MP-Opt-Model User's Manual

Version 5.0

Ray D. Zimmerman

July 12, 2025

Contents

List of Tables

1 Introduction

1.1 Background

MP-Opt-Model is a package of MATLAB language M-files[1] for constructing and solving mathematical programming and optimization problems. It provides an easy-to-use, object-oriented interface for building and solving your model. It also includes a unified interface for calling numerous LP, QP, QCQP, mixed-integer, and nonlinear solvers, with the ability to switch solvers simply by changing an input option. The MP-Opt-Model project page can be found at:

https://github.com/MATPOWER/mp-opt-model

MP-Opt-Model is based on code that was developed primarily by Ray D. Zimmerman of PSERC[2] at Cornell University, along with significant contributions from others, as part of the MATPOWER [1, 2] project.

Up until version 7 of MATPOWER, the code now included in MP-Opt-Model was distributed only as an integrated part of MATPOWER. After the release of MATPOWER 7, MP-Opt-Model was split out into a separate project, though it is still included with MATPOWER.

[1] Also compatible with GNU Octave [3].

[2] http://pserc.org/

1.2 License and Terms of Use

The code in MP-Opt-Model is distributed under the 3-clause BSD license [4]. The full text of the license can be found in the `LICENSE` file at the top level of the distribution or at https://github.com/MATPOWER/mp-opt-model/blob/master/LICENSE and reads as follows.

1.3 Citing MP-Opt-Model

We request that publications derived from the use of MP-Opt-Model explicitly acknowledge that fact by citing the MP-Opt-Model User's Manual [5]. The citation and DOI can be version-specific or general, as appropriate. For version 5.0, use:

> R. D. Zimmerman. MP-Opt-Model User's Manual, Version 5.0. 2025. [Online]. Available: `https://matpower.org/docs/MP-Opt-Model-manual-5.0.pdf`
> doi: 10.5281/zenodo.15871431

For a version non-specific citation, use the following citation and DOI, with <*YEAR*> replaced by the year of the most recent release:

> R. D. Zimmerman. MP-Opt-Model User's Manual. <*YEAR*>. [Online]. Available: `https://matpower.org/docs/MP-Opt-Model-manual.pdf`
> doi: 10.5281/zenodo.3818002

A list of versions of the User's Manual with release dates and version-specific DOI's can be found via the general DOI at `https://doi.org/10.5281/zenodo.3818002`.

1.4 MP-Opt-Model Development

The MP-Opt-Model project uses an open development paradigm, hosted on the MP-Opt-Model GitHub project page:

`https://github.com/MATPOWER/mp-opt-model`

The MP-Opt-Model GitHub project hosts the public Git code repository as well as a public issue tracker for handling bug reports, patches, and other issues and contributions. There are separate GitHub hosted repositories and issue trackers for MP-Opt-Model, MP-Test, MIPS, and MATPOWER, etc., all are available from `https://github.com/MATPOWER/`.

2 Getting Started

2.1 System Requirements

To use MP-Opt-Model 5.0 you will need:

- MATLAB® version 7.9 (R2009b) or later[3], or
- GNU Octave version 6.2 or later[4]
- MIPS, **M**ATPOWER **I**nterior **P**oint **S**olver [6, 7][5]
- MP-Test[6]

For the hardware requirements, please refer to the system requirements for the version of MATLAB[7] or Octave that you are using.

In this manual, references to MATLAB usually apply to Octave as well.

2.2 Installation

Note to MATPOWER users: *MP-Opt-Model and its prerequisites, MIPS and MP-Test, are included when you install* MATPOWER. *There is generally no need to install MP-Opt-Model separately. You can skip directly to step 3 to verify.*

Installation and use of MP-Opt-Model requires familiarity with the basic operation of MATLAB or Octave, including setting up your MATLAB/Octave path.

Step 1: Clone the repository or download and extract the zip file of the MP-Opt-Model distribution from the MP-Opt-Model project page[8] to the location of your choice. The files in the resulting `mp-opt-model` or `mp-opt-modelXXX` directory, where `XXX` depends on the version of MP-Opt-Model, should not need to be modified, so it is recommended that they be kept separate from your own code. We will use <*MPOM*> to denote the path to this directory.

---

[3] MATLAB is available from The MathWorks, Inc. (https://www.mathworks.com/). MATLAB is a registered trademark of The MathWorks, Inc.

[4] GNU Octave [3] is free software, available online at https://octave.org. All functionality except object copy constructors work on GNU Octave version 4.4 and later. MP-Opt-Model 4.2 and earlier required Octave 4.

[5] MIPS is available at https://github.com/MATPOWER/mips.

[6] MP-Test is available at https://github.com/MATPOWER/mptest.

[7] https://www.mathworks.com/support/sysreq/previous_releases.html

[8] https://github.com/MATPOWER/mp-opt-model

Step 2: Add the following directories to your MATLAB or Octave path:

- <*MPOM*>`/lib` – core MP-Opt-Model functions
- <*MPOM*>`/lib/t` – test scripts for MP-Opt-Model
- <*MPOM*>`/examples` – MP-Opt-Model examples

Step 3: At the MATLAB/Octave prompt, type `test_mp_opt_model` to run the test suite and verify that MP-Opt-Model is properly installed and functioning.[9] The result should resemble the following:

```
>> test_mp_opt_model
t_have_fcn.............ok
t_nested_struct_copy....ok
t_nleqs_master.........ok (30 of 150 skipped)
t_pnes_master..........ok
t_qps_master...........ok (144 of 504 skipped)
t_qcqps_master.........ok (94 of 651 skipped)
t_miqps_master.........ok (128 of 371 skipped)
t_nlps_master..........ok (16 of 540 skipped)
t_mp_opt_model.........ok
t_mm_solve_leqs........ok
t_mm_solve_nleqs.......ok (36 of 196 skipped)
t_mm_solve_pne.........ok
t_mm_solve_qcqps.......ok (6 of 214 skipped)
t_mm_solve_qps.........ok (120 of 449 skipped)
t_mm_solve_miqps.......ok (106 of 261 skipped)
t_mm_solve_nlps........ok (9 of 506 skipped)
t_opt_model............ok
t_om_solve_leqs........ok
t_om_solve_nleqs.......ok (36 of 196 skipped)
t_om_solve_pne.........ok
t_om_solve_qcqps.......ok (6 of 214 skipped)
t_om_solve_qps.........ok (120 of 449 skipped)
t_om_solve_miqps.......ok (106 of 261 skipped)
t_om_solve_nlps........ok (9 of 506 skipped)
All tests successful (6814 passed, 966 skipped of 7780)
Elapsed time 18.02 seconds.
```

2.3 Sample Usage

Suppose we have the following constrained 4-dimensional quadratic programming (QP) problem with two 2-dimensional variables, y and z, and two constraints, one

[9]The tests require functioning installations of MP-Test and MIPS.

equality and the other inequality, along with lower bounds on all of the variables.

$$\min_{y,z} \frac{1}{2} \left[\begin{array}{cc} y^\mathsf{T} & z^\mathsf{T} \end{array} \right] Q \left[\begin{array}{c} y \\ z \end{array} \right] \tag{2.1}$$

subject to

$$A_1 \left[\begin{array}{c} y \\ z \end{array} \right] = b_1 \tag{2.2}$$

$$A_2 y \leq u_2 \tag{2.3}$$

$$y \geq y_{\min} \tag{2.4}$$

$$z \leq z_{\max} \tag{2.5}$$

And suppose the data for the problem is provided as follows.

```
%% variable initial values
y0 = [1; 0];
z0 = [0; 1];

%% variable lower bounds
ymin = [0; 0];
zmax = [0; 2];

%% constraint data
A1 = [ 6 1 5 -4 ];  b1 = 4;
A2 = [ 4 9 ];       u2 = 2;

%% quadratic cost coefficients
Q = [ 8  1 -3 -4;
      1  4 -2 -1;
     -3 -2  5  4;
     -4 -1  4  12  ];
```

Below, we will show two approaches to construct and solve the problem. The first method, based on the mathematical programming and optimization model class `mp.opt_model`, allows you to add variables, constraints and costs to the model individually. Then `mp.opt_model` automatically assembles and solves the full model automatically.

```
%%-----  METHOD 1  -----
%% build model
mm = mp.opt_model;
mm.var.add('y', 2, y0, ymin);
mm.var.add('z', 2, z0, [], zmax);
mm.lin.add(mm.var, 'lincon1', A1, b1, b1);
mm.lin.add(mm.var, 'lincon2', A2, [], u2, {'y'});
mm.qdc.add(mm.var, 'cost', Q, []);

%% solve model
[x, f, exitflag, output, lambda] = mm.solve();
```

The second method requires you to construct the parameters for the full problem manually, then call the solver function directly.

```
%%-----  METHOD 2  -----
%% assemble model parameters manually
xmin = [ymin; -Inf(2,1)];
xmax = [ Inf(2,1); zmax];
x0 = [y0; z0];
A = [ A1; A2 0 0];
l = [ b1; -Inf ];
u = [ b1;  u2  ];

%% solve model
[x, f, exitflag, output, lambda] = qps_master(Q, [], A, l, u, xmin, xmax, x0);
```

The above examples are included in <*MPOM*>`examples/qp_ex1.m` along with some commands to print the results, yielding the output below for each approach:

```
f = 1.875      exitflag = 1

             var bound shadow prices
     x     lambda.lower  lambda.upper
   0.5000      0.0000        0.0000
   0.0000      5.1250        0.0000
  -0.0000      0.0000        8.7500
  -0.2500      0.0000        0.0000

constraint shadow prices
lambda.mu_l  lambda.mu_u
   1.2500       0.0000
   0.0000       0.6250
```

Both approaches can be applied to each of the types of problems that MP-Opt-Model handles, namely, LP, QP, MILP, MIQP, QCQP, NLP and linear and nonlinear equations, including families of parameterized nonlinear equations.

An options struct can be passed to the `solve` method or the `qps_master` function to select a specific solver, control the level of progress output, or modify a solver's default parameters.

2.4 Documentation

There are two primary sources of documentation for MP-Opt-Model. The first is this manual, which gives an overview of the capabilities and structure of MP-Opt-Model and describes the formulations behind the code. It can be found in your MP-Opt-Model distribution at <*MPOM*>`/docs/MP-Opt-Model-manual.pdf` and the latest version is always available at: `https://matpower.org/docs/MP-Opt-Model-manual.pdf`.

The second is the online MP-Opt-Model Reference Manual[10], whose content is also available via the built-in `help` command. As with the built-in functions and toolbox routines in MATLAB and Octave, you can type `help` followed by the name of a command or M-file to get help on that particular function. Many of the M-files in MP-Opt-Model have such documentation and this should be considered the main reference for the calling options for each function. See Appendix A for a list of MP-Opt-Model functions.

[10] `https://matpower.org/doc/mpom/`

3 MP-Opt-Model – Overview

MP-Opt-Model[11] and its functionality can be divided into two main parts, plus a few additional utility functions.

The first part consists of interfaces to various numerical optimization solvers and the wrapper functions that provide a single common interface to all supported solvers for a particular class of problems. There is currently a common interface provided for each of the following:

- linear (LP) and quadratic (QP) programming problems
- mixed-integer linear (MILP) and quadratic (MIQP) programming problems
- quadratically-constrained quadratic programming problems (QCQP)
- nonlinear programming problems (NLP)
- linear equations (LEQ)
- nonlinear equations (NLEQ)
- parameterized nonlinear equations (PNE)

The second part consists of a mathematical programming and optimization model class designed to help the user construct an optimization or zero-finding problem by adding variables, constraints and/or costs, then solve the problem and extract the solution in terms of the individual sets of variables, constraints and/or costs provided.

Finally, MP-Opt-Model includes a utlity function that can be used to get information about the availability of optional functionality, another to help with copying nested struct data, and a function that provides version information on the current MP-Opt-Model installation.

[11]The name MP-Opt-Model was originally derived from "MATPOWER Optimization Model," referring to the object used to encapsulate the optimization problem formed by MATPOWER when solving an optimal power flow (OPF) problem. However, given its subsequent expanded scope, it stands for "Mathematical Programming and Optimization Model"

4 Solver Interface Functions

4.1 LP/QP Solvers – `qps_master`

The `qps_master` function provides a common quadratic programming solver interface for linear programming (LP) and quadratic (QP) programming problems, that is, problems of the form:

$$\min_x \frac{1}{2} x^\mathsf{T} H x + c^\mathsf{T} x \tag{4.1}$$

subject to

$$l \leq Ax \leq u \tag{4.2}$$

$$x_{\min} \leq x \leq x_{\max}. \tag{4.3}$$

This function can be used to solve the problem with any of the available solvers by calling it as follows,

```
[x, f, exitflag, output, lambda] = ...
    qps_master(H, c, A, l, u, xmin, xmax, x0, opt);
```

where the input and output arguments are described in Tables 4-1 and 4-2, respectively, and the options in Table 4-3. Alternatively, the input arguments can be packaged as fields in a `problem` struct and passed in as a single argument, where all fields are (individually) optional.

```
[x, f, exitflag, output, lambda] = qps_master(problem);
```

The calling syntax is very similar to that used by `quadprog` from the Matlab Optimization Toolbox, with the primary difference that the linear constraints are specified in terms of a single doubly-bounded linear function ($l \leq Ax \leq u$) as opposed to separate equality constrained ($A_{eq}x = b_{eq}$) and upper bounded ($Ax \leq b$) functions.

The `qps_master` function is simply a master wrapper around corresponding functions specific to each solver, namely, `qps_bpmpd`, `qps_clp`, `qps_cplex`, `qps_glpk`, `qps_gurobi`, `qps_highs`, `qps_ipopt`, `qps_knitro`, `qps_mips`, `qps_mosek`, and `qps_ot`. Each of these functions has an interface identical to that of `qps_master`, with the exception of the options struct for `qps_mips`, which is a simple MIPS options struct.

Table 4-1: Input Arguments for `qps_master`[†]

name	description
`H`	(possibly sparse) matrix H of quadratic cost coefficients
`c`	column vector c of linear cost coefficients
`A`	(possibly sparse) matrix A of linear constraint coefficients
`l`	column vector l of lower bounds on Ax, defaults to $-\infty$
`u`	column vector u of upper bounds on Ax, defaults to $+\infty$
`xmin`	column vector $x_{\min}$ of lower bounds on x, defaults to $-\infty$
`xmax`	column vector $x_{\max}$ of upper bounds on x, defaults to $+\infty$
`x0`	optional starting value of optimization vector x *(ignored by some solvers)*
`opt`	optional options struct (all fields optional), see Table 4-3 for details
`problem`	alternative, single argument input struct with fields corresponding to arguments above

[†] All arguments are individually optional, though enough must be supplied to define a meaningful problem.

Table 4-2: Output Arguments for `qps_master`

name	description
`x`	solution vector x
`f`	final objective function value $f(x) = \frac{1}{2}x^{\mathsf{T}}Hx + c^{\mathsf{T}}x$
`exitflag`	exit flag
	1 – converged successfully
	≤ 0 – solver-specific failure code
`output`	output struct with the following fields:
	`alg` – algorithm code of solver used
	(others) – solver-specific fields
`lambda`	struct containing the Langrange and Kuhn-Tucker multipliers on the constraints, with fields:
	`mu_l` – lower (left-hand) limit on linear constraints
	`mu_u` – upper (right-hand) limit on linear constraints
	`lower` – lower bound on optimization variables
	`upper` – upper bound on optimization variables

Table 4-3: Options for `qps_master`

name	default	description
`alg`	`'DEFAULT'`	determines which solver to use `'DEFAULT'` – automatic, first available of Gurobi, CPLEX, MOSEK, Optimization Toolbox (if MATLAB), HiGHS, GLPK (LP only), BPMPD, MIPS `'BPMPD'` – BPMPD* `'CLP'` – CLP* `'CPLEX'` – CPLEX* `'GLPK'` – GLPK**(LP only)* `'GUROBI'` – Gurobi* `'HIGHS'` – HiGHS* `'IPOPT'` – IPOPT* `'MIPS'` – MIPS, **M**ATPOWER **I**nterior **P**oint **S**olver `'MOSEK'` – MOSEK* `'OT'` – MATLAB Opt Toolbox, `quadprog`, `linprog`
`verbose`	0	amount of progress info to be printed 0 – print no progress info 1 – print a little progress info 2 – print a lot of progress info 3 – print all progress info
`bp_opt`	*empty*	options vector for `bp`*
`clp_opt`	*empty*	options vector for CLP*
`cplex_opt`	*empty*	options struct for CPLEX*
`glpk_opt`	*empty*	options struct for GLPK*
`grb_opt`	*empty*	options struct for Gurobi*
`highs_opt`	*empty*	options struct for HiGHS*
`ipopt_opt`	*empty*	options struct for IPOPT*
`linprog_opt`	*empty*	options struct for `linprog`*
`mips_opt`	*empty*	options struct for MIPS
`mosek_opt`	*empty*	options struct for MOSEK*
`quadprog_opt`	*empty*	options struct for `quadprog`*

* Requires the installation of an optional package. See Appendix B for details on the corresponding package.

4.1.1 QP Example

The following code shows an example of using `qps_master` to solve a simple 4-dimensional QP problem[12] using the default solver.

```
H = [   1003.1  4.3     6.3     5.9;
        4.3     2.2     2.1     3.9;
        6.3     2.1     3.5     4.8;
        5.9     3.9     4.8     10  ];
c = zeros(4,1);
A = [   1       1       1       1;
        0.17    0.11    0.10    0.18    ];
l = [1; 0.10];
u = [1; Inf];
xmin = zeros(4,1);
x0 = [1; 0; 0; 1];
opt = struct('verbose', 2);
[x, f, s, out, lambda] = qps_master(H, c, A, l, u, xmin, [], x0, opt);
```

Other examples of using `qps_master` to solve LP and QP problems can be found in `t_qps_master.m`.

[12]From https://v8doc.sas.com/sashtml/iml/chap8/sect12.htm.

4.2 MILP/MIQP Solvers – miqps_master

The miqps_master function provides a common **m**ixed-**i**nteger **q**uadratic **p**rogramming **s**olver interface for mixed-integer linear programming (MILP) and mixed-integer quadratic programming (MIQP) problems. The form of the problem is identical to (4.1)–(4.3), with the addition of two possible additional constraints, namely,

$$x_i \in \mathbb{Z}, \qquad \forall i \in \mathcal{I} \tag{4.4}$$

$$x_j \in \{0, 1\}, \quad \forall j \in \mathcal{B}, \tag{4.5}$$

where $\mathcal{I}$ and $\mathcal{B}$ are the sets of indices of variables that are restricted to integer or binary values, respectively.

This function can be used to solve the problem with any of the available solvers by calling it as follows,

```
[x, f, exitflag, output, lambda] = ...
    miqps_master(H, c, A, l, u, xmin, xmax, x0, vtype, opt);
[x, f, exitflag, output, lambda] = miqps_master(problem);
```

The calling syntax for miqps_master is identical to that used by qps_master with the exception of a single new input argument, vtype, to specify the variable type, just before the options struct. The input arguments and options for miqps_master are described in Tables 4-4 and 4-5, respectively. The outputs are identical to those shown in Table 4-2 for qps_master.

Table 4-4: Input Arguments for miqps_master

name	description
all qps_master *input args from Table 4-1, with the following additions/modifications*	
vtype	character string of length n_x (number of elements in x), or 1 (value applies to all variables in x), specifying variable type; allowed values are:† 'C' – continuous (default) 'B' – binary 'I' – integer

† CPLEX and Gurobi also include 'S' for semi-continuous and 'N' for semi-integer, but these have not been tested.

By default, unless the skip_prices option is set to 1, once miqps_master has found the integer solution, it constrain the integer variables to their solved values and call qps_matpower on the resulting problem to determine the shadow prices in lambda.

Table 4-5: Options for `miqps_master`

name	default	description
`alg`	`'DEFAULT'`	determines which solver to use `'DEFAULT'` – automatic, first available of Gurobi, CPLEX, MOSEK, Optimization Toolbox (if MATLAB, MILP only), HiGHS (MILP only), GLPK (MILP only) `'CPLEX'` – CPLEX* `'GLPK'` – GLPK* *(LP only)* `'GUROBI'` – Gurobi* `'HIGHS'` – HiGHS* `'MOSEK'` – MOSEK* `'OT'` – MATLAB Opt Toolbox, `intlinprog`
`verbose`	0	amount of progress info to be printed 0 – print no progress info 1 – print a little progress info 2 – print a lot of progress info 3 – print all progress info
`skip_prices`	0	flag that specifies whether or not to skip the price computation stage, in which the problem is re-solved for only the continuous variables, with all others being constrained to their solved values
`price_stage_warn_tol`	10^{-7}	tolerance on the objective function value and primal variable relative mismatch required to avoid mismatch warning message
`cplex_opt`	*empty*	options struct for CPLEX*
`glpk_opt`	*empty*	options struct for GLPK*
`grb_opt`	*empty*	options struct for Gurobi*
`highs_opt`	*empty*	options struct for HiGHS*
`intlinprog_opt`	*empty*	options struct for `intlinprog`*
`mosek_opt`	*empty*	options struct for MOSEK*

* Requires the installation of an optional package. See Appendix B for details on the corresponding package.

The `miqps_master` function is simply a master wrapper around corresponding functions specific to each solver, namely, `miqps_cplex`, `miqps_glpk`, `miqps_gurobi`, `miqps_highs`, `miqps_mosek`, and `miqps_ot`. Each of these functions has an interface identical to that of `miqps_master`.

4.2.1 MILP Example

The following code shows an example of using `miqps_master` to solve a simple 2-dimensional MILP problem[13] using the default solver.

```
c = [-2; -3];
A = sparse([195 273; 4 40]);
u = [1365; 140];
xmax = [4; Inf];
vtype = 'I';
opt = struct('verbose', 2);
p = struct('c', c, 'A', A, 'u', u, 'xmax', xmax, 'vtype', vtype, 'opt', opt);
[x, f, s, out, lam] = miqps_master(p);
```

Other examples of using `miqps_master` to solve MILP and MIQP problems can be found in `t_miqps_master.m`.

4.3 QCQP Solvers – `qcqps_master`

The `qcqps_master` function provides a common **q**uadratically-**c**onstrained **q**uadratic **p**rogramming **s**olver interface for QCQP problems, that is, problems of the form:

$$\min_x \frac{1}{2} x^\mathsf{T} H x + c^\mathsf{T} x \tag{4.6}$$

subject to

$$l_{\mathrm{q}i} \le \tfrac{1}{2} x^\mathsf{T} Q_i x + b_i x \le u_{\mathrm{q}i}, \quad \forall i = 1, \dots, n_q \tag{4.7}$$

$$l \le Ax \le u \tag{4.8}$$

$$x_{\min} \le x \le x_{\max}. \tag{4.9}$$

where b_i is a row vector representing row i of a matrix B, and $l_{\mathrm{q}i}$ and $u_{\mathrm{q}i}$ are element i of lower and upper bound vectors l_q and u_q, respectively.

This function can be used to solve the problem with any of the available solvers by calling it as follows,

```
[x, f, exitflag, output, lambda] = ...
    qcqps_master(H, c, Q, B, lq, uq, A, l, u, xmin, xmax, x0, opt);
```

[13]From MOSEK 6.0 Guided Tour, section 7.13.1, https://docs.mosek.com/6.0/toolbox/node009.html.

where the input and output arguments are described in Tables 4-6 and 4-7, respectively, and the options in Table 4-8. Alternatively, the input arguments can be packaged as fields in a `problem` struct and passed in as a single argument, where all fields are (individually) optional.

```
[x, f, exitflag, output, lambda] = qcqps_master(problem);
```

Table 4-6: Input Arguments for `qcqps_master`†

name	description
`H`	(possibly sparse) matrix H of quadratic cost coefficients
`c`	column vector c of linear cost coefficients
`Q`	$n_q \times 1$ cell array of sparse quadratic matrices Q_i
`B`	(possibly sparse) matrix B of linear coefficients of quadratic constraints
`lq`	column vector l_q of lower bounds on quadratic constraints, defaults to $-\infty$
`uq`	column vector u_q of upper bounds on quadratic constraints, defaults to $+\infty$
`A`	(possibly sparse) matrix A of linear constraint coefficients
`l`	column vector l of lower bounds on Ax, defaults to $-\infty$
`u`	column vector u of upper bounds on Ax, defaults to $+\infty$
`xmin`	column vector $x_{\min}$ of lower bounds on x, defaults to $-\infty$
`xmax`	column vector $x_{\max}$ of upper bounds on x, defaults to $+\infty$
`x0`	optional starting value of optimization vector x *(ignored by some solvers)*
`opt`	optional options struct (all fields optional), see Table 4-8 for details
`problem`	alternative, single argument input struct with fields corresponding to arguments above

† All arguments are individually optional, though enough must be supplied to define a meaningful problem.

The `qcqps_master` function is simply a master wrapper around corresponding functions specific to each solver, namely, `qcqps_gurobi`, `qcqps_knitro`, and `qcqps_nlps`. Each of these functions has an interface identical to that of `qcqps_master`. IPOPT, MIPS, `fmincon`, and optionally Artelys Knitro, are handled by `qcqps_nlps` which calls `nlps_master` with the appropriate `alg` option to solve the problem.

Table 4-7: Output Arguments for `qcqps_master`

name	description
`x`	solution vector x
`f`	final objective function value $f(x) = \frac{1}{2}x^\mathsf{T}Hx + c^\mathsf{T}x$
`exitflag`	exit flag 1 – converged successfully ≤ 0 – solver-specific failure code
`output`	output struct with the following fields: `alg` – algorithm code of solver used *(others)* – solver-specific fields
`lambda`	struct containing the Langrange and Kuhn-Tucker multipliers on the constraints, with fields: `mu_l` – lower (left-hand) limit on linear constraints `mu_u` – upper (right-hand) limit on linear constraints `mu_lq` – lower (left-hand) limit on quadratic constraints `mu_uq` – upper (right-hand) limit on quadratic constraints `lower` – lower bound on optimization variables `upper` – upper bound on optimization variables

Table 4-8: Options for `qcqps_master`

name	default	description
`alg`	`'DEFAULT'`	determines which solver to use `'DEFAULT'` – automatic, first available of Ipopt, Artelys Knitro, `fmincon`, MIPS `'FMINCON'` – Matlab Opt Toolbox, `fmincon`* `'GUROBI'` – Gurobi* `'IPOPT'` – Ipopt* `'KNITRO'` – Artelys Knitro* `'KNITRO_NLP'` – Artelys Knitro,* via `nlps_master()` `'MIPS'` – MIPS, **M**atpower **I**nterior **P**oint **S**olver
`verbose`	0	amount of progress info to be printed 0 – print no progress info 1 – print a little progress info 2 – print a lot of progress info 3 – print all progress info
`fmincon_opt`	*empty*	options struct for `fmincon`*
`grb_opt`	*empty*	options struct for Gurobi*
`ipopt_opt`	*empty*	options struct for Ipopt*
`knitro_opt`	*empty*	options struct for Artelys Knitro*
`mips_opt`	*empty*	options struct for MIPS

* Requires the installation of an optional package. See Appendix B for details on the corresponding package.

4.3.1 QCQP Example

The following code shows an example of using `qcqps_master` to solve a simple 3-dimensional QCQP problem[14] using the default solver.

```
H = [];
c = [-1;0;0];
Q = { sparse([2 0 0; 0 2 0; 0 0 -2]); sparse([2 0 0; 0 0 -2; 0 -2 0]) };
B = zeros(2, 3);
lq = [-Inf;-Inf];
uq = [0; 0];
A = [ 1 1 1 ];  b = 1;
x0 = [0; 0; 1];
xmin = [0; 0; 0];
opt = struct('verbose', 2);
[x, f, exitflag, output, lambda] = ...
    qcqps_master(H, c, Q, B, lq, uq, A, b, b, xmin, [], x0, opt);
```

Other examples of using `qcqps_master` to solve QCQP problems can be found in `t_qcqps_master.m`.

4.4 NLP Solvers – `nlps_master`

The `nlps_master` function provides a common **n**on**l**inear **p**rogramming **s**olver interface for general nonlinear programming (NLP) problems, that is, problems of the form:

$$\min_x f(x) \tag{4.10}$$

subject to

$$g(x) = 0 \tag{4.11}$$
$$h(x) \leq 0 \tag{4.12}$$
$$l \leq Ax \leq u \tag{4.13}$$
$$x_{\min} \leq x \leq x_{\max} \tag{4.14}$$

where $f\colon \mathbb{R}^n \to \mathbb{R}$, $g\colon \mathbb{R}^n \to \mathbb{R}^m$ and $h\colon \mathbb{R}^n \to \mathbb{R}^p$.

This function can be used to solve the problem with any of the available solvers by calling it as follows,

[14] From https://docs.gurobi.com/projects/examples/en/current/examples/matlab/qcp.html.

```
[x, f, exitflag, output, lambda] = ...
    nlps_master(f_fcn, x0, A, l, u, xmin, xmax, gh_fcn, hess_fcn, opt);
```

where the input and output arguments are described in Tables 4-9 and 4-10, respectively. Alternatively, the input arguments can be packaged as fields in a `problem` struct and passed in as a single argument, where all fields except `f_fcn` and `x0` are optional.

```
[x, f, exitflag, output, lambda] = nlps_master(problem);
```

The calling syntax for `nlps_master` is nearly identical to that of MIPS and very similar to that used by `fmincon` from the MATLAB Optimization Toolbox. The primary difference from `fmincon` is that the linear constraints are specified in terms of a single doubly-bounded linear function ($l \leq Ax \leq u$) as opposed to separate equality constrained ($A_{eq}x = b_{eq}$) and upper bounded ($Ax \leq b$) functions.

The user-defined functions for evaluating the objective function, constraints and Hessian are identical to those required by MIPS. That is, they are identical to those required by `fmincon`, with one exception described below for the Hessian evaluation function. Specifically, `f_fcn` should return `f` as the scalar objective function value $f(x)$, `df` as an $n \times 1$ vector equal to ∇f and, unless `gh_fcn` is provided and the Hessian is computed by `hess_fcn`, `d2f` as an $n \times n$ matrix equal to the Hessian $\frac{\partial^2 f}{\partial x^2}$. Similarly, the constraint evaluation function `gh_fcn` must return the $m \times 1$ vector of nonlinear equality constraint violations $g(x)$, the $p \times 1$ vector of nonlinear inequality constraint violations $h(x)$ along with their gradients in `dg` and `dh`. Here `dg` is an $n \times m$ matrix whose j^{th} column is ∇g_j and `dh` is $n \times p$, with j^{th} column equal to ∇h_j. Finally, for cases with nonlinear constraints, `hess_fcn` returns the $n \times n$ Hessian $\frac{\partial^2 \mathcal{L}}{\partial x^2}$ of the Lagrangian function

$$\mathcal{L}(x, \lambda, \mu, \sigma) = \sigma f(x) + \lambda^{\mathsf{T}} g(x) + \mu^{\mathsf{T}} h(x) \tag{4.15}$$

for given values of the multipliers λ and μ, where σ is the `cost_mult` scale factor for the objective function. Unlike `fmincon`, some solvers, such as `mips`, pass this scale factor to the Hessian evaluation function in the 3rd input argument.

The use of `nargout` in `f_fcn` and `gh_fcn` is recommended so that the gradients and Hessian are only computed when required.

The `nlps_master` function is simply a master wrapper around corresponding functions specific to each solver, namely, `mips`, `nlps_fmincon`, `nlps_ipopt`, and `nlps_knitro`. Each of these functions has an interface identical to that of `nlps_master`, with the exception of the options struct for `mips`, which is a simple MIPS options struct.

Table 4-9: Input Arguments for `nlps_master`†

name	description
`f_fcn`	handle to function that evaluates the objective function, its gradients and Hessian‡ for a given value of x, with calling syntax: `[f, df, d2f] = f_fcn(x)`
`x0`	starting value of optimization vector x
`A`, `l`, `u`	define optional linear constraints $l \leq Ax \leq u$, where default values for elements of `l` and `u` are `-Inf` and `Inf`, respectively.
`xmin`, `xmax`	optional lower and upper bounds on x, with defaults `-Inf` and `Inf`, respectively
`gh_fcn`	handle to function that evaluates the optional nonlinear constraints and their gradients for a given value of x, with calling syntax: `[h, g, dh, dg] = gh_fcn(x)` where the columns of `dh` and `dg` are the gradients of the corresponding elements of `h` and `g`, i.e. `dh` and `dg` are transposes of the Jacobians of `h` and `g`, respectively
`hess_fcn`	handle to function that computes the Hessian‡of the Lagrangian for given values of x, λ and μ, where λ and μ are the multipliers on the equality and inequality constraints, g and h, respectively, with calling syntax: `Lxx = hess_fcn(x, lam, cost_mult)`, where $\lambda =$ `lam.eqnonlin`, $\mu =$ `lam.ineqnonlin` and `cost_mult` is a parameter used to scale the objective function
`opt`	optional options struct (all fields optional), see Table 4-11 for details
`problem`	alternative, single argument input struct with fields corresponding to arguments above

† All inputs are optional except `f_fcn` and `x0`.
‡ If `gh_fcn` is provided then `hess_fcn` is also required. Specifically, if there are nonlinear constraints, the Hessian information must be provided by the `hess_fcn` function and it need not be computed in `f_fcn`.

4.4.1 NLP Example 1

The following code, included as `nlps_master_ex1.m` in <*MPOM*>`examples`, shows a simple example of using `nlps_master` to solve a 2-dimensional unconstrained optimization of Rosenbrock's "banana" function[15]

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2. \tag{4.16}$$

First, create a function that will evaluate the objective function, its gradients and Hessian, for a given value of x. In this case, the coefficient of the first term is defined as a paramter `a`.

[15] https://en.wikipedia.org/wiki/Rosenbrock_function

Table 4-10: Output Arguments for `nlps_master`

name	description	
`x`	solution vector	
`f`	final objective function value, $f(x)$	
`exitflag`	exit flag	
	1 – converged successfully	
	≤ 0 – solver-specific failure code	
`output`	output struct with the following fields:	
	`alg` – algorithm code of solver used	
	(others) – solver-specific fields	
`lambda`	struct containing the Langrange and Kuhn-Tucker multipliers on the constraints, with fields:	
	`eqnonlin`	nonlinear equality constraints
	`ineqnonlin`	nonlinear inequality constraints
	`mu_l`	lower (left-hand) limit on linear constraints
	`mu_u`	upper (right-hand) limit on linear constraints
	`lower`	lower bound on optimization variables
	`upper`	upper bound on optimization variables

Table 4-11: Options for `nlps_master`

name	default	description
`alg`	`'DEFAULT'`	determines which solver to use
		`'DEFAULT'` – automatic, current default is MIPS
		`'MIPS'` – MIPS, **M**ATPOWER **I**nterior **P**oint **S**olver
		`'FMINCON'` – MATLAB Opt Toolbox, `fmincon`*
		`'IPOPT'` – IPOPT*
		`'KNITRO'` – Artelys Knitro*
`verbose`	0	amount of progress info to be printed
		0 – print no progress info
		1 – print a little progress info
		2 – print a lot of progress info
`mips_opt`	*empty*	options struct for MIPS
`fmincon_opt`	*empty*	options struct for `fmincon`*
`ipopt_opt`	*empty*	options struct for IPOPT*
`knitro_opt`	*empty*	options struct for Artelys Knitro*

* Requires the installation of an optional package. See Appendix B for details on the corresponding package.

```
function [f, df, d2f] = banana(x, a)
f = a*(x(2)-x(1)^2)^2+(1-x(1))^2;
if nargout > 1          %% gradient is required
    df = [  4*a*(x(1)^3 - x(1)*x(2)) + 2*x(1)-2;
            2*a*(x(2) - x(1)^2)                     ];
    if nargout > 2      %% Hessian is required
        d2f = 4*a*[ 3*x(1)^2 - x(2) + 1/(2*a),  -x(1);
                    -x(1)                        1/2 ];
    end
end
```

Then, create a handle to the function, defining the value of the paramter `a` to be 100, set up the starting value of x, and call the `nlps_master` function to solve it.

```
>> f_fcn = @(x)banana(x, 100);
>> x0 = [-1.9; 2];
>> [x, f] = nlps_master(f_fcn, x0)

x =

     1
     1


f =

     0
```

4.4.2 NLP Example 2

The second example[16] solves the following 3-dimensional constrained optimization, printing the details of the solver's progress:

$$\min_{x} f(x) = -x_1 x_2 - x_2 x_3 \tag{4.17}$$

subject to

$$\begin{aligned} x_1^2 - x_2^2 + x_3^2 - 2 &\leq 0 \quad (4.18) \\ x_1^2 + x_2^2 + x_3^2 - 10 &\leq 0. \quad (4.19) \end{aligned}$$

First, create a function to evaluate the objective function and its gradients,[17]

```
function [f, df, d2f] = f2(x)
f = -x(1)*x(2) - x(2)*x(3);
if nargout > 1           %% gradient is required
    df = -[x(2); x(1)+x(3); x(2)];
    if nargout > 2       %% Hessian is required
        d2f = -[0 1 0; 1 0 1; 0 1 0];   %% actually not used since
    end                                 %% 'hess_fcn' is provided
end
```

[16]From https://en.wikipedia.org/wiki/Nonlinear_programming#3-dimensional_example.

[17]Since the problem has nonlinear constraints and the Hessian is provided by `hess_fcn`, this function will never be called with three output arguments, so the code to compute `d2f` is actually not necessary.

one to evaluate the constraints, in this case inequalities only, and their gradients,

```
function [h, g, dh, dg] = gh2(x)
h = [ 1 -1 1; 1 1 1] * x.^2 + [-2; -10];
dh = 2 * [x(1) x(1); -x(2) x(2); x(3) x(3)];
g = []; dg = [];
```

and another to evaluate the Hessian of the Lagrangian.

```
function Lxx = hess2(x, lam, cost_mult)
if nargin < 3, cost_mult = 1; end   %% allows to be used with 'fmincon'
mu = lam.ineqnonlin;
Lxx = cost_mult * [0 -1 0; -1 0 -1; 0 -1 0] + ...
        [2*[1 1]*mu 0 0; 0 2*[-1 1]*mu 0; 0 0 2*[1 1]*mu];
```

Then create a `problem` struct with handles to these functions, a starting value for x and an option to print the solver's progress. Finally, pass this struct to `nlps_master` to solve the problem and print some of the return values to get the output below.

```
function nlps_master_ex2(alg)
if nargin < 1
    alg = 'DEFAULT';
end
problem = struct( ...
    'f_fcn',    @(x)f2(x), ...
    'gh_fcn',   @(x)gh2(x), ...
    'hess_fcn', @(x, lam, cost_mult)hess2(x, lam, cost_mult), ...
    'x0',       [1; 1; 0], ...
    'opt',      struct('verbose', 2, 'alg', alg) ...
);
[x, f, exitflag, output, lambda] = nlps_master(problem);
fprintf('\nf = %g   exitflag = %d\n', f, exitflag);
fprintf('\nx = \n');
fprintf('   %g\n', x);
fprintf('\nlambda.ineqnonlin =\n');
fprintf('   %g\n', lambda.ineqnonlin);
```

```
>> nlps_master_ex2
MATPOWER Interior Point Solver -- MIPS, Version 1.5.2, 12-Jul-2025
 (using built-in linear solver)
 it    objective   step size   feascond     gradcond     compcond     costcond
----  ------------ --------- ------------ ------------ ------------ ------------
  0           -1                       0          1.5            5            0
  1    -5.3250167     1.6875           0     0.894235     0.850653      2.16251
  2    -7.4708991    0.97413    0.129183   0.00936418     0.117278     0.339269
  3    -7.0553031    0.10406           0   0.00174933    0.0196518    0.0490616
  4    -7.0686267   0.034574           0   0.00041301    0.0030084   0.00165402
  5    -7.0706104  0.0065191           0  1.53531e-05  0.000337971  0.000245844
  6    -7.0710134 0.00062152           0  1.22094e-07  3.41308e-05  4.99387e-05
  7    -7.0710623 5.7217e-05           0  9.84878e-10  3.41587e-06  6.05875e-06
  8    -7.0710673 5.6761e-06           0  9.73553e-12  3.41615e-07  6.15483e-07
Converged!

f = -7.07107   exitflag = 1

x =
   1.58114
   2.23607
   1.58114

lambda.ineqnonlin =
   0
   0.707107
```

To use a different solver such as `fmincon`, assuming it is available, simply specify it in the `alg` option.

```
>> nlps_master_ex2('FMINCON')
                                              First-order      Norm of
 Iter F-count            f(x)  Feasibility   optimality         step
    0       1   -1.000000e+00    0.000e+00    1.000e+00
    1       2   -5.718566e+00    0.000e+00    1.230e+00    1.669e+00
    2       3   -8.395115e+00    1.875e+00    8.080e-01    8.259e-01
    3       4   -7.034187e+00    0.000e+00    3.752e-02    2.965e-01
    4       5   -7.050896e+00    0.000e+00    1.890e-02    5.339e-02
    5       6   -7.071406e+00    4.921e-04    1.133e-03    2.770e-02
    6       7   -7.070872e+00    0.000e+00    1.962e-04    2.332e-03
    7       8   -7.071066e+00    0.000e+00    1.958e-06    2.418e-04

Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.

<stopping criteria details>

f = -7.07107   exitflag = 1

x =
   1.58114
   2.23607
   1.58114

lambda.ineqnonlin =
   1.08013e-06
   0.707107
```

This example can be found in `nlps_master_ex2.m` in <*MPOM*>`examples`. More example problems for `nlps_master` can be found in `t_nlps_master.m` in <*MPOM*>`lib/t`.

4.5 Nonlinear Equation Solvers – `nleqs_master`

The `nleqs_master` function provides a common **n**on**l**inear **eq**uation **s**olver interface for general nonlinear equations (NLEQ), that is, problems of the form:

$$f(x) = 0 \tag{4.20}$$

where $f\colon \mathbb{R}^n \to \mathbb{R}^n$.

This function can be used to solve the problem with any of the available solvers by calling it as follows,

```
[x, f, exitflag, output, jac] = nleqs_master(fcn, x0, opt);
```

where the input and output arguments are described in Tables 4-12 and 4-13, respectively. Alternatively, the input arguments can be packaged as fields in a `problem` struct and passed in as a single argument, where the `opt` field is optional.

```
[x, f, exitflag, output, jac] = nleqs_master(problem);
```

The calling syntax for `nleqs_master` is identical to that used by `fsolve` from the MATLAB Optimization Toolbox.

Table 4-12: Input Arguments for `nleqs_master`

name	description
`fcn`	handle to function that evaluates the function $f(x)$ and optionally its Jacobian $J(x)$ for a given value of x, with calling syntax: `f = fcn(x)`, or `[f, J] = fcn(x)` where selected solver algorithm determines whether `fcn` is required to return the Jacobian or not
`x0`	starting value of vector x
`opt`	optional options struct (all fields optional), see Table 4-14 for details
`problem`	alternative, single argument input struct with fields corresponding to arguments above

Table 4-13: Output Arguments for `nleqs_master`†

name	description
`x`	solution vector
`f`	final function value, $f(x)$
`exitflag`	exit flag 1 – converged successfully ≤ 0 – solver-specific failure code
`output`	output struct with the following fields: `alg` – algorithm code of solver used *(others)* – solver-specific fields
`jac`	final value of Jacobian matrix

† All output arguments are optional.

Table 4-14: Options for `nleqs_master`

name	default	description
`alg`	`'DEFAULT'`	determines which solver to use `'DEFAULT'` – automatic, current default is `'NEWTON'` `'NEWTON'` – Newton's method `'CORE'` – core algorithm, with arbitrary update function¶ `'FD'` – fast-decoupled Newton's method† `'FSOLVE'` – MATLAB Opt Toolbox, `fsolve`* `'GS'` – Gauss-Seidel method‡
`verbose`	0	amount of progress info to be printed 0 – print no progress info 1 – print a little progress info 2 – print a lot of progress info
`max_it`	0	maximum number of iterations§
`tol`	0	termination tolerance on $f(x)$§
`core_sp`	*empty*	solver parameters struct for `nleqs_core`¶
`fd_opt`	*empty*	options struct for fast-decoupled Newton's method, `nleqs_fd_newton`†
`fsolve_opt`	*empty*	options struct for `fsolve`*
`gs_opt`	*empty*	options struct for Gauss-Seidel method, `nleqs_gauss_seidel`‡
`newton_opt`	*empty*	options struct for Newton's method, `nleqs_newton`

* The `fsolve` function is included with GNU Octave, but on MATLAB it is part of the MATLAB Optimization Toolbox. See Appendix B for more information on the MATLAB Optimization Toolbox.

† Fast-decoupled Newton requires setting `fd_opt.jac_approx_fcn` to a function handle that returns Jacobian approximations. See `help nleqs_fd_newton` for more details.

‡ Gauss-Seidel requires setting `gs_opt.x_update_fcn` to a function handle that updates x. See `help nleqs_gauss_seidel` for more details.

§ A value of 0 indicates to use the solver's own default.

¶ The `opt.core_sp` field is required when `alg` is set to `'CORE'`. See `help nleqs_core` for details.

The `nleqs_master` function is simply a master wrapper around corresponding solver-specific functions, namely, `nleqs_newton`, `nleqs_fd_newton`, `nleqs_gauss_seidel` and `nleqs_fsolve`. Each of these functions has an interface identical to that of `nleqs_master`.

There is also a more general function named `nleqs_core` which takes an arbitrary, user-defined update function. In fact, `nleqs_core` provides the core implementation for both `nleqs_newton` and `nleqs_gauss_seidel`. See `help nleqs_core` for details.

4.5.1 NLEQ Example 1

The following code, included as `nleqs_master_ex1.m` in <*MPOM*>`examples`, shows a simple example of using `nleqs_master` to solve a 2-dimensional nonlinear function[18]

$$f(x) = \begin{bmatrix} x_1 + x_2 - 1 \\ -x_1^2 + x_2 + 5 \end{bmatrix} \tag{4.21}$$

First, create a function that will evaluate the $f(x)$ and its Jacobian $J(x)$ for a given value of x.

```
function [f, J] = f1(x)
f = [  x(1)    + x(2) - 1;
      -x(1)^2 + x(2) + 5    ];
if nargout > 1
    J = [1 1; -2*x(1) 1];
end
```

Then, call the `nleqs_master` function with a handle to that function and a starting value for x.

```
>> x = nleqs_master(@f1, [0;0])

x =

    2.0000
   -1.0000
```

Or, alternatively, create a `problem` struct with a handle to the function, a starting value for x and an option to print the solver's progress. Then, pass this struct to `nleqs_master` to solve the problem and print some of the return values to get the output below.

[18] https://www.chilimath.com/lessons/advanced-algebra/systems-non-linear-equations/

```
function nleqs_master_ex1(alg)
if nargin < 1
    alg = 'DEFAULT';
end
problem = struct( ...
    'fcn',  @f1, ...
    'x0',   [0; 0], ...
    'opt',  struct('verbose', 2, 'alg', alg) ...
);
[x, f, exitflag, output, jac] = nleqs_master(problem);
fprintf('\nexitflag = %d\n', exitflag);
fprintf('\nx = \n');
fprintf('   %2g\n', x);
fprintf('\nf = \n');
fprintf('   %12g\n', f);
fprintf('\njac =\n');
fprintf('   %2g  %2g\n', jac');
```

```
>> nleqs_master_ex1

 it     max residual
----  ----------------
  0      5.000e+00
  1      3.600e+01
  2      7.669e+00
  3      1.056e+00
  4      3.818e-02
  5      5.795e-05
  6      1.343e-10
Newton's method converged in 6 iterations.

exitflag = 1

x =
    2
   -1

f =
    2.22045e-16
   -1.34308e-10

jac =
    1   1
   -4   1
```

To use a different solver such as `fsolve`, assuming it is available, simply specify it in the `alg` option.

```
>> nleqs_master_ex1('FSOLVE')

                                         Norm of      First-order   Trust-region
 Iteration  Func-count     f(x)          step         optimality    radius
     0          1              26                            4               1
     1          2         18.7537              1          3.65               1
     2          3         9.28396            2.5          12.9             2.5
     3          4          0.0148        1.30162         0.493             2.5
     4          5     3.37211e-07      0.0340793       0.00232            3.25
     5          6     1.81904e-16    0.000164239      5.39e-08            3.25

Equation solved.

fsolve completed because the vector of function values is near zero
as measured by the value of the function tolerance, and
the problem appears regular as measured by the gradient.


exitflag = 1

x =
    2
   -1

f =
              0
   -1.34872e-08

jac =
    1   1
   -4   1
```

4.5.2 NLEQ Example 2

The following code, included as `nleqs_master_ex2.m` in <*MPOM*>`examples`, shows another simple example of using `nleqs_master` to solve a 2-dimensional nonlinear function.[19] This example includes the update function required for Gauss-Seidel and the

[19]From Christi Patton Luks, https://www.youtube.com/watch?v=pJG4yhtgerg

Jacobian approximation function required for the fast-decoupled Newton's method.

$$f(x) = \begin{bmatrix} x_1^2 + x_1 x_2 - 10 \\ x_2 + 3x_1 x_2^2 - 57 \end{bmatrix} \tag{4.22}$$

```
function [f, J] = f2(x)
f = [  x(1)^2 +   x(1)*x(2)   - 10;
       x(2)   + 3*x(1)*x(2)^2 - 57  ];
if nargout > 1
    J = [   2*x(1)+x(2)    x(1);
            3*x(2)^2       6*x(1)*x(2)+1    ];
end
```

```
function JJ = jac_approx_fcn2()
J = [7 2; 27 37];
JJ = {J(1,1), J(2,2)};
```

```
function x = x_update_fcn2(x, f)
x(1) = sqrt(10 - x(1)*x(2));
x(2) = sqrt((57-x(2))/3/x(1));
```

```
function nleqs_master_ex2(alg)
if nargin < 1
    alg = 'DEFAULT';
end
x0 = [1; 2];
opt = struct( ...
    'verbose', 2, ...
    'alg', alg, ...
    'fd_opt', struct( ...
        'jac_approx_fcn', @jac_approx_fcn2, ...
        'labels', {{'P','Q'}}), ...
    'gs_opt', struct('x_update_fcn', @x_update_fcn2) );
[x, f, exitflag, output] = nleqs_master(@f2, x0, opt);
fprintf('\nexitflag = %d\n', exitflag);
fprintf('\nx = \n');
fprintf('   %2g\n', x);
fprintf('\nf = \n');
fprintf('   %12g\n', f);
```

Fast-decoupled Newton example results:

```
>> nleqs_master_ex2('FD')

 iteration    max residual    max residual
block    #        f[P]            f[Q]
------ ----  --------------  --------------
  -      0       7.000e+00       4.300e+01
  P      1       2.000e+00       3.100e+01
  Q      1       3.243e-01       5.842e+00
  P      2       5.367e-03       4.723e+00
  Q      2       2.558e-01       4.767e-02
  P      3       7.894e-04       1.012e+00
  Q      3       5.417e-02       2.058e-03
  P      4       3.606e-05       2.100e-01
  Q      4       1.133e-02       8.642e-05
  P      5       1.583e-06       4.374e-02
  Q      5       2.363e-03       3.727e-06
  P      6       6.892e-08       9.116e-03
  Q      6       4.927e-04       1.617e-07
  P      7       2.997e-09       1.901e-03
  Q      7       1.027e-04       7.028e-09
  P      8       1.303e-10       3.963e-04
  Q      8       2.142e-05       3.055e-10
  P      9       5.665e-12       8.262e-05
  Q      9       4.466e-06       1.327e-11
  P     10       2.451e-13       1.723e-05
  Q     10       9.311e-07       5.969e-13
  P     11       1.066e-14       3.591e-06
  Q     11       1.941e-07       1.421e-14
  P     12       0.000e+00       7.488e-07
  Q     12       4.048e-08       7.105e-15
  P     13       0.000e+00       1.561e-07
  Q     13       8.439e-09       7.105e-15
Fast-decoupled Newton's method converged in 13 P- and 13 Q-iterations.

exitflag = 1

x =
    2
    3

f =
    8.43887e-09
   -7.10543e-15
```

Gauss-Seidel example results:

```
>> nleqs_master_ex2('GS')

 it     max residual
----  ----------------
  0      4.300e+01
  1      5.201e+00
  2      1.690e+00
  3      6.481e-01
  4      2.141e-01
  5      7.413e-02
  6      2.523e-02
  7      8.638e-03
  8      2.951e-03
  9      1.009e-03
 10      3.449e-04
 11      1.179e-04
 12      4.030e-05
 13      1.378e-05
 14      4.709e-06
 15      1.610e-06
 16      5.503e-07
 17      1.881e-07
 18      6.430e-08
 19      2.198e-08
 20      7.513e-09
Gauss-Seidel method converged in 20 iterations.

exitflag = 1

x =
    2
    3

f =
   -7.51313e-09
    4.48558e-09
```

4.6 Parameterized Nonlinear Equation Solver – `pnes_master`

Continuation methods or branch tracing methods can be used to trace, beginning from an initial solution point, a curve of solutions to a parameterized system of nonlinear equations of the form

$$f(x) = 0, \tag{4.23}$$

where $f\colon \mathbb{R}^{n+1} \to \mathbb{R}^n$.

The `pnes_master` function provides a common **p**arameterized **n**onlinear **e**quation **s**olver interface for general parameterized nonlinear equations (PNE). The current implementation assumes that the function $f(x)$ arises from a parameterization, such as a homotopy, where the scalar parameter λ is by convention the last element of x. If we denote the first n elements of x as y, we have

$$x = \begin{bmatrix} y \\ \lambda \end{bmatrix} \tag{4.24}$$

In a typical application, we may have nonlinear functions $g_0, g\colon \mathbb{R}^n \to \mathbb{R}^n$, where we have a known solution y_0 to the equation $g_0(y) = 0$, but a good starting point for finding the solution to $g(y) = 0$ is not available. In this case, we can define $f(x)$ as a homotopy with parameter λ,

$$f(x) = (1 - \lambda)g_0(y) + \lambda g(y), \tag{4.25}$$

and use a continuation method to trace a solution curve from y_0 and $\lambda = 0$ to y^* and $\lambda = 1$, where y^* is the desired solution to $g(y) = 0$.

Currently MP-Opt-Model includes only a single solver implementation for PNE problems based on a numerical continuation method commonly known as a predictor-corrector method [8]. This method involves adding another equation to the system which identifies the location of the current solution with respect to the previous or next solution. The continuation process can be diagrammatically shown by (4.26).

$$x^j \quad \xrightarrow{\textit{Predictor}} \quad \hat{x}^{j+1} \quad \xrightarrow{\textit{Corrector}} \quad x^{j+1} \tag{4.26}$$

where, x^j represents the current solution at step j, $\hat{x}^{j+1}$ is the predicted solution for the next step, and x^{j+1} is the next solution on the curve.

4.6.1 Parameterization

The values of x along the solution curve can parameterized in a number of ways [9, 10]. Parameterization is a mathematical way of identifying each solution so that the *next* solution or *previous* solution can be quantified. MP-Opt-Model includes three parameterization scheme options to quantify this relationship, detailed below, where σ is the continuation step size parameter and λ is the last element of x.

- **Natural parameterization** simply uses λ directly as the parameter, so the new λ is simply the previous value plus the step size.

$$p^j(x) = \lambda - \lambda^j - \sigma^j = 0 \tag{4.27}$$

- **Arc length parameterization** results in the following relationship, where the step size is equal to the 2-norm of the distance from one solution to the next.

$$p^j(x) = \sum_i (x_i - x_i^j)^2 - (\sigma^j)^2 = 0 \tag{4.28}$$

- **Pseudo arc length parameterization** [11] is MP-Opt-Model's default parameterization scheme, where the next point x on the solution curve is constrained to lie in the hyperplane running through the predicted solution $\hat{x}^{j+1}$ orthogonal to the tangent line from the previous corrected solution x^j. This relationship can be quantified by the function

$$p^j(x) = \left(x - x^j\right)^{\mathsf{T}} \bar{z}^j - \sigma^j = 0, \tag{4.29}$$

  where $\bar{z}^j$ is the normalized tangent vector at x^j and σ^j is the continuation step size parameter.

4.6.2 Predictor

The predictor is used to produce an estimate for the next solution. The better the prediction, the faster is the convergence to the solution point. MP-Opt-Model uses a tangent predictor for estimating the curve to the next solution. At step j, the tangent vector z^j at the current solution x^j is found by solving the linear system

$$\left[\begin{array}{c} \frac{\partial f}{\partial x} \\ \frac{\partial p^{j-1}}{\partial x} \end{array}\right] z^j = \left[\begin{array}{c} 0 \\ 1 \end{array}\right]. \tag{4.30}$$

The matrix on the left-hand side is simply the Jacobian of $f(x)$ with an additional row added. The extra row, required to make the system non-singular and define the magnitude of z^j, is the derivative of $p^{j-1}(x)$, the parameterization function from the previous step.

The resulting tangent vector is then normalized

$$\bar{z}^j = \frac{z^j}{||z^j||_2} \tag{4.31}$$

and used to compute the predicted approximation $\hat{x}^{j+1}$ to the next solution x^{j+1} using

$$\hat{x}^{j+1} = x^j + \sigma^j \bar{z}^j, \tag{4.32}$$

where σ^j is the continuation step size.

4.6.3 Corrector

The corrector stage at step j finds the next solution x^{j+1} by correcting the approximation $\hat{x}^{j+1}$ estimated by the predictor. A method such as Newton's method is used to find the next solution by solving the $n+1$ dimensional system in (4.33), where one of (4.27)–(4.29) has been added as an additional constraint to the parameterized nonlinear equations of (4.23).

$$\begin{bmatrix} f(x) \\ p^j(x) \end{bmatrix} = 0 \tag{4.33}$$

The corrector in MP-Opt-Model uses `nleqs_master` with its variety of avaiable solvers to solve (4.33) for each new solution point on the curve.

4.6.4 Step Length Control

Step length control is a key element affecting the computational efficiency of a continuation method. It affects the continuation method with two issues: (1) speed – how fast the corrector converges to a specified accuracy, and (2) robustness – whether the corrector converges to a true solution given a predicted point. MP-Opt-Model's numerical continuation can optionally use adaptive steps, where the step size σ is adjusted by a scaling factor α within specified limits.

$$\sigma^{j+1} = \alpha^j \sigma^j, \qquad \sigma_{\min} \leq \sigma^{j+1} \leq \sigma_{\max} \tag{4.34}$$

This scaling factor α^j for step j is limited to a maximum of 2 and is calculated from an error estimation between the predicted and corrected solutions γ^j as follows,

$$\alpha^j = 1 + \beta \left(\frac{\epsilon}{\gamma^j} - 1 \right), \qquad \alpha^j \leq 2, \tag{4.35}$$

where β is a damping factor, ϵ is a specified tolerance, and γ^j is given by

$$\gamma^j = \left\| x^{j+1} - \hat{x}^{j+1} \right\|_\infty . \tag{4.36}$$

4.6.5 Event Detection and Location

A numerical continuation *event* is triggered when the value of one of the elements of an event function changes sign from one continuation step to the next. The event occurs at the point where the corresponding value of the event function passes through zero. MP-Opt-Model provides event functions to detect the location at which the continuation curve reaches the following:

- a specified target λ value
- a limit or nose point
- the end of a full trace

Each event function is registered with an event name, a flag indicating whether or not the location of the event should be pinpointed, and if so, to within what tolerance. For events that are to be located, when an event interval is detected, that is, when an element of the event function value changes sign, MP-Opt-Model adjusts the continuation step size via a False Position or Regula Falsi method until it locates the point of the zero-crossing to within the specified tolerance.

The detection of an event zero, or even an event interval, can be used to trigger further actions. MP-Opt-Model includes a callback functionality that can be used to handle events. For example, the numerical continuation termination for nose point, target λ or full trace modes are all based on callback functions in conjunction with event detection.

User-defined event detection functions for `pnes_master` can be provided via the `events` option.

4.6.6 Callback Functions

MP-Opt-Model's continuation method provides a callback mechanism to give the user access to the iteration process for executing custom code at each iteration, for example, to implement custom incremental plotting of a solution curve or to handle a detected event. This callback mechanism is used internally to handle default plotting functionality as well as to handle termination events. The `pne_callback_default` function, for example, is collects the λ and x results from each predictor and corrector iteration and optionally plots the continuation curve.

The prototype for a `pnes_master` callback function is

```
function [nx, cx, s] = pne_callback_user(k, nx, cx, px, s, opt)
```

and the input and output arguments are described in Tables 4-15 through 4-17 and in the help for `pne_callback_default`. Each registered callback function is called in three different contexts, distinguished by the value of the first argument `k` as follows:

1. *initial* – called with $k = 0$, after initial solution, before first continuation step
2. *iterations* – called with $k > 0$, at each iteration, after predictor-corrector step
3. *final* – called with $k < 0$, after exiting predictor-corrector loop, inputs identical to last iteration call, except `k` negated

Table 4-15: Callback Input Arguments

name	description
`k`	continuation step iteration count
`cx`	current continuation state,* corresponding to most recent successful step
`nx`	next continuation state,* corresponding to proposed next step
`px`	previous continuation state,* corresponding to last step prior to `cx`
`s`	container struct with various flags, etc, with fields:
`.done`	termination flag, 1 → terminate, 0 → continue
`.done_msg`	char array containing reason for termination
`.warmstart`	struct with information needed for warm-starting a continuation problem‡
`.rollback`	scalar flag to indicate that the current step should be rolled back and retried with a different step size, etc.
`.events`	struct array listing any events detected for this step‡
`.results`	current value of results struct whose fields are to be included in the `output` struct returned by `pnes_master`
`opt`	`pnes_master` options struct

* See Table 4-17 for details of the continuation state.
† See Table 4-22 for details.
‡ See `pne_detect_events` for details of the `events` field.

Table 4-16: Callback Output Arguments

name	description
All are updated versions of the corresponding input arguments, see Table 4-15 for more details.	
`cx`	current continuation state,* update values in `cx` such as `this_step` or `this_parm` if `s.rollback` is true
`nx`	next continuation state,* update values in this state if `s.rollback` is false
`s`	container struct with various flags, etc, with fields:
`.done`	callback may set this to request termination
`.done_msg`	callback may assign the reason for termination
`.warmstart`	callback may create this field to prepare for a subsequent warm-started call to `pnes_master`†
`.rollback`	callback can request a rollback step, even if it was not indicated by an event function‡
`.events`	`msg` field for a given event may be updated§
`.results`	updated version of results struct whose fields are to be included in the `output` struct returned by `pnes_master`

* See Table 4-17 for details of the continuation state.
† See Table 4-22 for details.
‡ In this case, the callback should also modify the step size or parameterization to be used for the re-try, by setting the `this_step` or `this_parm` fields in `cx`.
§ See `pne_detect_events` for details of the `events` field.

Table 4-17: Fields of Continuation State Struct

name	description
`x_hat`	solution vector from predictor
`x`	solution vector from corrector
`z`	normalized tangent vector, $\bar{z}$
`default_step`	default step size
`default_parm`	handle to function implementing parameterization used by default*
`this_step`	step size for this step only
`this_parm`	handle to function implementing parameterization used for this step only*
`step`	current step size
`parm`	handle to function implementing current parameterization*
`events`	event log, struct array, see `pne_detect_events` for details
`cbs`	callback state, callback functions may add fields containing any information the function would like to pass from one invokation to the next, taking care not to step on fields being used by other callbacks, such as the `'default'` field used by `pne_callback_default`.
`efv`	cell array of event function values

* Typically a handle to one of `pne_pfcn_natural`, `pne_pfcn_arc_len`, or `pne_pfcn_pseudo_arc_len`.

The user can define their own callback functions which take the same form and are called in the same contexts as `pne_callback_default`. User callback functions are included via the `callbacks` option to `pnes_master`. This option takes a single *callback specification* or a cell array of them if defining multiple callbacks, where a *callback specification* takes one of the following forms: `fcn`, {`fcn`}, or {`fcn, priority`}.

- `fcn` - function handle to the callback function
- `priority` - numerical value specifying callback priority,[20] default = 20

User-defined callback functions for `pnes_master` can be provided via the `callbacks` option.

[20]See `pne_register_callbacks` for details.

4.6.7 pnes_master

This function can be used to trace the parameterized solution curve with any of the available solvers[21] by calling it as follows,

```
[x, f, exitflag, output, jac] = pnes_master(fcn, x0, opt);
```

where the input and output arguments are described in Tables 4-18 and 4-19, respectively. Alternatively, the input arguments can be packaged as fields in a `problem` struct and passed in as a single argument, where the `opt` field is optional.

```
[x, f, exitflag, output, jac] = pnes_master(problem);
```

Table 4-18: Input Arguments for `pnes_master`

name	description
`fcn`	handle to function that evaluates the function $f(x)$ and its Jacobian $J(x)$ for a given value of x, with calling syntax: `f = fcn(x)`, or `[f, J] = fcn(x)` where f is $n \times 1$, x is $(n+1) \times 1$, and J is $n \times (n+1)$.
`x0`	starting value of vector x
`opt`	optional options struct (all fields also optional), see Table 4-20 for details
`problem`	alternative, single argument input struct with fields corresponding to arguments above

[21] The current implementation includes only a single solver based on a predictor-corrector continuation method.

Table 4-19: Output Arguments for `pnes_master`†

name	description
`x`	solution vector
`f`	final function value, $f(x)$
`exitflag`	exit flag 1 – converged successfully ≤ 0 – solver-specific failure code
`output`	output struct with the following fields:
`corrector`	`output` return value from `nleqs_master` from final corrector run, see Table 4-13 for details
`iterations`	N, total number of continuation steps performed
`events`	struct array of size n_e of events detected, with the following fields:
`k`	continuation step at which event was located
`name`	name of detected event
`idx`	index(es) of critical element(s) in corresponding event function
`msg`	descriptive text detailing the event
`done_msg`	message describing cause of continuation termination
`steps`	$(N + 1)$ row vector of stepsizes taken at each continuation step
`lam_hat`	$(N + 1)$ row vector of $\hat{\lambda}$ values from prediction steps
`lam`	$(N + 1)$ row vector of λ values from correction steps
`max_lam`	maximum value of parameter λ (from `output.lam`)
`warmstart`	optional output with information needed for warm-starting an updated continuation problem, see Table 4-22 for details
`x_hat`‡	$n \times (N + 1)$ matrix of solution values from prediction steps
`x`‡	$n \times (N + 1)$ matrix of solution values from correction steps
(others)	depends on `opt.output_fcn`, a custom output function can add arbitrary fields to `output`
`jac`	final value of Jacobian matrix

† All output arguments are optional.
‡ This field is created by the default output function and may not be present if using a custom output function defined by `opt.output_fcn`.

Table 4-20: Options for `pnes_master`

name	default	description
`alg`	`'DEFAULT'`	determines which solver to use*
`verbose`	0	amount of progress info to be printed 0 – print no progress info 1–5 – print increasing level of progress info
`nleqs_opt`	*empty*	options struct for `nleqs_master` used for corrector stage, see Table 4-14 for details
`solve_base`	1	0/1 flag that determines whether or not to run a corrector stage for initial solution point, `x0`
`parameterization`	3	choice of parameterization 1 – natural 2 – arc length 3 – pseudo arc length
`stop_at`	`'NOSE'`	determines stopping criterion `'NOSE'` – stop when limit or nose point is reached `'FULL'` – trace full continuation curve λ_{stop} – numeric, stop upon reaching target λ value λ_{stop}
`max_it`	2000	maximum number of continuation steps
`step`	0.05	continuation step size
`adapt_step`	0	toggle adaptive step size feature 0 – adaptive step size disabled 1 – adaptive step size enabled
`adapt_step_damping`	0.7	damping factor β from (4.35) for adaptive step sizing
`adapt_step_tol`	10^{-3}	tolerance ϵ from (4.35) for adaptive step sizing
`adapt_step_ws`	1	scale factor for default initial step size when warm-starting with adaptive step size enabled
`step_min`	10^{-4}	minimum allowed continuation step size, σ_{min} from (4.34)
`step_max`	0.2	maximum allowed continuation step size, σ_{max} from (4.34)
`default_event_tol`	10^{-3}	default tolerance for event functions
`target_lam_tol`	0	tolerance for target λ detection†
`nose_tol`	0	tolerance for nose point detection†
`events`	*empty*	cell array of specs for user-defined event functions‡
`callbacks`	*empty*	cell array of specs for user-defined callback functions§
`output_fcn`	*empty*	custom output function called by `pne_callback_default()`
`plot`	–	struct of options to contol plotting of continuation curve by `pne_callback_default()`, see Table 4-21 for details
`warmstart`	*empty*	struct with information needed for warm-starting a continuation problem, see Table 4-22 for details

* Currently `'DEFAULT'` is the only option.
† A value of 0 means use the value of `default_event_tol`.
‡ Passed as `my_events` arg to `pne_register_events()`. For details see `help pne_register_events`.
§ Passed as `my_cbacks` arg to `pne_register_callbacks()`. For details see `help pne_register_callbacks` and `help pne_callback_default`.

Table 4-21: Plot Options for `pnes_master`*

name	default	description
`level`	0	control plotting of continuation curve 0 – do not plot continuation curve 1 – plot when completed 2 – plot incrementally at each continuation step 3 – same as 2, with `pause` at each step
`idx`	*empty*	index of quantity to plot, passed to `yfcn()`
`idx_default`	*empty*	function to provide default value for `idx`, if none provided
`xname`	`'lam'`	name of field in `output` holding values that determine horizontal coordinates of plot
`yname`	`'x'`	name of field in `output` holding values that determine vertical coordinates of plot
`xfcn`	`@(x)x`	handle to function that maps a value from the indicated field of `output` to a horizontal coordinate for plotting†
`yfcn`	`@(y,idx)y(idx, :)`	handle to function that maps a value from the indicated field of `output` and an index to be applied to that value to a vertical coordinate for plotting†
`xlabel`	`'\lambda'`	label for horizontal axis
`ylabel`	`'Variable Value'`	label for vertical axis
`title`	`'Value of Variable %d'`	plot title used for plot of single variable§
`title2`	`'Value of Multiple Variables'`	plot title used for plot of multiple variables
`legend`	`'Variable %d'`	legend label§

* Defines the fields for optional input `opt.plot`.
† Relevant field is indicated by the value of `opt.plot.xname`.
‡ Relevant field is indicated by the value of `opt.plot.yname`.
§ Can use `%d` as placeholder for index `idx` of quantity to plot.

Table 4-22: Warm-start Data for `pnes_master`*

name	description
`cont_steps`	current value of continuation step counter
`direction`	$+1$ or -1, for tracing of curve in same or opposite direction, respectively
`dir_from_jac_eigs`	0/1 flag to indicate whether to use the sign of the smallest eigenvalue of the Jacobian to determine the initial direction
`x`	current solution vector
`z`	current tangent vector
`xp`	previous step solution vector
`zp`	previous step tangent vector
`parm`	function handle for current parameterization function
`default_parm`	function handle for default parameterization fcn
`default_step`	default step size
`events`	current event log, same as `output.events`
`cbs`	struct containing user-defined callback state

* Defines the fields for optional input `opt.warmstart` and optional output `output.warmstart`.

4.6.8 PNE Example

The following code is a simplified version of the example included as `pne_ex1.m` in <*MPOM*>`examples`. It illustrates the use of `pnes_master` to solve a 2-dimensional parameterized nonlinear function of 3 variables.[22] Recall that x_3, the last element of x, corresponds to the parameter λ.

$$f(x) = \left[\begin{array}{c} x_1 + x_2 + 6x_3 - 1 \\ -x_1^2 + x_2 + 5 \end{array} \right] \tag{4.37}$$

First, create a function that will evaluate the $f(x)$ and its Jacobian $J(x)$ for a given value of x.

```
function [f, J] = f1p(x)
f = [  x(1)   + x(2) + 6*x(3) - 1;
      -x(1)^2 + x(2) + 5    ];
if nargout > 1
    J = [1 1 6; -2*x(1) 1 0];
end
```

Then, call the `pnes_master` function with a handle to that function, a starting value for x, and an option to make it trace the full continuation curve.

```
>> x = pnes_master(@f1p, [-1;0;0], struct('stop_at', 'FULL'))

x =

    2.0000
   -1.0000
         0
```

Or, alternatively, create a `problem` struct to encapsulate the 3 inputs. Here we include additional options for verbose output and some plot options to make it plot the continuation curves for the first 2 variables. Then, pass this struct to `pnes_master` to solve the problem and print some of the return values to produce the output below and the plot shown in Figure 4-1.

[22] Based on a similar problem from https://www.chilimath.com/lessons/advanced-algebra/systems-non-linear-equations/.

```
function pne_ex1
opt = struct( 'verbose', 2, 'stop_at', 'FULL', 'step', 0.6);
opt.plot = struct( 'level', 2, 'idx', 1:2, ...
    'title2', 'PNE Continuation Example', 'legend', 'x_%d');
problem = struct('fcn', @f1p, 'x0', [-1;0;0], 'opt',  opt);
[x, f, exitflag, output, jac] = pnes_master(problem);
fprintf('\nexitflag = %d\n', exitflag);
fprintf('output.max_lam = %g\n', output.max_lam);
fprintf('\nx = \n');
fprintf('%4g\n', x);
fprintf('\nf = \n');
fprintf('%13g\n', f);
fprintf('\njac =\n');
fprintf('%4g%4g%4g\n', jac');
```

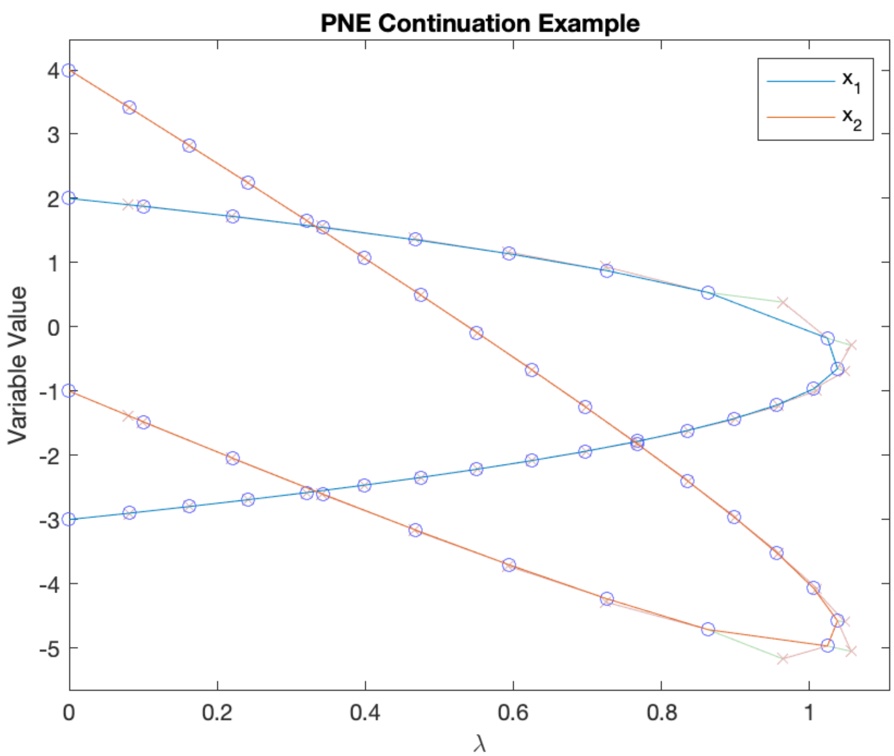

Figure 4-1: Continuation Curve for PNE Example

```
>> pne_ex1

MP-Opt-Model Version 5.0, 12-Jul-2025 -- Predictor/Corrector Continuation Method
step   0  :                         lambda =  0.000,  6 corrector steps
step   1  : PAL stepsize = 0.6      lambda =  0.081   2 corrector steps
step   2  : PAL stepsize = 0.6      lambda =  0.162   2 corrector steps
step   3  : PAL stepsize = 0.6      lambda =  0.241   2 corrector steps
step   4  : PAL stepsize = 0.6      lambda =  0.320   2 corrector steps
step   5  : PAL stepsize = 0.6      lambda =  0.398   2 corrector steps
step   6  : PAL stepsize = 0.6      lambda =  0.475   2 corrector steps
step   7  : PAL stepsize = 0.6      lambda =  0.551   2 corrector steps
step   8  : PAL stepsize = 0.6      lambda =  0.625   2 corrector steps
step   9  : PAL stepsize = 0.6      lambda =  0.697   2 corrector steps
step  10  : PAL stepsize = 0.6      lambda =  0.767   2 corrector steps
step  11  : PAL stepsize = 0.6      lambda =  0.835   2 corrector steps
step  12  : PAL stepsize = 0.6      lambda =  0.898   2 corrector steps
step  13  : PAL stepsize = 0.6      lambda =  0.956   2 corrector steps
step  14  : PAL stepsize = 0.6      lambda =  1.005   3 corrector steps
step  15  : PAL stepsize = 0.6      lambda =  1.038   3 corrector steps
step  16  : PAL stepsize = 0.6      lambda =  1.024   3 corrector steps
step  17  : PAL stepsize = 0.6      lambda =  0.863   3 corrector steps
step  18  : PAL stepsize = 0.6      lambda =  0.726   3 corrector steps
step  19  : PAL stepsize = 0.6      lambda =  0.595   3 corrector steps
step  20  : PAL stepsize = 0.6      lambda =  0.468   2 corrector steps
step  21  : PAL stepsize = 0.6      lambda =  0.343   2 corrector steps
step  22  : PAL stepsize = 0.6      lambda =  0.221   2 corrector steps
step  23  : PAL stepsize = 0.6      lambda =  0.100   2 corrector steps
step  24a : PAL stepsize = 0.6      lambda = -0.019   2 corrector steps ^ ROLLBACK
step  24  : NAT stepsize = 0.1      lambda =  0.000   3 corrector steps
CONTINUATION TERMINATION: Traced full continuation curve in 24 continuation steps

exitflag = 1
output.max_lam = 1.03783

x =
   2
  -1
   0

f =
            0
  -6.4837e-13

jac =
   1   1   6
  -4   1   0
```

5 Mathematical Model Class – mp.opt_model

The mp.opt_model class provides facilities for constructing a mathematical programming or optimization problem by adding and managing the indexing of sets of variables, constraints and costs. The model can then be solved by simply calling the solve method which automatically selects and calls the appropriate master solver function, i.e. qps_master, miqps_master, qcqps_master, nlps_master, nleqs_master or mplinsolve, depending on the type of problem.

In this manual, and in the code, mm is the name of the variable used by convention for mathematical model objects of the class mp.opt_model, which is typically created by calling the constructor mp.opt_model with no arguments.[23].

```
mm = mp.opt_model;
```

Variables, constraints and costs can then be added to the model using named sets. For variables and constraints, each set represents a column vector, and the sets are stacked in the order they are added to construct the full variable vector or full constraint vector. For costs, each set represents a component of a scalar cost, and the components are summed together to construct the full objective function value.

Important Note

MP-Opt-Model 5.0 introduced a major refactorization of the mathematical model code, with the legacy mathematical model class opt_model being replaced by the new mp.opt_model. The legacy class, described in Appendix C, provides backward compatibility with previous versions.

In both cases, the mathematical model class manages a number *set types*, such as variables and several types of constraints and costs. Each set type corresponds to a particular property in the model object, and is an instance of a subclass of mp.set_manager, as summarized in Table 5-1.

[23]The name om is used for legacy mathematical model objects of the class opt_model.

Table 5-1: Set Types

property name	mp.set_manager subclass	used to manage
var	mp.sm_variable	variables
lin	mp.sm_lin_constraint	linear constraints
qcn[†]	mp.sm_quad_constraint	quadratic constraints
nle	mp.sm_nln_constraint	nonlinear equality constraints
nli	mp.sm_nln_constraint	nonlinear inequality constraints
qdc	mp.sm_quad_cost[‡]	quadratic costs
nlc	mp.sm_nln_cost	general nonlinear costs

† While this feature is not officially supported, the quadratic cost property has also been added to the legacy opt_model class, and works just as in mp.opt_model.
‡ In the legacy opt_model class, the qdc property uses the mp.sm_quad_cost_legacy class for backward compatibility.

5.1 Adding Variables

```
mm.var.add(name, N);
mm.var.add(name, N, v0);
mm.var.add(name, N, v0, vl);
mm.var.add(name, N, v0, vl, vu);
mm.var.add(name, N, v0, vl, vu, vt);
mm.var.add(name, idx_list, N ...);
```

A named set of variables is added to the model using the `add` method of the `var` property, where `name` is a string containing the name of the set[24], `N` is the number n of variables in the set, `v0` is the initial value of the variables, `vl` and `vu` are the upper and lower bounds on the variables, and `vt` is the variable type. The accepted values for `vt` are:

- `'C'` – continuous
- `'I'` – integer
- `'B'` – binary, i.e. 0 or 1

The inputs `v0`, `vl` and `vu` are $n \times 1$ column vectors, `vt` is a scalar or a $1 \times n$ row vector. The defaults for the last four arguments, which are all optional, are for all to be continuous, unbounded and initialized to zero. That is, `v0`, `vl`, `vu`, and `vt` default to 0, $-\infty$, $+\infty$, and `'C'`, respectively.

For example, suppose our problem has variables u, v and w, which are vectors of length n_u, n_v, and n_w, respectively, where u is unbounded, v is non-negative and the lower and upper bounds on w are given in the vectors `wlb` and `wub`. Let us further

[24] A set name must be a valid field name for a struct.

suppose that the initial value of w is provided in `w0` and the first 3 elements of w are binary variables. And we will assume that the values of n_u, n_v, and n_w are available in the variables `nu`, `nv` and `nw`, respectively.

We can then add these variable sets to the model with the names **u**, **v**, and **w**, as follows:

```
wtype = repmat('C', 1, nw);  wt(1:3) = 'B';
mm.var.add('u', nu);
mm.var.add('v', nv, [], 0);
mm.var.add('w', nw, w0, wlb, wub, wtype);
```

In this case, then, the full variable vector is the $(n_u + n_v + n_w) \times 1$ vector

$$x = \left[\begin{array}{c} u \\ v \\ w \end{array} \right]. \tag{5.1}$$

See Section 5.7 for details on indexed named sets and the `idx_list` argument.

5.1.1 Variable Subsets

A key feature of MP-Opt-Model is that each set of constraints or costs can be defined in terms of the relevant variables only, as opposed to the entire vector x. This is done by specifying a variable subset, a cell array of the variable names of interest, in the `varsets` argument. Besides simplifying the constraint and cost definitions, another benefit of this approach is that it allows a model to be modified with new variables after some constraints and costs have already been added.

In the sections to follow, we will use the following two variable subsets for illustration purposes:

- {`'v'`} corresponding to $x_1 \equiv v$, and
- {`'u', 'w'`} corresponding to $x_2 \equiv \left[\begin{array}{c} u \\ w \end{array} \right]$.

5.2 Adding Constraints

A named set of constraints can be added to the model as soon as the variables on which it depends have been added. MP-Opt-Model currently supports four types of constraints, doubly-bounded linear constraints, doubly-bounded quadratic constraints, general nonlinear equality constraints, and general nonlinear inequality constraints.

5.2.1 Linear Constraints

```
mm.lin.add(mm.var, name, A, l, u);
mm.lin.add(mm.var, name, A, l, u, varsets);
mm.lin.add(mm.var, name, idx_list, A ...);
```

In MP-Opt-Model, linear constraints take the form

$$l \leq Ax \leq u, \tag{5.2}$$

where x here refers to either the full variable vector *(default)*, or the vector obtained by stacking the subset of variables specified in `varsets`. Here `A` contains the $n_A \times n_x$ matrix A and `l` and `u` are the $n_A \times 1$ vectors l and u.[25]

For example, suppose our problem has the following three sets of linear constraints,

$$l_1 \leq A_1 x_1 \leq u_1 \tag{5.3}$$
$$l_2 \leq A_2 x_2 \tag{5.4}$$
$$A_3 x \leq u_3, \tag{5.5}$$

where x_1 and x_2 are as defined in Section 5.1.1 and x is the full variable vector from (5.1). Notice that the number of columns in A_1 and A_2 correspond to n_v and $n_u + n_w$, respectively, whereas A_3 has the full set of columns corresponding to x.

These three linear constraint sets can be added to the model with the names **lincon1**, **lincon2**, and **lincon3**, using the `add` method of the `lin` property as follows:

```
mm.lin.add(mm.var, 'lincon1', A1, l1, u1, {'v'});
mm.lin.add(mm.var, 'lincon2', A2, l2, [], {'u', 'w'});
mm.lin.add(mm.var, 'lincon3', A3, [], u3);
```

See Section 5.7 for details on indexed named sets and the `idx_list` argument.

5.2.2 Quadratic Constraints

```
mm.qdc.add(mm.var, name, Q, B, l, u);
mm.qdc.add(mm.var, name, Q, B, l, u, varsets);
mm.qdc.add(mm.var, name, idx_list, Q, ...);
```

[25] The `A` matrix can be sparse.

In MP-Opt-Model, quadratic constraints take the form

$$l_{\mathrm{q}_i} \leq \frac{1}{2} x^\mathsf{T} Q_i x + b_i x \leq u_{\mathrm{q}_i}, \quad \forall i = 1, \ldots, n_q \tag{5.6}$$

where x here refers to either the full variable vector *(default)*, or the vector obtained by stacking the subset of variables specified in `varsets`. Here `Q` is an $n_q \times 1$ cell array of the $n_x \times n_x$ matrices Q_i, `B` contains the $n_q \times n_x$ matrix B, of which b_i is row i, and `l` and `u` are the $n_q \times 1$ vectors l_q and u_q.[26]

For example, suppose our problem has the following three sets of quadratic constraints,

$$l_1 \leq \left[\begin{array}{c} {x_1}^\mathsf{T} Q_{11} x_1 + b_{11} x_1 \\ {x_1}^\mathsf{T} Q_{12} x_1 + b_{12} x_1 \end{array} \right] \leq u_1 \tag{5.7}$$

$$l_2 \leq {x_2}^\mathsf{T} Q_2 x_2 + b_2 x_2 \tag{5.8}$$

$$x^\mathsf{T} Q_3 x + b_3 x \leq u_3, \tag{5.9}$$

where x_1 and x_2 are as defined in Section 5.1.1 and x is the full variable vector from (5.1). Notice that the number of columns in Q_{11}, Q_{12}, b_{11}, and b_{12} correspond to n_v and for Q_2 and b_2 to $n_u + n_w$. On the other hand, Q_3 and b_3 have the full set of columns corresponding to x.

These three quadratic constraint sets can be added to the model with the names **quadcon1**, **quadcon2**, and **quadcon3**, using the `mm.qcn.add` method as follows:

```
Q1 = {Q11; Q12}; B1 = [b11; b12];
mm.qcn.add(mm.var, 'quadcon1', Q1, B1, l1, u1, {'v'});
mm.qcn.add(mm.var, 'quadcon2', {Q2}, b2, l2, [], {'u', 'w'});
mm.qcn.add(mm.var, 'quadcon3', {Q3}, b3, [], u3);
```

See Section 5.7 for details on indexed named sets and the `idx_list` argument.

5.2.3 General Nonlinear Constraints

Equality Constraints

```
mm.nle.add(mm.var, name, N, fcn, hess);
mm.nle.add(mm.var, name, N, fcn, hess, varsets);
mm.nle.add(mm.var, name, idx_list, N ...);
```

Inequality Constraints

[26]The matrices in `Q` and `B` can be sparse.

```
mm.nli.add(mm.var, name, N, fcn, hess);
mm.nli.add(mm.var, name, N, fcn, hess, varsets);
mm.nli.add(mm.var, name, idx_list, N ...);
```

MP-Opt-Model allows the user to implement general nonlinear constraints of the form

$$g(x) = 0, \text{ or} \tag{5.10}$$
$$g(x) \leq 0 \tag{5.11}$$

by providing the handle `fcn` of a function that evaluates the constraint and its Jacobian and another handle `hess` of a function that evaluates the Hessian. The number of constraints in the set is given by `N`, and the `nle` property is used for an equality constraint or the `nli` property for an inequality.

The calling syntax for `fcn` is:

```
g = fcn(x);
[g, dg] = fcn(x);
```

Here `g` is the $n_g \times 1$ vector $g(x)$ and `dg` is the $n_g \times n_x$ Jacobian matrix $J(x)$, where $J_{ij} = \frac{\partial g_i}{\partial x_j}$.

Rather than computing the full three-dimensional Hessian, the `hess` function actually evaluates the Jacobian of the vector $J^{\mathsf{T}}(x)\lambda$ for a specified value of the vector λ. The calling syntax for `hess` is:

```
d2g = hess(x, lambda);
```

For both functions, the first input argument `x` takes one of two forms. If the constraint set is added with `varsets` empty or missing, then `x` will be the full variable vector. Otherwise it will be a cell array of vectors corresponding to the variable sets specified in `varsets`.

There is also the option for `name` to be a cell array of constraint set names, in which case `N` is a vector, specifying the number of constraints in each corresponding set. In this case, `fcn` and `hess` are each still a single function handle, but the values computed by each correspond to the entire stacked collection of constraint sets together, as if they were a single set.

For example, suppose our problem has the following three sets of nonlinear constraints,

$$g_1(x_1) \leq 0 \tag{5.12}$$

$$g_2(x_2) = 0 \tag{5.13}$$

$$g_3(x) \leq 0, \tag{5.14}$$

where x_1 and x_2 are as defined in Section 5.1.1 and x is the full variable vector from (5.1). Let `my_cons_fcn1`, `my_cons_fcn2`, and `my_cons_fcn3` be functions that evaluate $g_1(x_1)$, $g_2(x_2)$, and $g_3(x)$ and their gradients, respectively. Similarly, let `my_cons_hess1`, `my_cons_hess2`, and `my_cons_hess3` be Hessian evaluation functions for the same. The variables `ng1`, `ng2`, and `ng3` contain the number of constraints in the respective constraint sets.

These three nonlinear constraint sets can be added to the model with the names **nlncon1**, **nlncon2**, and **nlncon3**, using the `add` method of the `nle` and `nli` properties as follows:

```
fcn1 = @(x)my_cons_fcn1(x, <other_args>);
fcn2 = @(x)my_cons_fcn2(x, <other_args>);
fcn3 = @(x)my_cons_fcn3(x, <other_args>);
hess1 = @(x, lambda)my_cons_hess1(x, lambda, <other_args>);
hess2 = @(x, lambda)my_cons_hess2(x, lambda, <other_args>);
hess3 = @(x, lambda)my_cons_hess3(x, lambda, <other_args>);
mm.nli.add(mm.var, 'nlncon1', ng1, 0, fcn1, hess1 {'v'});
mm.nle.add(mm.var, 'nlncon2', ng2, 1, fcn2, hess2, {'u', 'w'});
mm.nli.add(mm.var, 'nlncon3', ng3, 0, fcn3, hess3);
```

In this case, the `x` variable passed to the `my_cons_fcn` and `my_cons_hess` functions will be as follows:

- `my_cons_fcn1`, `my_cons_hess1` $\longrightarrow$ `x` $= \{v\}$
- `my_cons_fcn2`, `my_cons_hess2` $\longrightarrow$ `x` $= \{u, w\}$
- `my_cons_fcn3`, `my_cons_hess3` $\longrightarrow$ `x` $= [u; v; w]$

See Section 5.7 for details on indexed named sets and the `idx_list` argument.

5.3 Adding Costs

The objective of an MP-Opt-Model optimization problem is to *minimize* the sum of all costs added to the model. As with constraints, a named set of costs can be added to the model as soon as the variables on which it depends have been added. MP-Opt-Model currently supports two types of costs, quadratic costs and general nonlinear costs.

5.3.1 Quadratic Costs

```
mm.qdc.add(mm.var, name, H, c);
mm.qdc.add(mm.var, name, H, c, k);
mm.qdc.add(mm.var, name, H, c, k, varsets);
mm.qdc.add(mm.var, name, idx_list, H ...);
```

A quadratic cost set takes the form:

$$f(x) = \frac{1}{2} x^\mathsf{T} H x + c^\mathsf{T} x + k \tag{5.15}$$

where x here refers to either the full variable vector *(default)*, or the vector obtained by stacking the subset of variables specified in `varsets`. Here `H` contains the $n_x \times n_x$ matrix H, `c` the $n_x \times 1$ vector c, and `k` the scalar k.[27]

Alternatively, if `H` is an $n_x \times 1$ vector, then $f(x)$ is also an $n_x \times 1$ vector, `k` can be $n_x \times 1$ or scalar, and the i-th element of $f(x)$ is given by

$$f_i(x) = \frac{1}{2} H_i x_i^2 + c_i x_i + k_i. \tag{5.16}$$

where $k_i =$ `k` for all i if `k` is scalar. If `H` is empty, then form (5.15) is implied by a scalar `k` and (5.16) by a vector `k`.

For example, suppose our problem has the following three sets of quadratic costs,

$$q_1(x_1) = \frac{1}{2} {x_1}^\mathsf{T} H_1 x_1 + {c_1}^\mathsf{T} x_1 + k_1 \tag{5.17}$$

$$q_2(x_2) = \frac{1}{2} {x_2}^\mathsf{T} H_2 x_2 + {c_2}^\mathsf{T} x_2 + k_2 \tag{5.18}$$

$$q_3(x) = \frac{1}{2} x^\mathsf{T} H_3 x + {c_3}^\mathsf{T} x + k_3, \tag{5.19}$$

where x_1 and x_2 are as defined in Section 5.1.1 and x is the full variable vector from (5.1). Notice that the dimensions of H_1 and H_2 (and c_1 and c_2) correspond to n_v and $n_u + n_w$, respectively, whereas H_3 (and c_3) correspond to the full x.

These three quadratic cost sets can be added to the model with the names **qcost1**, **qcost2**, and **qcost3**, using the `add_quad_cost` method as follows:

```
mm.qdc.add(mm.var, 'qcost1', H1, c1, k1, {'v'});
mm.qdc.add(mm.var, 'qcost2', H2, c2, k2, {'u', 'w'});
mm.qdc.add(mm.var, 'qcost3', H3, c3, k3);
```

See Section 5.7 for details on indexed named sets and the `idx_list` argument.

[27] The `H` matrix can be sparse.

5.3.2 General Nonlinear Costs

```
mm.nlc.add(mm.var, name, N, fcn);
mm.nlc.add(mm.var, name, N, fcn, varsets);
mm.nlc.add(mm.var, name, idx_list, N ...);
```

MP-Opt-Model allows the user to implement a general nonlinear cost by providing the handle `fcn` of a function that evaluates the cost $f(x)$, its gradient and Hessian H, as described below. The `N` parameter specifies the dimension for vector valued cost functions, which are not yet implemented. Currently `N` must equal 1 or it will throw an error.

For a cost function $f(x)$, `fcn` should point to a function with the following interface:

```
f = fcn(x)
[f, df] = fcn(x)
[f, df, d2f] = fcn(x)
```

where `f` is a scalar with the value of the function $f(x)$, `df` is the $n_x \times 1$ gradient of f, and `d2f` is the $n_x \times n_x$ Hessian H, where n_x is the number of elements in x.

The first input argument `x` takes one of two forms. If the constraint set is added with `varsets` empty or missing, then `x` will be the full variable vector. Otherwise it will be a cell array of vectors corresponding to the variable sets specified in `varsets`.

For example, suppose our problem has three sets of nonlinear costs, $f_1(x_1)$, $f_2(x_2)$, $f_3(x)$, where x_1 and x_2 are as defined in Section 5.1.1 and x is the full variable vector from (5.1). Let `my_cost_fcn1`, `my_cost_fcn2`, and `my_cost_fcn3` functions that evaluate $f_1(x)$, $f_2(x)$, and $f_3(x)$ and their gradients and Hessians, respectively.

These three nonlinear cost sets can be added to the model with the names **nlncost1**, **nlncost2**, and **nlncost3**, using the `add` method of the `nlc` property as follows:

```
fcn1 = @(x)my_cost_fcn1(x, <other_args>);
fcn2 = @(x)my_cost_fcn2(x, <other_args>);
fcn3 = @(x)my_cost_fcn3(x, <other_args>);
mm.nlc.add(mm.var, 'nlncost1', 1, fcn1 {'v'});
mm.nlc.add(mm.var, 'nlncost2', 1, fcn2, {'u', 'w'});
mm.nlc.add(mm.var, 'nlncost3', 1, fcn3);
```

In this case, the `x` variable passed to the `my_cost_fcn` functions will be as follows:

- `my_cost_fcn1` $\longrightarrow$ `x` $= \{v\}$
- `my_cost_fcn2` $\longrightarrow$ `x` $= \{u, w\}$
- `my_cost_fcn3` $\longrightarrow$ `x` $= [u; v; w]$

See Section 5.7 for details on indexed named sets and the `idx_list` argument.

5.4 Solving the Model

```
mm.solve()
[x, f, exitflag, output, jac] = mm.solve()
[x, f, exitflag, output, lambda] = mm.solve(opt)
[...] = mm.solve(opt)
```

After all variables, constraints and costs have been added to the model, the mathematical programming or optimization problem can be solved simply by calling the `solve` method. This method automatically selects and calls, depending on the problem type, `mplinsolve` or one of the master solver interface functions from Section 4, namely `qps_master`, `miqps_master`, `qcqps_master`, `nlps_master`, `nleqs_master`, or `pnes_master`. Note that one of the equation solvers is chosen if the model has no costs and no inequality constraints. In this case, if the number of variables is equal to the number of equality constraints, `mplinsolve` or `nleqs_master` is selected. If the number of variables is one more than the number of constraints `pnes_master` is chosen.

The results are stored in the `soln` field (see Section 5.5.5) of the MP-Opt-Model object and can be returned in the optional output arguments. The input options struct `opt`, summarized in Tables 5-2 and 5-3, is optional, as are all of its fields. For details on the return values see the descriptions of the individual solver functions in Sections 4.1, 4.2, 4.3, 4.4, 4.5, and 4.6. For linear equations, the `solver` and `opt` arguments for `mplinsolve`, described in Section 4.1 of the MIPS User's Manual, can be provided in the respective fields of `opt.leq_opt`.

Table 5-2: Options for `solve`

name	default	description
`alg`	`'DEFAULT'`	determines which solver to use, see Table 5-3
`verbose`	1	amount of progress info to be printed 0 – print no progress info 1–5 – print increasing level of progress info
`parse_soln`	0	flag that specifies whether or not to call the `parse_soln` method and place the return values in the `soln` property of the field type objects
`relax_integer`	0	relax integer constraints, if true
`x0`	*empty*	optional initial value of x, overrides value stored in model, *(ignored by some solvers)*
Additional Options for Specific Problem Types		
LP/QP		see Table 4-3
MILP/MIQP		see Table 4-5
QCQP		see Table 4-8
NLP		see Table 4-11
LEQ		see Section 4.1 of the MIPS User's Manual
`leq_opt.solver`	`''`	see `help mplinsolve`, input argument `solver`
`leq_opt.opt`	*empty*	see `help mplinsolve`, input argument `opt`
NLEQ		see Table 4-14
PNE		see Table 4-20

Table 5-3: Values for `alg` Option to `solve`

`alg` value	problem type(s)	description
`'DEFAULT'`	*all*	automatic, depends on problem type, uses first available of:
	LP	Gurobi, CPLEX, MOSEK, `linprog`,¶ HiGHS, GLPK, BPMPD, MIPS
	QP	Gurobi, CPLEX, MOSEK, `quadprog`,¶ HiGHS, BPMPD, MIPS
	MILP	Gurobi, CPLEX, MOSEK, `intlinprog`, HiGHS, GLPK
	MIQP	Gurobi, CPLEX, MOSEK
	QCQP	IPOPT, Artelys Knitro, `fmincon`,¶ MIPS
	NLP	MIPS
	MINLP	Artelys Knitro (not yet implemented)
	LEQ	built-in backslash operator
	NLEQ	Newton's method
	PNE	predictor/corrector continuation method
`'BPMPD'`	LP, QP	BPMPD*
`'CLP'`	LP, QP	CLP*
`'CPLEX'`	LP, QP, MILP, MIQP	CPLEX*
`'FD'`	NLEQ	fast-decoupled Newton's method†
`'FMINCON'`	QCQP, NLP	MATLAB Opt Toolbox, `fmincon`*
`'FSOLVE'`	NLEQ	MATLAB Opt Toolbox, `fsolve`§
`'GLPK'`	LP, MILP	GLPK* *(LP only)*
`'GS'`	NLEQ	Gauss-Seidel method‡
`'GUROBI'`	LP, QP, MILP, MIQP, QCQP	Gurobi*
`'HIGHS'`	LP, QP, MILP	HiGHS*
`'IPOPT'`	LP, QP, QCQP, NLP	IPOPT*
`'KNITRO'`	QCQP, NLP, MINLP	Artelys Knitro*
`'MIPS'`	LP, QP, QCQP, NLP	MIPS, **M**ATPOWER **I**nterior **P**oint **S**olver
`'MOSEK'`	LP, QP, MILP, MIQP	MOSEK*
`'NEWTON'`	NLEQ	Newton's method
`'OSQP'`	LP, QP	OSQP*
`'OT'`	LP, QP, MILP	MATLAB Opt Toolbox, `quadprog`, `linprog`, `intlinprog`

* Requires the installation of an optional package. See Appendix B for details on the corresponding package.
† Fast-decoupled Newton requires setting `fd_opt.jac_approx_fcn` to a function handle that returns Jacobian approximations. See `help nleqs_fd_newton` for more details.
‡ Gauss-Seidel requires setting `gs_opt.x_update_fcn` to a function handle that updates x. See `help nleqs_gauss_seidel` for more details.
§ The `fsolve` function is included with GNU Octave, but on MATLAB it is part of the MATLAB Optimization Toolbox. See Appendix B for more information on the MATLAB Optimization Toolbox.
¶ If running on MATLAB.

5.5 Accessing the Model

5.5.1 Indexing

For each set type, MP-Opt-Model maintains indexing information[28] for each named set that is added, including the number of elements and the starting and ending indices. This information is stored in the `idx` property of the set type object, consisting of a struct with fields `N`, `i1`, and `iN`, for storing number of elements, starting index and ending index, respectively. Each of these fields is also a struct with field names corresponding to the named sets.

For example, if we have added the **u**, **v**, and **w** variables as in Section 5.1, then the contents of `mm.var.idx` will be as shown in Table 5-4.

Table 5-4: Example Indexing Data

field	value	description
`mm.var.idx.N.u`	n_u	number of u variables
`mm.var.idx.N.v`	n_v	number of v variables
`mm.var.idx.N.w`	n_w	number of w variables
`mm.var.idx.i1.u`	1	starting index of u in full x
`mm.var.idx.i1.v`	$n_u + 1$	starting index of v in full x
`mm.var.idx.i1.w`	$n_u + n_v + 1$	starting index of w in full x
`mm.var.idx.iN.u`	n_u	ending index of u in full x
`mm.var.idx.iN.v`	$n_u + n_v$	ending index of v in full x
`mm.var.idx.iN.w`	$n_u + n_v + n_w$	ending index of w in full x

`get_idx`

```
[idx1, idx2, ...] = mm.get_idx(set_type1, set_type2, ...);
vv = mm.get_idx('var');
[ll, nne, nni] = mm.get_idx('lin', 'nle', 'nli');

vv = mm.get_idx()
[vv, ll] = mm.get_idx()
[vv, ll, nne] = mm.get_idx()
[vv, ll, nne, nni] = mm.get_idx()
[vv, ll, nne, nni, qq] = mm.get_idx()
[vv, ll, nne, nni, qq, nnc] = mm.get_idx()
[vv, ll, nne, nni, qq, nnc, qqcn] = mm.get_idx()
```

[28]This indexing information is managed by the `mp.set_manager` base class from which all set type objects inherit.

The `idx` property of indexing information for each set type is available via the `get_idx` method of `mp.opt_model`. When called with one or more set type property names as inputs, it returns the corresponding indexing structs. The list of valid set type strings is shown in Table 5-5. When called without input arguments, the indexing structs are simply returned in the order listed in the table.

Table 5-5: Valid Set Types

set type string	var name*	description
`'var'`	`vv`	variables
`'lin'`	`ll`	linear constraints
`'qcn'`	`qqcn`	quadratic constraints
`'nle'`	`nne`	nonlinear equality constraints
`'nli'`	`nni`	nonlinear inequality constraints
`'qdc'`	`qq`	quadratic costs
`'nlc'`	`nnc`	general nonlinear costs

* The name of the variable used by convention for this indexing struct.

For the example model built in Sections 5.1–5.3, where `x` and `lambda` are return values from the `solve` method, we can, for example, access the solved value of v and the shadow prices on the **nlncon3** constraints with the following code.

```
[vv, nne] = mm.get_idx('var', 'nle');
v = x(vv.i1.v:vv.iN.v);
lam_nln3 = lambda.ineqnonlin(nni.i1.nlncon3:nni.iN.nlncon3);
```

5.5.2 Variables

`var.params`

```
[v0, vl, vu] = mm.var.params()
[v0, vl, vu] = mm.var.params(name)
[v0, vl, vu] = mm.var.params(name, idx_list)
[v0, vl, vu, vt] = mm.var.params(...)
```

The `params` method of the `var` property returns the initial value `v0`, lower bound `vl` and upper bound `vu` for the full variable vector x, or for a specific named variable set. Optionally also returns a corresponding char vector `vt` of variable types, where `'C'`, `'I'` and `'B'` represent continuous, integer, and binary variables, respectively.

Examples:

```
[x0, xmin, xmax] = mm.var.params();
[w0, wlb, wub, wtype] = mm.var.params('w');
```

See Section 5.7 for details on indexed named sets and the idx_list argument.

5.5.3 Constraints

`lin.params`

```
[A, l, u] = mm.lin.params(mm.var)
[A, l, u] = mm.lin.params(mm.var, name)
[A, l, u] = mm.lin.params(mm.var, name, idx_list)
[A, l, u, vs] = mm.lin.params(mm.var, ...)
[A, l, u, vs, i1, in] = mm.lin.params(mm.var, ...)
```

With only the `mm.var` input paramter, the `params` method of the `lin` property assembles and returns the parameters for the aggregate linear constraints from all linear constraint sets added using `mm.lin.add`. The values of these parameters are cached for subsequent calls. The parameters are A, l, and u, where the linear constraint is of the form

$$l \leq Ax \leq u. \tag{5.20}$$

If a `name` is provided then it simply returns the parameters for the corresponding named set. An optional 4th output argument `vs` indicates the variable sets used by this constraint set. The size of `A` will be consistent with `vs`. Optional 5th and 6th output arguments `i1` and `iN` indicate the starting and ending row indices of the corresponding constraint set in the full aggregate constraint matrix.

Examples:

```
[A, l, u] = mm.lin.params(mm.var);
[A, l, u, vs, i1, iN] = mm.lin.params(mm.var, 'lincon2');
```

See Section 5.7 for details on indexed named sets and the idx_list argument.

`qcn.params`

See the reference documentation for the `params` method of `mp.sm_quad_constraint`.

`nle.params, nli.params`

Equality Constraints

```
N = mm.nle.params(mm.var, name)
N = mm.nle.params(mm.var, name, idx_list)
[N, fcn] = mm.nle.params(...)
[N, fcn, hess] = mm.nle.params(...)
[N, fcn, hess, vs] = mm.nle.params(...)
[N, fcn, hess, vs, include] = mm.nle.params(...)
```

Inequality Constraints

```
N = mm.nli.params(mm.var, name)
N = mm.nli.params(mm.var, name, idx_list)
[N, fcn] = mm.nli.params(...)
[N, fcn, hess] = mm.nli.params(...)
[N, fcn, hess, vs] = mm.nli.params(...)
[N, fcn, hess, vs, include] = mm.nli.params(...)
```

Returns the parameters `N`, and optionally `fcn`, and `hess` provided when the corresponding named nonlinear constraint set was added to the model. Likewise for indexed named sets specified by `name` and `idx_list`.

An optional 4th output argument `vs` indicates the variable sets used by this constraint set.

And, for constraint sets whose functions compute the constraints for another set, an optional 5th output argument returns a struct with a cell array of set names in the `'name'` field and an array of corresponding dimensions in the `'N'` field.

`lin.eval`

```
Ax_u = mm.lin.eval(mm.var, x)
Ax_u = mm.lin.eval(mm.var, x, name)
Ax_u = mm.lin.eval(mm.var, x, name, idx_list)
[Ax_u, l_Ax] = mm.lin.eval(...)
[Ax_u, l_Ax, A] = mm.lin.eval(...)
```

Builds and evaluates the linear constraints $Ax - u$ and, optionally $l - Ax$ for the full set of constraints or an individual named subset for a given value of the variable vector x, based on constraints added by `mm.lin.add`.

Examples:

```
[Ax_u, l_Ax, A] = mm.lin.eval(mm.var, x);
```

`qcn.eval`

See the reference documentation for the `eval` method of `mp.sm_quad_constraint`.

`nle.eval`, `nli.eval`

Equality Constraints

```
g = mm.nle.eval(mm.var, x)
g = mm.nle.eval(mm.var, x, name)
g = mm.nle.eval(mm.var, x, name, idx_list)
[g, dg] = mm.nle.eval(...)
```

Inequality Constraints

```
g = mm.nli.eval(mm.var, x)
g = mm.nli.eval(mm.var, x, name)
g = mm.nli.eval(mm.var, x, name, idx_list)
[g, dg] = mm.nli.eval(...)
```

Builds the nonlinear equality constraints $g(x)$ or inequality constraints $h(x)$ and optionally their gradients for the full set of constraints or an individual named subset for a given value of the variable vector x, based on constraints added by `mm.nle.add` or `mm.nli.add`, where $g(x) = 0$ and $h(x) \leq 0$.

Examples:

```
[g, dg] = mm.nle.eval(mm.var, x);
[h, dh] = mm.nli.eval(mm.var, x);
```

`nle.eval_hess`, `nli.eval_hess`

Equality Constraints

```
d2G = mm.nle.eval_hess(mm.var, x, lam)
```

Inequality Constraints

```
d2H = mm.nli.eval_hess(mm.var, x, lam)
```

Builds the Hessian of the full set of nonlinear equality constraints $g(x)$ or inequality constraints $h(x)$ for given values of the variable vector x and dual variables `lam`, based on constraints added by `mm.nle.add` or `mm.nli.add`, where $g(x) = 0$ and $h(x) \leq 0$.

Examples:

```
d2G = mm.nle.eval_hess(x, lam)
d2H = mm.nli.eval_hess(x, lam)
```

5.5.4 Costs

`qdc.params`

```
[H, c] = mm.qdc.params(mm.var)
[H, c] = mm.qdc.params(mm.var, name)
[H, c] = mm.qdc.params(mm.var, name, idx_list)
[H, c, k] = mm.qdc.params(...)
[H, c, k, vs] = mm.qdc.params(...)
```

With only the `mm.var` input paramter, the `mm.qdc.params` method assembles and returns the parameters for the aggregate quadratic cost from all quadratic cost sets added using `mm.qdc.add`. The values of these parameters are cached for subsequent calls. The parameters are H, c, and optionally k, where the quadratic cost is of the form

$$f(x) = \frac{1}{2} x^\mathsf{T} H x + c^\mathsf{T} x + k. \tag{5.21}$$

If a `name` is provided then it simply returns the parameters for the corresponding named set. In this case, `H` and `k` may be vectors, corresponding to a cost function $f(x)$ where the i-th element takes the form

$$f_i(x) = \frac{1}{2} H_i x_i^2 + c_i x_i + k_i, \tag{5.22}$$

depending on how the constraint set was initially specified.

An optional 4th output argument `vs` indicates the variable sets used by this cost set. The size of `H` and `c` will be consistent with `vs`.

Examples:

```
[H, c, k] = mm.qdc.params(mm.var);
[H, c, k, vs, i1, iN] = mm.qdc.params(mm.var, 'qcost2');
```

See Section 5.7 for details on indexed named sets and the `idx_list` argument.

nlc.params

```
[N, fcn] = mm.nlc.params(mm.var, name)
[N, fcn] = mm.nlc.params(mm.var, name, idx_list)
[N, fcn, vs] = mm.nlc.params(...)
```

Returns the parameters `N` and `fcn` provided when the corresponding named general nonlinear cost set was added to the model. Likewise for indexed named sets specified by `name` and `idx_list`.

An optional 3rd output argument `vs` indicates the variable sets used by this constraint set.

qdc.eval

```
f = mm.qdc.eval(mm.var, x ...)
[f, df] = mm.qdc.eval(mm.var, x ...)
[f, df, d2f] = mm.qdc.eval(mm.var, x ...)
[f, df, d2f] = mm.qdc.eval(mm.var, x, name)
[f, df, d2f] = mm.qdc.eval(mm.var, x, name, idx_list)
```

The `eval_quad_cost` method evaluates the cost function and its derivatives for an individual named set or the full set of quadratic costs for a given value of the variable vector x, based on costs added by `mm.qdc.add`.

Examples:

```
[f, df, d2f] = mm.qdc.eval(mm.var, x);
[f, df, d2f] = mm.qdc.eval(mm.var, x, 'qcost3');
```

See Section 5.7 for details on indexed named sets and the `idx_list` argument.

nlc.eval

```
f = mm.nlc.eval(mm.var, x)
[f, df] = mm.nlc.eval(mm.var, x)
[f, df, d2f] = mm.nlc.eval(mm.var, x)
[f, df, d2f] = mm.nlc.eval(mm.var, x, name)
[f, df, d2f] = mm.nlc.eval(mm.var, x, name, idx_list)
```

The `mm.nlc.eval` method evaluates the cost function and its derivatives for an individual named set or the full set of general nonlinear costs for a given value of the variable vector x, based on costs added by `mm.nlc.add`.

Examples:

```
[f, df, d2f] = mm.nlc.eval(mm.var, x);
[f, df, d2f] = mm.nlc.eval(mm.var, x, 'nlncost2');
```

See Section 5.7 for details on indexed named sets and the `idx_list` argument.

5.5.5 Model Solution

The solved results of a model, as returned by the `solve` method, are stored in the `soln` field of the MP-Opt-Model object as summarized in Table 5-6.

`is_solved`

```
TorF = mm.is_solved()
```

The `is_solved` method returns `1` if the model has been solved, `0` otherwise.

MP Set Manager `get_soln` Methods

The `get_soln` methods of the various set types can be used to extract solved results for a given named set of variables, constraints or costs. The input arguments for the `get_soln` methods are summarized in Table 5-7 and Table 5-8. The variable number of output arguments correspond to the `tags` input. If `tags` is empty or not specified, the calling context will define the number of outputs, returned in order of default tags for the given set type.

Examples:

Value of variable named `'P'` and shadow prices on its bounds.

```
[P, muPmin, muPmax] = mm.var.get_soln('P');
```

Shadow prices on upper and lower linear constraint set named `'lin_con_1'`.

```
[mu_u, mu_l] = mm.lin.get_soln(mm.var, {'mu_u', 'mu_l'}, 'lin_con_1');
```

Jacobian of the (2,3)-element of the indexed nonlinear equality constraint set named `'nle_con_b'`.

Table 5-6: Model Solution

field	description
`mm`	MP-Opt-Model object
`.soln`	model solution struct
`.x`	solution vector
`.f`	final function value,* $f(x)$
`.eflag`	exit flag
	1 – converged successfully
	≤ 0 – solver-specific failure code
`.output`	output struct with the following fields:
	`alg` – algorithm code of solver used
	`et` – solution elapsed time in seconds
	(others) – solver-specific fields
`.jac`	final value of Jacobian matrix (for LEQ/NLEQ)
`.lambda`	shadow prices on constraints
`.lower`	variable lower bound
`.upper`	variable upper bound
`.mu_l`	linear constraint lower bound
`.mu_u`	linear constraint upper bound
`.mu_lq`	quadratic constraint lower bound
`.mu_uq`	quadratic constraint upper bound
`.eqnonlin`	nonlinear equality constraints
`.ineqnonlin`	nonlinear inequality constraints
Parsed Solution†	
`.var.soln`	parsed solution for variables‡
`.lin.soln`	parsed solution for linear constraints‡
`.qcn.soln`	parsed solution for quadratic constraints‡
`.nle.soln`	parsed solution for nonlinear equality constraints‡
`.nli.soln`	parsed solution for nonlinear inequality constraints‡

* Objective function value for optimization problems, constraint function value for sets of equations.
† Only available after calling `parse_soln(true)` or calling `solve()` with the `opt.parse_soln` option set to 1.
‡ See Table 5-9 for details.

```
dg_b_2_3 = mm.nle.get_soln(mm.var, 'dg', 'nle_con_b', {2,3});
```

For more details, please see the reference documentation for the `get_soln` method of the respective subclass of `mp.set_manager`.

Table 5-7: Inputs for MP Set Manager `get_soln` Methods

name	default	description
`soln`		model solution struct `mm.soln`
`var`		variable set manager object `mm.var`
`tags`	*depends*	char array or cell array of char arrays specifying the desired output(s)[†]
`name`	*required*	char array specifying the name of the set
`idx`	*empty*	cell array specifying the indices of the set

[†] Valid values and defaults for `tags` depend on the set type and are summarized in Table 5-8.

Table 5-8: Values of `tags` Input to `get_soln` Methods

set type	valid tag values	description
`var`		default `tags` = {`'x'`, `'mu_l'`, `'mu_u'`}
	`'x'`	value of solution variable
	`'mu_l'`	shadow price on variable lower bound
	`'mu_u'`	shadow price on variable upper bound
`lin`		default `tags` = {`'f'`} for LEQ problems, {`'g'`, `'mu_l'`, `'mu_u'`} otherwise
	`'f'`†	equality constraint values, $Ax - u$
	`'g'`	1×2 cell array of upper and lower constraint values, $\{Ax - u, \, l - Ax\}$
	`'Ax_u'`	upper constraint value, $Ax - u$
	`'l_Ax'`	lower constraint value, $l - Ax$
	`'mu_l'`	shadow price on constraint lower bound
	`'mu_u'`	shadow price on constraint upper bound
`qcn`		default `tags` = {`'f'`} for NLEQ problems, {`'g'`, `'mu_l'`, `'mu_u'`} otherwise
	`'f'`†	equality constraint values, $g(x) - u$
	`'g'`	1×2 cell array of upper and lower constraint values, $\{g(x) - u, \, l - g(x)\}$
	`'Ax_u'`	upper constraint value, $g(x) - u$
	`'l_Ax'`	lower constraint value, $l - g(x)$
	`'mu_l'`	shadow price on constraint lower bound
	`'mu_u'`	shadow price on constraint upper bound
`nle`		default `tags` = {`'g'`, `'lam'`, `'dg'`}
	`'g'`	constraint value, $g(x)$
	`'lam'`	shadow price on constraint
	`'dg'`	Jacobian of constraint
`nli`		default `tags` = {`'h'`, `'mu'`, `'dh'`}
	`'h'`	constraint value, $h(x)$
	`'mu'`	shadow price on constraint
	`'dh'`	Jacobian of constraint
`nlc` or `qdc`		default `tags` = {`'f'`, `'df'`, `'d2f'`}
	`'f'`	cost function value, $f(x)$‡
	`'df'`	gradient of cost function
	`'d2f'`	Hession of cost function

† For LEQ/NLEQ problems only.
‡ For `qdc`, $f(x)$ can return be a vector.

parse_soln

```
ps = mm.parse_soln()
mm.parse_soln(stash)
```

The parse_soln method returns a struct of parsed solution vector and shadow price values for each named set of variables and constraints. The returned ps (parsed solution) struct has the format shown in Table 5-9, where each of the terminal elements is a struct with fields corresponding to the respective named sets.

Table 5-9: Output of parse_soln

fields	description
ps	
.var	variables
.val	struct of solution vectors
.mu_l	struct of lower bound shadow prices
.mu_u	struct of upper bound shadow prices
.lin	linear constraints
.mu_l	struct of lower bound shadow prices
.mu_u	struct of upper bound shadow prices
.qcn	quadratic constraints
.mu_l	struct of lower bound shadow prices
.mu_u	struct of upper bound shadow prices
.nle	nonlinear equality constraints
.lam	struct of shadow prices
.nli	nonlinear inequality constraints
.mu	struct of shadow prices

The value of each element in the returned struct can be obtained via the get_soln method as well, but parse_soln is generally more efficient if a complete set of values is needed.

If the optional stash input argument is present and true, the fields of the return struct are copied to the soln property of the corresponding set type object in mm.

has_parsed_soln

```
TorF = mm.has_parsed_soln()
```

The has_parsed_soln method returns 1 if the model has a parsed solution available in the soln property of the set type objects, 0 otherwise.

5.6 Modifying the Model

The parameters for an existing MP-Opt-Model object can be modified, rather than having to rebuild a new model from scratch.

MP Set Manager `set_params` Methods

`set_params`

The `set_params` methods of the various set types, inputs summarized in Table 5-10, can be used to modify any of the parameters associated with an existing variable, cost or constraint set.

Examples:

```
mm.var.set_params('Pg', 'v0', Pg0);
mm.lin.set_params(mm.var, 'y', {2,3}, {'l', 'u'}, {l, u});
mm.nle.set_params(mm.var, 'Pmis', 'all', {N, @fcn, @hess, vs});
```

For more details, please see the reference documentation for the `set_params` method of the respective subclass of `mp.set_manager`.

Table 5-10: Inputs for MP Set Manager `set_params` Methods

name	description
`var`	variable set manager object `mm.var`
`set_type`	one of the following, specifying the type of set, with the corresponding valid parameter names
`name`	char array specifying the name of the set
`idx`‡	cell array specifying the indices of the set
`params`	one of the following: `'all'` – indicates that `vals` is a cell array of values whose elements correspond to the input parameters of the respective `add` method *char array* – name of parameter to modify *cell array* – names of parameters to modify where the valid parameter names for the various set types are: `var` – variables: `N`, `v0`†, `vl`†, `vu`†, `vt`† `lin` – linear constraints: `A`, `l`, `u`†, `vs`† `qcn` – quadratic constraints: `Q`, `B`, `l`, `u`†, `vs`† `nle` – nonlinear equality constraints: `N`, `fcn`, `hess`, `vs`† `nli` – nonlinear inequality constraints: `N`, `fcn`, `hess`, `vs`† `nlc` – nonlinear costs: `N`, `fcn`, `vs`† `qdc` – quadratic costs: `H`, `c`†, `k`†, `vs`†
`vals`	new value or cell array of new values corresponding the parameter name(s) specified in `params`

† Optional when `params` = `'all'`.
‡ The `idx` argument is optional.

5.7 Indexed Sets

A variable, constraint or cost set is typically identified simply by a `name`, but it is also possible to use indexed names. For example, an optimal scheduling problem with a one week horizon might include a vector variable **y** for each day, indexed from 1 to 7, and another vector variable **z** for each hour of each day, indexed from (1, 1) to (7, 24).

In this case, we case use a single indexed named set for **y** and another for **z**. The dimensions are initialized via the `init_indexed_name` method of the set type before adding the variables to the model.

`init_indexed_name`

```
mm.(set_type).init_indexed_name(name, dim_list)
```

Examples:

```
mm.var.init_indexed_name('y', {7});
mm.var.init_indexed_name('z', {7, 24});
```

After initializing the dimensions, indexed named sets of variables, constraints or costs can be added by supplying the indices in the `idx_list` argument following the `name` argument in the call to the corresponding `add` method. The `idx_list` argument is simply a cell array containing the indices of interest.

Examples:

```
for d = 1:7
    mm.var.add('y', {d}, ny(d), y0{d}, yl{d}, yu{d}, yt{d});
end
for d = 1:7
    for h = 1:24
        mm.var.add('z', {d, h}, nz(d, h), z0{d, h}, zl{d, h}, zu{d, h});
    end
end
```

Other Methods

All of the MP Set Manager methods that take a `name` argument to specify a simple named set, can also take an `idx_list` argument immediately following `name` to handle the equivalent indexed named set. The `idx_list` argument is simply a cell array containing the indices of interest. This includes `get_N` and the `add`, `params`, and `eval` methods.[29]

For an indexed named set, the fields under the `N`, `i1` and `iN` fields in the `idx` property of a set type, i.e. the struct returned by `get_idx`, are now arrays of the appropriate dimension, not just scalars as in Table 5-4. For example, to find the starting index of the z variable for day 2, hour 13 in our example you would use `vv.i1.z(2, 13)`. Similarly for the values returned by `get_N` when specifying only the `name`.

Variable Subsets

A variable subset for a simple named set, usually specified by the variable `varsets` or else `vs`, is a cell array of variable set names. For indexed named sets of variables, on the other hand, it is a struct array with two fields `name` and `idx`. For each element of the struct array the `name` field contains the name of the variable set and the `idx` field contains a cell array of indices of interest.

For example, to specify a variable subset consisting of the **y** variable for day 3 and the **z** variable for day 3, hour 7, the variable subset could be defined as follows.

```
vs = struct('name', {'y', 'z'}, 'idx', {{3}, {3,7}});
```

5.8 Miscellaneous Methods

5.8.1 Public Methods

`copy`

```
mm2 = mm.copy()
```

The `copy` method can be used to make a copy of an MP-Opt-Model object.

`display`

```
mm
```

The `display` method displays the variable, constraint and cost sets that make up the model, along with their indexing data.

[29]Currently, `eval` and `eval_hess` for the `mp.sm_nln_constraint` class are only implemented for the full aggregate set of constraints and do not yet support evaluation of individual constraint sets.

display_soln

```
mm.display_soln()
```

The display_soln method displays the model solution, including values, bounds and shadow prices for variables, linear constraints, and quadratic constraints, values and shadow prices for nonlinear constraints, and individual cost components. Results are displayed for each set type. To display the solution for a given set type, use the display_soln method of the specific set type object.

from_struct

```
mm.from_struct(s)
```

Called by function mp.struct2object, after creating the object to copy the object data *from* a struct. Useful for recreating the object after loading struct data from a MAT-file in Octave.

get_set_types

```
set_types = mm.get_set_types()
```

The get_set_types method returns a cell array of the names of the properties containing the set types, that is those containing the mp.set_manager objects, as listed in Table 5-1.

get_userdata

```
data = mm.get_userdata(name)
```

MP-Opt-Model allows the user to store arbitrary data in fields of the userdata property, which is a simple struct. The get_userdata method returns the value of the field specified by name, or an empty matrix if the field does not exist in mm.userdata.

is_mixed_integer

```
TorF = mm.is_mixed_integer()
```

Returns 1 if any of the variables are binary or integer, 0 otherwise.

problem_type

```
prob_type = mm.problem_type()
prob_type = mm.problem_type(recheck)
```

Returns a string identifying the type of mathematical program represented by the current model, based on the variables, costs,and constraints that have been added to the model. Used to automatically select an appropriate solver.

Linear and nonlinear equations are models with no costs, no inequality constraints, and an equal number of continuous variables and equality constraints.

The `prob_type` string is one of the following:

- `'LEQ'` – linear equation
- `'NLEQ'` – nonlinear equation
- `'LP'` – linear program
- `'QP'` – quadratic program
- `'QCQP'` – quadratically-constrained quadratic program
- `'NLP'` – nonlinear program
- `'MILP'` – mixed-integer linear program
- `'MIQP'` – mixed-integer quadratic program
- `'MINLP'` – mixed-integer nonlinear program[30]

The output value is cached for future calls, but calling with a true value for the optional `recheck` argument will force it to recheck in case the problem type has changed due to modifying the variables, constraints or costs in the model.

to_struct

```
s = mm.to_struct()
```

Converts the object data *to* a struct that can later be converted back to an identical object using `mp.struct2object`. Useful for saving the object data to a MAT-file in Octave.

Methods for Variable Sets

The legacy methods for handling variable sets, `varsets_cell2struct`, `varsets_idx`, `varsets_len`, and `varsets_x` have been moved to `mp.sm_variable`. See the reference documentation for details.

[30] MP-Opt-Model does not yet implement solving MINLP problems.

5.9 MP Set Manager – mp.set_manager

The mp.opt_model class manages several ordered sets of entities, such as variables, various kinds of constraints, costs, etc. These sets consist of named (or named and indexed) subsets and are implemented as *set manager* objects implemented by the mp.set_manager class and subclasses.

Prior to MP-Opt-Model 5.0, much of the functionality for managing these sets was implemented in the container objects by opt_model and its parent mp_idx_manager. MP-Opt-Model 5.0 introduced a major refactorization in which most of this functionality was moved from the container class, into individual set type properties which were converted from simple structs into mostly backward-compatible set manager objects that inherit from the new mp.set_manager and mp.set_manager_opt_model classes, namely:

- mp.sm_lin_constraint - set manager class for linear constraints
- mp.sm_quad_constraint - set manager class for quadratic constraints
- mp.sm_nln_constraint - set manager class for nonlinear constraints
- mp.sm_nln_cost - set manager class for general nonlinear costs
- mp.sm_quad_cost - set manager class for quadratic costs
- mp.sm_variable - set manager class for variables

For more details on these classes, please see the online MP-Opt-Model Reference Manual[31].

The indexing functionality is handled by the base classes, with the remaining set type specific functionality for these various ordered set types being implemented in the individual subclasses. The properties and methods implemented by the base mp.set_manager class are shown in Table A-7.

By convention, the variable name used for a generic set manager object is sm, or for a specific type, sm_<type>, where <type> is the respective property name shown in Table 5-1.

[31] https://matpower.org/doc/mpom/

Table 5-11: MP Set Manager (`mp.set_manager`) Properties and Methods

name	description
Properties	
`idx`	struct with fields:
	`i1` – starting index of subset within full set*
	`iN` – ending index of subset within full set*
	`N` – number of elements in this subset*
`N`	total number of entities in the full set
`NS`	number of named or named/indexed subsets or blocks
`order`	struct array of blocks in order, with fields:
	`name` – name of the block
	`idx` – cell array of indices for the name
`data`	struct of additional set-type-specific data for each block*
Public Methods	
`mp.set_manager`	constructor for `mp.set_manager` class
`add`	add a named (and optionally indexed) subset of entities
`copy`	make a duplicate (shallow copy) of the object
`describe_idx`	provide/display name and index label for given indices, e.g. element 361 corresponds to `w(68)`, see also Section 5.9.1
`display`	display summary of indexing of subsets in object
`from_struct`	copy object data *from* a struct
`get_N`	return the number of elements in the set
`init_indexed_name`	initialize dimensions for an indexed named set
`set_type_idx_map`	map index back to named subset & index within set, e.g. element 361 corresponds to `w(68)`, see Section 5.9.1
`to_struct`	convert object data *to* a struct

* This field is a struct and the description applies to each field of the struct. The fields are the names corresponding to the subsets added.

5.9.1 MP Set Manager Methods

`add`

```
sm.add(name, N, ...)
sm.add(name, idx_list, N, ...)
```

This base class method handles the indexing part. Subclasses are expected to override it to handle any data that goes with each subset added for the given set type.

For example:

```
% Variable Set
mm.var.add(name, idx_list, N, v0, vl, vu, vt);

% Linear Constraint Set
mm.lin.add(name, idx_list, N, A, l, u, varsets);

% Nonlinear Equality Constraint Set
mm.nle.add(name, idx_list, N, fcn, hess, computed_by, varsets);

% Nonlinear Inequality Constraint Set
mm.nli.add(name, idx_list, N, fcn, hess, computed_by, varsets);

% Quadratic Cost Set
mm.qdc.add(name, idx_list, N, cp, varsets);

% General Nonlinear Cost Set
mm.nlc.add(name, idx_list, N, fcn, varsets);
```

See the online MP-Opt-Model Reference Manual for the implementing subclass for details.

`copy`

```
new_sm = sm.copy()
new_sm = sm.copy(new_class)
```

Make a shallow copy of the object by copying each of the top-level properties.

`describe_idx`

```
label = mm.describe_idx(set_type, idxs)
```

Calls `set_type_idx_map` and formats each element of the return data as character array, returning a cell array of the same dimensions as `idxs`, except in the case where `idxs` is scalar, in which case it returns a scalar.

Consider an example in which element 38 of the linear constraints corresponds to the 11th row of **lincon3** and elements 15 and 23 of the variable vector x correspond to element 7 of v and element 4 of w, respectively. The `describe_idx` method can be used to return this information as follows:

```
>> lin38 = mm.describe_idx('lin', 38)

lin38 =

    'lincon3(11)'


>> vars15_23 = mm.describe_idx('var', [15; 23])

vars15_23 =

  2x1 cell array

    {'v(7)'}
    {'w(4)'}
```

`display`

```
sm
```

Display summary of indexing of subsets in object.

This method is called automatically when omitting a semicolon on a line that retuns an object of this class.

Displays the details of the indexing for each subset or block, as well as the total number of elements and subsets.

`from_struct`

```
sm.from_struct(s)
```

Called by function `mp.struct2object`, after creating the object to copy the object data *from* a struct. Useful for recreating the object after loading struct data from a MAT-file in Octave.

`get_N`

```
N = sm.get_N()
N = sm.get_N(name)
N = sm.get_N(name, idx_list)
```

The get_N method can be used to get the number of elements in a particular named set, or the total for the set type. For example, the number n_v of elements in variable v and total number of elements in the full variable x can be obtained as follows.

```
nx = mm.var.get_N();
nv = mm.var.get_N('v');
```

See Section 5.7 for details on indexed named sets and the idx_list argument.

init_indexed_name

```
sm.init_indexed_name(name, dim_list)
```

A subset or block can be identified by a single name, such as 'foo', or by a name that is indexed by one or more indices, such as 'bar3,4'. For an indexed named set, before adding the indexed subsets themselves, the dimensions of the indexed set of names must be initialized by calling this method.

set_type_idx_map

```
s = sm.set_type_idx_map()
s = sm.set_type_idx_map(idxs)
s = sm.set_type_idx_map(idxs, group_by_name)
```

Given a particular index (or set of indices) for the full set of elements (e.g. variables or constraints) of a particular set type, the set_type_idx_map method can be used to determine which element of which particular named set the index corresponds to. If idxs is empty or not provided it defaults to [1:ns]', where ns is the full dimension of the set corresponding to the all elements for the specified set type. Results are returned in a struct s of the same dimensions as the input idxs, where each element specifies the details of the corresponding named set. The fields of s are (1) name, with the name of the corresponding set, (2) idx, a cell array of indices for the name, if the named set is indexed and, (3) i, the index of the element within the set.

If group_by_name is true, then the results are consolidated, with a single entry in s for each unique name index pair, where i field is a vector and there is an additional field named j that is a vector with the corresponding index of the set type, equal to a particular element of idxs. In this case s is 1 dimensional.

This method can be useful, for example, when a solver reports an issue with a particular variable or constraint and you want to map it back to the named sets you have added to your model. Consider an example in which element 38 of the linear constraints corresponds to the 11th row of **lincon3** and elements 15 and 23 of the variable vector x correspond to element 7 of v and element 4 of w, respectively. The `set_type_idx_map` method can be used to return this information as follows:

```
>> lin38 = mm.set_type_idx_map('lin', 38)

lin38 =

  struct with fields:

    name: 'lincon3'
     idx: []
       i: 11


>> s = mm.set_type_idx_map('var', [15; 23]);
>> var15 = s(1)

var15 =

  struct with fields:

    name: 'v'
     idx: []
       i: 7

>> var23 = s(2)

var23 =

  struct with fields:

    name: 'w'
     idx: []
       i: 4
```

`to_struct`

```
s = sm.to_struct()
```

Converts the object data *to* a struct that can later be converted back to an identical object using mp.struct2object. Useful for saving the object data to a MAT-file in Octave.

5.9.2 MP Set Manager Math Model Methods

The following are methods implemented by mp.set_manager_opt_model, a subclass of mp.set_manager inherited by all of the subclasses listed in Table 5-1.

See the online MP-Opt-Model Reference Manual for the implementing subclass for details.

params

```
[...] = sm.params()
[...] = sm.params(name)
[...] = sm.params(name, idx_list)
```

Returns set-type-specific parameters for the full set, if called without input arguments, or for a specific named or named and indexed subset. Outputs are determined by the implementing subclass.

set_params

```
sm.set_params(var, name, params, vals)
sm.set_params(var, name, idx_list, params, vals)
```

This method can be used to modify set-type-specific parameters for an existing subset. The var input is omitted for objects of the subclass mp.sm_variable.

display_soln

```
sm.display_soln(var, soln)
sm.display_soln(var, soln, name)
sm.display_soln(var, soln, name, idx_list)
sm.display_soln(var, soln, fid)
sm.display_soln(var, soln, fid, name)
sm.display_soln(var, soln, fid, name, idx_list)
```

Displays the solution values for all elements (default) or an individual named or named/indexed subset.

`has_parsed_soln`

```
TorF = sm.has_parsed_soln()
```

Returns true if parsed solution is available.

`parse_soln`

```
ps = sm.parse_soln(soln)
```

Parses a full solution struct into parts corresponding to individual subsets.

5.10 mp.opt_model Reference

5.10.1 Properties

The properties in mp.opt_model consist of one for each set type, one from the problem type, one for the problem solution, and one for storing arbitrary user data.

Table 5-12: mp.opt_model Properties

name	description
var	mp.sm_variable object for variables
lin	mp.sm_lin_constraint object for linear constraints
qcn	mp.sm_quad_constraint object for quadratic constraints
nle	mp.sm_nln_constraint object for nonlinear equality constraints
nli	mp.sm_nln_constraint object for nonlinear inequality constraints
qdc	mp.sm_quad_cost object for quadratic costs
nlc	mp.sm_nln_cost object for general nonlinear costs
prob_type	used to cache return value of problem_type method
soln	struct containing overall problem solution
userdata	struct for storing arbitrary user-defined data

5.10.2 Methods

Table 5-13: mp.opt_model Methods

name	description
get_set_types	return list of property names of set types managed by this class
copy	duplicate the object
display	displays the object
display_soln	displays solution values
from_struct	copy object data *from* a struct
get_idx	returns the idx struct for vars, constraints, costs
get_userdata	used to retrieve values of user data
has_parsed_soln	returns 1 if model has a parsed solution available, 0 otherwise
is_mixed_integer	returns 1 if any of the variables are binary or integer, 0 otherwise
is_solved	returns 1 if model has been solved, 0 otherwise
parse_soln	returns struct of all named solution vectors and shadow prices
problem_type	type of mathematical program represented by current model
solve	solves the model, see Section 5.4
to_struct	convert object data *to* a struct

6 Utility Functions

6.1 convert_lin_constraint

```
[ieq, igt, ilt, Ae, be, Ai, bi] = convert_lin_constraint(A, l, u)
[ieq, igt, ilt, A, b] = convert_lin_constraint(A, l, u)
```

This function converts linear constraints from a single set of doubly-bounded inequality constraints

$$l \leq Ax \leq u \tag{6.1}$$

to separate sets of equality and upper-bounded inequality constraints.

$$A_e x = b_e \tag{6.2}$$
$$A_i x \leq b_i \tag{6.3}$$

The first three return values are index vectors which satisfy the following.

```
Ae = A(ieq, :);
be = u(ieq, 1);
Ai  = [ A(ilt, :); -A(igt, :) ];
bi  = [ u(ilt, 1); -l(igt, 1) ];
```

Alternatively, the returned matrices and right hand side vectors can be stacked into a single set with the equalities first, then the inequalities.

```
A = [Ae; Ai]
b = [be; bi]
```

6.2 convert_quad_constraint

```
[ieq, igt, ilt, Qe, Be, de, Qi, Bi, di] = convert_quad_constraint(Q, B, l, u)
[ieq, igt, ilt, Q, B, d] = convert_quad_constraint(Q, B, l, u)
```

This function converts quadratic constraints from a single set of doubly-bounded inequality constraints

$$l_{\mathrm{q}_i} \leq \frac{1}{2} x^\mathsf{T} Q_i x + b_i x \leq u_{\mathrm{q}_i}, \quad \forall i = 1, \ldots, n_q \tag{6.4}$$

to separate sets of equality and upper-bounded inequality constraints.

$$\frac{1}{2}x^\mathsf{T} Q_{\mathrm{e}_j} x + b_{\mathrm{e}_j} x = d_{\mathrm{e}_j}, \quad \forall j = 1, \ldots, n_{\mathrm{e}_q} \tag{6.5}$$

$$\frac{1}{2}x^\mathsf{T} Q_{\mathrm{i}_j} x + b_{\mathrm{i}_j} x \leq d_{\mathrm{i}_j}, \quad \forall j = 1, \ldots, n_{\mathrm{i}_q} \tag{6.6}$$

The first three return values are index vectors which satisfy the following.

```
Qe = Q(ieq)
Be = B(ieq,:)
de = u(ieq)
Qi = [Q(ilt); -Q(igt)]
Bi = [B(ilt,:); -B(igt,:)]
```

Alternatively, the returned matrices and right hand side vectors can be stacked into a single set with the equalities first, then the inequalities.

```
Q = [Qe; Qi]
B = [Be; Bi]
d = [de; di]
```

6.3 convert_constraint_multipliers

```
[mu_l, mu_u] = convert_constraint_multipliers(lam, mu, ieq, igt, ilt)
```

This function converts the multipliers on linear or quadratic constraints from separate sets for equality and upper-bounded inequality constraints to those for doubly-bounded inequality constraints.

6.4 convert_lin_constraint_multipliers

```
[mu_l, mu_u] = convert_lin_constraint_multipliers(lam, mu, ieq, igt, ilt)
```

This function is deprecated. Instead, please use `convert_constraint_multipliers`.

6.5 have_fcn

This function is deprecated. Instead, please use `have_feature`, now included as part of MP-Test and described in the MP-Test `README` file. It is simply a drop-in replacement that has been reimplemented with an extensible, modular design, where the detection of a feature named `<tag>` is implemented by the function named `have_feature_<tag>`. The current `have_fcn` is a simple wrapper around `have_feature`.

6.6 mpomver

```
mpomver
v = mpomver
v = mpomver('all')
```

Prints or returns MP-Opt-Model version information for the current installation. When called without an input argument, it returns a string with the version number. Without an input argument it returns a struct with fields `Name`, `Version`, `Release`, and `Date`, all of which are strings. Calling `mpomver` without assigning the return value prints the version and release date of the current installation of MP-Opt-Model.

6.7 nested_struct_copy

```
ds = nested_struct_copy(d, s)
ds = nested_struct_copy(d, s, opt)
```

The `nested_struct_copy` function copies values from a source struct `s` to a destination struct `d` in a nested, recursive manner. That is, the value of each field in `s` is copied directly to the corresponding field in `d`, unless that value is itself a struct, in which case the copy is done via a recursive call to `nested_struct_copy`. Certain aspects of the copy behavior can be controled via the optional options struct `opt`, including the possible checking of valid field names.

6.8 mp.struct2object

```
obj = mp.struct2object(s)
```

As of version 10.x, Octave is still not able to save and load classdef objects. To aid in creating workarounds, this function allows objects to implement the following pattern with appropriately coded `to_struct()` and `from_struct()` methods:

```
s = obj.to_struct();            % convert to normal struct
new_obj = mp.struct2object(s);  % convert back to classdef object
isequal(new_obj, obj)           % returns true
```

The `to_struct()` method of the object must create a struct containing all of the data in the object, plus a char array field naned `class_` with the class name of the desired object, and an optional cell array field named `constructor_args_` with arguments to pass to the object constructor.

The `from_struct()` method takes a freshly constructed object and the struct above and copies the data from the struct back to the object.

This function creates an instance of the specified class, by calling its constructor with any specified arguments, then calling its `from_struct()` method.

6.9 Private Feature Detection Functions

The following are private functions that implement detection of specific optional functionality. They are not intended to be called directly, but rather are used to extend the capabilities of `have_feature`, a function included in MP-Test and described in the MP-Test `README` file.

6.9.1 `have_feature_bpmpd`

This function implements the `'bpmpd'` tag for `have_feature` to detect availability/version of BPMPD_MEX (interior point LP/QP solver). See also Appendix B.1.

6.9.2 `have_feature_catchme`

This function implements the `'catchme'` tag for `have_feature` to detect support for `catch me` syntax in `try/catch` constructs.

6.9.3 `have_feature_clp`

This function implements the `'clp'` tag for `have_feature` to detect availability/version of CLP (COIN-OR Linear Programming solver, LP/QP solver. See also Appendix B.2.

6.9.4 `have_feature_opti_clp`

This function implements the `'opti_clp'` tag for `have_feature` to detect the version of CLP distributed with OPTI Toolbox[32] [15]. See also Appendix B.2.

6.9.5 `have_feature_cplex`

This function implements the `'cplex'` tag for `have_feature` to detect availability/version of CPLEX, IBM ILOG CPLEX Optimizer. See also Appendix B.3.

[32]The OPTI Toolbox is available from https://www.inverseproblem.co.nz/OPTI/.

6.9.6 `have_feature_evalc`

This function implements the `'evalc'` tag for `have_feature` to detect support for `evalc()` function.

6.9.7 `have_feature_fmincon`

This function implements the `'fmincon'` tag for `have_feature` to detect availability/version of `fmincon`, solver from the MATLAB Optimization Toolbox. See also Appendix B.10.

6.9.8 `have_feature_fmincon_ipm`

This function implements the `'fmincon_ipm'` tag for `have_feature` to detect availability/version of `fmincon` with interior point solver from the MATLAB Optimization Toolbox 4.x and later. See also Appendix B.10.

6.9.9 `have_feature_fsolve`

This function implements the `'fsolve'` tag for `have_feature` to detect availability/version of `fsolve`, nonlinear equation solver from the MATLAB Optimization Toolbox or GNU Octave. See also Appendix B.10.

6.9.10 `have_feature_glpk`

This function implements the `'glpk'` tag for `have_feature` to detect availability/version of `glpk`, GNU Linear Programming Kit, LP/MILP solver. See also Appendix B.4.

6.9.11 `have_feature_gurobi`

This function implements the `'gurobi'` tag for `have_feature` to detect availability/version of `gurobi`, Gurobi optimizer. See also Appendix B.5.

6.9.12 `have_feature_highs`

This function implements the `'highs'` tag for `have_feature` to detect availability/version of `callhighs`, HiGHS optimizer. See also Appendix B.5.

6.9.13 `have_feature_intlinprog`

This function implements the `'intlinprog'` tag for `have_feature` to detect availability/version of `intlinprog`, MILP solver from the MATLAB Optimization Toolbox 7.0 (R2014a) and later.

6.9.14 `have_feature_ipopt`

This function implements the `'ipopt'` tag for `have_feature` to detect availability/version of IPOPT, a nonlinear programming solver from COIN-OR. See also Appendix B.7.

6.9.15 `have_feature_ipopt_auxdata`

This function implements the `'ipopt_auxdata'` tag for `have_feature` to detect support for `ipopt_auxdata()`, required by IPOPT 3.11 and later. See also Appendix B.7.

6.9.16 `have_feature_isequaln`

This function implements the `'isequaln'` tag for `have_feature` to detect support for `isequaln` function.

6.9.17 `have_feature_knitro`

This function implements the `'knitro'` tag for `have_feature` to detect availability/version of Artelys Knitro, a nonlinear programming solver. See also Appendix B.8.

6.9.18 `have_feature_knitromatlab`

This function implements the `'knitromatlab'` tag for `have_feature` to detect availability/version of Artelys Knitro 9.0.0 and later. See also Appendix B.8.

6.9.19 `have_feature_linprog`

This function implements the `'linprog'` tag for `have_feature` to detect availability/version of `linprog`, LP solver from the MATLAB Optimization Toolbox. See also Appendix B.10.

6.9.20 have_feature_linprog_ds

This function implements the 'linprog_ds' tag for have_feature to detect availability/version of linprog with support for the dual simplex method, from the MATLAB Optimization Toolbox 7.1 (R2014b) and later. See also Appendix B.10.

6.9.21 have_feature_mosek

This function implements the 'mosek' tag for have_feature to detect availability/version of MOSEK, LP/QP/MILP/MIQP solver. See also Appendix B.9.

6.9.22 have_feature_optim

This function implements the 'optim' tag for have_feature to detect availability/version of the Optimization Toolbox. See also Appendix B.10.

6.9.23 have_feature_optimoptions

This function implements the 'optimoptions' tag for have_feature to detect support for optimoptions, option setting funciton for the MATLAB Optimization Toolbox 6.3 and later. See also Appendix B.10.

6.9.24 have_feature_osqp

This function implements the 'osqp' tag for have_feature to detect availability/version of OSQP, **O**perator **S**plitting **Q**uadratic **P**rogram solver. See also Appendix B.11.

6.9.25 have_feature_quadprog

This function implements the 'quadprog' tag for have_feature to detect detect availability/version of quadprog, QP solver from the MATLAB Optimization Toolbox. See also Appendix B.10.

6.9.26 have_feature_quadprog_ls

This function implements the 'quadprog_ls' tag for have_feature to detect availability/version of quadprog with support for the large-scale interior point convex solver, from the MATLAB Optimization Toolbox 6.x and later. See also Appendix B.10.

6.9.27 have_feature_sdpt3

This function implements the 'sdpt3' tag for have_feature to detect availability/version of SDPT3 SDP solver, https://github.com/sqlp/sdpt3.

6.9.28 have_feature_sedumi

This function implements the 'sedumi' tag for have_feature to detect availability/version of SeDuMi SDP solver, http://sedumi.ie.lehigh.edu.

6.9.29 have_feature_yalmip

This function implements the 'yalmip' tag for have_feature to detect availability/version of YALMIP modeling platform, https://yalmip.github.io.

6.10 MATPOWER-related Functions

The following four functions are related specifically to MATPOWER, and are used for extracting relevant solver options from a MATPOWER options struct.

6.10.1 mpopt2nleqopt

```
nleqopt = mpopt2nleqopt(mpopt)
nleqopt = mpopt2nleqopt(mpopt, model)
nleqopt = mpopt2nleqopt(mpopt, model, alg)
```

The mpopt2nleqopt function returns an options struct suitable for nleqs_master or one of the solver specific equivalents. It is constructed from the relevant portions of mpopt, a MATPOWER options struct. The final alg argument allows the solver to be set explicitly (in nleqopt.alg). By default this value is set to 'DEFAULT', which currently selects Newton's method.

6.10.2 mpopt2nlpopt

```
nlpopt = mpopt2nlpopt(mpopt)
nlpopt = mpopt2nlpopt(mpopt, model)
nlpopt = mpopt2nlpopt(mpopt, model, alg)
```

The mpopt2nlpopt function returns an options struct suitable for nlps_master or one of the solver specific equivalents. It is constructed from the relevant portions of mpopt, a MATPOWER options struct. The final alg argument allows the solver to be set explicitly (in `nlpopt.alg`). By default this value is taken from `mpopt.opf.ac.solver`.

When the solver is set to `'DEFAULT'`, this function currently defaults to MIPS.

6.10.3 `mpopt2qpopt`

```
qpopt = mpopt2qpopt(mpopt)
qpopt = mpopt2qpopt(mpopt, model)
qpopt = mpopt2qpopt(mpopt, model, alg)
```

The `mpopt2qpopt` function returns an options struct suitable for `qps_master`, `miqps_master` or one of the solver specific equivalents. It is constructed from the relevant portions of `mpopt`, a MATPOWER options struct. The `model` argument specifies whether the problem to be solved is an LP, QP, MILP or MIQP problem to allow for the selection of a suitable default solver. The final `alg` argument allows the solver to be set explicitly (in `qpopt.alg`). By default this value is taken from `mpopt.opf.dc.solver`.

When the solver is set to `'DEFAULT'`, this function also selects the best available solver that is applicable[33] to the specific problem class, based on the following precedence: Gurobi, CPLEX, MOSEK, Optimization Toolbox, HiGHS, GLPK, BPMPD, MIPS.

6.10.4 `mpopt2qcqpopt`

```
qcqpopt = mpopt2qcqpopt(mpopt)
qcqpopt = mpopt2qcqpopt(mpopt, model)
qcqpopt = mpopt2qcqpopt(mpopt, model, alg)
```

The `mpopt2qcqpopt` function returns an options struct suitable for `qcqps_master` or one of the solver specific equivalents. It is constructed from the relevant portions of `mpopt`, a MATPOWER options struct. The `model` argument specifies whether the problem to be solved is an LP, QP, or QCQP problem to allow for the selection of a suitable default solver. The final `alg` argument allows the solver to be set explicitly (in `qcqpopt.alg`). By default this value is taken from `mpopt.opf.ac.solver`.

When the solver is set to `'DEFAULT'`, this function also selects the best available solver that is applicable to the specific problem class. For LP and QP problems

[33]GLPK is not available for problems with quadratic costs (QP and MIQP), BPMPD and MIPS are not available for mixed-integer problems (MILP and MIQP), and the Optimization Toolbox and HiGHS are not options for problems that combine the two (MIQP).

mpopt2qpopt is called, but for QCQP problems the precedence is: IPOPT, Artelys Knitro, fmincon, MIPS.

6.10.5 mpopt2pneopt

```
pneopt = mpopt2pneopt(mpopt)
pneopt = mpopt2pneopt(mpopt, model)
pneopt = mpopt2pneopt(mpopt, model, alg)
```

The mpopt2pneopt function returns an options struct suitable for pnes_master. It is constructed from the relevant portions of mpopt, a MATPOWER options struct. The final alg argument allows the solver to be set explicitly (in pneopt.alg). By default this value is set to 'DEFAULT', which is currently the only available method.

7 Acknowledgments

The authors would like to acknowledge the support of the research grants and contracts that have contributed directly and indirectly to the development of MP-Opt-Model. This includes funding from the Power Systems Engineering Research Center (PSerc), the U.S. Department of Energy,[34] and the National Science Foundation.[35]

The authors would also like to explicitly thank and acknowledge Shrirang Abhyankar and Alexander Flueck for their contributions to the continuation power flow code and documentation in Matpower upon which the predictor-corrector continuation method for parameterized nonlinear equations in MP-Opt-Model is based. And we thank Wilson González Vanegas for contributing the quadratic constraint implementation and QCQP handling functionality.

[34]Supported in part by the Consortium for Electric Reliability Technology Solutions (Certs) and the Office of Electricity Delivery and Energy Reliability, Transmission Reliability Program of the U.S. Department of Energy under the National Energy Technology Laboratory Cooperative Agreement No. DE-FC26-09NT43321.

[35]This material is based upon work supported in part by the National Science Foundation under Grant Nos. 0532744, 1642341 and 1931421. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Appendix A MP-Opt-Model Files, Functions and Classes

This appendix lists all of the files, functions and classes that MP-Opt-Model provides. In most cases, the function is found in a MATLAB M-file in the `lib` directory of the distribution, where the `.m` extension is omitted from this listing. For more information on each, at the MATLAB/Octave prompt, simply type `help` followed by the name of the function. For documentation and other files, the filename extensions are included.

Table A-1: MP-Opt-Model Files and Functions

name	description
`AUTHORS`	list of authors and contributors
`CHANGES`	MP-Opt-Model change history
`CITATION`	info on how to cite MP-Opt-Model
`CONTRIBUTING.md`	notes on how to contribute to the MP-Opt-Model project
`LICENSE`	MP-Opt-Model license (3-clause BSD license)
`README.md`	basic introduction to MP-Opt-Model
`docs/`	
`MP-Opt-Model-manual.pdf`	MP-Opt-Model User's Manual
`sphinx/`	contains Sphinx source for MP-Opt-Model Reference Manual
`src/MP-Opt-Model-manual/`	
`MP-Opt-Model-manual.tex`	LaTeX source for MP-Opt-Model User's Manual
`examples/`	MP-Opt-Model example functions, live scripts, etc.
`lib/`	MP-Opt-Model software (see Tables A-2, A-3, A-6, A-9 and A-10)
`t/`	MP-Opt-Model tests (see Table A-13)

Table A-2: Solver Master Functions

name	description
`miqps_master`	Mixed-Integer Quadratic Program Solver wrapper function, provides a unified interface for various MIQP/MILP solvers
`nleqs_master`	Nonlinear Equation Solver wrapper function, provides a unified interface for various nonlinear equation (NLEQ) solvers
`nlps_master`	Nonlinear Program Solver wrapper function, provides a unified interface for various NLP solvers
`qps_master`	Quadratic Program Solver wrapper function, provides a unified interface for various QP/LP solvers
`qcqps_master`	Quadratically-Constrained Quadratic Program Solver wrapper function, provides a unified interface for various QCQP solvers
`pnes_master`	Parameterized Nonlinear Equation Solver wrapper function, provides a unified interface for parameterized nonlinear equation (PNE) solvers
	deprecated functions
`miqps_matpower`	use `miqps_master` instead
`qps_matpower`	use `qps_master` instead

† Requires the installation of an optional package. See Appendix B for details on the corresponding package.

Table A-3: Solver Implementation Functions

name	description
`miqps_cplex`	MIQP/MILP solver API implementation for CPLEX (`cplexmiqp` and `cplexmilp`)†
`miqps_glpk`	MILP solver API implementation for GLPK†
`miqps_gurobi`	MIQP/MILP solver API implementation for Gurobi†
`miqps_highs`	MILP solver API implementation for HiGHS†
`miqps_mosek`	MIQP/MILP solver API implementation for MOSEK (`mosekopt`)†
`miqps_ot`	QP/MILP solver API implementation for MATLAB Opt Toolbox's `intlinprog`, `quadprog`, `linprog`
`nleqs_core`	core NLEQ solver API implementation with arbitrary update function, used to implement `nleqs_gauss_seidel` and `nleqs_newton`
`nleqs_fd_newton`	NLEQ solver API implementation for fast-decoupled Newton's method solver
`nleqs_fsolve`	NLEQ solver API implementation for `fsolve`
`nleqs_gauss_seidel`	NLEQ solver API implementation for Gauss-Seidel method solver
`nleqs_newton`	NLEQ solver API implementation for Newton's method solver
`nlps_fmincon`	NLP solver API implementation for MATLAB Opt Toolbox's `fmincon`
`nlps_ipopt`	NLP solver API implementation for IPOPT-based solver†
`nlps_knitro`	NLP solver API implementation for Artelys Knitro-based solver†
`qcqps_gurobi`	QCQP solver API implementation for Gurobi†
`qcqps_knitro`	QCQP solver API implementation for Artelys Knitro†
`qcqps_nlps`	QCQP solver API implementation for `nlps_master` (e.g. supports `fmincon`, IPOPT, Artelys Knitro, MIPS)
`qps_bpmpd`	QP/LP solver API implementation for BPMPD_MEX†
`qps_clp`	QP/LP solver API implementation for CLP†
`qps_cplex`	QP/LP solver API implementation for CPLEX (`cplexqp` and `cplexlp`)†
`qps_glpk`	QP/LP solver API implementation for GLPK†
`qps_gurobi`	QP/LP solver API implementation for Gurobi†
`qps_highs`	QP/LP solver API implementation for HiGHS†
`qps_ipopt`	QP/LP solver API implementation for IPOPT-based solver†
`qps_knitro`	QP/LP solver API implementation for Artelys Knitro-based solver†
`qps_mosek`	QP/LP solver API implementation for MOSEK (`mosekopt`)†
`qps_osqp`	QP/LP solver API implementation for OSQP†
`qps_ot`	QP/LP solver API implementation for MATLAB Opt Toolbox's `quadprog`, `linprog`

† Requires the installation of an optional package. See Appendix B for details on the corresponding package.

Table A-4: PNE Implementation Functions*

name	description
`pne_callback_default`	default callback function
`pne_callback_nose`	callback function for handling nose point detection events
`pne_callback_target_lam`	callback function for handling target λ events
`pne_detect_events`	detect events from event function values
`pne_detected_event`	returns detected event of a particular name
`pne_event_nose`	event function to detect the limit or nose point
`pne_event_target_lam`	event function to detect a target λ value
`pne_pfcn_arc_len`	arc length parameterization function
`pne_pfcn_natural`	natural parameterization function
`pne_pfcn_pseudo_arc_len`	pseudo arc length parameterization function
`pne_register_callbacks`	registers callback functions
`pne_register_events`	registers event functions

* Used to implement the predictor/corrector continuation method in `pnes_master`.

Table A-5: Solver Options, etc.

name	description
`artelys_knitro_options`	default options for Artelys Knitro solver†
`clp_options`	default options for CLP solver†
`cplex_options`	default options for CPLEX solver†
`glpk_options`	default options for GLPK solver†
`gurobi_options`	default options for Gurobi solver†
`highs_options`	default options for HiGHS solver†
`gurobiver`	prints version information for Gurobi/Gurobi_MEX
`highsver`	prints version information for HiGHS
`knitrover`	prints version information for Artelys Knitro
`ipopt_options`	default options for IPOPT solver†
`mosek_options`	default options for MOSEK solver†
`mosek_symbcon`	symbolic constants to use for MOSEK solver options†
`osqp_options`	default options for OSQP solver†
`osqpver`	prints version information for OSQP

† Requires the installation of an optional package. See Appendix B for details on the corresponding package.

Table A-6: Mathematical Model Class

name	description
mp.opt_model	mathematical programming and optimization model class
nlp_consfcn†	evaluates nonlinear constraints and gradients for mp.opt_model
nlp_costfcn†	evaluates nonlinear costs, gradients, Hessian for mp.opt_model
nlp_hessfcn†	evaluates nonlinear constraint Hessians for mp.opt_model

† Ideally should be implemented as a method of the mp.opt_model class, but a bug in Octave 4.2.x and earlier prevents it from finding an inherited method via a function handle, which MP-Opt-Model requires. Also used by the legacy opt_model.

Table A-7: MP Set Manager Classes

name	description
mp.set_manager	MP Set Manager base class, implements functionality to manage indexing and ordering of various named and indexed blocks of elements such as variables, constraints, etc.
mp.set_manager_opt_model	implements functionality to handle parameters and solution data for set type properties of mp.opt_model objects
mp.sm_lin_constraint	MP Set Manager class for linear constraints (e.g. mm.lin)
mp.sm_nln_constraint	MP Set Manager class for general nonlinear constraints (e.g. mm.nle, mm.nli)
mp.sm_nln_cost	MP Set Manager class for general nonlinear costs (e.g. mm.nlc)
mp.sm_quad_constraint	MP Set Manager class for quadratic constraints (e.g. mm.qcn)
mp.sm_quad_cost	MP Set Manager class for quadratic costs (e.g. mm.qdc)
mp.sm_variable	MP Set Manager class for variables (e.g. mm.var)

Table A-8: Legacy Mathematical Model Class*

name	description
`@opt_model/`	mathematical model class (subclass of `mp_idx_manager`)
`opt_model`	constructor for the `opt_model` class
`add_lin_constraint`	adds a named subset of linear constraints to the model
`add_nln_constraint`	adds a named subset of nonlinear constraints to the model
`add_nln_cost`	adds a named subset of general nonlinear costs to the model
`add_quad_cost`	adds a named subset of quadratic costs to the model
`add_var`	adds a named subset of variables to the model
`display`	called to display object when statement not ended with semicolon
`display_soln`	displays model solution
`eval_lin_constraint`	computes linear constraint values
`eval_nln_constraint`	computes nonlinear equality or inequality constraints and their gradients
`eval_nln_constraint_hess`	returns Hessian for full set of nonlinear equality or inequality constraints
`eval_nln_cost`	evaluates general nonlinear costs and derivatives
`eval_quad_cost`	evaluates quadratic costs and derivatives
`get_idx`	returns the idx struct for vars, constraints, costs
`get_soln`	returns named/indexed results for solved model
`init_indexed_name`	initializes dimensions for indexed named set of costs, constraints or variables
`is_mixed_integer`	indicates whether any of the variables are binary or integer
`params_lin_constraint`	returns individual or full set of linear constraint parameters
`params_nln_constraint`	returns individual nonlinear constraint parameters
`params_nln_cost`	returns individual general nonlinear cost parameters
`params_quad_cost`	returns individual or full set of quadratic cost coefficients
`params_var`	returns initial values, bounds and variable type for optimimization vector $\hat{x}$‡
`parse_soln`	returns struct of all named solution vectors and shadow prices
`problem_type`	indicates type of mathematical program (e.g. LP, QP, MILP, MIQP, or NLP)
`solve`	solves the model
`varsets_cell2struct`†	converts variable set list from cell array to struct array
`varsets_idx`	returns vector of indices into opt vector $\hat{x}$ for variable set list
`varsets_len`	returns total number of variables for variable set list
`varsets_x`	assembles cell array of sub-vectors of opt vector $\hat{x}$ specified by variable set list

* Deprecated, please use `mp.opt_model` instead, except as needed for backward compatibility.
† Private method for internal use only.
‡ For all, or alternatively, only for a named (and possibly indexed) subset.

Table A-9: Legacy MATPOWER Index Manager Class*

name	description
`@mp_idx_manager/`	MATPOWER Index Manager abstract class used to manage indexing and ordering of various sets of parameters, etc. (e.g. variables, constraints, costs for OPF Model objects).
`mp_idx_manager`	constructor for the `mp_idx_manager` class
`add_named_set`†	add named subset of a particular type to the object
`describe_idx`	describes indices of a given set type, e.g. variable 361 corresponds to `w(68)`
`get_idx`	returns index structure(s) for specified set type(s), with starting/ending indices and number of elements for each named (and optionally indexed) block
`get_userdata`	retreives values of user data stored in the object
`get`	returns the value of a field of the object
`getN`	returns the number of elements of any given set type‡
`init_indexed_name`	initializes dimensions for a particular indexed named set
`set_type_idx_map`	maps indices of a given set type, e.g. variable 361 corresponds to `w(68)`
`valid_named_set_type`†	returns label for given named set type if valid, empty otherwise

* Deprecated, please use `mp.set_manager` instead, except as needed for backward compatibility. The functionality previously implemented in `mp_idx_manager`, a parent container class for managing various set types, has been moved to `mp.set_manager`, a base class for the set type objects themselves.
† Private method for internal use only.
‡ For all, or alternatively, only for a named (and possibly indexed) subset.

Table A-10: Utility Functions

name	description
`have_fcn`	checks for availability of optional functionality*
`mpomver`	prints version information for MP-Opt-Model
`mpopt2nleqopt`	create/modify `nleqs_master` options struct from MATPOWER options struct
`mpopt2nlpopt`	create/modify `nlps_master` options struct from MATPOWER options struct
`mpopt2qpopt`	create/modify `mi/qps_master` options struct from MATPOWER options struct
`mpopt2qcqpopt`	create/modify `qcqps_master` options struct from MATPOWER options struct
`mpopt2pneopt`	create/modify `pnes_master` options struct from MATPOWER options struct
`nested_struct_copy`	copies the contents of nested structs

* Deprecated. Please use `have_feature` from MP-Test instead.

Table A-11: Feature Detection Functions*

name	description
have_feature_bpmpd	bp, BPMPD interior point LP/QP solver
have_feature_catchme	support for catch me syntax in try/catch constructs
have_feature_clp	CLP, LP/QP solver, https://github.com/coin-or/Clp
have_feature_opti_clp	version of CLP distributed with OPTI Toolbox, https://www.inverseproblem.co.nz/OPTI/
have_feature_cplex	CPLEX, IBM ILOG CPLEX Optimizer
have_feature_evalc	support for evalc() function
have_feature_fmincon	fmincon, solver from Optimization Toolbox
have_feature_fmincon_ipm	fmincon with interior point solver from Optimization Toolbox 4.x+
have_feature_fsolve	fsolve, nonlinear equation solver from Optimization Toolbox
have_feature_glpk	glpk, GNU Linear Programming Kit, LP/MILP solver
have_feature_gurobi	gurobi, Gurobi solver, https://www.gurobi.com/
have_feature_highs	callhighs, HiGHS solver, LP/QP/MILP solver
have_feature_intlinprog	intlinprog, MILP solver from Optimization Toolbox 7.0 (R2014a)+
have_feature_ipopt	Ipopt, NLP solver, https://github.com/coin-or/Ipopt
have_feature_ipopt_auxdata	support for ipopt_auxdata(), required by Ipopt 3.11 and later
have_feature_isequaln	support for isequaln function
have_feature_knitro	Artelys Knitro, NLP solver, https://www.artelys.com/solvers/knitro/
have_feature_knitromatlab	Artelys Knitro, version 9.0.0+
have_feature_linprog	linprog, LP solver from Optimization Toolbox
have_feature_linprog_ds	linprog w/dual-simplex solver from Optimization Toolbox 7.1 (R2014b)+
have_feature_mosek	MOSEK, LP/QP solver, https://www.mosek.com/
have_feature_optim	Optimization Toolbox
have_feature_optimoptions	optimoptions, option setting funciton for Optimization Toolbox 6.3+
have_feature_osqp	OSQP, **O**perator **S**plitting **Q**uadratic **P**rogram solver, https://osqp.org
have_feature_quadprog	quadprog, QP solver from Optimization Toolbox
have_feature_quadprog_ls	quadprog with large-scale interior point convex solver from Optimization Toolbox 6.x+
have_feature_sdpt3	SDPT3 SDP solver, https://github.com/sqlp/sdpt3
have_feature_sedumi	SeDuMi SDP solver, http://sedumi.ie.lehigh.edu
have_feature_yalmip	YALMIP modeling platform, https://yalmip.github.io

* These functions implement new tags and the detection of the corresponding features for have_feature which is part of MP-Test.

Table A-12: MP-Opt-Model Examples

name	description
`examples/`	MP-Opt-Model examples
`lp_ex1`	code for LP Example
`milp_ex1`	code for MILP Example
`milp_example1.mlx`	live script for MILP Example 1
`nleqs_master_ex1`	code for NLEQ Example 1 (see Section 4.5.1) for `nleqs_master`
`nleqs_master_ex2`	code for NLEQ Example 2 (see Section 4.5.2) for `nleqs_master`
`nlps_master_ex1`	code for NLP Example 1 (see Section 4.4.1) for `nlps_master`
`nlps_master_ex2`	code for NLP Example 2 (see Section 4.4.2) for `nlps_master`
`pne_ex1`	code for PNE Example (see Section 4.6.8) for `pnes_master`
`qcqp_ex1`	code for QCQP Example
`qcqp_example1.mlx`	live script for QCQP Example
`qp_ex1`	code for QP Example from Section 2.3

Table A-13: MP-Opt-Model Tests

name	description
lib/t/	MP-Opt-Model tests
test_mp_opt_model	runs full MP-Opt-Model test suite
t_have_fcn	runs tests for (deprecated) have_fcn
t_miqps_master	runs tests of MILP/MIQP solvers via miqps_master
t_mp_opt_model	runs tests for mp.opt_model objects
t_mm_solve_leqs	runs tests of LEQ solvers via mm.solve()
t_mm_solve_miqps	runs tests of MILP/MIQP solvers via mm.solve()
t_mm_solve_nleqs	runs tests of NLEQ solvers via mm.solve()
t_mm_solve_nlps	runs tests of NLP solvers via mm.solve()
t_mm_solve_pne	runs tests of PNE solvers via mm.solve()
t_mm_solve_qcqps	runs tests of QCQP solvers via mm.solve()
t_mm_solve_qps	runs tests of LP/QP solvers via mm.solve()
t_nested_struct_copy	runs tests for nested_struct_copy
t_nleqs_master	runs tests of NLEQ solvers via nleqs_master
t_nlps_master	runs tests of NLP solvers via nlps_master
t_om_solve_leqs	runs tests of LEQ solvers via om.solve()
t_om_solve_miqps	runs tests of MILP/MIQP solvers via om.solve()
t_om_solve_nleqs	runs tests of NLEQ solvers via om.solve()
t_om_solve_nlps	runs tests of NLP solvers via om.solve()
t_om_solve_pne	runs tests of PNE solvers via om.solve()
t_om_solve_qcqps	runs tests of QCQP solvers via om.solve()
t_om_solve_qps	runs tests of LP/QP solvers via om.solve()
t_opt_model	runs tests for opt_model objects
t_pnes_master	runs tests of PNE solvers via pnes_master
t_qcqps_master	runs tests of QCQP solvers via qcqps_master
t_qps_master	runs tests of LP/QP solvers via qps_master

Appendix B Optional Packages

There are a number of optional packages, not included in the MP-Opt-Model distribution, that MP-Opt-Model can utilize if they are installed in your Matlab/Octave path.

B.1 BPMPD_MEX – MEX interface for BPMPD

BPMPD_MEX [12, 13] is a Matlab MEX interface to BPMPD, an interior point solver for quadratic programming developed by Csaba Mészáros at the MTA SZTAKI, Computer and Automation Research Institute, Hungarian Academy of Sciences, Budapest, Hungary. It can be used by MP-Opt-Model's QP/LP solver interface.

This MEX interface for BPMPD was coded by Carlos E. Murillo-Sánchez, while he was at Cornell University. It does not provide all of the functionality of BPMPD, however. In particular, the stand-alone BPMPD program is designed to read and write results and data from MPS and QPS format files, but this MEX version does not implement reading data from these files into Matlab.

The current version of the MEX interface is based on version 2.21 of the BPMPD solver, implemented in Fortran. Builds are available for Linux (32-bit), Mac OS X (PPC, Intel 32-bit) and Windows (32-bit) at `http://www.pserc.cornell.edu/bpmpd/`.

When installed BPMPD_MEX can be used to solve general LP and QP problems via MP-Opt-Model's common QP solver interface `qps_master` with the algorithm option set to `'BPMPD'`, or by calling `qps_bpmpd` directly.

B.2 CLP – COIN-OR Linear Programming

The CLP [14] (**C**OIN-OR **L**inear **P**rogramming) solver is an open-source linear programming solver written in C++ by John Forrest. It can solve both linear programming (LP) and quadratic programming (QP) problems. It is primarily meant to be used as a callable library, but a basic, stand-alone executable version exists as well. It is available from the COIN-OR initiative at `https://github.com/coin-or/Clp`.

To use CLP with MP-Opt-Model, a MEX interface is required[36]. For Microsoft

[36]According to David Gleich at `http://web.stanford.edu/~dgleich/notebook/2009/03/coinor_clop_for_matlab.html`, there was a Matlab MEX interface to CLP written by Johan Lofberg and available (at some point in the past) at `http://control.ee.ethz.ch/~joloef/mexclp.zip`. Unfortunately, at the time of this writing, it seems it is no longer available there, but Davide Barcelli makes some precompiled MEX files for some platforms available here

Windows users, a pre-compiled MEX version of CLP (and numerous other solvers, such as GLPK and IPOPT) are easily installable as part of the OPTI Toolbox[37] [15]

With the MATLAB interface to CLP installed, it can be used to solve general LP and QP problems via MP-Opt-Model's common QP solver interface `qps_master` with the algorithm option set to `'CLP'`, or by calling `qps_clp` directly.

B.3 CPLEX – High-performance LP, QP, MILP and MIQP Solvers

The IBM ILOG CPLEX Optimizer, or simply CPLEX, is a collection of optimization tools that includes high-performance solvers for large-scale linear programming (LP) and quadratic programming (QP) problems, among others. More information is available at https://www.ibm.com/analytics/cplex-optimizer.

Although CPLEX is a commercial package, at the time of this writing the full version is available to academics at no charge through the IBM Academic Initiative program for teaching and non-commercial research. See http://www.ibm.com/support/docview.wss?uid=swg21419058 for more details.

When the MATLAB interface to CPLEX is installed, it can also be used to solve general LP, QP problems via MP-Opt-Model's common QP solver interface `qps_master`, or MILP and MIQP problems via `miqps_master`, with the algorithm option set to `'CPLEX'`, or by calling `qps_cplex` or `miqps_cplex` directly.

B.4 GLPK – GNU Linear Programming Kit

The GLPK [16] (**G**NU **L**inear **P**rogramming **K**it) package is intended for solving large-scale linear programming (LP), mixed-integer programming (MIP), and other related problems. It is a set of routines written in ANSI C and organized in the form of a callable library.

To use GLPK with MP-Opt-Model, a MEX interface is required[38]. For Microsoft Windows users, a pre-compiled MEX version of GLPK (and numerous other solvers, such as CLP and IPOPT) are easily installable as part of the OPTI Toolbox[39] [15].

---

http://www.dii.unisi.it/~barcelli/software.php, and the ZIP file linked as Clp 1.14.3 contains the MEX source as well as a `clp.m` wrapper function with some rudimentary documentation.

[37] The OPTI Toolbox is available from https://www.inverseproblem.co.nz/OPTI/.

[38] The http://glpkmex.sourceforge.net site and Davide Barcelli's page http://www.dii.unisi.it/~barcelli/software.php may be useful in obtaining the MEX source or pre-compiled binaries for Mac or Linux platforms.

[39] The OPTI Toolbox is available from https://www.inverseproblem.co.nz/OPTI/.

When GLPK is installed, either as part of Octave or with a MEX interface for MATLAB, it can be used to solve general LP problems via MP-Opt-Model's common QP solver interface `qps_master`, or MILP problems via `miqps_master`, with the algorithm option set to `'GLPK'`, or by calling `qps_glpk` or `miqps_glpk` directly.

B.5 Gurobi – High-performance LP, QP, QCQP, MILP and MIQP Solvers

Gurobi [17] is a collection of optimization tools that includes high-performance solvers for large-scale linear programming (LP) and quadratic programming (QP) problems, among others. The project was started by some former CPLEX developers. More information is available at https://www.gurobi.com/.

Although Gurobi is a commercial package, at the time of this writing their is a free academic license available. See https://www.gurobi.com/academia/for-universities for more details.

When Gurobi is installed, it can be used to solve general LP and QP problems via MP-Opt-Model's common QP solver interface `qps_master`, or MILP and MIQP problems via `miqps_master`, with the algorithm option set to `'GUROBI'`, or by calling `qps_gurobi` or `miqps_gurobi` directly.

B.6 HiGHS – High Performance Open-Source LP/QP/MILP Solvers

HiGHS[40] [19] is high performance serial and parallel software for solving large-scale sparse linear programming (LP), mixed-integer programming (MIP) and quadratic programming (QP) models, developed in C++11, with interfaces to C, C#, FORTRAN, Julia and Python. HiGHS is freely available under the MIT licence, and is downloaded from GitHub.[41]

To use HiGHS with MP-Opt-Model, a MEX interface is required. MEX source and pre-compiled binaries for Mac and Windows platforms are available from the HiGHSMEX project.[42]

When HiGHS and it's MEX interface is installed, it can be used to solve general LP and QP problems via MP-Opt-Model's common QP solver interface `qps_master`, or MILP problems via `miqps_master`, with the algorithm option set to `'HIGHS'`, or by calling `qps_highs` or `miqps_highs` directly.

[40] https://highs.dev

[41] https://github.com/ERGO-Code/HiGHS

[42] https://github.com/savyasachi/HiGHSMEX

B.7 Ipopt – Interior Point Optimizer

Ipopt [18] (**I**nterior **P**oint **OPT**imizer, pronounced I-P-Opt) is a software package for large-scale nonlinear optimization. It is is written in C++ and is released as open source code under the Common Public License (CPL). It is available from the COIN-OR initiative at https://github.com/coin-or/Ipopt. The code has been written by Carl Laird and Andreas Wächter, who is the COIN project leader for Ipopt.

MP-Opt-Model requires the Matlab MEX interface to Ipopt, which is included in some versions of the Ipopt source distribution, but must be built separately. Additional information on the MEX interface is available at https://projects.coin-or.org/Ipopt/wiki/MatlabInterface. Please consult the Ipopt documentation, web-site and mailing lists for help in building and installing the Ipopt Matlab interface. This interface uses callbacks to Matlab functions to evaluate the objective function and its gradient, the constraint values and Jacobian, and the Hessian of the Lagrangian.

Precompiled MEX binaries for a high-performance version of Ipopt, using the PARDISO linear solver [20, 21], are available from the PARDISO project[43]. For Microsoft Windows users, a pre-compiled MEX version of Ipopt (and numerous other solvers, such as CLP and GLPK) are easily installable as part of the OPTI Toolbox[44] [15].

When installed, Ipopt can be used by MP-Opt-Model to solve general LP, QP and NLP problems via MP-Opt-Model's common QP and NLP solver interfaces `qps_master` and `nlps_master` with the algorithm option set to `'IPOPT'`, or by calling `qps_ipopt` or `nlps_ipopt` directly.

B.8 Artelys Knitro – LP, QP, QCQP, and Non-Linear Programming Solver

Artelys Knitro [22] is a general purpose optimization solver specializing in nonlinear problems, available from Artelys. More information is available at https://www.artelys.com/solvers/knitro/ and https://www.artelys.com/docs/knitro/.

Although Artelys Knitro is a commercial package, at the time of this writing there is a free academic license available, with details on their download page.

When installed, Knitro's Matlab interface functions can be used by MP-Opt-Model to solve general NLP problems via MP-Opt-Model's common NLP solver interface

[43] See https://pardiso-project.org/ for the download links.

[44] The OPTI Toolbox is available from https://www.inverseproblem.co.nz/OPTI/.

nlps_master with the algorithm option set to 'KNITRO', or by calling nlps_knitro directly. As of MP-Opt-Model version 5.x, it can also solve LP/QP and QCQP problems via the common LP/QP and QCQP solver interfaces, qps_master and qcqps_master, respectively, with the algorithm option set to 'KNITRO', or by calling qps_knitro or qcqps_knitro directly.

B.9 MOSEK – High-performance LP, QP, MILP and MIQP Solvers

MOSEK is a collection of optimization tools that includes high-performance solvers for large-scale linear programming (LP) and quadratic programming (QP) problems, among others. More information is available at https://www.mosek.com/.

Although MOSEK is a commercial package, at the time of this writing there is a free academic license available. See https://www.mosek.com/products/academic-licenses/ for more details.

When the MATLAB interface to MOSEK is installed, it can be used to solve general LP and QP problems via MP-Opt-Model's common QP solver interface qps_master, or MILP and MIQP problems via miqps_master, with the algorithm option set to 'MOSEK', or by calling qps_mosek or miqps_mosek directly.

B.10 Optimization Toolbox – LP, QP, NLP, NLEQ and MILP Solvers

MATLAB's Optimization Toolbox [23, 24], available from The MathWorks, provides a number of high-performance solvers that MP-Opt-Model can take advantage of.

It includes fsolve for nonlinear equations (NLEQ), fmincon for nonlinear programming problems (NLP), and linprog and quadprog for linear programming (LP) and quadratic programming (QP) problems, respectively. For mixed-integer linear programs (MILP), it provides intlingprog. Each solver implements a number of different solution algorithms. More information is available from The MathWorks, Inc. at https://www.mathworks.com/.

When available, the Optimization Toolbox solvers can be used to solve general LP and QP problems via MP-Opt-Model's common QP solver interface qps_master, or MILP problems via miqps_master, with the algorithm option set to 'OT', or by calling qps_ot or miqps_ot directly. It can be to solve general NLP problems via MP-Opt-Model's common NLP solver interface nlps_master with the algorithm option set to 'FMINCON', or by calling nlps_fmincon directly. It can also be used to solve general NLEQ problems via MP-Opt-Model's common NLEQ solver interface nleqs_master with the algorithm option set to 'FSOLVE', or by calling nleqs_fsolve directly.

B.11 OSQP – Operator Splitting Quadratic Program Solver

OSQP [25] is a numerical optimization package for solving convex quadratic programming problems. It uses a custom ADMM-based first-order method requiring only a single matrix factorization in the setup phase. More information is available at https://osqp.org.

OSQP is a free, open-source package distributed under the Apache 2.0 License.

When the MATLAB interface to OSQP is installed, it can be used to solve general LP and QP problems via MP-Opt-Model's common QP solver interface qps_master with the algorithm option set to 'OSQP', or by calling qps_osqp directly.

Appendix C Legacy Mathematical Model Class – opt_model

With the release of version 5.0, the legacy mathematical model class opt_model and it's legacy parent class mp_idx_manager were replaced by mp.opt_model. This new class relies on mp.set_manager and its subclasses to implement core functionality for each set type (variables, linear constraints, etc.). They use a new syntax for adding elements, retrieving parameters, evaluating, displaying, etc. and do not include the deprecated methods listed in Table C-1.

The legacy classes, opt_model and mp_idx_manager, were retained for backward compatibility. While they have been updated internally to be based on mp.set_manager and its subclasses, they should be backward-compatible with previous versions. The deprecated methods are often simple wrappers around corresponding calls to mp.set_manager methods.

See Section 5.9 and the online MP-Opt-Model Reference Manual[45] for more details and up-to-date documentation.

By convention we use om to refer to opt_model objects, and mm to refer to mp.opt_model objects.

Aside from lacking the deprecated methods in Table C-1, the name for the quadratic coefficient parameter used for quadratic costs has been renamed from Q to H for consistency of notation across MP-Opt-Model.

Conversion between the new mm and legacy om type objects is simple and straightforward using respective copy constructors.

```
mm = mp.opt_model(om);
om = opt_model(struct(mm));
```

What follows is the documentation for the legacy classes.

The opt_model class provides facilities for constructing a mathematical programming or optimization problem by adding and managing the indexing of sets of variables, constraints and costs. The model can then be solved by simply calling the solve method which automatically selects and calls the appropriate master solver function, i.e. qps_master, miqps_master, nlps_master, nleqs_master or mplinsolve, depending on the type of problem.

[45] https://matpower.org/doc/mpom/

Table C-1: Deprecated Methods†

deprecated opt_model methods	mp.set_manager methods to use instead
om.add_named_set()	om.<sm>.add()
om.describe_idx()	om.<sm>.describe_idx()
om.getN()	om.<sm>.get_N()
om.init_indexed_name()	om.<sm>.init_indexed_name()
om.set_type_idx_map()	om.<sm>.set_type_idx_map()
om.add_lin_constraint()	om.lin.add()
om.add_nln_constraint()	om.nle.add() or om.nli.add()
om.add_nln_cost()	om.nlc.add()
om.add_quad_cost()	om.qdc.add()
om.add_var()	om.var.add()
om.eval_lin_constraint()	om.lin.eval()
om.eval_nln_constraint()	om.nle.eval() or om.nli.eval()
om.eval_nln_constraint_hess()	om.nle.eval_hess() or om.nli.eval_hess()
om.eval_nln_cost()	om.nlc.eval()
om.eval_quad_cost()	om.qdc.eval()
om.init_indexed_name()	om.<sm>.init_indexed_name()
om.params_lin_constraint()	om.lin.params()
om.params_nln_constraint()	om.nle.params() or om.nli.params()
om.params_nln_cost()	om.nlc.params()
om.params_quad_cost()	om.qdc.params()
om.params_var()	om.var.params()
om.set_params()	om.<sm>.set_params()
om.varsets_cell2struct()	om.var.varsets_cell2struct()
om.varsets_idx()	om.var.varsets_idx()
om.varsets_len()	om.var.varsets_len()
om.varsets_x()	om.var.varsets_x()

† In cases where a method appears in both opt_model and mp_idx_manager, it is deprecated in both.

In this manual, and in the code, om is the name of the variable used by convention for the legacy mathematical model object, which is typically created by calling the constructor opt_model with no arguments.

```
om = opt_model;
```

Variables, constraints and costs can then be added to the model using named sets. For variables and constraints, each set represents a column vector, and the sets are stacked in the order they are added to construct the full variable vector or full constraint vector. For costs, each set represents a component of a scalar cost, and the components are summed together to construct the full objective function value.

C.1 Adding Variables

```
om.add_var(name, N);
om.add_var(name, N, v0);
om.add_var(name, N, v0, vl);
om.add_var(name, N, v0, vl, vu);
om.add_var(name, N, v0, vl, vu, vt);
om.add_var(name, idx_list, N ...);
```

A named set of variables is added to the model using the `add_var` method, where `name` is a string containing the name of the set[46], `N` is the number n of variables in the set, `v0` is the initial value of the variables, `vl` and `vu` are the upper and lower bounds on the variables, and `vt` is the variable type. The accepted values for `vt` are:

- `'C'` – continuous
- `'I'` – integer
- `'B'` – binary, i.e. 0 or 1

The inputs `v0`, `vl` and `vu` are $n \times 1$ column vectors, `vt` is a scalar or a $1 \times n$ row vector. The defaults for the last four arguments, which are all optional, are for all to be continuous, unbounded and initialized to zero. That is, `v0`, `vl`, `vu`, and `vt` default to 0, $-\infty$, $+\infty$, and `'C'`, respectively.

For example, suppose our problem has variables u, v and w, which are vectors of length n_u, n_v, and n_w, respectively, where u is unbounded, v is non-negative and the lower and upper bounds on w are given in the vectors `wlb` and `wub`. Let us further suppose that the initial value of w is provided in `w0` and the first 3 elements of w are binary variables. And we will assume that the values of n_u, n_v, and n_w are available in the variables `nu`, `nv` and `nw`, respectively.

We can then add these variable sets to the model with the names **u**, **v**, and **w**, as follows:

```
wtype = repmat('C', 1, nw);  wt(1:3) = 'B';
om.add_var('u', nu);
om.add_var('v', nv, [], 0);
om.add_var('w', nw, w0, wlb, wub, wtype);
```

In this case, then, the full variable vector is the $(n_u + n_v + n_w) \times 1$ vector

$$x = \begin{bmatrix} u \\ v \\ w \end{bmatrix}. \tag{C.1}$$

See Section C.7 for details on indexed named sets and the `idx_list` argument.

[46] A set name must be a valid field name for a struct.

C.1.1 Variable Subsets

A key feature of MP-Opt-Model is that each set of constraints or costs can be defined in terms of the relevant variables only, as opposed to the entire variable vector x. This is done by specifying a variable subset, a cell array of the variable names of interest, in the `varsets` argument. Besides simplifying the constraint and cost definitions, another benefit of this approach is that it allows a model to be modified with new variables after some constraints and costs have already been added.

In the sections to follow, we will use the following two variable subsets for illustration purposes:

- {`'v'`} corresponding to $x_1 \equiv v$, and
- {`'u', 'w'`} corresponding to $x_2 \equiv \left[\begin{array}{c} u \\ w \end{array}\right]$.

C.2 Adding Constraints

A named set of constraints can be added to the model as soon as the variables on which it depends have been added. MP-Opt-Model currently supports three types of constraints, doubly-bounded linear constraints, general nonlinear equality constraints, and general nonlinear inequality constraints.

C.2.1 Linear Constraints

```
om.add_lin_constraint(name, A, l, u);
om.add_lin_constraint(name, A, l, u, varsets);
om.add_lin_constraint(name, idx_list, A ...);
```

In MP-Opt-Model, linear constraints take the form

$$l \leq Ax \leq u, \tag{C.2}$$

where x here refers to either the full variable vector *(default)*, or the vector obtained by stacking the subset of variables specified in `varsets`. Here `A` contains the $n_A \times n_x$ matrix A and `l` and `u` are the $n_A \times 1$ vectors l and u.[47]

For example, suppose our problem has the following three sets of linear constraints,

$$l_1 \leq A_1 x_1 \leq u_1 \tag{C.3}$$

$$l_2 \leq A_2 x_2 \tag{C.4}$$

$$A_3 x \leq u_3, \tag{C.5}$$

[47]The `A` matrix can be sparse.

where x_1 and x_2 are as defined in Section C.1.1 and x is the full variable vector from (C.1). Notice that the number of columns in A_1 and A_2 correspond to n_v and $n_u + n_w$, respectively, whereas A_3 has the full set of columns corresponding to x.

These three linear constraint sets can be added to the model with the names **lincon1**, **lincon2**, and **lincon3**, using the `add_lin_constraint` method as follows:

```
om.add_lin_constraint('lincon1', A1, l1, u1, {'v'});
om.add_lin_constraint('lincon2', A2, l2, [], {'u', 'w'});
om.add_lin_constraint('lincon3', A3, [], u3);
```

See Section C.7 for details on indexed named sets and the `idx_list` argument.

C.2.2 General Nonlinear Constraints

```
om.add_nln_constraint(name, N, iseq, fcn, hess);
om.add_nln_constraint(name, N, iseq, fcn, hess, varsets);
om.add_nln_constraint(name, idx_list, N ...);
```

MP-Opt-Model allows the user to implement general nonlinear constraints of the form

$$g(x) = 0, \text{ or} \tag{C.6}$$

$$g(x) \leq 0 \tag{C.7}$$

by providing the handle `fcn` of a function that evaluates the constraint and its Jacobian and another handle `hess` of a function that evaluates the Hessian. The number of constraints in the set is given by `N`, and `iseq` is set to 1 to specify an equality constraint or 0 for an inequality.

The calling syntax for `fcn` is:

```
g = fcn(x);
[g, dg] = fcn(x);
```

Here `g` is the $n_g \times 1$ vector $g(x)$ and `dg` is the $n_g \times n_x$ Jacobian matrix $J(x)$, where $J_{ij} = \frac{\partial g_i}{\partial x_j}$.

Rather than computing the full three-dimensional Hessian, the `hess` function actually evaluates the Jacobian of the vector $J^{\mathsf{T}}(x)\lambda$ for a specified value of the vector λ. The calling syntax for `hess` is:

```
d2g = hess(x, lambda);
```

For both functions, the first input argument `x` takes one of two forms. If the constraint set is added with `varsets` empty or missing, then `x` will be the full variable vector. Otherwise it will be a cell array of vectors corresponding to the variable sets specified in `varsets`.

There is also the option for `name` to be a cell array of constraint set names, in which case `N` is a vector, specifying the number of constraints in each corresponding set. In this case, `fcn` and `hess` are each still a single function handle, but the values computed by each correspond to the entire stacked collection of constraint sets together, as if they were a single set.

For example, suppose our problem has the following three sets of nonlinear constraints,

$$g_1(x_1) \leq 0 \tag{C.8}$$

$$g_2(x_2) = 0 \tag{C.9}$$

$$g_3(x) \leq 0, \tag{C.10}$$

where x_1 and x_2 are as defined in Section C.1.1 and x is the full variable vector from (C.1). Let `my_cons_fcn1`, `my_cons_fcn2`, and `my_cons_fcn3` be functions that evaluate $g_1(x_1)$, $g_2(x_2)$, and $g_3(x)$ and their gradients, respectively. Similarly, let `my_cons_hess1`, `my_cons_hess2`, and `my_cons_hess3` be Hessian evaluation functions for the same. The variables `ng1`, `ng2`, and `ng3` contain the number of constraints in the respective constraint sets.

These three nonlinear constraint sets can be added to the model with the names **nlncon1**, **nlncon2**, and **nlncon3**, using the `add_nln_constraint` method as follows:

```
fcn1 = @(x)my_cons_fcn1(x, <other_args>);
fcn2 = @(x)my_cons_fcn2(x, <other_args>);
fcn3 = @(x)my_cons_fcn3(x, <other_args>);
hess1 = @(x, lambda)my_cons_hess1(x, lambda, <other_args>);
hess2 = @(x, lambda)my_cons_hess2(x, lambda, <other_args>);
hess3 = @(x, lambda)my_cons_hess3(x, lambda, <other_args>);
om.add_nln_constraint('nlncon1', ng1, 0, fcn1, hess1 {'v'});
om.add_nln_constraint('nlncon2', ng2, 1, fcn2, hess2, {'u', 'w'});
om.add_nln_constraint('nlncon3', ng3, 0, fcn3, hess3);
```

In this case, the `x` variable passed to the `my_cons_fcn` and `my_cons_hess` functions will be as follows:

- `my_cons_fcn1`, `my_cons_hess1` $\longrightarrow$ `x` $= \{v\}$
- `my_cons_fcn2`, `my_cons_hess2` $\longrightarrow$ `x` $= \{u, w\}$
- `my_cons_fcn3`, `my_cons_hess3` $\longrightarrow$ `x` $= [u; v; w]$

See Section C.7 for details on indexed named sets and the `idx_list` argument.

C.3 Adding Costs

The objective of an MP-Opt-Model optimization problem is to *minimize* the sum of all costs added to the model. As with constraints, a named set of costs can be added to the model as soon as the variables on which it depends have been added. MP-Opt-Model currently supports two types of costs, quadratic costs and general nonlinear costs.

C.3.1 Quadratic Costs

```
om.add_quad_cost(name, Q, c);
om.add_quad_cost(name, Q, c, k);
om.add_quad_cost(name, Q, c, k, varsets);
om.add_quad_cost(name, idx_list, Q ...);
```

A quadratic cost set takes the form:

$$f(x) = \frac{1}{2} x^\mathsf{T} Q x + c^\mathsf{T} x + k \tag{C.11}$$

where x here refers to either the full variable vector *(default)*, or the vector obtained by stacking the subset of variables specified in `varsets`. Here `Q` contains the $n_x \times n_x$ matrix Q, `c` the $n_x \times 1$ vector c, and `k` the scalar k.[48]

Alternatively, if `Q` is an $n_x \times 1$ vector or empty, then $f(x)$ is also an $n_x \times 1$ vector, `k` can be $n_x \times 1$ or scalar, and the i-th element of $f(x)$ is given by

$$f_i(x) = \frac{1}{2} Q_i x_i^2 + c_i x_i + k_i. \tag{C.12}$$

where $k_i =$ `k` for all i if `k` is scalar.

For example, suppose our problem has the following three sets of quadratic costs,

$$q_1(x_1) = \frac{1}{2} {x_1}^\mathsf{T} Q_1 x_1 + {c_1}^\mathsf{T} x_1 + k_1 \tag{C.13}$$

$$q_2(x_2) = \frac{1}{2} {x_2}^\mathsf{T} Q_2 x_2 + {c_2}^\mathsf{T} x_2 + k_2 \tag{C.14}$$

$$q_3(x) = \frac{1}{2} x^\mathsf{T} Q_3 x + {c_3}^\mathsf{T} x + k_3, \tag{C.15}$$

where x_1 and x_2 are as defined in Section C.1.1 and x is the full variable vector from (C.1). Notice that the dimensions of Q_1 and Q_2 (and c_1 and c_2) correspond to n_v and $n_u + n_w$, respectively, whereas Q_3 (and c_3) correspond to the full x.

[48] The `Q` matrix can be sparse.

These three quadratic cost sets can be added to the model with the names **qcost1**, **qcost2**, and **qcost3**, using the `add_quad_cost` method as follows:

```
om.add_quad_cost('qcost1', Q1, c1, k1, {'v'});
om.add_quad_cost('qcost2', Q2, c2, k2, {'u', 'w'});
om.add_quad_cost('qcost3', Q3, c3, k3);
```

See Section C.7 for details on indexed named sets and the `idx_list` argument.

C.3.2 General Nonlinear Costs

```
om.add_nln_cost(name, N, fcn);
om.add_nln_cost(name, N, fcn, varsets);
om.add_nln_cost(name, idx_list, N ...);
```

MP-Opt-Model allows the user to implement a general nonlinear cost by providing the handle `fcn` of a function that evaluates the cost $f(x)$, its gradient and Hessian H, as described below. The `N` parameter specifies the dimension for vector valued cost functions, which are not yet implemented. Currently `N` must equal 1 or it will throw an error.

For a cost function $f(x)$, `fcn` should point to a function with the following interface:

```
f = fcn(x)
[f, df] = fcn(x)
[f, df, d2f] = fcn(x)
```

where `f` is a scalar with the value of the function $f(x)$, `df` is the $n_x \times 1$ gradient of f, and `d2f` is the $n_x \times n_x$ Hessian H, where n_x is the number of elements in x.

The first input argument `x` takes one of two forms. If the constraint set is added with `varsets` empty or missing, then `x` will be the full variable vector. Otherwise it will be a cell array of vectors corresponding to the variable sets specified in `varsets`.

For example, suppose our problem has three sets of nonlinear costs, $f_1(x_1)$, $f_2(x_2)$, $f_3(x)$, where x_1 and x_2 are as defined in Section C.1.1 and x is the full variable vector from (C.1). Let `my_cost_fcn1`, `my_cost_fcn2`, and `my_cost_fcn3` functions that evaluate $f_1(x)$, $f_2(x)$, and $f_3(x)$ and their gradients and Hessians, respectively.

These three nonlinear cost sets can be added to the model with the names **nlncost1**, **nlncost2**, and **nlncost3**, using the `add_nln_cost` method as follows:

```
fcn1 = @(x)my_cost_fcn1(x, <other_args>);
fcn2 = @(x)my_cost_fcn2(x, <other_args>);
fcn3 = @(x)my_cost_fcn3(x, <other_args>);
om.add_nln_cost('nlncost1', 1, fcn1 {'v'});
om.add_nln_cost('nlncost2', 1, fcn2, {'u', 'w'});
om.add_nln_cost('nlncost3', 1, fcn3);
```

In this case, the `x` variable passed to the `my_cost_fcn` functions will be as follows:

- `my_cost_fcn1` $\longrightarrow$ `x` $= \{v\}$
- `my_cost_fcn2` $\longrightarrow$ `x` $= \{u, w\}$
- `my_cost_fcn3` $\longrightarrow$ `x` $= [u; v; w]$

See Section C.7 for details on indexed named sets and the `idx_list` argument.

C.4 Solving the Model

```
om.solve()
[x, f, exitflag, output, jac] = om.solve()
[x, f, exitflag, output, lambda] = om.solve(opt)
[...] = om.solve(opt)
```

After all variables, constraints and costs have been added to the model, the mathematical programming or optimization problem can be solved simply by calling the `solve` method. This method automatically selects and calls, depending on the problem type, `mplinsolve` or one of the master solver interface functions from Section 4, namely `qps_master`, `miqps_master`, `nlps_master`, `nleqs_master`, or `pnes_master`. Note that one of the equation solvers is chosen if the model has no costs and no inequality constraints. In this case, if the number of variables is equal to the number of equality constraints, `mplinsolve` or `nleqs_master` is selected. If the number of variables is one more than the number of constraints `pnes_master` is chosen.

The results are stored in the `soln` field (see Section C.5.5) of the MP-Opt-Model object and can be returned in the optional output arguments. The input options struct `opt`, summarized in Tables C-2 and C-3, is optional, as are all of its fields. For details on the return values see the descriptions of the individual solver functions in Sections 4.1, 4.2, 4.4, 4.5, and 4.6. For linear equations, the `solver` and `opt` arguments for `mplinsolve`, described in Section 4.1 of the MIPS User's Manual, can be provided in the respective fields of `opt.leq_opt`.

Table C-2: Options for `solve`

name	default	description
`alg`	`'DEFAULT'`	determines which solver to use, see Table C-3
`verbose`	1	amount of progress info to be printed 0 – print no progress info 1–5 – print increasing level of progress info
`parse_soln`	0	flag that specifies whether or not to call the `parse_soln` method and place the return values in the `soln` property of the field type objects
`relax_integer`	0	relax integer constraints, if true
`x0`	*empty*	optional initial value of x, overrides value stored in model, *(ignored by some solvers)*
Additional Options for Specific Problem Types		
LP/QP		see Table 4-3
MILP/MIQP		see Table 4-5
NLP		see Table 4-11
LEQ		see Section 4.1 of the MIPS User's Manual
`leq_opt.solver`	`''`	see `help mplinsolve`, input argument `solver`
`leq_opt.opt`	*empty*	see `help mplinsolve`, input argument `opt`
NLEQ		see Table 4-14
PNE		see Table 4-20

Table C-3: Values for `alg` Option to `solve`

`alg` value	problem type(s)	description
`'DEFAULT'`	*all*	automatic, depends on problem type, uses first available of:
	LP	Gurobi, CPLEX, MOSEK, `linprog`,¶ HiGHS, GLPK, BPMPD, MIPS
	QP	Gurobi, CPLEX, MOSEK, `quadprog`,¶ HiGHS, BPMPD, MIPS
	MILP	Gurobi, CPLEX, MOSEK, `intlinprog`, HiGHS, GLPK
	MIQP	Gurobi, CPLEX, MOSEK
	NLP	MIPS
	MINLP	Artelys Knitro (not yet implemented)
	LEQ	built-in backslash operator
	NLEQ	Newton's method
	PNE	predictor/corrector continuation method
`'BPMPD'`	LP, QP	BPMPD*
`'CLP'`	LP, QP	CLP*
`'CPLEX'`	LP, QP, MILP, MIQP	CPLEX*
`'FD'`	NLEQ	fast-decoupled Newton's method†
`'FMINCON'`	NLP	MATLAB Opt Toolbox, `fmincon`*
`'FSOLVE'`	NLEQ	MATLAB Opt Toolbox, `fsolve`§
`'GLPK'`	LP, MILP	GLPK* *(LP only)*
`'GS'`	NLEQ	Gauss-Seidel method‡
`'GUROBI'`	LP, QP, MILP, MIQP	Gurobi*
`'HIGHS'`	LP, QP, MILP	HiGHS*
`'IPOPT'`	LP, QP, NLP	IPOPT*
`'KNITRO'`	NLP, MINLP	Artelys Knitro*
`'MIPS'`	LP, QP, NLP	MIPS, **M**ATPOWER **I**nterior **P**oint **S**olver
`'MOSEK'`	LP, QP, MILP, MIQP	MOSEK*
`'NEWTON'`	NLEQ	Newton's method
`'OSQP'`	LP, QP	OSQP*
`'OT'`	LP, QP, MILP	MATLAB Opt Toolbox, `quadprog`, `linprog`, `intlinprog`

* Requires the installation of an optional package. See Appendix B for details on the corresponding package.
† Fast-decoupled Newton requires setting `fd_opt.jac_approx_fcn` to a function handle that returns Jacobian approximations. See `help nleqs_fd_newton` for more details.
‡ Gauss-Seidel requires setting `gs_opt.x_update_fcn` to a function handle that updates x. See `help nleqs_gauss_seidel` for more details.
§ The `fsolve` function is included with GNU Octave, but on MATLAB it is part of the MATLAB Optimization Toolbox. See Appendix B for more information on the MATLAB Optimization Toolbox.
¶ If running on MATLAB.

C.5 Accessing the Model

C.5.1 Indexing

For each type of variable, constraint or cost, MP-Opt-Model maintains indexing information for each named set that is added, including the number of elements and the starting and ending indices. For each set type, this information is stored in a struct `idx` with fields `N`, `i1`, and `iN`, for storing number of elements, starting index and ending index, respectively. Each of these fields is also a struct with field names corresponding to the named sets.

For example, if `vv` is the struct of indexing information for variables, and we have added the **u**, **v**, and **w** variables as in Section C.1, then the contents of `vv` will be as shown in Table C-4.

Table C-4: Example Indexing Data

field	value	description
`vv.N.u`	n_u	number of u variables
`vv.N.v`	n_v	number of v variables
`vv.N.w`	n_w	number of w variables
`vv.i1.u`	1	starting index of u in full x
`vv.i1.v`	$n_u + 1$	starting index of v in full x
`vv.i1.w`	$n_u + n_v + 1$	starting index of w in full x
`vv.iN.u`	n_u	ending index of u in full x
`vv.iN.v`	$n_u + n_v$	ending index of v in full x
`vv.iN.w`	$n_u + n_v + n_w$	ending index of w in full x

`get_idx`

```
[idx1, idx2, ...] = om.get_idx(set_type1, set_type2, ...);
vv = om.get_idx('var');
[ll, nne, nni] = om.get_idx('lin', 'nle', 'nli');

vv = om.get_idx()
[vv, ll] = om.get_idx()
[vv, ll, nne] = om.get_idx()
[vv, ll, nne, nni] = om.get_idx()
[vv, ll, nne, nni, qq] = om.get_idx()
[vv, ll, nne, nni, qq, nnc] = om.get_idx()
```

The `idx` struct of indexing information for each set type is available via the `get_idx` method. When called with one or more set type strings as inputs, it returns the corresponding indexing structs. The list of valid set type strings is shown in Table C-5. When called without input arguments, the indexing structs are simply returned in the order listed in the table.

Table C-5: Valid Set Types

set type string	var name*	description
'var'	vv	variables
'lin'	ll	linear constraints
'nle'	nne	nonlinear equality constraints
'nli'	nni	nonlinear inequality constraints
'qdc'	qq	quadratic costs
'nlc'	nnc	general nonlinear costs

* The name of the variable used by convention for this indexing struct.

For the example model built in Sections C.1–C.3, where `x` and `lambda` are return values from the `solve` method, we can, for example, access the solved value of v and the shadow prices on the **nlncon3** constraints with the following code.

```
[vv, nne] = om.get_idx('var', 'nle');
v = x(vv.i1.v:vv.iN.v);
lam_nln3 = lambda.ineqnonlin(nni.i1.nlncon3:nni.iN.nlncon3);
```

`getN`

```
N = om.getN(set_type)
N = om.getN(set_type, name)
N = om.getN(set_type, name, idx_list)
```

The `getN` method can be used to get the number of elements in a particular named set, or the total for the set type. For example, the number n_v of elements in variable v and total number of elements in the full variable vector x can be obtained as follows.

```
nx = om.getN('var');
nv = om.getN('var', 'v');
```

See Section C.7 for details on indexed named sets and the `idx_list` argument.

set_type_idx_map

```
s = om.set_type_idx_map(set_type, idxs)
s = om.set_type_idx_map(set_type)
s = om.set_type_idx_map(set_type, idxs)
```

Given a particular index (or set of indices) for the full set of elements (e.g. variables or constraints) of a particular set type, the `set_type_idx_map` method can be used to determine which element of which particular named set the index corresponds to. If `idxs` is empty or not provided it defaults to `[1:ns]'`, where `ns` is the full dimension of the set corresponding to the all elements for the specified set type. Results are returned in a struct `s` of the same dimensions as the input `idxs`, where each element specifies the details of the corresponding named set. The fields of `s` are (1) `name`, with the name of the corresponding set, (2) `idx`, a cell array of indices for the name, if the named set is indexed and, (3) `i`, the index of the element within the set.

If `group_by_name` is true, then the results are consolidated, with a single entry in `s` for each unique name index pair, where `i` field is a vector and there is an additional field named `j` that is a vector with the corresponding index of the set type, equal to a particular element of `idxs`. In this case `s` is 1 dimensional.

This method can be useful, for example, when a solver reports an issue with a particular variable or constraint and you want to map it back to the named sets you have added to your model. Consider an example in which element 38 of the linear constraints corresponds to the 11th row of **lincon3** and elements 15 and 23 of the variable vector x correspond to element 7 of v and element 4 of w, respectively. The `set_type_idx_map` method can be used to return this information as follows:

```
>> lin38 = om.set_type_idx_map('lin', 38)

lin38 =

  struct with fields:

    name: 'lincon3'
     idx: []
       i: 11


>> s = om.set_type_idx_map('var', [15; 23]);
>> var15 = s(1)

var15 =

  struct with fields:

    name: 'v'
     idx: []
       i: 7

>> var23 = s(2)

var23 =

  struct with fields:

    name: 'w'
     idx: []
       i: 4
```

`describe_idx`

```
label = om.describe_idx(set_type, idxs)
```

Calls `set_type_idx_map` and formats each element of the return data as character array, returning a cell array of the same dimensions as `idxs`, except in the case where `idxs` is scalar, in which case it returns a scalar.

Consider an example in which element 38 of the linear constraints corresponds to the 11th row of **lincon3** and elements 15 and 23 of the variable vector x correspond to element 7 of v and element 4 of w, respectively. The `describe_idx` method can be used to return this information as follows:

```
>> lin38 = om.describe_idx('lin', 38)

lin38 =

    'lincon3(11)'


>> vars15_23 = om.describe_idx('var', [15; 23])

vars15_23 =

  2x1 cell array

    {'v(7)'}
    {'w(4)'}
```

C.5.2 Variables

`params_var`

```
[v0, vl, vu] = om.params_var()
[v0, vl, vu] = om.params_var(name)
[v0, vl, vu] = om.params_var(name, idx_list)
[v0, vl, vu, vt] = params_var(...)
```

The `params_var` method returns the initial value `v0`, lower bound `vl` and upper bound `vu` for the full variable variable vector x, or for a specific named variable set. Optionally also returns a corresponding char vector `vt` of variable types, where `'C'`, `'I'` and `'B'` represent continuous integer and binary variables, respectively.

Examples:

```
[x0, xmin, xmax] = om.params_var();
[w0, wlb, wub, wtype] = om.params_var('w');
```

See Section C.7 for details on indexed named sets and the `idx_list` argument.

C.5.3 Constraints

`params_lin_constraint`

```
[A, l, u] = om.params_lin_constraint()
[A, l, u] = om.params_lin_constraint(name)
[A, l, u] = om.params_lin_constraint(name, idx_list)
[A, l, u, vs] = om.params_lin_constraint(...)
[A, l, u, vs, i1, in] = om.params_lin_constraint(...)
```

With no input parameters, the `params_lin_constraint` method assembles and returns the parameters for the aggregate linear constraints from all linear constraint sets added using `add_lin_constraint`. The values of these parameters are cached for subsequent calls. The parameters are A, l, and u, where the linear constraint is of the form

$$l \leq Ax \leq u. \tag{C.16}$$

If a `name` is provided then it simply returns the parameters for the corresponding named set. An optional 4th output argument `vs` indicates the variable sets used by this constraint set. The size of `A` will be consistent with `vs`. Optional 5th and 6th output arguments `i1` and `iN` indicate the starting and ending row indices of the corresponding constraint set in the full aggregate constraint matrix.

Examples:

```
[A, l, u] = om.params_lin_constraint();
[A, l, u, vs, i1, iN] = om.params_lin_constraint('lincon2');
```

See Section C.7 for details on indexed named sets and the `idx_list` argument.

`params_nln_constraint`

```
N = om.params_nln_constraint(iseq, name)
N = om.params_nln_constraint(iseq, name, idx_list)
[N, fcn] = om.params_nln_constraint(...)
[N, fcn, hess] = om.params_nln_constraint(...)
[N, fcn, hess, vs] = om.params_nln_constraint(...)
[N, fcn, hess, vs, include] = om.params_nln_constraint(...)
```

Returns the parameters `N`, and optionally `fcn`, and `hess` provided when the corresponding named nonlinear constraint set was added to the model. Likewise for indexed named sets specified by `name` and `idx_list`. The `iseq` input should be set to 1 for equality constrainst and to 0 for inequality constraints.

An optional 4th output argument `vs` indicates the variable sets used by this constraint set.

And, for constraint sets whose functions compute the constraints for another set, an optional 5th output argument returns a struct with a cell array of set names in the `'name'` field and an array of corresponding dimensions in the `'N'` field.

`eval_lin_constraint`

```
Ax_u = om.eval_lin_constraint(x)
Ax_u = om.eval_lin_constraint(x, name)
Ax_u = om.eval_lin_constraint(x, name, idx_list)
[Ax_u, l_Ax] = om.eval_lin_constraint(...)
[Ax_u, l_Ax, A] = om.eval_lin_constraint(...)
```

Builds and evaluates the linear constraints $Ax - u$ and, optionally $l - Ax$ for the full set of constraints or an individual named subset for a given value of the variable vector x, based on constraints added by `add_lin_constraint`.

Examples:

```
[Ax_u, l_Ax, A] = om.eval_lin_constraint(x);
```

`eval_nln_constraint`

```
g = om.eval_nln_constraint(x, iseq)
g = om.eval_nln_constraint(x, iseq, name)
g = om.eval_nln_constraint(x, iseq, name, idx_list)
[g, dg] = om.eval_nln_constraint(...)
```

Builds the nonlinear equality constraints $g(x)$ or inequality constraints $h(x)$ and optionally their gradients for the full set of constraints or an individual named subset for a given value of the variable vector x, based on constraints added by `add_nln_constraint`, where $g(x) = 0$ and $h(x) \leq 0$.

Examples:

```
[g, dg] = om.eval_nln_constraint(x, 1);
[h, dh] = om.eval_nln_constraint(x, 0);
```

eval_nln_constraint_hess

```
d2G = om.eval_nln_constraint_hess(x, lam, iseq)
```

Builds the Hessian of the full set of nonlinear equality constraints $g(x)$ or inequality constraints $h(x)$ for given values of the variable vector x and dual variables `lam`, based on constraints added by `add_nln_constraint`, where $g(x) = 0$ and $h(x) \leq 0$.

Examples:

```
d2G = om.eval_nln_constraint_hess(x, lam, 1)
d2H = om.eval_nln_constraint_hess(x, lam, 0)
```

C.5.4 Costs

params_quad_cost

```
[Q, c] = om.params_quad_cost()
[Q, c] = om.params_quad_cost(name)
[Q, c] = om.params_quad_cost(name, idx_list)
[Q, c, k] = om.params_quad_cost(...)
[Q, c, k, vs] = om.params_quad_cost(...)
```

With no input parameters, the `params_quad_cost` method assembles and returns the parameters for the aggregate quadratic cost from all quadratic cost sets added using `add_quad_cost`. The values of these parameters are cached for subsequent calls. The parameters are Q, c, and optionally k, where the quadratic cost is of the form

$$f(x) = \frac{1}{2}x^{\mathsf{T}}Qx + c^{\mathsf{T}}x + k. \tag{C.17}$$

If a `name` is provided then it simply returns the parameters for the corresponding named set. In this case, `Q` and `k` may be vectors, corresponding to a cost function $f(x)$ where the i-th element takes the form

$$f_i(x) = \frac{1}{2}Q_i x_i^2 + c_i x_i + k_i, \tag{C.18}$$

depending on how the constraint set was initially specified.

An optional 4th output argument `vs` indicates the variable sets used by this cost set. The size of `Q` and `c` will be consistent with `vs`.

Examples:

```
[Q, c, k] = om.params_quad_cost();
[Q, c, k, vs, i1, iN] = om.params_quad_cost('qcost2');
```

See Section C.7 for details on indexed named sets and the `idx_list` argument.

`params_nln_cost`

```
[N, fcn] = om.params_nln_cost(name)
[N, fcn] = om.params_nln_cost(name, idx_list)
[N, fcn, vs] = om.params_nln_cost(...)
```

Returns the parameters `N` and `fcn` provided when the corresponding named general nonlinear cost set was added to the model. Likewise for indexed named sets specified by `name` and `idx_list`.

An optional 3rd output argument `vs` indicates the variable sets used by this constraint set.

`eval_quad_cost`

```
f = om.eval_quad_cost(x ...)
[f, df] = om.eval_quad_cost(x ...)
[f, df, d2f] = om.eval_quad_cost(x ...)
[f, df, d2f] = om.eval_quad_cost(x, name)
[f, df, d2f] = om.eval_quad_cost(x, name, idx_list)
```

The `eval_quad_cost` method evaluates the cost function and its derivatives for an individual named set or the full set of quadratic costs for a given value of the variable vector x, based on costs added by `add_quad_cost`.

Examples:

```
[f, df, d2f] = om.eval_quad_cost(x);
[f, df, d2f] = om.eval_quad_cost(x, 'qcost3');
```

See Section C.7 for details on indexed named sets and the `idx_list` argument.

eval_nln_cost

```
f = om.eval_nln_cost(x)
[f, df] = om.eval_nln_cost(x)
[f, df, d2f] = om.eval_nln_cost(x)
[f, df, d2f] = om.eval_nln_cost(x, name)
[f, df, d2f] = om.eval_nln_cost(x, name, idx_list)
```

The `eval_nln_cost` method evaluates the cost function and its derivatives for an individual named set or the full set of general nonlinear costs for a given value of the variable vector x, based on costs added by `add_nln_cost`.

Examples:

```
[f, df, d2f] = om.eval_nln_cost(x);
[f, df, d2f] = om.eval_nln_cost(x, 'nlncost2');
```

See Section C.7 for details on indexed named sets and the `idx_list` argument.

C.5.5 Model Solution

The solved results of a model, as returned by the `solve` method, are stored in the `soln` field of the MP-Opt-Model object as summarized in Table C-6.

is_solved

```
TorF = om.is_solved()
```

The `is_solved` method returns `1` if the model has been solved, `0` otherwise.

get_soln

```
vals = om.get_soln(set_type, name)
vals = om.get_soln(set_type, name, idx_list)
vals = om.get_soln(set_type, tags, name)
vals = om.get_soln(set_type, tags, name, idx_list)
```

The `get_soln` method can be used to extract solved results for a given named set of variables, constraints or costs. The input arguments for `get_soln` are summarized in Table C-7 and Table C-8. The variable number of output arguments correspond to the `tags` input. If `tags` is empty or not specified, the calling context will define the number of outputs, returned in order of default tags for the specified `set_type`.

Table C-6: Model Solution

field	description
`om`	MP-Opt-Model object
`.soln`	model solution struct
`.x`	solution vector
`.f`	final function value,* $f(x)$
`.eflag`	exit flag
	1 – converged successfully
	≤ 0 – solver-specific failure code
`.output`	output struct with the following fields:
	`alg` – algorithm code of solver used
	`et` – solution elapsed time in seconds
	(others) – solver-specific fields
`.jac`	final value of Jacobian matrix (for LEQ/NLEQ)
`.lambda`	shadow prices on constraints
`.lower`	variable lower bound
`.upper`	variable upper bound
`.mu_l`	linear constraint lower bound
`.mu_u`	linear constraint upper bound
`.eqnonlin`	nonlinear equality constraints
`.ineqnonlin`	nonlinear inequality constraints
Parsed Solution†	
`.var.soln`	parsed solution for variables‡
`.lin.soln`	parsed solution for linear constraints‡
`.nle.soln`	parsed solution for nonlinear equality constraints‡
`.nli.soln`	parsed solution for nonlinear inequality constraints‡
Parsed Solution (deprecated)†	
`.var`	parsed solution for variables‡
`.lin`	parsed solution for linear constraints‡
`.nle`	parsed solution for nonlinear equality constraints‡
`.nli`	parsed solution for nonlinear inequality constraints‡

* Objective function value for optimization problems, constraint function value for sets of equations.
† Only available after calling `parse_soln(true)` or calling `solve()` with the `opt.parse_soln` option set to 1.
‡ See Table C-9 for details.

Examples:

Value of variable named `'P'` and shadow prices on its bounds.

```
[P, muPmin, muPmax] = om.get_soln('var', 'P');
```

Shadow prices on upper and lower linear constraint set named 'lin_con_1'.

```
[mu_u, mu_l] = om.get_soln('lin', {'mu_u', 'mu_l'}, 'lin_con_1');
```

Jacobian of the (2,3)-element of the indexed nonlinear equality constraint set named 'nle_con_b'.

```
dg_b_2_3 = om.get_soln('nle', 'dg', 'nle_con_b', {2,3});
```

Table C-7: Inputs for get_soln

name	default	description
set_type	*required*	one of the following, specifying the type of set 'var' – variables 'lin' – linear constraints 'nle' – nonlinear equality constraints 'nli' – nonlinear inequality constraints 'nlc' – nonlinear costs 'qdc' – quadratic costs
tags	*depends*	char array or cell array of char arrays specifying the desired output(s)†
name	*required*	char array specifying the name of the set
idx	*empty*	cell array specifying the indices of the set

† Valid values and defaults for tags depend on set_type. See Table C-8 for details.

Table C-8: Values of `tags` input to `get_soln`

set type	valid tag values	description
`'var'`		default `tags` = {`'x'`, `'mu_l'`, `'mu_u'`}
	`'x'`	value of solution variable
	`'mu_l'`	shadow price on variable lower bound
	`'mu_u'`	shadow price on variable upper bound
`'lin'`		default `tags` = {`'f'`} for LEQ problems, {`'g'`, `'mu_l'`, `'mu_u'`} otherwise
	`'f'`†	equality constraint values, $Ax - u$
	`'g'`	1×2 cell array of upper and lower constraint values, $\{Ax - u, \, l - Ax\}$
	`'Ax_u'`	upper constraint value, $Ax - u$
	`'l_Ax'`	lower constraint value, $l - Ax$
	`'mu_l'`	shadow price on constraint lower bound
	`'mu_u'`	shadow price on constraint upper bound
`'nle'`		default `tags` = {`'g'`, `'lam'`, `'dg'`}
	`'g'`	constraint value, $g(x)$
	`'lam'`	shadow price on constraint
	`'dg'`	Jacobian of constraint
`'nli'`		default `tags` = {`'h'`, `'mu'`, `'dh'`}
	`'h'`	constraint value, $h(x)$
	`'mu'`	shadow price on constraint
	`'dh'`	Jacobian of constraint
`'nlc'` or `'qdc'`		default `tags` = {`'f'`, `'df'`, `'d2f'`}
	`'f'`	cost function value, $f(x)$‡
	`'df'`	gradient of cost function
	`'d2f'`	Hession of cost function

† For LEQ problems only.
‡ For `'qdc'`, $f(x)$ can return be a vector.

`parse_soln`

```
ps = om.parse_soln()
om.parse_soln(stash)
```

The `parse_soln` method returns a struct of parsed solution vector and shadow price values for each named set of variables and constraints. The returned `ps` (parsed solution) struct has the format shown in Table C-9, where each of the terminal elements is a struct with fields corresponding to the respective named sets.

Table C-9: Output of `parse_soln`

fields	description
`ps`	
`.var`	variables
`.val`	struct of solution vectors
`.mu_l`	struct of lower bound shadow prices
`.mu_u`	struct of upper bound shadow prices
`.lin`	linear constraints
`.mu_l`	struct of lower bound shadow prices
`.mu_u`	struct of upper bound shadow prices
`.nle`	nonlinear equality constraints
`.lam`	struct of shadow prices
`.nli`	nonlinear inequality constraints
`.mu`	struct of shadow prices

The value of each element in the returned struct can be obtained via the `get_soln` method as well, but `parse_soln` is generally more efficient if a complete set of values is needed.

If the optional `stash` input argument is present and true, the fields of the return struct are copied to the `soln` property of the corresponding set type object in `om`. They are also copied to `om.soln` where they are available with others listed in Table C-6, but this is deprecated.

`has_parsed_soln`

```
TorF = om.has_parsed_soln()
```

The `has_parsed_soln` method returns `1` if the model has a parsed solution available in the `soln` property of the set type objects, `0` otherwise.

C.6 Modifying the Model

The parameters for an existing MP-Opt-Model object can be modified, rather than having to rebuild a new model from scratch.

`set_params`

```
om.set_params(set_type, name, params, vals)
om.set_params(set_type, name, idx_list, params, vals)
```

The `set_params` method, inputs summarized in Table C-10, can be used to modify any of the parameters associated with an existing variable, cost or constraint set.

Examples:

```
om.set_params('var', 'Pg', 'v0', Pg0);
om.set_params('lin', 'y', {2,3}, {'l', 'u'}, {l, u});
om.set_params('nle', 'Pmis', 'all', {N, @fcn, @hess, vs});
```

Table C-10: Inputs for `set_params`

name	description
`set_type`	one of the following, specifying the type of set, with the corresponding valid parameter names `'var'` – variables: `N`, `v0`†, `vl`†, `vu`†, `vt`† `'lin'` – linear constraints: `A`, `l`, `u`†, `vs`† `'nle'` – nonlinear equality constraints: `N`, `fcn`, `hess`, `vs`† `'nli'` – nonlinear inequality constraints: `N`, `fcn`, `hess`, `vs`† `'nlc'` – nonlinear costs: `N`, `fcn`, `vs`† `'qdc'` – quadratic costs: `Q`, `c`†, `k`†, `vs`†
`name`	char array specifying the name of the set
`idx`‡	cell array specifying the indices of the set
`params`	one of the following: `'all'` – indicates that `vals` is a cell array of values whose elements correspond to the input parameters of the respective `add_*` method *char array* – name of parameter to modify *cell array* – names of parameters to modify
`vals`	new value or cell array of new values corresponding the parameter name(s) specified in `params`

† Optional when `params = 'all'`.
‡ The `idx` argument is optional.

C.7 Indexed Sets

A variable, constraint or cost set is typically identified simply by a `name`, but it is also possible to use indexed names. For example, an optimal scheduling problem with a one week horizon might include a vector variable **y** for each day, indexed from 1 to 7, and another vector variable **z** for each hour of each day, indexed from (1, 1) to (7, 24).

In this case, we case use a single indexed named set for **y** and another for **z**. The dimensions are initialized via the `init_indexed_name` method before adding the variables to the model.[49]

`init_indexed_name`

```
om.init_indexed_name(set_type, name, dim_list)
```

Examples:

```
om.init_indexed_name('var', 'y', {7});
om.init_indexed_name('var', 'z', {7, 24});
```

After initializing the dimensions, indexed named sets of variables, constraints or costs can be added by supplying the indices in the `idx_list` argument following the `name` argument in the call to the corresponding `add_var`, `add_lin_constraint`, `add_nln_constraint`, `add_quad_cost`, or `add_nln_cost` method. The `idx_list` argument is simply a cell array containing the indices of interest.

Examples:

```
for d = 1:7
    om.add_var('y', {d}, ny(d), y0{d}, yl{d}, yu{d}, yt{d});
end
for d = 1:7
    for h = 1:24
        om.add_var('z', {d, h}, nz(d, h), z0{d, h}, zl{d, h}, zu{d, h});
    end
end
```

[49]The same is true for indexed named sets of constraints or costs.

Other Methods

All of the methods that take a `name` argument to specify a simple named set, can also take an `idx_list` argument immediately following `name` to handle the equivalent indexed named set. The `idx_list` argument is simply a cell array containing the indices of interest. This includes `getN` and the methods that begin with `add_`, `params_`, and `eval_`.[50]

For an indexed named set, the fields under the `N`, `i1` and `iN` fields in the index information struct returned by `get_idx` are now arrays of the appropriate dimension, not just scalars as in Table C-4. For example, to find the starting index of the z variable for day 2, hour 13 in our example you would use `vv.i1.z(2, 13)`. Similarly for the values returned by `getN` when specifying only the `set_type` and `name`.

Variable Subsets

A variable subset for a simple named set, usually specified by the variable `varsets` or else `vs`, is a cell array of variable set names. For indexed named sets of variables, on the other hand, it is a struct array with two fields `name` and `idx`. For each element of the struct array the `name` field contains the name of the variable set and the `idx` field contains a cell array of indices of interest.

For example, to specify a variable subset consisting of the **y** variable for day 3 and the **z** variable for day 3, hour 7, the variable subset could be defined as follows.

```
vs = struct('name', {'y', 'z'}, 'idx', {{3}, {3,7}});
```

C.8 Miscellaneous Methods

C.8.1 Public Methods

`copy`

```
om2 = om.copy()
```

The `copy` method can be used to make a copy of an MP-Opt-Model object.

[50]Currently, `eval_nln_constraint` and `eval_nln_constraint_hess` are only implemented for the full aggregate set of constraints and do not yet support evaluation of individual constraint sets.

display

```
om
```

The `display` method displays the variable, constraint and cost sets that make up the model, along with their indexing data.

display_soln

```
om.display_soln()
om.display_soln(set_type)
om.display_soln(set_type, name)
om.display_soln(set_type, name, idx_list)
```

The `display_soln` method displays the model solution, including values, bounds and shadow prices for variables and linear constraints, values and shadow prices for nonlinear constraints, and individual cost components. Results are displayed for each `set_type` or specified `set_type` and for each named/indexed set or a specified `name`/`idx`.

get_userdata

```
data = om.get_userdata(name)
```

MP-Opt-Model allows the user to store arbitrary data in fields of the `userdata` property, which is a simple struct. The `get_userdata` method returns the value of the field specified by `name`, or an empty matrix if the field does not exist in `om.userdata`.

is_mixed_integer

```
TorF = om.is_mixed_integer()
```

Returns 1 if any of the variables are binary or integer, 0 otherwise.

problem_type

```
prob_type = om.problem_type()
prob_type = om.problem_type(recheck)
```

Returns a string identifying the type of mathematical program represented by the current model, based on the variables, costs,and constraints that have been added to the model. Used to automatically select an appropriate solver.

Linear and nonlinear equations are models with no costs, no inequality constraints, and an equal number of continuous variables and equality constraints.

The `prob_type` string is one of the following:

- `'LEQ'` – linear equation
- `'NLEQ'` – nonlinear equation
- `'LP'` – linear program
- `'QP'` – quadratic program
- `'NLP'` – nonlinear program
- `'MILP'` – mixed-integer linear program
- `'MIQP'` – mixed-integer quadratic program
- `'MINLP'` – mixed-integer nonlinear program[51]

The output value is cached for future calls, but calling with a true value for the optional `recheck` argument will force it to recheck in case the problem type has changed due to modifying the variables, constraints or costs in the model.

varsets_cell2struct

```
varsets = om.varsets_cell2struct(varsets)
```

Converts variable subset `varsets` from a cell array to a struct array, if necessary.

varsets_idx

```
k = om.varsets_idx(varsets)
```

Returns a vector of indices into the full variable vector x corresponding to the variable sets specified by `varsets`.

varsets_len

```
nv = om.varsets_len(varsets)
```

Returns the total number of elements in the variable sub-vector specified by `varsets`.

[51]MP-Opt-Model does not yet implement solving MINLP problems.

`varsets_x`

```
x = om.varsets_x(x, varsets)
x = om.varsets_x(x, varsets, 'vector')
```

Returns a cell array of sub-vectors of x specified by `varsets`, or the full variable vector x, if `varsets` is empty.

If a 3rd argument is present (value is ignored) the returned value is a single numeric vector with the individual components stacked vertically.

C.8.2 Private Methods

`def_set_types`

```
om.def_set_types()
```

The `def_set_types` method is a *private* method that assigns a struct to the `set_types` property of the object. The fields of the struct correspond to the valid set types listed in Table C-5 and the values are labels used by the `display` method.

`init_set_types`

```
om.init_set_types()
```

Initializes the base data structures for each set type.

C.9 Matpower Index Manager Base Class – `mp_idx_manager`

Most of the functionality of the `opt_model` class related to managing the indexing of the various set types is inherited from the Matpower Index Manager base class named `mp_idx_manager`. The properties and methods implemented in this base class and inherited or overridden by `opt_model` are listed in Table C-11.

With MP-Opt-Model 5.0, the implementation for most of the functionality for the Matpower Index Manager base class has been moved to `mp.set_manager`. Now it is essentially a backward compatible container base class for `mp.set_manager` objects and many of its methods are just wrappers around the corresponding `mp.set_manager` method.

The Matpower Index Manager base class initializes and manages the data that is common across all set types. Table C-12 illustrates for an example `'var'` set type, such as defined in `opt_model`, what the data structure looks like, but it is the same for any other set types defined by child classes, such as `opt_model`.

Table C-11: MATPOWER Index Manager (`mp_idx_manager`) Properties and Methods

name	description
Properties	
`set_types`	struct whose fields define the valid set types*
`userdata`	struct for storing arbitrary user-defined data
Public Methods	
`mp_idx_manager`	constructor for `mp_idx_manager` class
`copy`	makes a copy of an existing `mp_idx_manager` object
`describe_idx`	wrapper around `describe_idx` for a given set object
`display_set`	wrapper around `display` for a given set object, typically called by `display`
`from_struct`	copy object data *from* a struct
`get`	access (possibly nested) fields of the object
`get_idx`	returns index structure(s) for specified set type(s), with starting/ending indices and number of elements for each named (and optionally indexed) block
`get_userdata`	retreives values of user data stored in the object
`getN`	returns the number of elements of any given set type†
`init_indexed_name`	wrapper around `init_indexed_name` for a given set object
`set_type_idx_map`	wrapper around `set_type_idx_map` for a given set object
`to_struct`	convert object data *to* a struct
Private Methods‡	
`add_named_set`	wrapper around `add` for a given set object
`init_set_types`	initializes the data structures for each set type
`valid_named_set_type`	returns label or set object for given named set type if valid, empty otherwise

* This value is initialized automatically by the `def_set_types` method of the subclass.
† For all, or alternatively, only for a named (and possibly indexed) subset.
‡ For internal use only.

Table C-12: Matpower Index Manager (`mp_idx_manager`) Object Structure

name	description
`obj`	
`.set_types`	struct whose fields define the valid set types
`.var`	`mp.set_manager` object with data for `'var'` set type, e.g. variable sets that make up the full variable variable x
`.idx`	
`.i1`	starting index within x
`.iN`	ending index within x
`.N`	number of elements in this variable set
`.N`	total number of elements in x
`.NS`	number of variable sets or named blocks
`.data`	additional set-type-specific data for each block†
`.order`	struct array of names/indices for variable blocks in the order they appear in x
`.name`	name of the block, e.g. `z`
`.idx`	indices for name, $\{$`2,3`$\}$ $\rightarrow$ `z(2,3)`
`.<other-set-types>`	other `mp.set_manager` objects, with structure identical to `var`
`.userdata`	struct for storing arbitrary user-defined data

† For the `'var'` set type in `opt_model`, this is a struct with fields `v0`, `vl`, `vu`, and `vt` for storing initial value, lower and upper bounds, and variable type. For other set types

C.10 Reference

C.10.1 Properties

The properties in `opt_model` consist of those inherited from the base class, plus one corresponding to each set type.

Table C-13: `opt_model` Properties

name	description
`set_types`†	struct whose fields define the valid set types*
`prob_type`	used to cache return value of `problem_type` method
`var`‡	data for `'var'` set type, variables
`lin`‡	data for `'lin'` set type, linear constraints
`qcn`‡	data for `'qcn'` set type, quadratic constraints
`nle`‡	data for `'nle'` set type, nonlinear equality constraints
`nli`‡	data for `'nli'` set type, nonlinear inequality constraints
`qdc`‡	data for `'qdc'` set type, quadratic costs
`nlc`‡	data for `'nlc'` set type, general nonlinear costs
`userdata`†	struct for storing arbitrary user-defined data

* This value is initialized automatically by the `def_set_types` method of the subclass.
† Inherited from MATPOWER Index Manager base class, `mp_idx_manager`.
‡ See `var` field in Table C-12 for details of the structure of this field. The only difference between set types is the structure of the `data` sub-field.

C.10.2 Methods

Table C-14: `opt_model` Methods

name	description
Public Methods	
`add_lin_constraint`	add linear constraint set, see Section C.2.1
`add_nln_constraint`	add general nonlinear constraint set, see Section C.2.2
`add_nln_cost`	add general nonlinear cost set, see Section C.3.2
`add_quad_cost`	add quadratic cost set, see Section C.3.1
`add_var`	add variable set, see Section C.1
`display`	displays variable, constraint and cost sets, see Section C.8.1
`display_soln`	displays model solution, see Section C.8.1
`eval_lin_constraint`	computes linear constraint values, see Section C.5.3
`eval_nln_constraint`	builds full set of nonlinear equality or inequality constraints and their gradients, see Section C.5.3
`eval_nln_constraint_hess`	builds Hessian for full set of nonlinear equality or inequality constraints, see Section C.5.3
`eval_nln_cost`	evaluates nonlinear cost function and its derivatives‡, see Section C.5.4
`eval_quad_cost`	evaluates quadratic cost function and its derivatives‡, see Section C.5.4
`get_soln`	returns named/indexed results for solved model
`has_parsed_soln`	returns 1 if model has a parsed solution available, 0 otherwise
`is_mixed_integer`	returns 1 if any of the variables are binary or integer, 0 otherwise
`is_solved`	returns 1 if model has been solved, 0 otherwise
`params_lin_constraint`	assembles and returns parameters for linear constraints‡
`params_nln_constraint`	assembles and returns parameters for nonlinear constraints‡
`params_nln_cost`	assembles and returns parameters for general nonlinear costs‡
`params_quad_cost`	assembles and returns parameters for quadratic costs‡
`params_var`	assembles and returns inital values, bounds, types for variables‡
`parse_soln`	returns struct of all named solution vectors and shadow prices
`problem_type`	type of mathematical program represented by current model
`solve`	solves the model, see Section C.4
`varsets_cell2struct`	converts variable subset `varsets` from cell array to struct array
`varsets_idx`	returns vector of indices into x corresponding to `varsets`
`varsets_len`	returns number of elements in sub-vector specified by `varsets`
`varsets_x`	returns cell array of sub-vectors of x specified by `varsets`
Inherited Public Methods†	
`copy`, `describe_idx`, `display_set`, `get`, `get_idx`, `get_userdata`, `getN`, `init_indexed_name, set_type_idx_map`	
*Private Methods**	
`def_set_types`	initializes the `set_types` property
`init_set_types`§	initializes the data structures for each set type
`valid_named_set_type`†	returns label for given named set type if valid, empty otherwise

* For internal use only.
† Inherited from MATPOWER Index Manager base class, `mp_idx_manager`, see Table C-11.
‡ For all, or alternatively, only for a named (and possibly indexed) subset.
§ Overrides and augments method inherited from MATPOWER Index Manager base class, `mp_idx_manager`.

Appendix D Release History

The full release history can be found in `CHANGES.md` or online at `https://github.com/MATPOWER/mp-opt-model/blob/master/CHANGES.md`.

D.1 Version 0.7 – Jun 20, 2019

This release history begins with the code that was part of the MATPOWER 7.0 release.

D.2 Version 0.8 – Apr 29, 2020 *(not released publicly)*

This version consists of functionality moved directly from MATPOWER.[52] There is no User's Manual yet.

New Features

- New unified interface `nlps_master()` for nonlinear programming solvers MIPS, `fmincon`, IPOPT and Artelys Knitro.
- New functions:
  - `mpopt2nlpopt()` creates or modifies an options struct for `nlps_master()` from a MATPOWER options struct.
  - `nlps_fmincon()` provides implementation of unified nonlinear programming solver interface for `fmincon`.
  - `nlps_ipopt()` provides implementation of unified nonlinear programming solver interface interface for IPOPT.
  - `nlps_knitro()` provides implementation of unified nonlinear programming solver interface interface for IPOPT.
  - `nlps_master()` provides a single wrapper function for calling any of MP-Opt-Model's nonlinear programming solvers.

Other Improvements

- Significant performance improvement for some problems when constructing sparse matrices for linear constraints or quadratic costs. *Thanks to Daniel Muldrew.*

[52]From the current `master` branch in the MATPOWER GitHub repository at the time.

- Significant performance improvement for CPLEX on small problems by eliminating call to `cplexoptimset()`, which was a huge bottleneck.
- Add four new methods to `opt_model` class:
  - `copy()` – works around issues with inheritance in constructors that was preventing copy constructor from working in Octave 5.2 and earlier (see also https://savannah.gnu.org/bugs/?52614)
  - `is_mixed_integer()` – returns true if the model includes any binary or integer variables
  - `problem_type()` – returns one of the following strings, based on the characteristics of the variables, costs and constraints in the model:
    - `'LP'` – linear program
    - `'QP'` – quadratic program
    - `'NLP'` – nonlinear program
    - `'MILP'` – mixed-integer linear program
    - `'MIQP'` – mixed-integer quadratic program
    - `'MINLP'` – mixed-integer nonlinear program
  - `solve()` - solves the model using `qps_master()`, `miqps_master()`, or `nlps_master()`, depending on the problem type (`'MINLP'` problems are not yet implemented)

Bugs Fixed

- Artelys Knitro 12.1 compatibility fix.
- Fix CPLEX 12.10 compatibility issue #90.
- Fix issue with missing objective function value from `miqps_mosek()` and `qps_mosek()` when return status is "Stalled at or near optimal solution."
- Fix bug orginally in `ktropf_solver()` (code now moved to `nlps_knitro()`) where Artelys Knitro was still using `fmincon` options.

Incompatible Changes

- Modify order of default output arguments of `opt_model.get_idx()` (again), removing the one related to legacy costs.

- MP-Opt-Model has renamed the following functions and modified the order of their input args so that the MP-Opt-Model object appears first. Ideally, these would be defined as methods of the `opt_model` class, but Octave 4.2 and earlier is not able to find them via a function handle (as used in the `solve()` method) if they are inherited by a subclass.
  - `opf_consfcn()` → `nlp_consfcn()`
  - `opf_costfcn()` → `nlp_costfcn()`
  - `opf_hessfcn()` → `nlp_hessfcn()`

D.3 Version 1.0 – released May 8, 2020

This is the first public release of MP-Opt-Model as its own package. The MP-Opt-Model 1.0 User's Manual is available online.[53]

New Documentation

- Add MP-Opt-Model User's Manual with LaTeX source code included in `docs/src`.

Other Improvements

- Refactor `opt_model` class to inherit from new abstract base class `mp_idx_manager` which can be used to manage the indexing of other sets of parameters, etc. in other contexts.

D.4 Version 2.0 – released Jul 8, 2020

The MP-Opt-Model 2.0 User's Manual is available online.[54]

New Features

- Add new `'fsolve'` tag to `have_fcn()` to check for availability of `fsolve()` function.

- Add `nleqs_master()` function as unified interface for solving nonlinear equations, including implementations for `fsolve` and Newton's method in functions `nleqs_fsolve()` and `nleqs_newton()`, respectively.

[53] https://matpower.org/docs/MP-Opt-Model-manual-1.0.pdf
[54] https://matpower.org/docs/MP-Opt-Model-manual-2.0.pdf

- Add support for nonlinear equations (NLEQ) to `opt_model`. For problems with only nonlinear equality constraints and no costs, the `problem_type()` method returns `'NLEQ'` and the `solve()` method calls `nleqs_master()` to solve the problem.
- New functions:
  - `mpopt2nleqopt()` creates or modifies an options struct for `nleqs_master()` from a MATPOWER options struct.
  - `nleqs_fsolve()` provides implementation of unified nonlinear equation solver interface for `fsolve`.
  - `nleqs_master()` provides a single wrapper function for calling any of MP-Opt-Model's nonlinear equation solvers.
  - `nleqs_newton()` provides implementation of Newton's method solver with a unified nonlinear equation solver interface.
  - `opt_model.params_nln_constraint()` method returns parameters for a named (and optionally indexed) set of nonlinear constraints.
  - `opt_model.params_nln_cost()` method returns parameters for a named (and optionally indexed) set of general nonlinear costs.

Other Changes

- Add to `eval_nln_constraint()` method the ability to compute constraints for a single named set.
- Skip evaluation of gradient if `eval_nln_constraint()` is called with a single output argument.
- Remove redundant MIPS tests from `test_mp_opt_model.m`.
- Add tests for solving LP/QP, MILP/MIQP, NLP and NLEQ problems via `opt_model.solve()`.
- Add Table 6.1 of valid `have_fcn()` input tags to User's Manual.

D.5 Version 2.1 – released Aug 25, 2020

The MP-Opt-Model 2.1 User's Manual is available online.[55]

New Features

- Fast-decoupled Newton's and Gauss-Seidel solvers for nonlinear equations.
- New linear equation (`'LEQ'`) problem type for models with equal number of variables and linear equality constraints, no costs, and no inequality or nonlinear equality constraints. Solved via `mplinsolve()`.
- The `solve()` method of `opt_model` can now automatically handle mixed systems of equations, with both linear and nonlinear equality constraints.
- New core nonlinear equation solver function with arbitrary, user-defined update function, used to implement Gauss-Seidel and Newton solvers.
- New functions:
  - `nleqs_fd_newton()` solves a nonlinear set of equations via a fast-decoupled Newton's method.
  - `nleqs_gauss_seidel()` solves a nonlinear set of equations via a Gauss-Seidel method.
  - `nleqs_core()` implements core nonlinear equation solver with arbitrary update function.

Incompatible Changes

- In `output` return value from `nleqs_newton()`, changed the `normF` field of `output.hist` to `normf`, for consistency in using lowercase `f` everywhere.

[55] https://matpower.org/docs/MP-Opt-Model-manual-2.1.pdf

D.6 Version 3.0 – released Oct 8, 2020

The MP-Opt-Model 3.0 User's Manual is available online.[56]

New Features

- Support for OSQP solver for LP and QP problems (`https://osqp.org`).
- Support for modifying parameters of an existing MP-Opt-Model object.
- Support for extracting specific named/indexed variables, costs, constraint values and shadow prices, etc. from a solved MP-Opt-Model object.
- Results of the `solve()` method saved to the `soln` field of the MP-Opt-Model object.
- Allow `v0`, `vl`, and `vu` inputs to `opt_model.add_var()` method, and `l` and `u` inputs to `opt_model.add_lin_constraint()` to be scalars that get expanded automatically to the appropriate vector dimension.
- New functions:
  - `opt_model.set_params()` method modifies parameters for a given named set of existing variables, costs, or constraints of an MP-Opt-Model object.
  - `opt_model.get_soln()` method extracts solved results for a given named set of variables, constraints or costs.
  - `opt_model.parse_soln()` method returns a complete set of solution vector and shadow price values for a solved model.
  - `opt_model.eval_lin_constraint()` method computes the constraint values for the full set or an individual named subset of linear constraints.
  - `qps_osqp()` provides standardized interface for using OSQP to solve LP/QP problems
  - `osqp_options()` initializes options for OSQP solver
  - `osqpver()` returns/displays version information for OSQP
  - ...plus 29 individual feature detection functions for `have_feature()`, see Table A-11 for details.

[56] `https://matpower.org/docs/MP-Opt-Model-manual-3.0.pdf`

Bugs Fixed

- Starting point supplied to `solve()` via `opt.x0` is no longer ignored for nonlinear equations.
- Calling `params_var()` method with empty `idx` no longer results in fatal error.
- For `opt_model`, incorrect evaluation of constant term has been fixed for vector valued quadratic costs with constant term supplied as a vector.

Other Changes

- Simplified logic to determine whether a quadratic cost for an MP-Opt-Model object is vector vs. scalar valued. If the quadratic coefficient is supplied as a matrix, the cost is scalar varied, otherwise it is vector valued.
- Deprecated `have_fcn()` and made it a simple wrapper around the new modular and extensible `have_feature()`, which has now been moved to MP-Test.[57]

D.7 Version 4.0 – released Oct 18, 2021

The MP-Opt-Model 4.0 User's Manual is available online.[58]

New Features

- Support for new class of problems – parameterized nonlinear equations (PNE). Either create a model with only equality constraints (no inequalities or costs) and with number of variables equal to 1 more than number of constraints, *or* call `pnes_master()` directly. See Section 4.6 of User's Manual for details.
  - Predictor/corrector numerical continuation method for tracing solution curves for PNE problems.
  - Plotting of solution curves.
  - User-defined event functions and callback functions.
  - Warm-start capabilities.

  Thanks to Shrirang Abhyankar and Alexander Flueck for contributions to this feature, which is based on the continuation power flow code in MATPOWER *7.1.*

[57] MP-Test is available at `https://github.com/MATPOWER/mptest`.
[58] `https://matpower.org/docs/MP-Opt-Model-manual-4.0.pdf`

- Optional threshold for detecting failure of LEQ solve, by setting the `leq_opt.thresh` option. If the absolute value of any element of the solution vector exceeds the threshold, `exitflag` is set to 0, indicating failure.
- New functions:
  - `pnes_master()` provides unified interface for parameterized nonlinear equation (PNE) solvers.
  - `pne_callback_default()` collects PNE results and optionally plots solution curve.
  - `pne_callback_nose()` handles event signaling a nose point or limit has been reached.
  - `pne_callback_target_lam()` handles event signaling a target value of parameter λ has been reached.
  - `pne_detect_events()` detects events from event function values.
  - `pne_detected_event()` returns detected event details for events with a particular name.
  - `pne_event_nose()` detects the limit or nose point.
  - `pne_event_target_lam()` detects a target λ value.
  - `pne_pfcn_arc_length()` implements arc length parameterization.
  - `pne_pfcn_natural()` implements natural parameterization.
  - `pne_pfcn_pseudo_arc_length()` implements pseudo arc length parameterization.
  - `pne_register_callbacks()` registers callback functions.
  - `pne_register_events()` registers event functions.
  - `mp_idx_manager.set_type_idx_map()` method returns information about mapping of indices for a given set type back to the corresponding named (and possibly indexed) sets.
  - `mpopt2pneopt()` creates or modifies an options struct for `pnes_master()` from a MATPOWER options struct.

Bugs Fixed

- Calling the `problem_type()` or `is_mixed_integer()` method on an empty model no longer causes a fatal error.

Other Changes

- Labels from the set_types property are now used as headers for opt_model.display() to simplify things facilitate use by subclasses.
- Refactored describe_idx into a new method, set_type_idx_map, that returns in information in a programmatically usable form, and an updated describe_idx that calls the new method, then formats the results in the expected char array(s).

D.8 Version 4.1 – released Dec 13, 2022

The MP-Opt-Model 4.1 User's Manual is available online.[59]

New Features

- Add support to qps_master() for calling qps_<my_solver>() by setting opt.alg to '<MY_SOLVER>' to allow for handling custom LP/QP solvers.

Other Changes

- Update for compatibility with Artelys Knitro 13.1 and later.
- Add elapsed time in seconds to results of the solve() method of opt_model, returned in om.soln.output.et.
- Add runtime field to output argument of qps_glpk() and qps_mosek().

D.9 Version 4.2 – released May 10, 2024

The MP-Opt-Model 4.2 User's Manual is available online.[60]

New Features

- Option for opt_model.add_lin_constraint() to provide/store the transpose of the A matrix instead of the original. This can potentially save significant memory for sparse matrices with many more columns than rows. E.g. storage constraints in MOST for 8760 hour planning horizon.

[59] https://matpower.org/docs/MP-Opt-Model-manual-4.1.pdf

[60] https://matpower.org/docs/MP-Opt-Model-manual-4.2.pdf

- Add support to `nlps_master()` for calling `nlps_<my_solver>()` by setting `opt.alg` to `'<MY_SOLVER>'` to allow for handling custom NLP solvers.
- Add support to `miqps_master()` for calling `miqps_<my_solver>()` by setting `opt.alg` to `'<MY_SOLVER>'` to allow for handling custom MILP/MIQP solvers.
- Add to the `parse_soln()` method of `opt_model` an optional `stash` input argument that, if present and true, causes the parsed solution to be stored back in the object, as the `solve()` method was already doing when `opt.parse_soln` is true.
- New Sphinx-based Reference documentation.[61]
- New functions:
  - `convert_lin_constraint()` converts linear constraints from a single set of doubly-bounded inequality constraints to separate sets of equality and upper-bounded inequality constraints.
  - `convert_lin_constraint_multipliers()` converts multipliers on linear constraints from separate sets for equality and upper-bounded inequality constraints to those for doubly-bounded inequality constraints.
- New `opt_model` methods:
  - `is_solved()` indicates whether the model has been solved.
  - `has_parsed_soln()` indicates whether a parsed solution is available in the model.
  - `display_soln()` display the results of a solved model, including values, bounds and shadow prices for variables and linear constraints, values and shadow prices for nonlinear constraints, and individual cost components.

Bugs Fixed

- Clear cached parameters after updating linear constraints or quadratic costs via `opt_model.set_params()`.
- In `miqps_mosek()` the lower and upper bounds of binary variables got overwritten with 0 and 1, respectively, effectively relaxing any potentially tighter bounds provided as input.

[61] https://matpower.org/doc/mpom/

- Fix false positive in `have_feature_fsolve` in case where the file is present, but without a valid license.

Other Changes

- Update for compatibility with MATLAB R2023a (Optimization Toolbox 9.5) and later, which removed `x0` as a valid input to `linprog`.

- Update `have_feature_ipopt()` to recognize IPOPT MEX installations from Enrico Bertolazzi's mexIPOPT[62], which include MEX files that have been renamed to architecture-specific names along with an `ipopt.m` wrapper function to call the appropriate one. *Thanks to Carlos Murillo-Sánchez.*
  Note: *While MP-Opt-Model no longer requires this, my recommendation is still to simply rename the MEX file to* `ipopt.<mexext>`*, with the appropriate architecture-specific extension, and delete the unnecessary* `ipopt.m` *entirely.*

- Always skip price computation stage in `miqps_<solver>()` functions for pure (as opposed to mixed) integer problems.

- Add caching of aggregate output parameters in `opt_model.params_var()`.

D.10 Version 5.0 – released Jul 12, 2025

The MP-Opt-Model 5.0 User's Manual is available online.[63]

New Features

- Support for quadratic constraints in `opt_model` and quadratically-constrained quadratic programming (QCQP) problems, including functions `qcqps_master()`, `qcqps_gurobi()`, `qcqps_knitro()`, `qcqps_nlps()`, and more. *Thanks to Wilson González Vanegas.*

- Support for the open-source HiGHS[64] solver for LP, QP and MILP problems, including functions `miqps_highs()`, `qps_highs()`, `have_feature_highs()`, `highsver()`, and `highs_options()` based on the HiGHSMEX[65] interface.

[62] mexIPOPT is available at https://github.com/ebertolazzi/mexIPOPT.
[63] https://matpower.org/docs/MP-Opt-Model-manual-5.0.pdf
[64] https://highs.dev
[65] https://github.com/savyasachi/HiGHSMEX

- New `relax_integer` option for `opt_model.solve()`. Set to true to easily solve the LP or QP relaxation of a mixed integer LP or QP.

- Support for Artelys Knitro solver for LP and QP problems, including functions `qps_knitro()`, `knitrover()`, and `artelys_knitro_options()`. *Thanks to Wilson González Vanegas.*

- Support for Artelys Knitro 15.x, which required changes to the prior options handling.

- Support for conversion between objects and structs to facilitate workarounds for Octave's inability to save and load classdef objects.

- New classes:

  - `mp.opt_model` replaces legacy `opt_model` and `mp_idx_manager` classes with a new modeling API. *The legacy classes are retained for backward compatibility.*

  - `mp.set_manager` encapsulates `mp_idx_manager` functionality into an individual field object representing a specific set type, rather than in the container class.

  - `mp.set_manager_opt_model` is a subclass of `mp.set_manager` that handles common functionality, e.g. related to handling solution data, for all of the set manager subclasses used by `opt_model`.

  - `mp.sm_lin_constraint` - set manager class for linear constraints

  - `mp.sm_quad_constraint` - set manager class for quadratic constraints

  - `mp.sm_nln_constraint` - set manager class for nonlinear constraints

  - `mp.sm_nln_cost` - set manager class for general nonlinear costs

  - `mp.sm_quad_cost` - set manager class for quadratic costs

  - `mp.sm_quad_cost_legacy` - backward compatible set manager class for quadratic costs

  - `mp.sm_variable` - set manager class for variables

- New functions:

  - `artelys_knitro_options()` sets options for Artelys Knitro.

  - `convert_constraint_multipliers()` replaces `convert_lin_constraint_multipliers()`.

- `convert_quad_constraint()` converts bounded quadratic constraints to equality/inequality pairs.
- `have_feature_highs()` feature detection function for HiGHS solver.
- `highsver()` displays version of installed HiGHS.
- `highs_options()` sets options for HiGHS.
- `knitrover()` displays version of installed Artelys Knitro.
- `miqps_highs()` provides standardized interface for using HiGHS to solve MILP problems.
- `mpopt2qcqpopt()` creates/modifies `qcqps_master` options struct from MATPOWER options struct.
- `mp.struct2object()` converts a struct back to the object from which it was created.
- `qcqps_gurobi()` provides standardized interface for using Gurobi to solve QCQP problems.
- `qcqps_knitro()` provides standardized interface for using Artelys Knitro to solve QCQP problems.
- `qcqps_nlps()` provides standardized interface for using `nlps_master` to solve QCQP problems via `fmincon`, IPOPT, Artelys Knitro, or MIPS.
- `qcqps_master()` provides a single wrapper function for calling any of MP-Opt-Model's QCQP solvers.
- `qps_highs()` provides standardized interface for using HiGHS to solve LP/QP problems.
- `qps_knitro()` provides standardized interface for using Artelys Knitro to solve LP/QP problems.

New Documentation

Two live scripts illustrate the use of new features.

- `milp_example1.mlx` (in `examples`) illustrates the use of MP-Opt-Model and the new `mp.opt_model` class to build and solve an optimization (MILP) model.
- `qcqp_example1.mlx` (in `examples`) illustrates the new quadratic constraint features and two methods of building and solving a quadratically-constrained quadratic programming (QCQP) model.

Bugs Fixed

- Using `miqps_master()` with `'DEFAULT'` solver to solve an LP/QP problem without a MILP/MIQP solver available incorrectly threw a fatal error stating there was no solver available.

Other Changes

- Major refactor of `mp_idx_manager` to use new `mp.set_manager` class.
- Major refactor of `opt_model` to use new `mp.set_manager_opt_model` subclasses:
  - `mp.sm_lin_constraint` - set manager class for linear constraints
  - `mp.sm_quad_constraint` - set manager class for quadratic constraints
  - `mp.sm_nln_constraint` - set manager class for nonlinear constraints
  - `mp.sm_nln_cost` - set manager class for general nonlinear costs
  - `mp.sm_quad_cost_legacy` - backward compatible set manager class for quadratic costs
  - `mp.sm_variable` - set manager class for variables
- Deprecate the following `opt_model` methods in favor of methods of the individual `mp.set_manager` objects contained by the `opt_model` object:
  - `add_named_set()` – use `mp.set_manager.add()`
  - `describe_idx()` – use `mp.set_manager.describe_idx()`
  - `getN()` – use `mp.set_manager.get_N()`
  - `init_indexed_name()` – use `mp.set_manager.init_indexed_name()`
  - `set_type_idx_map()` – use `mp.set_manager.set_type_idx_map()`
  - `add_lin_constraint()` – use `mp.sm_lin_constraint.add()`
  - `add_nln_constraint()` – use `mp.sm_nln_constraint.add()`
  - `add_nln_cost()` – use `mp.sm_nln_cost.add()`
  - `add_quad_cost()` – use `mp.sm_quad_cost.add()`
  - `add_var()` – use `mp.sm_variable.add()`
  - `eval_lin_constraint()` – use `mp.sm_lin_constraint.eval()`
  - `eval_nln_constraint()` – use `mp.sm_nln_constraint.eval()`

- eval_nln_constraint_hess() – use mp.sm_nln_constraint.eval_hess()
- eval_nln_cost() – use mp.sm_nln_cost.eval()
- eval_quad_cost() – use mp.sm_quad_cost.eval()
- init_indexed_name() – use mp.set_manager.init_indexed_name()
- params_lin_constraint() – use mp.sm_lin_constraint.params()
- params_nln_constraint() – use mp.sm_nln_constraint.params()
- params_nln_cost() – use mp.sm_nln_cost.params()
- params_quad_cost() – use mp.sm_quad_cost.params()
- params_var() – use mp.sm_variable.params()
- set_params() – use mp.set_manager.set_params()
- varsets_cell2struct() – use mp.sm_variable.varsets_cell2struct()
- varsets_idx() – use mp.sm_variable.varsets_idx()
- varsets_len() – use mp.sm_variable.varsets_len()
- varsets_x() – use mp.sm_variable.varsets_x()

- Update `mosek_options()` for MOSEK 11.x compatibility.
- Update `miqps_<solver>()` functions to avoid changing MIP solution values in price computation stage. It was rounding integer variables, potentionally causing a small discrepancy between the objective value reported by the solver and the value obtained by computing directly from the returned solution x.
- Deprecate the `convert_lin_constraint_multipliers()` in favor of `convert_constraint_multipliers()`.

Incompatible Changes

- Parsed solution information was moved from the `soln` property of the `opt_model` object to the `soln` property of the individual child `mp.set_manager_opt_model` objects. Currently it is still available at the original location, but this is now deprecated.
- The `knitro_opts` field of the `opt` input to `nlps_master()` and `nlps_knitro()` and the `solve()` method of `opt_model` has been redesigned. It is now a raw Artelys Knitro options struct, so the `opts`, `tol_x` and `tol_f` fields are no longer valid. For `tol_x` and `tol_f`, use `xtol` and `ftol`, and the contents of `opts` should be placed directly in the top level of the `knitro_opts` field.

- Remove support for older versions of Artelys Knitro, including all references to `ktrlink` for pre-v9 versions. Currently supports Artelys Knitro version 13.1 and later.

References

[1] R. D. Zimmerman, C. E. Murillo-Sánchez, and R. J. Thomas, "Matpower: Steady-State Operations, Planning and Analysis Tools for Power Systems Research and Education," *Power Systems, IEEE Transactions on*, vol. 26, no. 1, pp. 12–19, Feb. 2011. doi: 10.1109/TPWRS.2010.2051168 1.1

[2] R. D. Zimmerman, C. E. Murillo-Sánchez (2025). Matpower [Software]. Available: `https://matpower.org` doi: 10.5281/zenodo.3236535 1.1

[3] John W. Eaton, David Bateman, Søren Hauberg, Rik Wehbring (2024). *GNU Octave version 9.2.0 manual: a high-level interactive language for numerical computations.* Available: `https://docs.octave.org/v9.2.0/`. 1, 4

[4] The BSD 3-Clause License. [Online]. Available: `https://opensource.org/licenses/BSD-3-Clause`. 1.2

[5] R. D. Zimmerman. MP-Opt-Model User's Manual. 2025. [Online]. Available: `https://matpower.org/docs/MP-Opt-Model-manual.pdf` doi: 10.5281/zenodo.3818002 1.3

[6] H. Wang, C. E. Murillo-Sánchez, R. D. Zimmerman, and R. J. Thomas, "On Computational Issues of Market-Based Optimal Power Flow," *Power Systems, IEEE Transactions on*, vol. 22, no. 3, pp. 1185–1193, August 2007. doi: 10.1109/TPWRS.2007.901301 2.1

[7] R. D. Zimmerman, H. Wang. **M**atpower **I**nterior **P**oint **S**olver (MIPS) User's Manual. 2025. [Online]. Available: `https://matpower.org/docs/MIPS-manual.pdf` doi: 10.5281/zenodo.3236506 2.1

[8] E. L. Allgower, K. Georg, *Introduction to Numerical Continuation Methods*, Society for Industrial and Applied Mathematics, 2003. doi: 10.1137/1.9780898719154 4.6

[9] H.-D. Chiang, A. Flueck, K. Shah, and N. Balu, "CPFLOW: A Practical Tool for Tracing Power System Steady-State Stationary Behavior Due to Load and Generation Variations," *Power Systems, IEEE Transactions on*, vol. 10, no. 2, pp. 623–634, May 1995. 4.6.1

[10] S. H. Li and H. D. Chiang, "Nonlinear Predictors and Hybrid Corrector for Fast Continuation Power Flow", *Generation, Transmission Distribution, IET*, 2(3):341–354, 2008. 4.6.1

[11] H. Mori and S. Yamada, "Continuation Power Flow with the Nonlinear Predictor of the Lagrange's Polynomial Interpolation Formula, " In *Transmission and Distribution Conference and Exhibition 2002: Asia Pacific. IEEE/PES*, vol. 2, pp. 1133–1138, Oct 6–10, 2002. 4.6.1

[12] BPMPD_MEX. [Online]. Available: `http://www.pserc.cornell.edu/bpmpd/`. B.1

[13] C. Mészáros, *The Efficient Implementation of Interior Point Methods for Linear Programming and their Applications*, Ph.D. thesis, Eötvös Loránd University of Sciences, Budapest, Hungary, 1996. B.1

[14] COIN-OR Linear Programming (CLP) Solver. [Online]. Available: `https://github.com/coin-or/Clp`. B.2

[15] J. Currie and D. I. Wilson,"OPTI: Lowering the Barrier Between Open Source Optimizers and the Industrial MATLAB User," *Foundations of Computer-Aided Process Operations*, Georgia, USA, 2012. 6.9.4, B.2, B.4, B.7

[16] GLPK. [Online]. Available: `https://www.gnu.org/software/glpk/`. B.4

[17] Gurobi Optimization, Inc., "Gurobi Optimizer Reference Manual," 2016. [Online]. Available: `https://www.gurobi.com/`. B.5

[18] A. Wächter and L. T. Biegler, "On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming," *Mathematical Programming*, 106(1):25—57, 2006. B.7

[19] Q. Huangfu and J. A. J. Hall, "Parallelizing the dual revised simplex method", *Mathematical Programming Computation*, 10 (1), 119–142, 2018. doi: 10.1007/s12532-017-0130-5 B.6

[20] O. Shenk and K. Gärtner, "Solving unsymmetric sparse systems of linear equations with PARDISO," *Journal of Future Generation Computer Systems*, 20(3):475–487, 2004. B.7

[21] A. Kuzmin, M. Luisier and O. Shenk, "Fast methods for computing selected elements of the Greens function in massively parallel nanoelectronic device simulations," in F. Wolf, B. Mohr and D. Mey, editors, *Euro-Par 2013 Parallel Processing*, Vol. 8097, *Lecture Notes in Computer Science*, pp. 533–544, Springer Berlin Heidelberg, 2013. B.7

[22] R. H. Byrd, J. Nocedal, and R. A. Waltz, "KNITRO: An Integrated Package for Nonlinear Optimization", *Large-Scale Nonlinear Optimization*, G. di Pillo and M. Roma, eds, pp. 35–59 (2006), Springer-Verlag. doi: 10.1007/0-387-30065-1_4 B.8

[23] *Optimization Toolbox*, The MathWorks, Inc. [Online]. Available: https://www.mathworks.com/products/optimization/. B.10

[24] *Optimization Toolbox Users's Guide*, The MathWorks, Inc., 2024. [Online]. Available: https://www.mathworks.com/help/pdf_doc/optim/optim.pdf. B.10

[25] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd, S., "OSQP: An Operator Splitting Solver for Quadratic Programs", *Mathematical Programming Computation*, 2020. doi: 10.1007/s12532-020-00179-2 B.11