**(Yade-DEM)**

V. Smilauer et al. (2021), Yade Documentation 3rd ed.

**(Yanxon2020)**
Yanxon, Zagaceta, Tang, Matteson, Zhu, Mach. Learn.: Sci. Technol. 2, 027001 (2020).

**(Yeh)**
Yeh and Berkowitz, J Chem Phys, 111, 3155 (1999).

**(Yuan2010a)**
Yuan, Huang, Li, Lykotrafitis, Zhang, Phys. Rev. E, 82, 011905(2010).

**(Yuan2010b)**
Yuan, Huang, Zhang, Soft. Matter, 6, 4571(2010).

**(Zagaceta2020)**
Zagaceta, Yanxon, Zhu, J Appl Phys, 128, 045113 (2020).

**(ZBL)**
J.F. Ziegler, J.P. Biersack, U. Littmark, 'Stopping and Ranges of Ions in Matter' Vol 1, 1985, Pergamon Press.

**(Zhang1)**
Zhang and Makse, Phys Rev E, 72, p 011301 (2005).

**(Zhang2)**
Zhang and Trinkle, Computational Materials Science, 124, 204-210 (2016).

**(Zhang3)**
Zhang, Glotzer, Nanoletters, 4, 1407-1413 (2004).

**(Zhang4)**
Zhang, J Chem Phys, 106, 6102 (1997).

**(Zhang5)**
Zhang, Lussetti, de Souza, Muller-Plathe, J Phys Chem B, 109, 15060-15067 (2005).

**(Zhigilei1)**
Volkov and Zhigilei, ACS Nano 4, 6187 (2010).

**(Zhigilei2)**
Volkov, Simov, Zhigilei, ASME paper IMECE2008, 68021 (2008).

**(Zhigilei3)**
Volkov, Zhigilei, J. Phys. Chem. C 114, 5513 (2010).

**(Zhigilei4)**
Wittmaack, Banna, Volkov, Zhigilei, Carbon 130, 69 (2018).

**(Zhigilei5)**
Wittmaack, Volkov, Zhigilei, Compos. Sci. Technol. 166, 66 (2018).

**(Zhigilei6)**
Wittmaack, Volkov, Zhigilei, Carbon 143, 587 (2019).

**(Zhigilei7)**
Volkov, Zhigilei, Phys. Rev. Lett. 104, 215902 (2010).

**(Zhigilei8)**
Volkov, Shiga, Nicholson, Shiomi, Zhigilei, J. Appl. Phys. 111, 053501 (2012).

**(Zhigilei9)**
Volkov, Zhigilei, Appl. Phys. Lett. 101, 043113 (2012).

**(Zhigilei10)**
Jacobs, Nicholson, Zemer, Volkov, Zhigilei, Phys. Rev. B 86, 165414 (2012).

**(Zhou1)**
Zhou, Saidi, Fichthorn, J Phys Chem C, 118(6), 3366-3374 (2014).

**(Zhou3)**
X. W. Zhou, M. E. Foster, R. E. Jones, P. Yang, H. Fan, and F. P. Doty, J. Mater. Sci. Res., 4, 15 (2015).

**(Zhou4)**
X. W. Zhou, M. E. Foster, J. A. Ronevich, and C. W. San Marchi, J. Comp. Chem., 41, 1299 (2020).

**(Zhu)**
Zhu, Tajkhorshid, and Schulten, Biophys. J. 83, 154 (2002).

**(Ziegler)**
J.F. Ziegler, J. P. Biersack and U. Littmark, "The Stopping and Range of Ions in Matter", Volume 1, Pergamon, 1985.

**(Zimmerman2004)**
Zimmerman, JA; Webb, EB; Hoyt, JJ;. Jones, RE; Klein, PA; Bammann, DJ, "Calculation of stress in atomistic simulation." Special Issue of Modelling and Simulation in Materials Science and Engineering (2004),12:S319.

**(Zimmerman2010)**
Zimmerman, JA; Jones, RE; Templeton, JA, "A material frame approach for evaluating continuum variables in atomistic simulations." Journal of Computational Physics (2010), 229:2364.

**(electronic stopping)**
Wikipedia - Electronic Stopping Power: https://en.wikipedia.org/wiki/Stopping_power_%28particle_radiation%29

# Part IV

# Indices and tables

# INDEX

## Symbols


_LMP_STYLE_CONST (*C++ enum*), 604
_LMP_TYPE_CONST (*C++ enum*), 604
_LMP_VAR_CONST (*C++ enum*), 605
__version__ (*in module lammps*), 708


## A


accelerator_config (*lammps.lammps property*), 723
addstep_compute() (*fortran subroutine*), **661**
addstep_compute() (*lammps.lammps method*), 717
addstep_compute_all() (*fortran subroutine*), **661**
addstep_compute_all() (*lammps.lammps method*), 717
angle_coeff, 975
angle_style, 976
angle_style amoeba, 2841
angle_style charmm, 2843
angle_style charmm/intel, 2843
angle_style charmm/kk, 2843
angle_style charmm/omp, 2843
angle_style class2, 2844
angle_style class2/kk, 2844
angle_style class2/omp, 2844
angle_style class2/p6, 2844
angle_style class2/p6/kk, 2844
angle_style class2/p6/omp, 2844
angle_style class2xe, 2844
angle_style class2xe/kk, 2844
angle_style class2xe/omp, 2844
angle_style cosine, 2847
angle_style cosine/buck6d, 2848
angle_style cosine/buck6d/omp, 2848
angle_style cosine/delta, 2850
angle_style cosine/delta/kk, 2850
angle_style cosine/delta/omp, 2850
angle_style cosine/kk, 2847
angle_style cosine/omp, 2847
angle_style cosine/periodic, 2851
angle_style cosine/periodic/kk, 2851
angle_style cosine/periodic/omp, 2851
angle_style cosine/shift, 2852

angle_style cosine/shift/exp, 2853
angle_style cosine/shift/exp/kk, 2853
angle_style cosine/shift/exp/omp, 2853
angle_style cosine/shift/kk, 2852
angle_style cosine/shift/omp, 2852
angle_style cosine/squared, 2854
angle_style cosine/squared/kk, 2854
angle_style cosine/squared/omp, 2854
angle_style cosine/squared/restricted, 2855
angle_style cosine/squared/restricted/kk, 2855
angle_style cosine/squared/restricted/omp, 2855
angle_style cross, 2856
angle_style cross/kk, 2856
angle_style cross/omp, 2856
angle_style dipole, 2858
angle_style dipole/kk, 2858
angle_style dipole/omp, 2858
angle_style fourier, 2859
angle_style fourier/kk, 2859
angle_style fourier/omp, 2859
angle_style fourier/simple, 2861
angle_style fourier/simple/kk, 2861
angle_style fourier/simple/omp, 2861
angle_style gaussian, 2862
angle_style gaussian/kk, 2862
angle_style gaussian/omp, 2862
angle_style harmonic, 2863
angle_style harmonic/intel, 2863
angle_style harmonic/kk, 2863
angle_style harmonic/omp, 2863
angle_style hybrid, 2864
angle_style hybrid/kk, 2864
angle_style lepton, 2866
angle_style lepton/omp, 2866
angle_style mesocnt, 2868
angle_style mm3, 2870
angle_style mm3/kk, 2870
angle_style mm3/omp, 2870
angle_style mwlc, 2871
angle_style mwlc/kk, 2871
angle_style mwlc/omp, 2871
angle_style none, 2873
angle_style quartic, 2873
angle_style quartic/kk, 2873
angle_style quartic/omp, 2873
angle_style spica, 2874
angle_style spica/kk, 2874
angle_style spica/omp, 2874
angle_style table, 2876
angle_style table/omp, 2876
angle_style zero, 2878
angle_write, 978
ArgInfo (*C++ class*), 938
ArgInfo::ArgInfo (*C++ function*), 939

ArgInfo::ArgTypes (*C++ enum*), 938
ArgInfo::ArgTypes::BIN1D (*C++ enumerator*), 939
ArgInfo::ArgTypes::BIN2D (*C++ enumerator*), 939
ArgInfo::ArgTypes::BIN3D (*C++ enumerator*), 939
ArgInfo::ArgTypes::BINCYLINDER (*C++ enumerator*), 939
ArgInfo::ArgTypes::BINSPHERE (*C++ enumerator*), 939
ArgInfo::ArgTypes::COMPUTE (*C++ enumerator*), 938
ArgInfo::ArgTypes::DENSITY_MASS (*C++ enumerator*), 939
ArgInfo::ArgTypes::DENSITY_NUMBER (*C++ enumerator*), 939
ArgInfo::ArgTypes::DNAME (*C++ enumerator*), 939
ArgInfo::ArgTypes::ERROR (*C++ enumerator*), 938
ArgInfo::ArgTypes::F (*C++ enumerator*), 938
ArgInfo::ArgTypes::FIX (*C++ enumerator*), 938
ArgInfo::ArgTypes::INAME (*C++ enumerator*), 939
ArgInfo::ArgTypes::KEYWORD (*C++ enumerator*), 938
ArgInfo::ArgTypes::MASS (*C++ enumerator*), 939
ArgInfo::ArgTypes::MOLECULE (*C++ enumerator*), 939
ArgInfo::ArgTypes::NONE (*C++ enumerator*), 938
ArgInfo::ArgTypes::TEMPERATURE (*C++ enumerator*), 939
ArgInfo::ArgTypes::TYPE (*C++ enumerator*), 939
ArgInfo::ArgTypes::UNKNOWN (*C++ enumerator*), 938
ArgInfo::ArgTypes::V (*C++ enumerator*), 938
ArgInfo::ArgTypes::VARIABLE (*C++ enumerator*), 938
ArgInfo::ArgTypes::X (*C++ enumerator*), 938
ArgInfo::copy_name (*C++ function*), 940
ArgInfo::get_dim (*C++ function*), 940
ArgInfo::get_index1 (*C++ function*), 940
ArgInfo::get_index2 (*C++ function*), 940
ArgInfo::get_name (*C++ function*), 940
ArgInfo::get_type (*C++ function*), 939
Atom (*C++ class*), 896
Atom::add_custom (*C++ function*), 897
Atom::Atom (*C++ function*), 896
Atom::extract (*C++ function*), 898
Atom::extract_datatype (*C++ function*), 900
Atom::extract_size (*C++ function*), 900
Atom::find_custom (*C++ function*), 896
Atom::find_custom_ghost (*C++ function*), 897
Atom::PerAtom (*C++ struct*), 900
Atom::remove_custom (*C++ function*), 897
atom_modify, 979
atom_style, 982
available_ids() (*lammps.lammps method*), 724
available_plugins() (*lammps.lammps method*), 725
available_styles() (*lammps.lammps method*), 724

## B

balance, 988
bond_coeff, 995
bond_style, 997
bond_style bpm/rotational, 2797
bond_style bpm/spring, 2801
bond_style bpm/spring/plastic, 2804
bond_style class2, 2807

bond_style class2/kk, 2807
bond_style class2/omp, 2807
bond_style fene, 2808
bond_style fene/expand, 2810
bond_style fene/expand/kk, 2810
bond_style fene/expand/omp, 2810
bond_style fene/intel, 2808
bond_style fene/kk, 2808
bond_style fene/nm, 2808
bond_style fene/nm/kk, 2808
bond_style fene/nm/omp, 2808
bond_style fene/omp, 2808
bond_style gaussian, 2811
bond_style gaussian/kk, 2811
bond_style gaussian/omp, 2811
bond_style gromos, 2812
bond_style gromos/kk, 2812
bond_style gromos/omp, 2812
bond_style harmonic, 2813
bond_style harmonic/intel, 2813
bond_style harmonic/kk, 2813
bond_style harmonic/omp, 2813
bond_style harmonic/restrain, 2814
bond_style harmonic/restrain/kk, 2814
bond_style harmonic/restrain/omp, 2814
bond_style harmonic/shift, 2816
bond_style harmonic/shift/cut, 2817
bond_style harmonic/shift/cut/kk, 2817
bond_style harmonic/shift/cut/omp, 2817
bond_style harmonic/shift/kk, 2816
bond_style harmonic/shift/omp, 2816
bond_style hybrid, 2818
bond_style hybrid/kk, 2818
bond_style lepton, 2819
bond_style lepton/omp, 2819
bond_style mesocnt, 2822
bond_style mm3, 2823
bond_style mm3/kk, 2823
bond_style mm3/omp, 2823
bond_style morse, 2824
bond_style morse/kk, 2824
bond_style morse/omp, 2824
bond_style none, 2825
bond_style nonlinear, 2826
bond_style nonlinear/kk, 2826
bond_style nonlinear/omp, 2826
bond_style oxdna/fene, 2827
bond_style oxdna2/fene, 2827
bond_style oxdna3/fene, 2827
bond_style oxrna2/fene, 2827
bond_style quartic, 2830
bond_style quartic/exp, 2832
bond_style quartic/exp/kk, 2832
bond_style quartic/kk, 2830

bond_style quartic/omp, 2830
bond_style rheo/shell, 2833
bond_style special, 2835
bond_style table, 2837
bond_style table/omp, 2837
bond_style zero, 2839
bond_write, 999
boundary, 1000

## C

change_box, 1002
CiteMe (*C++ class*), 903
CiteMe::~CiteMe (*C++ function*), 903
CiteMe::add (*C++ function*), 903
CiteMe::CiteMe (*C++ function*), 903
CiteMe::flush (*C++ function*), 904
clear, 1007
clearstep_compute() (*fortran subroutine*), **661**
clearstep_compute() (*lammps.lammps method*), 717
close() (*fortran subroutine*), **646**
close() (*lammps.lammps method*), 710
cmd (*lammps.lammps property*), 709
comm_modify, 1007
comm_style, 1010
command() (*fortran subroutine*), **647**
command() (*lammps.lammps method*), 711
commands_list() (*fortran subroutine*), **647**
commands_list() (*lammps.lammps method*), 711
commands_string() (*fortran subroutine*), **647**
commands_string() (*lammps.lammps method*), 711
compute, 1011
compute ackland/atom, 2045
compute ackland/atom/omp, 2045
compute adf, 2046
compute aggregate/atom, 2078
compute angle, 2049
compute angle/local, 2050
compute angmom/chunk, 2052
compute ave/sphere/atom, 2053
compute ave/sphere/atom/kk, 2053
compute ave/sphere/atom/omp, 2053
compute basal/atom, 2055
compute body/local, 2056
compute bond, 2057
compute bond/local, 2058
compute born/matrix, 2061
compute centro/atom, 2064
compute centro/atom/omp, 2064
compute centroid/stress/atom, 2251
compute chunk/atom, 2066
compute chunk/spread/atom, 2075
compute cluster/atom, 2078
compute cna/atom, 2080
compute cna/atom/omp, 2080

compute cnp/atom, 2082
compute cnp/atom/omp, 2082
compute com, 2084
compute com/chunk, 2085
compute composition/atom, 2086
compute composition/atom/kk, 2086
compute contact/atom, 2088
compute coord/atom, 2089
compute coord/atom/kk, 2089
compute coord/atom/omp, 2089
compute count/type, 2091
compute damage/atom, 2093
compute dihedral, 2094
compute dihedral/local, 2095
compute dilatation/atom, 2097
compute dipole, 2097
compute dipole/chunk, 2099
compute dipole/tip4p, 2097
compute dipole/tip4p/chunk, 2099
compute displace/atom, 2100
compute dpd, 2102
compute dpd/atom, 2103
compute edpd/temp/atom, 2104
compute efield/atom, 2105
compute efield/wolf/atom, 2106
compute entropy/atom, 2108
compute entropy/atom/omp, 2108
compute erotate/asphere, 2110
compute erotate/asphere/kk, 2110
compute erotate/rigid, 2111
compute erotate/sphere, 2112
compute erotate/sphere/atom, 2113
compute erotate/sphere/atom/omp, 2113
compute erotate/sphere/kk, 2112
compute event/displace, 2115
compute fabric, 2116
compute fep, 2118
compute fep/ta, 2122
compute force/tally, 2260
compute fragment/atom, 2078
compute gaussian/grid/local, 2123
compute gaussian/grid/local/kk, 2123
compute global/atom, 2125
compute group/group, 2128
compute gyration, 2130
compute gyration/chunk, 2131
compute gyration/shape, 2133
compute gyration/shape/chunk, 2134
compute hbond/local, 2135
compute heat/flux, 2140
compute heat/flux/tally, 2260
compute heat/flux/virial/tally, 2260
compute hexorder/atom, 2144
compute hexorder/atom/omp, 2144

compute hma, 2145
compute improper, 2148
compute improper/local, 2149
compute inertia/chunk, 2150
compute ke, 2152
compute ke/atom, 2152
compute ke/atom/eff, 2154
compute ke/atom/omp, 2152
compute ke/eff, 2155
compute ke/rigid, 2156
compute mliap, 2157
compute momentum, 2159
compute msd, 2160
compute msd/chunk, 2161
compute msd/nongauss, 2163
compute nbond/atom, 2164
compute omega/chunk, 2165
compute orientorder/atom, 2167
compute orientorder/atom/kk, 2167
compute orientorder/atom/omp, 2167
compute pace, 2169
compute pair, 2173
compute pair/local, 2174
compute pe, 2176
compute pe/atom, 2178
compute pe/mol/tally, 2260
compute pe/tally, 2260
compute plasticity/atom, 2179
compute pod/atom, 2180
compute pod/global, 2180
compute pod/local, 2180
compute podd/atom, 2180
compute pressure, 2182
compute pressure/alchemy, 2184
compute pressure/uef, 2185
compute property/atom, 2186
compute property/chunk, 2190
compute property/grid, 2192
compute property/local, 2193
compute ptm/atom, 2196
compute rattlers/atom, 2198
compute rdf, 2199
compute rdf/omp, 2199
compute reaxff/atom, 2202
compute reaxff/atom/kk, 2202
compute reduce, 2204
compute reduce/chunk, 2207
compute reduce/region, 2204
compute rheo/property/atom, 2210
compute rigid/local, 2211
compute saed, 2214
compute slcsa/atom, 2218
compute slice, 2220
compute smd/contact/radius, 2222

compute smd/damage, 2223
compute smd/hourglass/error, 2223
compute smd/internal/energy, 2224
compute smd/plastic/strain, 2225
compute smd/plastic/strain/rate, 2226
compute smd/rho, 2226
compute smd/tlsph/defgrad, 2227
compute smd/tlsph/dt, 2228
compute smd/tlsph/num/neighs, 2229
compute smd/tlsph/shape, 2230
compute smd/tlsph/strain, 2230
compute smd/tlsph/strain/rate, 2231
compute smd/tlsph/stress, 2232
compute smd/triangle/vertices, 2233
compute smd/ulsph/effm, 2234
compute smd/ulsph/num/neighs, 2235
compute smd/ulsph/strain, 2236
compute smd/ulsph/strain/rate, 2236
compute smd/ulsph/stress, 2237
compute smd/vol, 2238
compute sna/atom, 2239
compute sna/grid, 2239
compute sna/grid/kk, 2239
compute sna/grid/local, 2239
compute sna/grid/local/kk, 2239
compute snad/atom, 2239
compute snap, 2239
compute snav/atom, 2239
compute sph/e/atom, 2247
compute sph/rho/atom, 2248
compute sph/t/atom, 2249
compute spin, 2250
compute stress/atom, 2251
compute stress/cartesian, 2255
compute stress/cylinder, 2256
compute stress/mop, 2258
compute stress/mop/profile, 2258
compute stress/spherical, 2256
compute stress/tally, 2260
compute tdpd/cc/atom, 2263
compute temp, 2264
compute temp/asphere, 2265
compute temp/body, 2268
compute temp/chunk, 2270
compute temp/com, 2273
compute temp/com/kk, 2273
compute temp/cs, 2275
compute temp/deform, 2276
compute temp/deform/eff, 2279
compute temp/deform/kk, 2276
compute temp/drude, 2280
compute temp/eff, 2282
compute temp/kk, 2264
compute temp/partial, 2283

compute temp/profile, 2284
compute temp/ramp, 2287
compute temp/region, 2288
compute temp/region/eff, 2290
compute temp/rotate, 2291
compute temp/sphere, 2292
compute temp/sphere/kk, 2292
compute temp/uef, 2294
compute ti, 2295
compute torque/chunk, 2297
compute vacf, 2298
compute vacf/chunk, 2299
compute vcm/chunk, 2301
compute viscosity/cos, 2302
compute voronoi/atom, 2304
compute xrd, 2307
compute_modify, 1018
config_accelerator() (*fortran function*), **677**
config_has_exceptions() (*fortran function*), **676**
config_has_ffmpeg_support() (*fortran function*), **675**
config_has_gzip_support() (*fortran function*), **674**
config_has_jpeg_support() (*fortran function*), **675**
config_has_mpi_support() (*fortran function*), **674**
config_has_omp_support() (*fortran function*), **674**
config_has_package() (*fortran function*), **676**
config_has_png_support() (*fortran function*), **675**
config_package_count() (*fortran function*), **676**
config_package_name() (*fortran subroutine*), **676**
create_atoms, 1019
create_atoms() (*fortran function*), **671**
create_atoms() (*lammps.lammps method*), 720
create_bonds, 1028
create_box, 1031
create_molecule() (*fortran subroutine*), **671**
create_molecule() (*lammps.lammps method*), 720

## D

darray() (*lammps.numpy_wrapper.numpy_wrapper method*), 734
decode_image_flags() (*fortran subroutine*), **681**
decode_image_flags() (*lammps.lammps method*), 720
delete_atoms, 1034
delete_bonds, 1037
dielectric, 1039
dihedral_coeff, 1040
dihedral_style, 1042
dihedral_style charmm, 2881
dihedral_style charmm/intel, 2881
dihedral_style charmm/kk, 2881
dihedral_style charmm/omp, 2881
dihedral_style charmmfsw, 2881
dihedral_style charmmfsw/kk, 2881
dihedral_style charmmfsw/omp, 2881
dihedral_style class2, 2883
dihedral_style class2/kk, 2883

dihedral_style class2/omp, 2883
dihedral_style class2xe, 2883
dihedral_style class2xe/kk, 2883
dihedral_style class2xe/omp, 2883
dihedral_style cosine/shift/exp, 2887
dihedral_style cosine/shift/exp/kk, 2887
dihedral_style cosine/shift/exp/omp, 2887
dihedral_style cosine/squared/restricted, 2889
dihedral_style cosine/squared/restricted/kk, 2889
dihedral_style cosine/squared/restricted/omp, 2889
dihedral_style fourier, 2890
dihedral_style fourier/intel, 2890
dihedral_style fourier/kk, 2890
dihedral_style fourier/omp, 2890
dihedral_style harmonic, 2891
dihedral_style harmonic/intel, 2891
dihedral_style harmonic/kk, 2891
dihedral_style harmonic/omp, 2891
dihedral_style helix, 2892
dihedral_style helix/kk, 2892
dihedral_style helix/omp, 2892
dihedral_style hybrid, 2893
dihedral_style hybrid/kk, 2893
dihedral_style lepton, 2895
dihedral_style lepton/omp, 2895
dihedral_style multi/harmonic, 2897
dihedral_style multi/harmonic/kk, 2897
dihedral_style multi/harmonic/omp, 2897
dihedral_style nharmonic, 2898
dihedral_style nharmonic/kk, 2898
dihedral_style nharmonic/omp, 2898
dihedral_style none, 2899
dihedral_style opls, 2900
dihedral_style opls/intel, 2900
dihedral_style opls/kk, 2900
dihedral_style opls/omp, 2900
dihedral_style quadratic, 2901
dihedral_style quadratic/kk, 2901
dihedral_style quadratic/omp, 2901
dihedral_style spherical, 2902
dihedral_style spherical/kk, 2902
dihedral_style spherical/omp, 2902
dihedral_style table, 2904
dihedral_style table/cut, 2904
dihedral_style table/cut/omp, 2904
dihedral_style table/omp, 2904
dihedral_style zero, 2908
dihedral_write, 1044
dimension, 1045
displace_atoms, 1046
dump, 2933
dump atom, 2933
dump atom/adios, 2946
dump atom/gz, 2933

dump atom/zstd, 2933
dump cfg, 2933
dump cfg/gz, 2933
dump cfg/uef, 2947
dump cfg/zstd, 2933
dump custom, 2933
dump custom/adios, 2946
dump custom/gz, 2933
dump custom/zstd, 2933
dump dcd, 2933
dump extxyz, 2933
dump grid, 2933
dump grid/vtk, 2933
dump h5md, 2948
dump image, 2949
dump local, 2933
dump local/gz, 2933
dump local/zstd, 2933
dump molfile, 2983
dump movie, 2949
dump netcdf, 2985
dump netcdf/mpiio, 2985
dump vtk, 2986
dump xtc, 2933
dump xyz, 2933
dump xyz/gz, 2933
dump xyz/zstd, 2933
dump yaml, 2933
dump_modify, 2970
dynamical_matrix, 1048
dynamical_matrix/kk, 1048

## E

echo, 1050
encode_image_flags() (*fortran function*), **681**
encode_image_flags() (*lammps.lammps method*), 719
error() (*fortran subroutine*), **646**
error() (*lammps.lammps method*), 710
eval() (*fortran function*), **661**
eval() (*lammps.lammps method*), 718
expand() (*lammps.lammps method*), 711
extract_atom() (*fortran function*), **653**
extract_atom() (*lammps.lammps method*), 715
extract_atom() (*lammps.numpy_wrapper.numpy_wrapper method*), 730
extract_atom_datatype() (*lammps.lammps method*), 714
extract_atom_size() (*lammps.lammps method*), 715
extract_box() (*fortran subroutine*), **649**
extract_box() (*lammps.lammps method*), 712
extract_compute() (*fortran function*), **654**
extract_compute() (*lammps.lammps method*), 716
extract_compute() (*lammps.numpy_wrapper.numpy_wrapper method*), 730
extract_fix() (*fortran function*), **656**
extract_fix() (*lammps.lammps method*), 716
extract_fix() (*lammps.numpy_wrapper.numpy_wrapper method*), 731

extract_global() (*fortran function*), **651**
extract_global() (*lammps.lammps method*), 713
extract_global_datatype() (*lammps.lammps method*), 713
extract_pair() (*lammps.lammps method*), 714
extract_pair_dimension() (*lammps.lammps method*), 714
extract_setting() (*fortran function*), **651**
extract_setting() (*lammps.lammps method*), 713
extract_variable() (*fortran function*), **659**
extract_variable() (*lammps.lammps method*), 717
extract_variable() (*lammps.numpy_wrapper.numpy_wrapper method*), 731

## F

file() (*fortran subroutine*), **646**
file() (*lammps.lammps method*), 711
finalize() (*lammps.lammps method*), 710
find() (*lammps.NeighList method*), 737
find() (*lammps.numpy_wrapper.NumPyNeighList method*), 737
find_compute_neighlist() (*fortran function*), **673**
find_compute_neighlist() (*lammps.lammps method*), 729
find_fix_neighlist() (*fortran function*), **672**
find_fix_neighlist() (*lammps.lammps method*), 729
find_pair_neighlist() (*fortran function*), **672**
find_pair_neighlist() (*lammps.lammps method*), 728
fitpod, 1064
fix, 1050
fix accelerate/cos, 1383
fix acks2/reaxff, 1384
fix acks2/reaxff/kk, 1384
fix adapt, 1386
fix adapt/fep, 1394
fix add/heat, 1398
fix addforce, 1400
fix addforce/kk, 1400
fix addtorque/atom, 1402
fix addtorque/group, 1404
fix alchemy, 1406
fix align/self, 1408
fix amoeba/bitorsion, 1409
fix amoeba/pitorsion, 1412
fix append/atoms, 1414
fix atom/swap, 1416
fix atom_weight/apip, 1419
fix ave/atom, 1421
fix ave/chunk, 1423
fix ave/correlate, 1430
fix ave/correlate/long, 1435
fix ave/grid, 1437
fix ave/histo, 1444
fix ave/histo/weight, 1444
fix ave/moments, 1449
fix ave/time, 1453
fix aveforce, 1458
fix aveforce/kk, 1458
fix balance, 1459

fix baoab, 1465
fix bocs, 1467
fix bond/break, 1469
fix bond/create, 1472
fix bond/create/angle, 1472
fix bond/react, 1476
fix bond/swap, 1489
fix box/relax, 1492
fix brownian, 1497
fix brownian/asphere, 1497
fix brownian/sphere, 1497
fix charge/regulation, 1501
fix cmap, 1505
fix cmap/kk, 1505
fix colvars, 1507
fix colvars/kk, 1507
fix controller, 1510
fix damping/cundall, 1513
fix deform, 1515
fix deform/kk, 1515
fix deform/pressure, 1524
fix deposit, 1529
fix dpd/energy, 1534
fix dpd/energy/kk, 1534
fix drag, 1536
fix drag/kk, 1536
fix drude, 1538
fix drude/transform/direct, 1539
fix drude/transform/inverse, 1539
fix dt/reset, 1541
fix dt/reset/kk, 1541
fix edpd/source, 1535
fix efield, 1543
fix efield/kk, 1543
fix efield/lepton, 1546
fix efield/tip4p, 1543
fix ehex, 1549
fix electrode/conp, 1552
fix electrode/conp/intel, 1552
fix electrode/conq, 1552
fix electrode/conq/intel, 1552
fix electrode/thermo, 1552
fix electrode/thermo/intel, 1552
fix electron/stopping, 1558
fix electron/stopping/fit, 1558
fix electron/stopping/kk, 1558
fix enforce2d, 1561
fix enforce2d/kk, 1561
fix eos/cv, 1562
fix eos/table, 1563
fix eos/table/rx, 1564
fix eos/table/rx/kk, 1564
fix evaporate, 1567
fix external, 1569

fix external/kk, 1569
fix ffl, 1572
fix filter/corotate, 1573
fix flow/gauss, 1574
fix freeze, 1577
fix freeze/kk, 1577
fix gcmc, 1578
fix gemc, 1584
fix gjf, 1587
fix gld, 1589
fix gle, 1592
fix graphics/arrows, 1594
fix graphics/chunk, 1597
fix graphics/isosurface, 1600
fix graphics/labels, 1604
fix graphics/lines, 1612
fix graphics/objects, 1614
fix graphics/periodic, 1618
fix graphics/replica, 1619
fix gravity, 1622
fix gravity/kk, 1622
fix gravity/omp, 1622
fix grem, 1624
fix halt, 1626
fix heat, 1629
fix heat/flow, 1631
fix hmc, 1632
fix hyper/global, 1635
fix hyper/local, 1638
fix ilves, 1645
fix imd, 1650
fix indent, 1652
fix ipi, 1656
fix lambda/apip, 1658
fix lambda/la/csp/apip, 1661
fix lambda_thermostat/apip, 1664
fix langevin, 1666
fix langevin/drude, 1670
fix langevin/eff, 1674
fix langevin/kk, 1666
fix langevin/spin, 1676
fix lb/fluid, 1677
fix lb/momentum, 1683
fix lb/viscous, 1684
fix lineforce, 1685
fix lineforce/kk, 1685
fix manifoldforce, 1686
fix mdi/qm, 1687
fix mdi/qmmm, 1691
fix meso/move, 1694
fix mol/swap, 1698
fix momentum, 1700
fix momentum/chunk, 1700
fix momentum/kk, 1700

fix move, 1702
fix msst, 1706
fix mvv/dpd, 1708
fix mvv/edpd, 1708
fix mvv/tdpd, 1708
fix neb, 1710
fix neb/spin, 1713
fix neighbor/swap, 1714
fix nonaffine/displacement, 1733
fix nph, 1718
fix nph/asphere, 1735
fix nph/asphere/omp, 1735
fix nph/body, 1737
fix nph/eff, 1728
fix nph/kk, 1718
fix nph/omp, 1718
fix nph/sphere, 1739
fix nph/sphere/kk, 1739
fix nph/sphere/omp, 1739
fix nphug, 1741
fix nphug/omp, 1741
fix npt, 1718
fix npt/asphere, 1744
fix npt/asphere/omp, 1744
fix npt/body, 1746
fix npt/cauchy, 1748
fix npt/eff, 1728
fix npt/gpu, 1718
fix npt/intel, 1718
fix npt/kk, 1718
fix npt/omp, 1718
fix npt/sphere, 1756
fix npt/sphere/kk, 1756
fix npt/sphere/omp, 1756
fix npt/uef, 1730
fix numdiff, 1758
fix numdiff/virial, 1760
fix nve, 1761
fix nve/asphere, 1762
fix nve/asphere/gpu, 1762
fix nve/asphere/intel, 1762
fix nve/asphere/kk, 1762
fix nve/asphere/noforce, 1764
fix nve/body, 1764
fix nve/bpm/sphere, 1765
fix nve/dot, 1767
fix nve/dotc/langevin, 1767
fix nve/eff, 1770
fix nve/gpu, 1761
fix nve/intel, 1761
fix nve/kk, 1761
fix nve/limit, 1770
fix nve/limit/kk, 1770
fix nve/line, 1772

fix nve/manifold/rattle, 1773
fix nve/noforce, 1774
fix nve/noforce/kk, 1774
fix nve/omp, 1761
fix nve/sphere, 1775
fix nve/sphere/kk, 1775
fix nve/sphere/omp, 1775
fix nve/spin, 1777
fix nve/tri, 1778
fix nvk, 1779
fix nvt, 1718
fix nvt/asphere, 1780
fix nvt/asphere/omp, 1780
fix nvt/body, 1782
fix nvt/eff, 1728
fix nvt/gpu, 1718
fix nvt/intel, 1718
fix nvt/kk, 1718
fix nvt/manifold/rattle, 1784
fix nvt/omp, 1718
fix nvt/sllod, 1785
fix nvt/sllod/eff, 1789
fix nvt/sllod/intel, 1785
fix nvt/sllod/kk, 1785
fix nvt/sllod/omp, 1785
fix nvt/sphere, 1790
fix nvt/sphere/kk, 1790
fix nvt/sphere/omp, 1790
fix nvt/uef, 1730
fix oneway, 1792
fix oneway/kk, 1792
fix orient/bcc, 1794
fix orient/bcc/omp, 1794
fix orient/eco, 1797
fix orient/eco/omp, 1797
fix orient/fcc, 1794
fix orient/fcc/omp, 1794
fix pafi, 1799
fix pair, 1801
fix phonon, 1803
fix pimd/langevin, 1805
fix pimd/langevin/bosonic, 1805
fix pimd/nvt, 1805
fix pimd/nvt/bosonic, 1805
fix planeforce, 1815
fix planeforce/kk, 1815
fix plumed, 1816
fix polarize/bem/gmres, 1817
fix polarize/bem/icc, 1817
fix polarize/functional, 1817
fix pour, 1821
fix precession/spin, 1824
fix press/berendsen, 1827
fix press/langevin, 1830

fix print, 1834
fix propel/self, 1836
fix property/atom, 1838
fix property/atom/kk, 1838
fix python/invoke, 1842
fix python/move, 1844
fix qbmsst, 1845
fix qeq/comb, 1854
fix qeq/comb/omp, 1854
fix qeq/ctip, 1849
fix qeq/ctip/omp, 1849
fix qeq/dynamic, 1849
fix qeq/dynamic/omp, 1849
fix qeq/fire, 1849
fix qeq/fire/omp, 1849
fix qeq/point, 1849
fix qeq/point/omp, 1849
fix qeq/reaxff, 1856
fix qeq/reaxff/kk, 1856
fix qeq/reaxff/omp, 1856
fix qeq/rel/reaxff, 1858
fix qeq/shielded, 1849
fix qeq/shielded/omp, 1849
fix qeq/slater, 1849
fix qeq/slater/omp, 1849
fix qmmm, 1861
fix qtb, 1862
fix qtpie/reaxff, 1864
fix rattle, 1920
fix reaxff/bonds, 1867
fix reaxff/bonds/kk, 1867
fix reaxff/species, 1869
fix reaxff/species/kk, 1869
fix recenter, 1873
fix recenter/kk, 1873
fix restrain, 1875
fix rheo, 1879
fix rheo/oxidation, 1882
fix rheo/pressure, 1883
fix rheo/thermal, 1885
fix rheo/viscosity, 1886
fix rhok, 1888
fix rigid, 1889
fix rigid/meso, 1901
fix rigid/nph, 1889
fix rigid/nph/omp, 1889
fix rigid/nph/small, 1889
fix rigid/npt, 1889
fix rigid/npt/omp, 1889
fix rigid/npt/small, 1889
fix rigid/nve, 1889
fix rigid/nve/omp, 1889
fix rigid/nve/small, 1889
fix rigid/nvt, 1889

fix rigid/nvt/omp, 1889
fix rigid/nvt/small, 1889
fix rigid/omp, 1889
fix rigid/small, 1889
fix rigid/small/omp, 1889
fix rx, 1905
fix rx/kk, 1905
fix saed/vtk, 1909
fix set, 1911
fix setforce, 1914
fix setforce/kk, 1914
fix setforce/spin, 1914
fix settorque/atom, 1916
fix sgcmc, 1917
fix shake, 1920
fix shake/kk, 1920
fix shardlow, 1924
fix shardlow/kk, 1924
fix smd, 1926
fix smd/adjust_dt, 1928
fix smd/integrate_tlsph, 1929
fix smd/integrate_ulsph, 1930
fix smd/move_tri_surf, 1931
fix smd/setvel, 1932
fix smd/wall_surface, 1933
fix sph, 1935
fix sph/stationary, 1936
fix spring, 1936
fix spring/chunk, 1939
fix spring/kk, 1936
fix spring/rg, 1940
fix spring/self, 1942
fix spring/self/kk, 1942
fix srd, 1943
fix store/force, 1948
fix store/state, 1950
fix surface/global, 1953
fix surface/local, 1958
fix tdpd/source, 1535
fix temp/berendsen, 1961
fix temp/berendsen/kk, 1961
fix temp/csld, 1964
fix temp/csvr, 1964
fix temp/rescale, 1966
fix temp/rescale/eff, 1969
fix temp/rescale/kk, 1966
fix tfmc, 1970
fix tgnpt/drude, 1972
fix tgnvt/drude, 1972
fix thermal/conductivity, 1976
fix ti/spring, 1978
fix tmd, 1980
fix ttm, 1982
fix ttm/grid, 1982

fix ttm/mod, 1982
fix ttm/thermal, 1982
fix tune/kspace, 1989
fix vector, 1991
fix viscosity, 1993
fix viscous, 1995
fix viscous/kk, 1995
fix viscous/nonlinear, 1997
fix viscous/sphere, 1998
fix viscous/sphere/kk, 1998
fix wall/body/polygon, 2009
fix wall/body/polyhedron, 2011
fix wall/colloid, 2000
fix wall/ees, 2012
fix wall/flow, 2015
fix wall/flow/kk, 2015
fix wall/gran, 2018
fix wall/gran/kk, 2018
fix wall/gran/region, 2022
fix wall/harmonic, 2000
fix wall/harmonic/kk, 2000
fix wall/harmonic/outside, 2000
fix wall/lepton, 2000
fix wall/lj1043, 2000
fix wall/lj1043/kk, 2000
fix wall/lj126, 2000
fix wall/lj126/kk, 2000
fix wall/lj93, 2000
fix wall/lj93/kk, 2000
fix wall/morse, 2000
fix wall/morse/kk, 2000
fix wall/piston, 2027
fix wall/reflect, 2028
fix wall/reflect/kk, 2028
fix wall/reflect/stochastic, 2031
fix wall/region, 2033
fix wall/region/ees, 2012
fix wall/region/kk, 2033
fix wall/srd, 2037
fix wall/table, 2000
fix widom, 2040
fix_external_get_force() (*fortran function*), **683**
fix_external_get_force() (*lammps.lammps method*), 726
fix_external_get_force() (*lammps.numpy_wrapper.numpy_wrapper method*), 732
fix_external_set_energy_global() (*fortran subroutine*), **684**
fix_external_set_energy_global() (*lammps.lammps method*), 726
fix_external_set_energy_peratom() (*fortran subroutine*), **685**
fix_external_set_energy_peratom() (*lammps.lammps method*), 727
fix_external_set_energy_peratom() (*lammps.numpy_wrapper.numpy_wrapper method*), 733
fix_external_set_vector() (*fortran subroutine*), **686**
fix_external_set_vector() (*lammps.lammps method*), 727
fix_external_set_vector_length() (*fortran subroutine*), **686**
fix_external_set_vector_length() (*lammps.lammps method*), 727
fix_external_set_virial_global() (*fortran subroutine*), **684**

fix_external_set_virial_global() (*lammps.lammps method*), 726
fix_external_set_virial_peratom() (*lammps.lammps method*), 727
fix_external_set_virial_peratom() (*lammps.numpy_wrapper.numpy_wrapper method*), 733
fix_modify, 1061
flush_buffers() (*fortran subroutine*), **687**
flush_buffers() (*lammps.lammps method*), 717
force_timeout() (*fortran subroutine*), **687**
force_timeout() (*lammps.lammps method*), 722

## G

gather() (*fortran subroutine*), **666**
gather_angles() (*fortran subroutine*), **665**
gather_angles() (*lammps.lammps method*), 719
gather_angles() (*lammps.numpy_wrapper.numpy_wrapper method*), 732
gather_atoms() (*fortran subroutine*), **661**
gather_atoms_concat() (*fortran subroutine*), **662**
gather_atoms_subset() (*fortran subroutine*), **663**
gather_bonds() (*fortran subroutine*), **664**
gather_bonds() (*lammps.lammps method*), 718
gather_bonds() (*lammps.numpy_wrapper.numpy_wrapper method*), 731
gather_concat() (*fortran subroutine*), **668**
gather_dihedrals() (*fortran subroutine*), **667**
gather_dihedrals() (*lammps.lammps method*), 719
gather_dihedrals() (*lammps.numpy_wrapper.numpy_wrapper method*), 732
gather_impropers() (*fortran subroutine*), **667**
gather_impropers() (*lammps.lammps method*), 719
gather_impropers() (*lammps.numpy_wrapper.numpy_wrapper method*), 732
gather_subset() (*fortran subroutine*), **669**
get() (*lammps.NeighList method*), 737
get() (*lammps.numpy_wrapper.NumPyNeighList method*), 737
get_gpu_device_info() (*fortran subroutine*), **678**
get_gpu_device_info() (*lammps.lammps method*), 723
get_last_error_message() (*fortran subroutine*), **688**
get_mpi_comm() (*fortran function*), **650**
get_mpi_comm() (*lammps.lammps method*), 710
get_natoms() (*fortran function*), **647**
get_natoms() (*lammps.lammps method*), 711
get_neighlist() (*lammps.lammps method*), 728
get_neighlist() (*lammps.numpy_wrapper.numpy_wrapper method*), 733
get_neighlist_element_neighbors() (*lammps.lammps method*), 728
get_neighlist_element_neighbors() (*lammps.numpy_wrapper.numpy_wrapper method*), 733
get_neighlist_size() (*lammps.lammps method*), 728
get_os_info() (*fortran subroutine*), **674**
get_os_info() (*lammps.lammps method*), 710
get_thermo() (*fortran function*), **647**
get_thermo() (*lammps.lammps method*), 712
geturl, 1068
group, 1070
group2ndx, 1075

## H

has_curl_support (*lammps.lammps property*), 723
has_error() (*fortran function*), **688**
has_exceptions (*lammps.lammps property*), 722

has_ffmpeg_support (*lammps.lammps property*), 722
has_gpu_device (*lammps.lammps property*), 723
has_gpu_device() (*fortran function*), **678**
has_gzip_support (*lammps.lammps property*), 722
has_id() (*fortran function*), **679**
has_id() (*lammps.lammps method*), 724
has_jpeg_support (*lammps.lammps property*), 722
has_mpi_support (*lammps.lammps property*), 721
has_omp_support (*lammps.lammps property*), 721
has_package() (*lammps.lammps method*), 723
has_png_support (*lammps.lammps property*), 722
has_style() (*fortran function*), **678**
has_style() (*lammps.lammps method*), 724
hyper, 1076

## I

iarray() (*lammps.numpy_wrapper.numpy_wrapper method*), 734
id_count() (*fortran function*), **679**
id_name() (*fortran subroutine*), **680**
if, 1079
image() (*lammps.ipython.wrapper method*), 735
improper_coeff, 1082
improper_style, 1083
improper_style amoeba, 2911
improper_style class2, 2912
improper_style class2/kk, 2912
improper_style class2/omp, 2912
improper_style cossq, 2914
improper_style cossq/kk, 2914
improper_style cossq/omp, 2914
improper_style cvff, 2915
improper_style cvff/intel, 2915
improper_style cvff/kk, 2915
improper_style cvff/omp, 2915
improper_style distance, 2916
improper_style distance/kk, 2916
improper_style distance/omp, 2916
improper_style distharm, 2918
improper_style distharm/kk, 2918
improper_style distharm/omp, 2918
improper_style fourier, 2919
improper_style fourier/kk, 2919
improper_style fourier/omp, 2919
improper_style harmonic, 2921
improper_style harmonic/intel, 2921
improper_style harmonic/kk, 2921
improper_style harmonic/omp, 2921
improper_style hybrid, 2922
improper_style hybrid/kk, 2922
improper_style inversion/harmonic, 2924
improper_style inversion/harmonic/kk, 2924
improper_style inversion/harmonic/omp, 2924
improper_style none, 2925
improper_style ring, 2926

improper_style ring/kk, 2926
improper_style ring/omp, 2926
improper_style sqdistharm, 2927
improper_style sqdistharm/kk, 2927
improper_style sqdistharm/omp, 2927
improper_style umbrella, 2928
improper_style umbrella/kk, 2928
improper_style umbrella/omp, 2928
improper_style zero, 2930
include, 1085
info, 1085
Input (*C++ class*), 901
Input::file (*C++ function*), 901
Input::Input (*C++ function*), 901
Input::one (*C++ function*), 901
installed_packages (*lammps.lammps property*), 724
installed_packages() (*fortran subroutine*), **677**
InvalidFloatException (*C++ class*), 937
InvalidFloatException::InvalidFloatException (*C++ function*), 937
InvalidIntegerException (*C++ class*), 937
InvalidIntegerException::InvalidIntegerException (*C++ function*), 937
ipython (*lammps.lammps property*), 710
is_running (*lammps.lammps property*), 721
is_running() (*fortran function*), **687**

## J

jump, 1087

## K

kim_commands, 1089
kspace_modify, 1108
kspace_style esp, 1116
kspace_style ewald, 1116
kspace_style ewald/dipole, 1116
kspace_style ewald/dipole/spin, 1116
kspace_style ewald/disp, 1116
kspace_style ewald/disp/dipole, 1116
kspace_style ewald/electrode, 1116
kspace_style ewald/gpu, 1116
kspace_style ewald/omp, 1116
kspace_style msm, 1116
kspace_style msm/cg, 1116
kspace_style msm/cg/omp, 1116
kspace_style msm/dielectric, 1116
kspace_style msm/omp, 1116
kspace_style pppm, 1116
kspace_style pppm/cg, 1116
kspace_style pppm/cg/omp, 1116
kspace_style pppm/dielectric, 1116
kspace_style pppm/dipole, 1116
kspace_style pppm/dipole/spin, 1116
kspace_style pppm/disp, 1116
kspace_style pppm/disp/dielectric, 1116
kspace_style pppm/disp/intel, 1116

kspace_style pppm/disp/omp, 1116
kspace_style pppm/disp/tip4p, 1116
kspace_style pppm/disp/tip4p/omp, 1116
kspace_style pppm/electrode, 1116
kspace_style pppm/electrode/intel, 1116
kspace_style pppm/gpu, 1116
kspace_style pppm/intel, 1116
kspace_style pppm/kk, 1116
kspace_style pppm/omp, 1116
kspace_style pppm/rk, 1116
kspace_style pppm/stagger, 1116
kspace_style pppm/tip4p, 1116
kspace_style pppm/tip4p/omp, 1116
kspace_style scafacos, 1116
kspace_style zero, 1116

## L

label, 1125
labelmap, 1126
LabelMap (*C++ class*), 948
LabelMap::check_labels (*C++ function*), 951
LabelMap::checkflag (*C++ member*), 953
LabelMap::create_lmap2lmap (*C++ function*), 952
LabelMap::find_label (*C++ function*), 952
LabelMap::find_type (*C++ function*), 952
LabelMap::infer_angletype (*C++ function*), 949
LabelMap::infer_bondtype (*C++ function*), 948, 949
LabelMap::infer_dihedraltype (*C++ function*), 949, 950
LabelMap::infer_impropertype (*C++ function*), 950
LabelMap::is_complete (*C++ function*), 952
LabelMap::LabelMap (*C++ function*), 951
LabelMap::merge_lmap (*C++ function*), 952
LabelMap::modify_lmap (*C++ function*), 951
LabelMap::parse_typelabel (*C++ function*), 953
LabelMap::read_restart (*C++ function*), 951
LabelMap::write_data (*C++ function*), 951
LabelMap::write_restart (*C++ function*), 951
lammps
    module, 708
LAMMPS (*C++ class*), 894
lammps (*class in lammps*), 708
lammps (*fortran type*), **642**
lammps() (*fortran function*), **645**
lammps.formats
    module, 739
LAMMPS::~LAMMPS (*C++ function*), 895
LAMMPS::argv_pointers (*C++ function*), 896
LAMMPS::LAMMPS (*C++ function*), 895
LAMMPS::match_style (*C++ function*), 895
LAMMPS::non_pair_suffix (*C++ function*), 895
lammps_addstep_compute (*C++ function*), 603
lammps_addstep_compute_all (*C++ function*), 603
lammps_clearstep_compute (*C++ function*), 603
lammps_close (*C++ function*), 575

lammps_command (*C++ function*), 578
lammps_commands_list (*C++ function*), 579
lammps_commands_string (*C++ function*), 579
lammps_config_accelerator (*C++ function*), 626
lammps_config_has_exceptions (*C++ function*), 625
lammps_config_has_ffmpeg_support (*C++ function*), 625
lammps_config_has_gzip_support (*C++ function*), 624
lammps_config_has_jpeg_support (*C++ function*), 624
lammps_config_has_mpi_support (*C++ function*), 624
lammps_config_has_omp_support (*C++ function*), 624
lammps_config_has_package (*C++ function*), 625
lammps_config_has_png_support (*C++ function*), 624
lammps_config_package_count (*C++ function*), 625
lammps_config_package_name (*C++ function*), 626
lammps_create_atoms (*C++ function*), 618
lammps_create_molecule (*C++ function*), 619
lammps_decode_image_flags (*C++ function*), 630
lammps_encode_image_flags (*C++ function*), 630
lammps_error (*C++ function*), 577
lammps_eval (*C++ function*), 602
lammps_expand (*C++ function*), 579
lammps_extract_atom (*C++ function*), 595
lammps_extract_atom_datatype (*C++ function*), 594
lammps_extract_atom_size (*C++ function*), 594
lammps_extract_box (*C++ function*), 583
lammps_extract_compute (*C++ function*), 596
lammps_extract_fix (*C++ function*), 597
lammps_extract_global (*C++ function*), 589
lammps_extract_global_datatype (*C++ function*), 588
lammps_extract_pair (*C++ function*), 593
lammps_extract_pair_dimension (*C++ function*), 593
lammps_extract_setting (*C++ function*), 585
lammps_extract_variable (*C++ function*), 599
lammps_extract_variable_datatype (*C++ function*), 599
lammps_file (*C++ function*), 578
lammps_find_compute_neighlist (*C++ function*), 620
lammps_find_fix_neighlist (*C++ function*), 620
lammps_find_pair_neighlist (*C++ function*), 620
lammps_fix_external_get_force (*C++ function*), 631
lammps_fix_external_set_energy_global (*C++ function*), 632
lammps_fix_external_set_energy_peratom (*C++ function*), 632
lammps_fix_external_set_vector (*C++ function*), 634
lammps_fix_external_set_vector_length (*C++ function*), 634
lammps_fix_external_set_virial_global (*C++ function*), 633
lammps_fix_external_set_virial_peratom (*C++ function*), 633
lammps_flush_buffers (*C++ function*), 635
lammps_force_timeout (*C++ function*), 635
lammps_free (*C++ function*), 635
lammps_gather (*C++ function*), 615
lammps_gather_angles (*C++ function*), 611
lammps_gather_atoms (*C++ function*), 606
lammps_gather_atoms_concat (*C++ function*), 607
lammps_gather_atoms_subset (*C++ function*), 607
lammps_gather_bonds (*C++ function*), 609

lammps_gather_concat (*C++ function*), 616
lammps_gather_dihedrals (*C++ function*), 612
lammps_gather_impropers (*C++ function*), 613
lammps_gather_subset (*C++ function*), 616
lammps_get_gpu_device_info (*C++ function*), 627
lammps_get_last_error_message (*C++ function*), 636
lammps_get_mpi_comm (*C++ function*), 584
lammps_get_natoms (*C++ function*), 581
lammps_get_os_info (*C++ function*), 623
lammps_get_thermo (*C++ function*), 582
lammps_has_error (*C++ function*), 636
lammps_has_gpu_device (*C++ function*), 626
lammps_has_id (*C++ function*), 628
lammps_has_style (*C++ function*), 627
lammps_id_count (*C++ function*), 628
lammps_id_name (*C++ function*), 629
lammps_is_running (*C++ function*), 635
lammps_kokkos_finalize (*C++ function*), 576
lammps_last_thermo (*C++ function*), 582
lammps_map_atom (*C++ function*), 593
lammps_memory_usage (*C++ function*), 584
lammps_mpi_finalize (*C++ function*), 576
lammps_mpi_init (*C++ function*), 575
lammps_neighlist_element_neighbors (*C++ function*), 621
lammps_neighlist_num_elements (*C++ function*), 621
lammps_open (*C++ function*), 573
lammps_open_fortran (*C++ function*), 575
lammps_open_no_mpi (*C++ function*), 574
lammps_plugin_finalize (*C++ function*), 577
lammps_python_api_version (*C++ function*), 637
lammps_python_finalize (*C++ function*), 576
lammps_reset_box (*C++ function*), 584
lammps_scatter (*C++ function*), 617
lammps_scatter_atoms (*C++ function*), 608
lammps_scatter_atoms_subset (*C++ function*), 609
lammps_scatter_subset (*C++ function*), 618
lammps_set_fix_external_callback (*C++ function*), 630
lammps_set_internal_variable (*C++ function*), 601
lammps_set_show_error (*C++ function*), 637
lammps_set_string_variable (*C++ function*), 601
lammps_set_variable (*C++ function*), 600
lammps_style (*fortran type*), **645**
lammps_style_count (*C++ function*), 627
lammps_style_name (*C++ function*), 627
lammps_type (*fortran type*), **646**
lammps_variable_info (*C++ function*), 602
lammps_version (*C++ function*), 623
last_thermo() (*fortran function*), **648**
last_thermo() (*lammps.lammps method*), 712
last_thermo_step (*lammps.lammps property*), 712
lattice, 1127
LIBLAMMPS (*module*), **642**
LMP_SIZE_COLS (*C++ enumerator*), 605
LMP_SIZE_ROWS (*C++ enumerator*), 605

LMP_SIZE_VECTOR (*C++ enumerator*), 605
LMP_STYLE_ATOM (*C++ enumerator*), 604
LMP_STYLE_GLOBAL (*C++ enumerator*), 604
LMP_STYLE_LOCAL (*C++ enumerator*), 604
LMP_TYPE_ARRAY (*C++ enumerator*), 605
LMP_TYPE_SCALAR (*C++ enumerator*), 605
LMP_TYPE_VECTOR (*C++ enumerator*), 605
LMP_VAR_ATOM (*C++ enumerator*), 605
LMP_VAR_EQUAL (*C++ enumerator*), 605
LMP_VAR_STRING (*C++ enumerator*), 605
LMP_VAR_VECTOR (*C++ enumerator*), 605
log, 1132

## M

map_atom() (*lammps.lammps method*), 714
mass, 1133
MathEigen::jacobi3 (*C++ function*), 957
MathSpecial::cube (*C++ function*), 932
MathSpecial::exp2_x86 (*C++ function*), 931
MathSpecial::expmsq (*C++ function*), 931
MathSpecial::factorial (*C++ function*), 930
MathSpecial::fm_exp (*C++ function*), 931
MathSpecial::my_erfcx (*C++ function*), 931
MathSpecial::powint (*C++ function*), 932
MathSpecial::powsign (*C++ function*), 932
MathSpecial::powsinxx (*C++ function*), 932
MathSpecial::square (*C++ function*), 931
mdi, 1134
memory_usage() (*fortran subroutine*), **650**
min_modify, 1139
min_style, 1143
min_style spin, 1142
minimize, 1146
minimize/kk, 1146
module
    lammps, 708
    lammps.formats, 739
molecule, 1150
multitype::_LMP_DATATYPE_CONST (*C++ enum*), 603
multitype::_LMP_DATATYPE_CONST::LAMMPS_DOUBLE (*C++ enumerator*), 604
multitype::_LMP_DATATYPE_CONST::LAMMPS_DOUBLE_2D (*C++ enumerator*), 604
multitype::_LMP_DATATYPE_CONST::LAMMPS_INT (*C++ enumerator*), 604
multitype::_LMP_DATATYPE_CONST::LAMMPS_INT64 (*C++ enumerator*), 604
multitype::_LMP_DATATYPE_CONST::LAMMPS_INT64_2D (*C++ enumerator*), 604
multitype::_LMP_DATATYPE_CONST::LAMMPS_INT_2D (*C++ enumerator*), 604
multitype::_LMP_DATATYPE_CONST::LAMMPS_NONE (*C++ enumerator*), 604
multitype::_LMP_DATATYPE_CONST::LAMMPS_STRING (*C++ enumerator*), 604
MyPage (*C++ class*), 954
MyPage::get (*C++ function*), 955
MyPage::init (*C++ function*), 954
MyPage::MyPage (*C++ function*), 954
MyPage::reset (*C++ function*), 955
MyPage::size (*C++ function*), 955
MyPage::status (*C++ function*), 955

MyPage::vget (*C++ function*), 955
MyPage::vgot (*C++ function*), 955
MyPoolChunk (*C++ class*), 955
MyPoolChunk::~MyPoolChunk (*C++ function*), 956
MyPoolChunk::get (*C++ function*), 956
MyPoolChunk::MyPoolChunk (*C++ function*), 956
MyPoolChunk::put (*C++ function*), 956
MyPoolChunk::size (*C++ function*), 956
MyPoolChunk::status (*C++ function*), 957

## N

ndx2group, 1075
neb, 1168
neb/spin, 1174
neigh_modify, 1178
neighbor, 1183
NeighList (*class in lammps*), 736
neighlist_element_neighbors() (*fortran subroutine*), **673**
neighlist_num_elements() (*fortran function*), **673**
newton, 1184
next, 1185
numpy (*lammps.lammps property*), 709
numpy_wrapper (*class in lammps.numpy_wrapper*), 730
NumPyNeighList (*class in lammps.numpy_wrapper*), 737

## P

package, 1187
pair_coeff, 1199
pair_coeff ldd indicator dpd, 2522
pair_coeff ldd indicator lucy, 2522
pair_coeff ldd indicator shell, 2522
pair_coeff ldd indicator smooth, 2522
pair_coeff ldd indicator sphere, 2522
pair_coeff ldd potential constant, 2522
pair_coeff ldd potential linear, 2523
pair_coeff ldd potential mdpd, 2523
pair_coeff ldd potential noforce, 2522
pair_coeff ldd potential quadratic, 2523
pair_coeff ldd potential table/lin, 2523
pair_coeff ldd potential table/spline, 2523
pair_modify, 1201
pair_style, 1206
pair_style adp, 2311
pair_style adp/kk, 2311
pair_style adp/omp, 2311
pair_style agni, 2313
pair_style agni/omp, 2313
pair_style aip/water/2dm, 2315
pair_style aip/water/2dm/opt, 2315
pair_style airebo, 2318
pair_style airebo/bc, 2318
pair_style airebo/bc/omp, 2318
pair_style airebo/intel, 2318
pair_style airebo/morse, 2318

pair_style airebo/morse/intel, 2318
pair_style airebo/morse/omp, 2318
pair_style airebo/omp, 2318
pair_style amoeba, 2322
pair_style amoeba/gpu, 2322
pair_style atm, 2326
pair_style atm/omp, 2326
pair_style beck, 2329
pair_style beck/gpu, 2329
pair_style beck/kk, 2329
pair_style beck/omp, 2329
pair_style body/nparticle, 2330
pair_style body/nparticle/omp, 2330
pair_style body/rounded/polygon, 2332
pair_style body/rounded/polygon/omp, 2332
pair_style body/rounded/polyhedron, 2335
pair_style body/rounded/polyhedron/omp, 2335
pair_style bondval, 2338
pair_style bondval/kk, 2338
pair_style bondval/vec, 2338
pair_style bondval/vec/kk, 2338
pair_style bop, 2342
pair_style born, 2347
pair_style born/coul/dsf, 2347
pair_style born/coul/dsf/cs, 2392
pair_style born/coul/dsf/cs/omp, 2392
pair_style born/coul/dsf/omp, 2347
pair_style born/coul/long, 2347
pair_style born/coul/long/cs, 2392
pair_style born/coul/long/cs/gpu, 2392
pair_style born/coul/long/cs/omp, 2392
pair_style born/coul/long/gpu, 2347
pair_style born/coul/long/kk, 2347
pair_style born/coul/long/omp, 2347
pair_style born/coul/msm, 2347
pair_style born/coul/msm/omp, 2347
pair_style born/coul/wolf, 2347
pair_style born/coul/wolf/cs, 2392
pair_style born/coul/wolf/cs/gpu, 2392
pair_style born/coul/wolf/cs/omp, 2392
pair_style born/coul/wolf/gpu, 2347
pair_style born/coul/wolf/kk, 2347
pair_style born/coul/wolf/omp, 2347
pair_style born/gauss, 2350
pair_style born/gauss/kk, 2350
pair_style born/gauss/omp, 2350
pair_style born/gpu, 2347
pair_style born/kk, 2347
pair_style born/omp, 2347
pair_style bpm/spring, 2352
pair_style brownian, 2354
pair_style brownian/kk, 2354
pair_style brownian/omp, 2354
pair_style brownian/poly, 2354

pair_style brownian/poly/omp, 2354
pair_style buck, 2356
pair_style buck/coul/cut, 2356
pair_style buck/coul/cut/gpu, 2356
pair_style buck/coul/cut/intel, 2356
pair_style buck/coul/cut/kk, 2356
pair_style buck/coul/cut/omp, 2356
pair_style buck/coul/long, 2356
pair_style buck/coul/long/cs, 2392
pair_style buck/coul/long/cs/omp, 2392
pair_style buck/coul/long/gpu, 2356
pair_style buck/coul/long/intel, 2356
pair_style buck/coul/long/kk, 2356
pair_style buck/coul/long/omp, 2356
pair_style buck/coul/msm, 2356
pair_style buck/coul/msm/omp, 2356
pair_style buck/gpu, 2356
pair_style buck/intel, 2356
pair_style buck/kk, 2356
pair_style buck/long/coul/long, 2361
pair_style buck/long/coul/long/omp, 2361
pair_style buck/mdf, 2582
pair_style buck/mdf/kk, 2582
pair_style buck/mdf/omp, 2582
pair_style buck/omp, 2356
pair_style buck6d/coul/gauss/dsf, 2359
pair_style buck6d/coul/gauss/dsf/kk, 2359
pair_style buck6d/coul/gauss/dsf/omp, 2359
pair_style buck6d/coul/gauss/long, 2359
pair_style buck6d/coul/gauss/long/kk, 2359
pair_style buck6d/coul/gauss/long/omp, 2359
pair_style colloid, 2370
pair_style colloid/gpu, 2370
pair_style colloid/kk, 2370
pair_style colloid/omp, 2370
pair_style comb, 2373
pair_style comb/omp, 2373
pair_style comb3, 2373
pair_style cosine/squared, 2376
pair_style cosine/squared/kk, 2376
pair_style cosine/squared/omp, 2376
pair_style coul/ctip, 2378
pair_style coul/cut, 2378
pair_style coul/cut/dielectric, 2395
pair_style coul/cut/global, 2378
pair_style coul/cut/global/kk, 2378
pair_style coul/cut/global/omp, 2378
pair_style coul/cut/gpu, 2378
pair_style coul/cut/kk, 2378
pair_style coul/cut/omp, 2378
pair_style coul/cut/soft, 2445
pair_style coul/cut/soft/gapsys, 2445
pair_style coul/cut/soft/gapsys/omp, 2445
pair_style coul/cut/soft/omp, 2445

pair_style coul/debye, 2378
pair_style coul/debye/gpu, 2378
pair_style coul/debye/kk, 2378
pair_style coul/debye/omp, 2378
pair_style coul/diel, 2385
pair_style coul/diel/kk, 2385
pair_style coul/diel/omp, 2385
pair_style coul/dsf, 2378
pair_style coul/dsf/gpu, 2378
pair_style coul/dsf/kk, 2378
pair_style coul/dsf/omp, 2378
pair_style coul/esp, 2378
pair_style coul/exclude, 2378
pair_style coul/exclude/omp, 2378
pair_style coul/long, 2378
pair_style coul/long/cs, 2392
pair_style coul/long/cs/gpu, 2392
pair_style coul/long/cs/omp, 2392
pair_style coul/long/dielectric, 2395
pair_style coul/long/gpu, 2378
pair_style coul/long/kk, 2378
pair_style coul/long/omp, 2378
pair_style coul/long/soft, 2445
pair_style coul/long/soft/omp, 2445
pair_style coul/msm, 2378
pair_style coul/msm/omp, 2378
pair_style coul/shield, 2387
pair_style coul/shield/kk, 2387
pair_style coul/shield/omp, 2387
pair_style coul/slater, 2388
pair_style coul/slater/cut, 2388
pair_style coul/slater/cut/omp, 2388
pair_style coul/slater/long, 2388
pair_style coul/slater/long/gpu, 2388
pair_style coul/slater/long/kk, 2388
pair_style coul/slater/long/omp, 2388
pair_style coul/streitz, 2378
pair_style coul/streitz/omp, 2378
pair_style coul/tt, 2390
pair_style coul/wolf, 2378
pair_style coul/wolf/cs, 2392
pair_style coul/wolf/cs/omp, 2392
pair_style coul/wolf/kk, 2378
pair_style coul/wolf/omp, 2378
pair_style dispersion/d3, 2403
pair_style dpd, 2405
pair_style dpd/coul/slater/long, 2408
pair_style dpd/coul/slater/long/gpu, 2408
pair_style dpd/ext, 2411
pair_style dpd/ext/kk, 2411
pair_style dpd/ext/omp, 2411
pair_style dpd/ext/tstat, 2411
pair_style dpd/ext/tstat/kk, 2411
pair_style dpd/ext/tstat/omp, 2411

pair_style dpd/fdt, 2414
pair_style dpd/fdt/energy, 2414
pair_style dpd/fdt/energy/kk, 2414
pair_style dpd/gpu, 2405
pair_style dpd/intel, 2405
pair_style dpd/kk, 2405
pair_style dpd/omp, 2405
pair_style dpd/tstat, 2405
pair_style dpd/tstat/gpu, 2405
pair_style dpd/tstat/kk, 2405
pair_style dpd/tstat/omp, 2405
pair_style drip, 2416
pair_style drip/omp, 2416
pair_style dsmc, 2419
pair_style e3b, 2421
pair_style e3b/omp, 2421
pair_style eam, 2424
pair_style eam/alloy, 2424
pair_style eam/alloy/gpu, 2424
pair_style eam/alloy/intel, 2424
pair_style eam/alloy/kk, 2424
pair_style eam/alloy/omp, 2424
pair_style eam/alloy/opt, 2424
pair_style eam/apip, 2430
pair_style eam/cd, 2424
pair_style eam/cd/old, 2424
pair_style eam/fs, 2424
pair_style eam/fs/apip, 2430
pair_style eam/fs/gpu, 2424
pair_style eam/fs/intel, 2424
pair_style eam/fs/kk, 2424
pair_style eam/fs/omp, 2424
pair_style eam/fs/opt, 2424
pair_style eam/gpu, 2424
pair_style eam/he, 2424
pair_style eam/intel, 2424
pair_style eam/kk, 2424
pair_style eam/omp, 2424
pair_style eam/opt, 2424
pair_style edip, 2432
pair_style edip/multi, 2432
pair_style edip/omp, 2432
pair_style edpd, 2599
pair_style edpd/gpu, 2599
pair_style eff/cut, 2435
pair_style eim, 2439
pair_style eim/omp, 2439
pair_style exp6/rx, 2441
pair_style exp6/rx/kk, 2441
pair_style extep, 2444
pair_style extep/omp, 2444
pair_style gauss, 2454
pair_style gauss/cut, 2454
pair_style gauss/cut/kk, 2454

pair_style gauss/cut/omp, 2454
pair_style gauss/gpu, 2454
pair_style gauss/kk, 2454
pair_style gauss/omp, 2454
pair_style gayberne, 2457
pair_style gayberne/gpu, 2457
pair_style gayberne/intel, 2457
pair_style gayberne/omp, 2457
pair_style gran/hertz/history, 2460
pair_style gran/hertz/history/omp, 2460
pair_style gran/hooke, 2460
pair_style gran/hooke/history, 2460
pair_style gran/hooke/history/kk, 2460
pair_style gran/hooke/history/omp, 2460
pair_style gran/hooke/omp, 2460
pair_style granular, 2464
pair_style granular/superellipsoid, 2478
pair_style gw, 2486
pair_style gw/omp, 2486
pair_style gw/zbl, 2486
pair_style gw/zbl/omp, 2486
pair_style harmonic/cut, 2488
pair_style harmonic/cut/kk, 2488
pair_style harmonic/cut/omp, 2488
pair_style hbond/dreiding/lj, 2490
pair_style hbond/dreiding/lj/angleoffset, 2490
pair_style hbond/dreiding/lj/angleoffset/omp, 2490
pair_style hbond/dreiding/lj/omp, 2490
pair_style hbond/dreiding/morse, 2490
pair_style hbond/dreiding/morse/angleoffset, 2490
pair_style hbond/dreiding/morse/angleoffset/omp, 2490
pair_style hbond/dreiding/morse/omp, 2490
pair_style hdnnp, 2494
pair_style hippo, 2322
pair_style hippo/gpu, 2322
pair_style hybrid, 2497
pair_style hybrid/kk, 2497
pair_style hybrid/molecular, 2497
pair_style hybrid/molecular/omp, 2497
pair_style hybrid/omp, 2497
pair_style hybrid/overlay, 2497
pair_style hybrid/overlay/kk, 2497
pair_style hybrid/overlay/omp, 2497
pair_style hybrid/scaled, 2497
pair_style hybrid/scaled/kk, 2497
pair_style hybrid/scaled/omp, 2497
pair_style ilp/graphene/hbn, 2505
pair_style ilp/graphene/hbn/opt, 2505
pair_style ilp/tmd, 2508
pair_style ilp/tmd/opt, 2508
pair_style kim, 2510
pair_style kolmogorov/crespi/full, 2512
pair_style kolmogorov/crespi/z, 2514
pair_style lambda/input/apip, 2515

pair_style lambda/input/csp/apip, 2515
pair_style lambda/zone/apip, 2517
pair_style lcbop, 2519
pair_style lcbop/omp, 2519
pair_style ldd, 2520
pair_style lebedeva/z, 2524
pair_style lennard/mdf, 2582
pair_style lennard/mdf/kk, 2582
pair_style lennard/mdf/omp, 2582
pair_style lepton, 2525
pair_style lepton/coul, 2525
pair_style lepton/coul/omp, 2525
pair_style lepton/omp, 2525
pair_style lepton/sphere, 2525
pair_style lepton/sphere/omp, 2525
pair_style line/lj, 2529
pair_style line/lj/omp, 2529
pair_style list, 2531
pair_style lj/charmm/coul/charmm, 2363
pair_style lj/charmm/coul/charmm/gpu, 2363
pair_style lj/charmm/coul/charmm/implicit, 2363
pair_style lj/charmm/coul/charmm/implicit/kk, 2363
pair_style lj/charmm/coul/charmm/implicit/omp, 2363
pair_style lj/charmm/coul/charmm/intel, 2363
pair_style lj/charmm/coul/charmm/kk, 2363
pair_style lj/charmm/coul/charmm/omp, 2363
pair_style lj/charmm/coul/long, 2363
pair_style lj/charmm/coul/long/gpu, 2363
pair_style lj/charmm/coul/long/intel, 2363
pair_style lj/charmm/coul/long/kk, 2363
pair_style lj/charmm/coul/long/omp, 2363
pair_style lj/charmm/coul/long/opt, 2363
pair_style lj/charmm/coul/long/soft, 2445
pair_style lj/charmm/coul/long/soft/omp, 2445
pair_style lj/charmm/coul/msm, 2363
pair_style lj/charmm/coul/msm/omp, 2363
pair_style lj/charmmfsw/coul/charmmfsh, 2363
pair_style lj/charmmfsw/coul/charmmfsh/omp, 2363
pair_style lj/charmmfsw/coul/long, 2363
pair_style lj/charmmfsw/coul/long/kk, 2363
pair_style lj/charmmfsw/coul/long/omp, 2363
pair_style lj/class2, 2367
pair_style lj/class2/coul/cut, 2367
pair_style lj/class2/coul/cut/kk, 2367
pair_style lj/class2/coul/cut/omp, 2367
pair_style lj/class2/coul/cut/soft, 2445
pair_style lj/class2/coul/cut/soft/omp, 2445
pair_style lj/class2/coul/long, 2367
pair_style lj/class2/coul/long/cs, 2392
pair_style lj/class2/coul/long/cs/omp, 2392
pair_style lj/class2/coul/long/gpu, 2367
pair_style lj/class2/coul/long/kk, 2367
pair_style lj/class2/coul/long/omp, 2367
pair_style lj/class2/coul/long/soft, 2445

pair_style lj/class2/coul/long/soft/omp, 2445
pair_style lj/class2/gpu, 2367
pair_style lj/class2/kk, 2367
pair_style lj/class2/omp, 2367
pair_style lj/class2/soft, 2445
pair_style lj/class2/soft/omp, 2445
pair_style lj/cubic, 2537
pair_style lj/cubic/gpu, 2537
pair_style lj/cubic/kk, 2537
pair_style lj/cubic/omp, 2537
pair_style lj/cut, 2534
pair_style lj/cut/coul/cut, 2539
pair_style lj/cut/coul/cut/dielectric, 2395
pair_style lj/cut/coul/cut/dielectric/omp, 2395
pair_style lj/cut/coul/cut/gpu, 2539
pair_style lj/cut/coul/cut/kk, 2539
pair_style lj/cut/coul/cut/omp, 2539
pair_style lj/cut/coul/cut/soft, 2445
pair_style lj/cut/coul/cut/soft/gpu, 2445
pair_style lj/cut/coul/cut/soft/omp, 2445
pair_style lj/cut/coul/debye, 2539
pair_style lj/cut/coul/debye/dielectric, 2395
pair_style lj/cut/coul/debye/dielectric/omp, 2395
pair_style lj/cut/coul/debye/gpu, 2539
pair_style lj/cut/coul/debye/kk, 2539
pair_style lj/cut/coul/debye/omp, 2539
pair_style lj/cut/coul/dsf, 2539
pair_style lj/cut/coul/dsf/gpu, 2539
pair_style lj/cut/coul/dsf/kk, 2539
pair_style lj/cut/coul/dsf/omp, 2539
pair_style lj/cut/coul/esp, 2539
pair_style lj/cut/coul/long, 2539
pair_style lj/cut/coul/long/cs, 2392
pair_style lj/cut/coul/long/cs/omp, 2392
pair_style lj/cut/coul/long/dielectric, 2395
pair_style lj/cut/coul/long/dielectric/omp, 2395
pair_style lj/cut/coul/long/gpu, 2539
pair_style lj/cut/coul/long/intel, 2539
pair_style lj/cut/coul/long/kk, 2539
pair_style lj/cut/coul/long/omp, 2539
pair_style lj/cut/coul/long/opt, 2539
pair_style lj/cut/coul/long/soft, 2445
pair_style lj/cut/coul/long/soft/gpu, 2445
pair_style lj/cut/coul/long/soft/omp, 2445
pair_style lj/cut/coul/msm, 2539
pair_style lj/cut/coul/msm/dielectric, 2395
pair_style lj/cut/coul/msm/gpu, 2539
pair_style lj/cut/coul/msm/omp, 2539
pair_style lj/cut/coul/wolf, 2539
pair_style lj/cut/coul/wolf/kk, 2539
pair_style lj/cut/coul/wolf/omp, 2539
pair_style lj/cut/dipole/cut, 2397
pair_style lj/cut/dipole/cut/gpu, 2397
pair_style lj/cut/dipole/cut/kk, 2397

pair_style lj/cut/dipole/cut/omp, 2397
pair_style lj/cut/dipole/long, 2397
pair_style lj/cut/dipole/long/gpu, 2397
pair_style lj/cut/dipole/long/omp, 2397
pair_style lj/cut/gpu, 2534
pair_style lj/cut/intel, 2534
pair_style lj/cut/kk, 2534
pair_style lj/cut/omp, 2534
pair_style lj/cut/opt, 2534
pair_style lj/cut/soft, 2445
pair_style lj/cut/soft/gapsys, 2445
pair_style lj/cut/soft/omp, 2445
pair_style lj/cut/sphere, 2543
pair_style lj/cut/sphere/kk, 2543
pair_style lj/cut/sphere/omp, 2543
pair_style lj/cut/thole/long, 2766
pair_style lj/cut/thole/long/omp, 2766
pair_style lj/cut/tip4p/cut, 2546
pair_style lj/cut/tip4p/cut/omp, 2546
pair_style lj/cut/tip4p/long, 2546
pair_style lj/cut/tip4p/long/gpu, 2546
pair_style lj/cut/tip4p/long/omp, 2546
pair_style lj/cut/tip4p/long/opt, 2546
pair_style lj/cut/tip4p/long/soft, 2445
pair_style lj/cut/tip4p/long/soft/omp, 2445
pair_style lj/expand, 2550
pair_style lj/expand/coul/long, 2550
pair_style lj/expand/coul/long/gpu, 2550
pair_style lj/expand/coul/long/kk, 2550
pair_style lj/expand/coul/long/omp, 2550
pair_style lj/expand/gpu, 2550
pair_style lj/expand/kk, 2550
pair_style lj/expand/omp, 2550
pair_style lj/expand/sphere, 2551
pair_style lj/expand/sphere/omp, 2551
pair_style lj/gromacs, 2484
pair_style lj/gromacs/coul/gromacs, 2484
pair_style lj/gromacs/coul/gromacs/kk, 2484
pair_style lj/gromacs/coul/gromacs/omp, 2484
pair_style lj/gromacs/gpu, 2484
pair_style lj/gromacs/kk, 2484
pair_style lj/gromacs/omp, 2484
pair_style lj/long/coul/long, 2554
pair_style lj/long/coul/long/dielectric, 2395
pair_style lj/long/coul/long/intel, 2554
pair_style lj/long/coul/long/omp, 2554
pair_style lj/long/coul/long/opt, 2554
pair_style lj/long/dipole/long, 2397
pair_style lj/long/tip4p/long, 2554
pair_style lj/long/tip4p/long/omp, 2554
pair_style lj/mdf, 2582
pair_style lj/mdf/kk, 2582
pair_style lj/mdf/omp, 2582
pair_style lj/pirani, 2558

pair_style lj/pirani/kk, 2558
pair_style lj/pirani/omp, 2558
pair_style lj/relres, 2560
pair_style lj/relres/omp, 2560
pair_style lj/sf/dipole/sf, 2397
pair_style lj/sf/dipole/sf/gpu, 2397
pair_style lj/sf/dipole/sf/omp, 2397
pair_style lj/smooth, 2564
pair_style lj/smooth/gpu, 2564
pair_style lj/smooth/kk, 2564
pair_style lj/smooth/linear, 2566
pair_style lj/smooth/linear/omp, 2566
pair_style lj/smooth/omp, 2564
pair_style lj/spica, 2723
pair_style lj/spica/coul/long, 2723
pair_style lj/spica/coul/long/gpu, 2723
pair_style lj/spica/coul/long/kk, 2723
pair_style lj/spica/coul/long/omp, 2723
pair_style lj/spica/coul/msm, 2723
pair_style lj/spica/coul/msm/omp, 2723
pair_style lj/spica/gpu, 2723
pair_style lj/spica/kk, 2723
pair_style lj/spica/omp, 2723
pair_style lj/switch3/coulgauss/long, 2567
pair_style lj/switch3/coulgauss/long/kk, 2567
pair_style lj/switch3/coulgauss/long/omp, 2567
pair_style lj96/cut, 2536
pair_style lj96/cut/gpu, 2536
pair_style lj96/cut/kk, 2536
pair_style lj96/cut/omp, 2536
pair_style local/density, 2570
pair_style local/density/omp, 2570
pair_style lubricate, 2573
pair_style lubricate/omp, 2573
pair_style lubricate/poly, 2573
pair_style lubricate/poly/omp, 2573
pair_style lubricateU, 2576
pair_style lubricateU/poly, 2576
pair_style mbx, 2579
pair_style mdpd, 2599
pair_style mdpd/gpu, 2599
pair_style mdpd/rhosum, 2599
pair_style meam, 2585
pair_style meam/kk, 2585
pair_style meam/ms, 2585
pair_style meam/ms/kk, 2585
pair_style meam/spline, 2592
pair_style meam/spline/omp, 2592
pair_style meam/sw/spline, 2594
pair_style mesocnt, 2596
pair_style mesocnt/viscous, 2596
pair_style mgpt, 2606
pair_style mie/cut, 2608
pair_style mie/cut/gpu, 2608

pair_style mie/cut/kk, 2608
pair_style mie/cut/omp, 2608
pair_style mliap, 2610
pair_style mliap/kk, 2610
pair_style mm3/switch3/coulgauss/long, 2567
pair_style mm3/switch3/coulgauss/long/kk, 2567
pair_style mm3/switch3/coulgauss/long/omp, 2567
pair_style momb, 2614
pair_style momb/kk, 2614
pair_style momb/omp, 2614
pair_style morse, 2615
pair_style morse/gpu, 2615
pair_style morse/kk, 2615
pair_style morse/omp, 2615
pair_style morse/opt, 2615
pair_style morse/smooth/linear, 2615
pair_style morse/smooth/linear/kk, 2615
pair_style morse/smooth/linear/omp, 2615
pair_style morse/soft, 2445
pair_style morse/soft/omp, 2445
pair_style multi/lucy, 2617
pair_style multi/lucy/rx, 2620
pair_style multi/lucy/rx/kk, 2620
pair_style nb3b/harmonic, 2623
pair_style nb3b/harmonic/omp, 2623
pair_style nb3b/screened, 2623
pair_style nm/cut, 2625
pair_style nm/cut/coul/cut, 2625
pair_style nm/cut/coul/cut/kk, 2625
pair_style nm/cut/coul/cut/omp, 2625
pair_style nm/cut/coul/long, 2625
pair_style nm/cut/coul/long/kk, 2625
pair_style nm/cut/coul/long/omp, 2625
pair_style nm/cut/kk, 2625
pair_style nm/cut/omp, 2625
pair_style nm/cut/split, 2625
pair_style nm/cut/split/omp, 2625
pair_style none, 2628
pair_style oxdna/coaxstk, 2629
pair_style oxdna/excv, 2629
pair_style oxdna/hbond, 2629
pair_style oxdna/stk, 2629
pair_style oxdna/xstk, 2629
pair_style oxdna2/coaxstk, 2634
pair_style oxdna2/dh, 2634
pair_style oxdna2/excv, 2634
pair_style oxdna2/hbond, 2634
pair_style oxdna2/stk, 2634
pair_style oxdna2/xstk, 2634
pair_style oxdna3/coaxstk, 2639
pair_style oxdna3/dh, 2639
pair_style oxdna3/excv, 2639
pair_style oxdna3/hbond, 2639
pair_style oxdna3/stk, 2639

pair_style oxdna3/xstk, 2639
pair_style oxrna2/coaxstk, 2643
pair_style oxrna2/dh, 2643
pair_style oxrna2/excv, 2643
pair_style oxrna2/hbond, 2643
pair_style oxrna2/stk, 2643
pair_style oxrna2/xstk, 2643
pair_style pace, 2649
pair_style pace/apip, 2652
pair_style pace/extrapolation, 2649
pair_style pace/extrapolation/kk, 2649
pair_style pace/fast/apip, 2652
pair_style pace/kk, 2649
pair_style pace/precise/apip, 2652
pair_style pedone, 2654
pair_style pedone/kk, 2654
pair_style pedone/omp, 2654
pair_style peri/eps, 2655
pair_style peri/lps, 2655
pair_style peri/lps/omp, 2655
pair_style peri/pmb, 2655
pair_style peri/pmb/omp, 2655
pair_style peri/ves, 2655
pair_style pod, 2659
pair_style pod/kk, 2659
pair_style polymorphic, 2661
pair_style python, 2666
pair_style quip, 2670
pair_style rann, 2671
pair_style reaxff, 2677
pair_style reaxff/kk, 2677
pair_style reaxff/omp, 2677
pair_style rebo, 2318
pair_style rebo/intel, 2318
pair_style rebo/omp, 2318
pair_style rebomos, 2683
pair_style rebomos/omp, 2683
pair_style resquared, 2685
pair_style resquared/gpu, 2685
pair_style resquared/omp, 2685
pair_style rheo, 2688
pair_style rheo/solid, 2689
pair_style runner, 2690
pair_style saip/metal, 2695
pair_style saip/metal/opt, 2695
pair_style saip/metal/tmd, 2695
pair_style saip/metal/tmd/opt, 2695
pair_style sdpd/taitwater/isothermal, 2698
pair_style smatb, 2699
pair_style smatb/single, 2699
pair_style smd/hertz, 2701
pair_style smd/tlsph, 2702
pair_style smd/tri_surface, 2703
pair_style smd/ulsph, 2704

pair_style smtbq, 2705
pair_style snap, 2709
pair_style snap/intel, 2709
pair_style snap/kk, 2709
pair_style soft, 2713
pair_style soft/gpu, 2713
pair_style soft/kk, 2713
pair_style soft/omp, 2713
pair_style sph/heatconduction, 2715
pair_style sph/heatconduction/gpu, 2715
pair_style sph/idealgas, 2717
pair_style sph/lj, 2718
pair_style sph/lj/gpu, 2718
pair_style sph/rhosum, 2719
pair_style sph/taitwater, 2720
pair_style sph/taitwater/gpu, 2720
pair_style sph/taitwater/morris, 2722
pair_style spin/dipole/cut, 2725
pair_style spin/dipole/long, 2725
pair_style spin/dmi, 2727
pair_style spin/exchange, 2728
pair_style spin/exchange/biquadratic, 2728
pair_style spin/magelec, 2731
pair_style spin/neel, 2732
pair_style srp, 2733
pair_style srp/react, 2733
pair_style surf/granular, 2736
pair_style sw, 2738
pair_style sw/angle/table, 2742
pair_style sw/gpu, 2738
pair_style sw/intel, 2738
pair_style sw/kk, 2738
pair_style sw/mod, 2738
pair_style sw/mod/omp, 2738
pair_style sw/omp, 2738
pair_style table, 2746
pair_style table/gpu, 2746
pair_style table/kk, 2746
pair_style table/omp, 2746
pair_style table/rx, 2750
pair_style table/rx/kk, 2750
pair_style tdpd, 2599
pair_style tersoff, 2753
pair_style tersoff/gpu, 2753
pair_style tersoff/intel, 2753
pair_style tersoff/kk, 2753
pair_style tersoff/mod, 2758
pair_style tersoff/mod/c, 2758
pair_style tersoff/mod/c/omp, 2758
pair_style tersoff/mod/gpu, 2758
pair_style tersoff/mod/kk, 2758
pair_style tersoff/mod/omp, 2758
pair_style tersoff/omp, 2753
pair_style tersoff/table, 2753

pair_style tersoff/table/omp, 2753
pair_style tersoff/zbl, 2761
pair_style tersoff/zbl/gpu, 2761
pair_style tersoff/zbl/kk, 2761
pair_style tersoff/zbl/omp, 2761
pair_style thole, 2766
pair_style threebody/table, 2768
pair_style threebody/table/omp, 2768
pair_style tip4p/cut, 2378
pair_style tip4p/cut/omp, 2378
pair_style tip4p/long, 2378
pair_style tip4p/long/omp, 2378
pair_style tip4p/long/soft, 2445
pair_style tip4p/long/soft/omp, 2445
pair_style tracker, 2772
pair_style tri/lj, 2774
pair_style tri/lj/omp, 2774
pair_style uf3, 2776
pair_style uf3/kk, 2776
pair_style ufm, 2779
pair_style ufm/gpu, 2779
pair_style ufm/kk, 2779
pair_style ufm/omp, 2779
pair_style ufm/opt, 2779
pair_style vashishta, 2781
pair_style vashishta/gpu, 2781
pair_style vashishta/kk, 2781
pair_style vashishta/omp, 2781
pair_style vashishta/table, 2781
pair_style vashishta/table/omp, 2781
pair_style wf/cut, 2784
pair_style wf/cut/kk, 2784
pair_style wf/cut/omp, 2784
pair_style ylz, 2786
pair_style ylz/kk, 2786
pair_style ylz/omp, 2786
pair_style yukawa, 2789
pair_style yukawa/colloid, 2790
pair_style yukawa/colloid/gpu, 2790
pair_style yukawa/colloid/kk, 2790
pair_style yukawa/colloid/omp, 2790
pair_style yukawa/gpu, 2789
pair_style yukawa/kk, 2789
pair_style yukawa/omp, 2789
pair_style zbl, 2792
pair_style zbl/gpu, 2792
pair_style zbl/kk, 2792
pair_style zbl/omp, 2792
pair_style zero, 2794
pair_write, 1216
partition, 1218
platform::chdir (*C++ function*), 908
platform::compiler_info (*C++ function*), 904
platform::compress_info (*C++ function*), 905

platform::compressed_read (*C++ function*), 911
platform::compressed_write (*C++ function*), 911
platform::cputime (*C++ function*), 904
platform::cxx_standard (*C++ function*), 905
platform::disk_free (*C++ function*), 907
platform::dlclose (*C++ function*), 910
platform::dlerror (*C++ function*), 911
platform::dlopen (*C++ function*), 910
platform::dlsym (*C++ function*), 911
platform::END_OF_FILE (*C++ member*), 908
platform::file_is_readable (*C++ function*), 907
platform::file_is_writable (*C++ function*), 907
platform::file_redirect (*C++ function*), 907
platform::filepathsep (*C++ member*), 905
platform::find_exe_path (*C++ function*), 910
platform::fseek (*C++ function*), 909
platform::ftell (*C++ function*), 908
platform::ftruncate (*C++ function*), 909
platform::guesspath (*C++ function*), 906
platform::has_compress_extension (*C++ function*), 911
platform::is_console (*C++ function*), 907
platform::list_directory (*C++ function*), 907
platform::list_pathenv (*C++ function*), 910
platform::mkdir (*C++ function*), 908
platform::mpi_info (*C++ function*), 905
platform::mpi_vendor (*C++ function*), 905
platform::openmp_standard (*C++ function*), 905
platform::os_info (*C++ function*), 904
platform::path_basename (*C++ function*), 906
platform::path_dirname (*C++ function*), 906
platform::path_join (*C++ function*), 906
platform::pathvarsep (*C++ member*), 906
platform::pclose (*C++ function*), 909
platform::popen (*C++ function*), 909
platform::putenv (*C++ function*), 909
platform::rmdir (*C++ function*), 908
platform::unlink (*C++ function*), 908
platform::unsetenv (*C++ function*), 910
platform::usleep (*C++ function*), 904
platform::walltime (*C++ function*), 904
plugin, 1219
plugin_count() (*fortran function*), **680**
plugin_name() (*fortran subroutine*), **680**
Pointers (*C++ class*), 896
PotentialFileReader (*C++ class*), 944
PotentialFileReader::~PotentialFileReader (*C++ function*), 945
PotentialFileReader::ignore_comments (*C++ function*), 945
PotentialFileReader::next_bigint (*C++ function*), 946
PotentialFileReader::next_double (*C++ function*), 946
PotentialFileReader::next_dvector (*C++ function*), 945
PotentialFileReader::next_int (*C++ function*), 946
PotentialFileReader::next_line (*C++ function*), 945
PotentialFileReader::next_string (*C++ function*), 946
PotentialFileReader::next_tagint (*C++ function*), 946

PotentialFileReader::next_values (*C++ function*), 945
PotentialFileReader::PotentialFileReader (*C++ function*), 944
PotentialFileReader::rewind (*C++ function*), 945
PotentialFileReader::set_bufsize (*C++ function*), 945
PotentialFileReader::skip_line (*C++ function*), 945
prd, 1220
print, 1224
processors, 1226
python, 1231

## Q

quit, 1240

## R

read_data, 1241
read_dump, 1264
read_restart, 1269
region, 1272
region2vmd, 1278
replicate, 1280
rerun, 1282
reset_atoms, 1285
reset_box() (*fortran subroutine*), **650**
reset_box() (*lammps.lammps method*), 712
reset_timestep, 1288
restart, 1289
run, 1292
run_style, 1295

## S

SafeFilePtr (*C++ class*), 940
SafeFilePtr::operator FILE* (*C++ function*), 941
SafeFilePtr::operator= (*C++ function*), 941
SafeFilePtr::set_pclose (*C++ function*), 941
scatter() (*fortran subroutine*), **669**
scatter_atoms() (*fortran subroutine*), **663**
scatter_atoms_subset() (*fortran subroutine*), **664**
scatter_subset() (*fortran subroutine*), **670**
set, 1300
set_fix_external_callback() (*fortran subroutine*), **682**
set_fix_external_callback() (*lammps.lammps method*), 725
set_fix_external_set_virial_peratom() (*fortran subroutine*), **685**
set_internal_variable() (*fortran subroutine*), **660**
set_internal_variable() (*lammps.lammps method*), 718
set_show_error() (*lammps.lammps method*), 721
set_string_variable() (*fortran subroutine*), **660**
set_string_variable() (*lammps.lammps method*), 718
set_variable() (*fortran subroutine*), **660**
set_variable() (*lammps.lammps method*), 717
shell, 1309
size (*lammps.NeighList property*), 737
special_bonds, 1311
style_count() (*fortran function*), **678**
style_name() (*fortran subroutine*), **679**

suffix, 1314

## T

tad, 1316
temper, 1320
temper/grem, 1322
temper/npt, 1323
TextFileReader (*C++ class*), 943
TextFileReader::~TextFileReader (*C++ function*), 943
TextFileReader::ignore_comments (*C++ member*), 944
TextFileReader::next_dvector (*C++ function*), 944
TextFileReader::next_line (*C++ function*), 943
TextFileReader::next_values (*C++ function*), 944
TextFileReader::rewind (*C++ function*), 943
TextFileReader::set_bufsize (*C++ function*), 943
TextFileReader::skip_line (*C++ function*), 943
TextFileReader::TextFileReader (*C++ function*), 943
thermo, 1324
thermo_modify, 1325
thermo_style, 1329
third_order, 1336
third_order/kk, 1336
timer, 1337
timestep, 1339
Tokenizer (*C++ class*), 933
Tokenizer::as_vector (*C++ function*), 935
Tokenizer::contains (*C++ function*), 934
Tokenizer::count (*C++ function*), 935
Tokenizer::has_next (*C++ function*), 934
Tokenizer::matches (*C++ function*), 934
Tokenizer::next (*C++ function*), 934
Tokenizer::reset (*C++ function*), 934
Tokenizer::skip (*C++ function*), 934
Tokenizer::Tokenizer (*C++ function*), 934
TokenizerException (*C++ class*), 935
TokenizerException::TokenizerException (*C++ function*), 935
TokenizerException::what (*C++ function*), 935

## U

ubuf (*C++ union*), 958
ubuf::d (*C++ member*), 958
ubuf::i (*C++ member*), 958
ubuf::ubuf (*C++ function*), 958
uncompute, 1340
undump, 1340
unfix, 1341
units, 1342
utils::binary_search (*C++ function*), 930
utils::bnumeric (*C++ function*), 915
utils::bounds (*C++ function*), 924
utils::bounds_typelabel (*C++ function*), 925
utils::check_packages_for_style (*C++ function*), 929
utils::count_words (*C++ function*), 919
utils::current_date (*C++ function*), 929

utils::date2num (*C++ function*), 929
utils::errorurl (*C++ function*), 928
utils::expand_args (*C++ function*), 925
utils::expand_type (*C++ function*), 926
utils::fgets_trunc (*C++ function*), 913
utils::flush_buffers (*C++ function*), 928
utils::get_conversion_factor (*C++ function*), 924
utils::get_potential_date (*C++ function*), 923
utils::get_potential_file_path (*C++ function*), 923
utils::get_potential_units (*C++ function*), 923
utils::get_supported_conversions (*C++ function*), 923
utils::getsyserror (*C++ function*), 929
utils::has_utf8 (*C++ function*), 918
utils::inumeric (*C++ function*), 914, 915
utils::is_double (*C++ function*), 922
utils::is_id (*C++ function*), 922
utils::is_integer (*C++ function*), 922
utils::is_type (*C++ function*), 923
utils::join (*C++ function*), 920
utils::join_words (*C++ function*), 920
utils::logical (*C++ function*), 916
utils::logmesg (*C++ function*), 927
utils::lowercase (*C++ function*), 917
utils::merge_sort (*C++ function*), 930
utils::missing_cmd_args (*C++ function*), 928
utils::numeric (*C++ function*), 914
utils::open_potential (*C++ function*), 924
utils::parse_grid_id (*C++ function*), 926
utils::point_to_error (*C++ function*), 928
utils::print (*C++ function*), 927
utils::read_lines_from_file (*C++ function*), 913
utils::sfgets (*C++ function*), 912
utils::sfread (*C++ function*), 912
utils::split_lines (*C++ function*), 921
utils::split_words (*C++ function*), 920
utils::star_subst (*C++ function*), 918
utils::strcompress (*C++ function*), 918
utils::strdup (*C++ function*), 917
utils::strfind (*C++ function*), 922
utils::strip_style_suffix (*C++ function*), 918
utils::strmatch (*C++ function*), 921
utils::strsame (*C++ function*), 921
utils::timespec2seconds (*C++ function*), 929
utils::tnumeric (*C++ function*), 915, 916
utils::trim (*C++ function*), 917
utils::trim_and_count_words (*C++ function*), 919
utils::trim_comment (*C++ function*), 917
utils::uppercase (*C++ function*), 917
utils::utf8_subst (*C++ function*), 919

## V

ValueTokenizer (*C++ class*), 935
ValueTokenizer::contains (*C++ function*), 936
ValueTokenizer::count (*C++ function*), 937

ValueTokenizer::has_next (*C++ function*), 936
ValueTokenizer::matches (*C++ function*), 936
ValueTokenizer::next_bigint (*C++ function*), 936
ValueTokenizer::next_double (*C++ function*), 936
ValueTokenizer::next_int (*C++ function*), 936
ValueTokenizer::next_string (*C++ function*), 936
ValueTokenizer::next_tagint (*C++ function*), 936
ValueTokenizer::skip (*C++ function*), 937
ValueTokenizer::ValueTokenizer (*C++ function*), 935
variable, 1346
velocity, 1369
version() (*fortran function*), **674**
version() (*lammps.lammps method*), 710
video() (*lammps.ipython.wrapper method*), 735

## W

wrapper (*class in lammps.ipython*), 735
write_coeff, 1373
write_data, 1373
write_dump, 1376
write_molecule, 1378
write_restart, 1380