Implementation of a compressible-flow simulation code in the D programming language

Peter Jacobs[a *] and Rowan Gollan[b]

School of Mechanical and Mining Engineering,

The University of Queensland, Brisbane, Australia

[a]peterj@mech.uq.edu.au, [b]r.gollan@uq.edu.au

Keywords: compressible-flow, simulation, programming

Abstract. We describe the formulation and implementation of the Eilmer4 compressible-flow solver as well as discuss the features of the D programming language that we have found useful for writing scientific software. An example of use is provided to show the features of the user-input scripting and the performance of the main simulation code when run in parallel with block-marching.

Introduction

The Eilmer series of compressible-flow simulation codes [1,2,3] have been developed for more than 25 years. Initially written in C, and then in C++, the simulation code is now being completely rebuilt in the D programming language [4]. The motivation for the rebuild is that the Eilmer3 C++ code base has become difficult to maintain and extend.

So far, the gas-dynamic simulation core of the C++ code has been ported to D, along with a new approach to boundary condition specification and a simplified implementation of the flow derivative calculations for viscous terms. The new code is capable of simulating viscous, reacting, compressible flows in two- and three-dimensional domains, in fixed or rotating frames of reference. It also includes a transient heat-conduction solver that is consistently coupled to the gas-dynamic flow, however, only the gas-dynamics part of the new code will be discussed in this paper.

Formulation

The flow simulation program is based on a finite-volume formulation of the gas dynamics. The flow domain is partitioned into a large number of small (finite-volume) cells and the conservation relations of mass, momentum, energy and chemical species is used to determine a set of update equations for the average properties within each cell [5]. With boundary conditions specified around the periphery of the flow domain and initial gas state specified throughout the domain, the simulation proceeds in small time steps as shown in Fig. 1, with the major physical processes being coupled loosely within the controlling super-loop.

Within each iteration of the super-loop, the convective and molecular-scale transport components are updated in a number of stages (as shown on the right of Fig. 1), typically with an explicit time integration scheme such as Euler, predictor-corrector or a third-order Runge-Kutta method. As a precursor to each update stage, the boundary condition effects are applied as lists of ghost-cell effects and boundary-face effects, depending on the nature of the specific boundary condition.

Fluxes of mass, momentum and energy are then computed at all inter-cell faces and these fluxes are used to update the state of the conserved quantities in each cell. To achieve a reasonably high-order spatial truncation error for the convective flux calculation, the flow-field data is reconstructed either-side of each inter-cell face using a quasi-one-dimensional interpolation of the cell-average properties, prior to one of a number of up-winding flux calculators being applied.

Viscous transport fluxes are computed from local estimates of flow-quantity spatial derivatives combined with molecular transport coefficients that may also be dependent on the local gas state. The spatial derivatives are computed via the application of the vector divergence theorem or by least-squares fitting of a simple linear model for the flow-quantity variation.

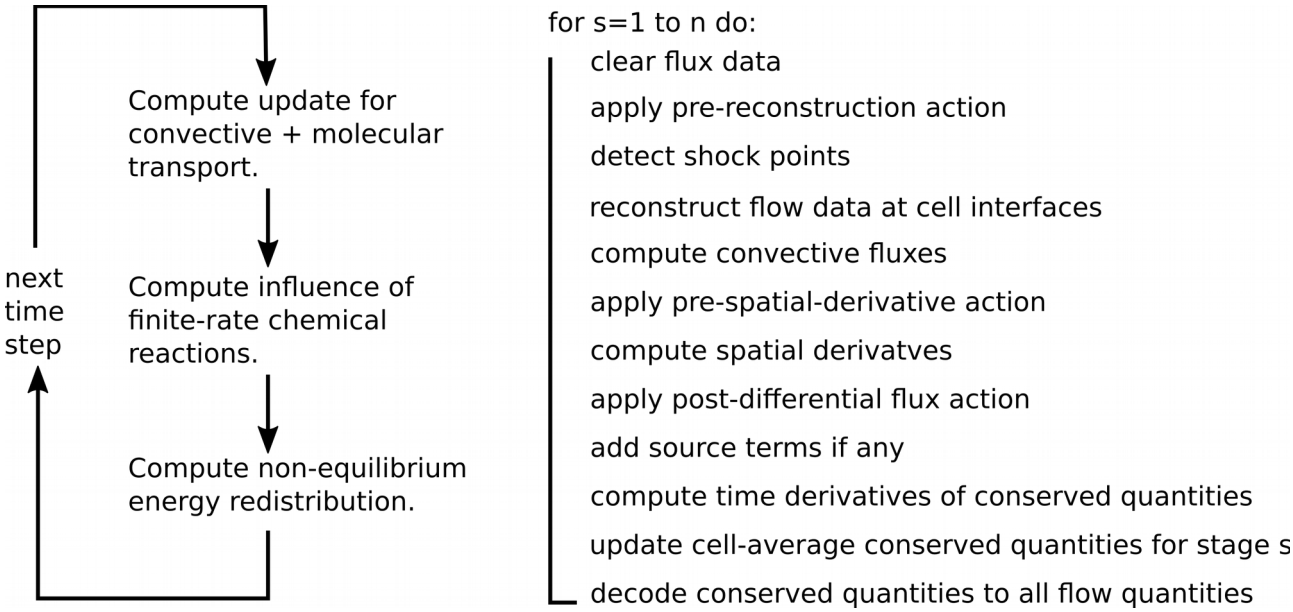

Fig. 1: Super-loop (left) for operator-splitting of physical-process updates and the n-stage gas-dynamic update process (right) that is applied to all cells.

Following the gas-dynamic update, updates to conserved quantities due to other physical processes are applied. These processes include finite-rate chemical reactions and non-equilibrium thermodynamic processes that redistribute energy between the internal energy modes of each molecular species.

Implementation

The actual code to implement the methods described above has to use a specific way of retaining the data associated with all of the cells and then provide convenient access to that data in an efficient way. Presently, the code uses an array of block-structured meshes to define the gas-flow domain as patches in two dimensions or sub-volumes in three dimensions. There may be several, possibly many, blocks to be defined with boundary faces aligned to the domain boundaries. Having a flow domain comprised of many blocks allows the option of doing the flow updates in parallel. We choose to apply the coarse-gain parallel approach of assigning the updates of the blocks to a pool of processors, with the update of any particular block being done independently of the update for any other block.

Choice of programming language. The predecessor code, Eilmer3, was written in a mixture of Python and C++ and is bound together using an automated interface generator tool. Parallel execution of a simulation is coordinated via the MPI message-passing library and run-time customization of the code is done via user-supplied scripts written in the Lua programming language [6].

There are a number of reasons that we have been growing unhappy with the current arrangements for the Eilmer3 code. The C++ code base is not easy for students to understand and subsequently contribute to, partly because of the growth of the code itself (with a design that could be better) and partly because the complexity of the C++ programming language for *occasional* programmers such as ourselves. On top of that, end users of the code might have to learn two programming languages (Python and Lua) to write their input scripts. Finally, for new users of the software, it can be difficult to get the code to build on each new machine that becomes available for simulations, principally because of all of the code building tools that need to be installed on the machine. To use reasonably modern features of C++11, we need to have up-to-date compilers, interface-generator tools and libraries.

With the recent maturing of the programming language D [4] as a good alternative to C++ for statically-checked, natively-compiled code, we have taken the opportunity to rebuild our simulation code. The D programming language provides the conveniences of Python, the run-time performance of C++ and the ability to be directly linked to C language libraries. It appears that we can have it all and we can have it now. One good example of where the rebuild has resulted in significant improvements is the viscous-flux calculation code. In the C++ code, about 2500 lines were used, and this had the extra complication of requiring the M4 preprocessor to produce the actual C++ code (of length 5580 lines) that was given to the compiler. The new D language code amounts to 733 lines.

Also, because the connection to Lua is easy enough not to require an automated interface generator and also that most of our thermochemical configuration and database was written in Lua, we have opted to use only Lua for all of the user-supplied input at run time. This reduces by one the number of languages for new users to learn.

Parallelism. Although we could use the MPI library for parallelism from D, we have opted to stay with the shared-memory parallelism offered directly by the D compiler. See Fig. 2 for an example of the almost-trivial conversion of the serial code to parallel code. The "parallel" macro expands the loop to one that allocates each iteration of the loop body to one of the threads (effectively a core) in a thread pool and coordinates all of the house-keeping that needs to be done to ensure that all blocks are processed. For this to work well, there is a need for all of the data used within one iteration to be completely separate to the data written by another iteration. We achieve this with our block-level parallelism by having each block hold its own configuration, gas model and mesh data. Compared with the previous MPI code, the simplification afforded by the D-language constructs is very welcome.

```
1  // Determine the allowable time step -- serial version.
2  double dt_allow = 1.0e9;
3  foreach (myblk; gasBlocks) {
4      if (!myblk.active) continue;
5      double local_dt_allow = myblk.determine_time_step_size(dt_global);
6      dt_allow = min(dt_allow, local_dt_allow);
7  }
8
9
10
11 // Determine the allowable time step -- parallel version.
12 shared double dt_allow = 1.0e9;
13 foreach (myblk; parallel(gasBlocks,1)) {
14     if (!myblk.active) continue;
15     double local_dt_allow = myblk.determine_time_step_size(dt_global);
16     dt_allow = min(dt_allow, local_dt_allow);
17 }
```

Fig. 2: Collecting the low-hanging fruit of parallelism.

Although the use of a thread-pool to implement the parallel updates for the blocks means we need to run parallel jobs on a multi-core computer, we seem to have ready of access to workstations and server computers with between 8 and 64 cores and lots of RAM. This situation will likely be true for most users of the code in the near future.

If the flow is expected to become steady, has a dominant supersonic flow direction, and the domain is relatively long in that direction, it may be convenient to progressively compute the flow along the duct a small section at a time. To enable this within the multi-block simulation, we arrange to divide the full flow domain into many relatively-small blocks and then iterate in time on small subsets of blocks that span the flow domain. Starting at a supersonic inlet, we allow the flow

in the initial slice of blocks to settle and then move on downstream allowing the next-downstream slice of blocks to be integrated, leaving the upstream blocks to be no longer updated. With the shared-memory code, this is simple to arrange with the addition of a flag to indicate whether a block is part of the *active* set. If the active set is a small fraction of the overall domain, the computational load is greatly reduced. The code required to implement this block-marching is quite small, having only to manage the activity flag for each block, assuming that the blocks are arranged in a simple rectangular array.

Example of Use

Figure 3 shows the computed pressure for the simulation of a two-dimensional supersonic flow in a channel with a circular-arc bump. With a dominant supersonic flow direction along the length of the duct (*i.e.* the x-direction), this is a good example to show the benefits of block-marching. Combined with parallel calculation on a small 4-core workstation with AMD Phenom II 840 processors, this case runs in less than a minute.

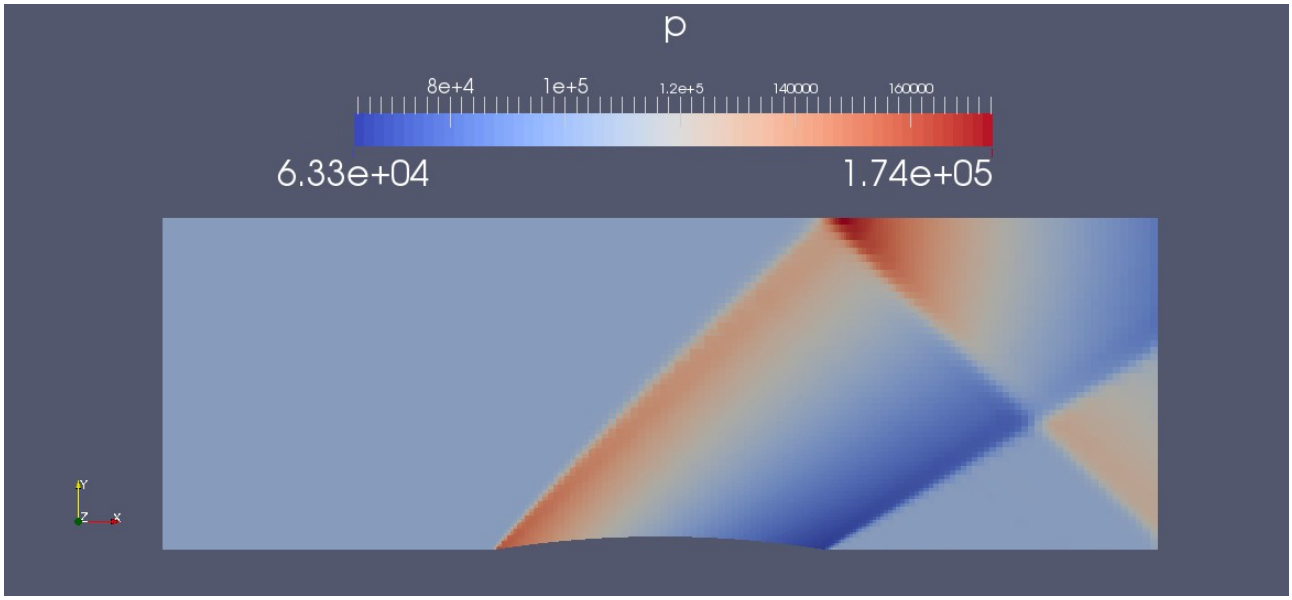

Fig. 3: Pressure field (in Pa) for supersonic flow in a channel with a bump.

The required input script is shown in Fig. 4 as a Lua script that is run in the context of the preparation stage of the simulation. Once the main program starts running, it sets up a Lua interpreter that, in turn, imports and executes the user's input script. Having a full Lua interpreter available at this stage means that it is easy to define the flow conditions and the flow domain in a fully-parametric manner. For example, lines 13 through 20 establish the use of the ideal-air thermochemical model for the gas and define the inflow condition with the convenient parameters of Mach number, static pressure and temperature. The other parameters needed for the full definition of the inflow condition are computed within the script. The resulting values are also printed to standard output so that they can be checked.

Lines 23 through 33 define the flow domain as a set of three patches, with the middle patch having a circular arc as its southern boundary. The parameters of interest to the user are the characteristic length, L, for the domain and the height of the bump, h. The script then defines some points of interest as Vector objects. Note that the script makes use of the object convention described in the Lua book [6]. Once the points are available, they may be used to define Path objects such as straight lines and circular arcs. The middle patch (patch1) is then defined, on line 31 and 32, as the set of paths that bound its north, east south and west extents. The upstream patch (patch0) and downstream patch (patch2) have all straight-line edges and so can be conveniently

defined from their corner locations only. This is seen on lines 30 and 33. The principal structured data type in Lua is the *table*. Tables appear in Lua code as sequences of expressions bounded by braces {} and early all objects will require their data to be passed as a table

```
1  -- bump.lua
2  -- Ported for Eilmer4 by PJ, 2015-05-28
3  -- This test appeared in papers on Euler solvers for supersonic flows.
4  -- For example:
5  -- S. Eidelman, P. Colella and R.P. Shreeve (1984)
6  -- Application of the Godunov method and its second-order extension to
7  -- cascade flow modelling.
8  -- AIAA Journal Vol. 22 No. 11 pp
9
10 config.title = "Channel with circular-arc bump."
11 print(config.title)
12
13 nsp, nmodes = setGasModel('ideal-air-gas-model.lua')
14 print("GasModel set to ideal air. nsp= ", nsp, " nmodes= ", nmodes)
15 p_inf = 101.3e3 -- Pa
16 T_inf = 288.0 -- degree K
17 a_inf = math.sqrt(1.4*287.0*T_inf)
18 u_inf = 1.65 * a_inf -- m/s
19 print("Inflow: p=", p_inf, " T=", T_inf, " velx=", u_inf)
20 inflow = FlowState:new{p=p_inf, T=T_inf, velx=u_inf, vely=0.0}
21
22 -- Geometry of flow domain.
23 L = 1.0 -- will use for both length and height of domain
24 h = 0.04 * L -- height of bump
25 a0 = Vector3:new{0.0,0.0}; a1 = Vector3:new{0.0,L}
26 b0 = Vector3:new{L,0.0}; b1 = Vector3:new{L,L}
27 c0 = Vector3:new{1.5*L,h}
28 d0 = Vector3:new{2.0*L,0.0}; d1 = Vector3:new{2.0*L,L}
29 e0 = Vector3:new{3.0*L,0.0}; e1 = Vector3:new{3.0*L,L}
30 patch0 = CoonsPatch:new{p00=a0, p10=b0, p11=b1, p01=a1}
31 patch1 = makePatch{Line:new{b1,d1}, Line:new{d0,d1},
32             Arc3:new{b0,c0,d0}, Line:new{b0,b1}}
33 patch2 = CoonsPatch:new{p00=d0, p10=e0, p11=e1, p01=d1}
34
35 -- Mesh the patches, with particular discretisation.
36 rcfx0 = RobertsFunction:new{end0=false, end1=true, beta=1.2}
37 rcfx1 = RobertsFunction:new{end0=true, end1=true, beta=1.2}
38 rcfx2 = RobertsFunction:new{end0=true, end1=false, beta=1.2}
39 rcfy = RobertsFunction:new{end0=true, end1=false, beta=1.2}
40 ni0 = 64; nj0 = 64 -- We'll scale discretization off these values
41 factor = 1.0
42 ni0 = math.floor(ni0*factor); nj0 = math.floor(nj0*factor)
43 grid0 = StructuredGrid:new{psurface=patch0, cfList={rcfx0,rcfy,rcfx0,rcfy}, niv=ni0+1, njv=nj0+1}
44 grid1 = StructuredGrid:new{psurface=patch1, cfList={rcfx1,rcfy,rcfx1,rcfy}, niv=ni0+1, njv=nj0+1}
45 grid2 = StructuredGrid:new{psurface=patch2, cfList={rcfx2,rcfy,rcfx2,rcfy}, niv=ni0+1, njv=nj0+1}
46 -- Define the flow-solution blocks and set boundary conditions.
47 blk0 = SBlockArray{grid=grid0, fillCondition=inflow, nib=16, njb=2,
48                    bcList={west=SupInBC:new{flowCondition=inflow}}}
49 blk1 = SBlockArray{grid=grid1, fillCondition=inflow, nib=16, njb=2}
50 blk2 = SBlockArray{grid=grid2, fillCondition=inflow, nib=16, njb=2,
51                    bcList={east=ExtrapolateOutBC:new{}}}
52 identifyBlockConnections()
53
54 config.block_marching = true
55 config.nib = 48
56 config.njb = 2
57 config.propagate_inflow_data = true
58 config.flux_calc = "adaptive"
59 config.gasdynamic_update_scheme = "classic-rk3"
60 config.cfl_target = 1.0
61 config.max_time = 20.0*L/u_inf -- long enough, tunnel has 15ms steady time
62 config.max_step = 50000
63 config.dt_init = 1.0e-6
64 config.dt_plot = config.max_time
65
```

Fig. 4: User written input script for channel with bump simulation.

Once the patches are defined, they may be discretized as structured blocks of small cells. Lines 36 through 39 set up some clustering functions to provide a nonuniform distribution of cell sizes along each coordinate direction. This non-uniformity is chosen to cluster cells into the regions of the domain where interesting flow behaviour needs to to be resolved. For this simple flow, the regions at the beginning and end of the bump need to be well resolved. Lines 40 to 42 set the numbers of cells using a single factor to allow the overall resolution of the mesh to be conveniently

changed (on line 41). The actual meshes are constructed on lines 43 through 45 with calls to the built-in grid generator. For more complex flow domains, it may be convenient to generate the meshes with an external tool and import them. The code can presently import multi-block meshes prepared by GridPro [7].

Given a mesh, a flow block (SBlock object) is constructed by adding an initial flow condition and appropriate boundary conditions. Lines 47 though 51 construct arrays of SBlock objects so that the domain can be subdivided into many slices of blocks for the block-marching process to take advantage of the dominant supersonic flow along the duct.

The final few lines set a number of simulation configuration parameters such as the number of blocks in each index direction (i,j), the type of gas-dynamic update to apply each time step, the size of the time steps and the duration of the simulation.

When this simulation is run as a block-marching simulation, the time required to compute the solution is 58 seconds. Changing the block_marching flag on line 57 from true to false results in a run time of 25 minutes 13 seconds. Clearly the block-marching is effective if the resulting flow is suitable.

Parallel performance. In separate experiments on a moderately-large time-marching simulation, the scaling of the run time on an 8-core workstation (with Intel Xeon cores) indicated that the fraction of the computational work done in parallel is somewhere between 85% and 91% of the total work done by the code. Although this allows moderate speed-ups for up to 8 cores, we will want to increase the fraction of work done in parallel before seriously using the code on our 64-core workstations.

Conclusion

We have spent much of a year building a new compressible flow simulation code from scratch, in a relatively new language. So far, our experience with the D programming language has been positive, with a fairly capable simulation code being constructed with a few months of effort by two people. Together with some code redesign, the D programming language has allowed the construction of a code base that is much simpler than its C++ predecessor but is just as fast.

Acknowledgements

Many people have supported the development of the Eilmer series of compressible-flow simulation codes, through contributions of code, case studies and suffering through buggy implementations. We have tried to list them on the front cover of the Eilmer3 user guide [3].

References

[1] http://cfcfd.mechmining.uq.edu.au

[2] R.J. Gollan and P.A. Jacobs, About the formulation, verification and validation of the hypersonic flow solver Eilmer. International Journal for Numerical Methods in Fluids, 73 1 (2013): 19-57.

[3] P.A. Jacobs, R.J. Gollan, I. Jahn and D.F. Potter, The Eilmer3 Code: User Guide and Example Book, 2015 Edition. School of Mechanical and Mining Engineering Report 2015/07.

[4] A. Alexandrescu, The D Programming Language, Addison-Wesley, Upper Saddle River, NJ, 2010.

[5] P.A. Jacobs, R.J. Gollan, A.J. Denman, B.T. O'Flaherty, D.F. Potter, P.J. Petrie-Repar, and I.A. Johnston, Eilmer's theory book: Basic models for gas dynamics and thermochemistry. Mechanical Engineering Report 2010/09, The University of Queensland, Brisbane, Australia, 2010.

[6] R. Ierusalimschy, Programming in Lua, 2nd Edition, Lua.org, 2006.

[7] http://www.gridpro.com/