MP-Opt-Model User's Manual

Version 2.1

Ray D. Zimmerman

August 25, 2020

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, 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 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.

```
Copyright (c) 2004-2020, Power Systems Engineering Research Center
(PSERC) and individual contributors (see AUTHORS file for details).
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
```

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 2.1, use:

> R. D. Zimmerman. MP-Opt-Model User's Manual, Verision 2.1. 2020. [Online]. Available: `https://matpower.org/docs/MP-Opt-Model-manual-2.1.pdf`
> doi: 10.5281/zenodo.4001106

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 2.1 you will need:

- MATLAB® version 8.6 (R2015b) or later[3], or
- GNU Octave version 4.2 or later[4]
- MIPS, **M**ATPOWER **I**nterior **P**oint **S**olver [6, 7][5]
- MP-Test, for running the MP-Opt-Model test suite.[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 them 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 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://www.gnu.org/software/octave/.

[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

Step 3: At the MATLAB 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_nested_struct_copy....ok
t_have_fcn..............ok
t_nleqs_master..........ok (30 of 150 skipped)
t_qps_master............ok (100 of 396 skipped)
t_miqps_master..........ok (68 of 288 skipped)
t_nlps_master...........ok
t_opt_model.............ok
t_om_solve_leqs.........ok
t_om_solve_nleqs........ok (36 of 170 skipped)
t_om_solve_qps..........ok (79 of 319 skipped)
t_om_solve_miqps........ok (12 of 72 skipped)
t_om_solve_nlps.........ok
All tests successful (2713 passed, 325 skipped of 3038)
Elapsed time 3.29 seconds.
```

2.3 Sample Usage

Suppose we have the following constrained 4-dimensional quadratic programming (QP) problem[10] with two 2-dimensional variables, y and z, and two constraints, one equality and the other inequality, along with lower bounds on all of the variables.

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

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

[10] Based on the one from `https://v8doc.sas.com/sashtml/iml/chap8/sect12.htm`.

subject to

$$
\begin{aligned}
A_1 \begin{bmatrix} y \\ z \end{bmatrix} &= b_1 && (2.2) \\
l_2 \leq & A_2 \begin{bmatrix} y \\ z \end{bmatrix} && (2.3) \\
y_{\min} \leq & y && (2.4) \\
z_{\min} \leq & z && (2.5)
\end{aligned}
$$

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];
zmin = [0; 0];

%% constraint data
A1 = [ 1 1 1 1 ];               b1 = 1;
A2 = [ 0.17 0.11 0.10 0.18 ];   l2 = 0.1;

%% quadratic cost coefficients
Q = [   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  ];
```

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

```
%%-----  METHOD 1  -----
%% build model
om = opt_model;
om.add_var('y', 2, y0, ymin);
om.add_var('z', 2, z0, zmin);
om.add_lin_constraint('lincon1', A1, b1, b1, {'y', 'z'});
om.add_lin_constraint('lincon2', A2, l2, [], {'y', 'z'});
om.add_quad_cost('cost', Q, [], [], {'y', 'z'});

%% solve model
[x, f, exitflag, output, lambda] = om.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; zmin];
x0 = [y0; z0];
A = [ A1; A2 ];
l = [ b1; l2 ];
u = [ b1; Inf ];

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

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

```
f = 1.09667   exitflag = 1

x =
   0.0000
   0.9333
   0.0667
   0.0000

lambda.lower (var bound shadow price) =
   2.2400
   0.0000
   0.0000
   1.7667

lambda.mu_l (constraint shadow price) =
   2.1933
   0.0000
```

Both approaches can be applied to each of the types of problems that MP-Opt-Model handles, namely, LP, QP, MILP, MIQP, NLP and 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`.

And second is 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.

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
- nonlinear programming problems (NLP)
- nonlinear equations (NLEQ)

The second part consists of an optimization model class designed to help the user construct an optimization problem by adding variables, constraints and costs, then solve the problem and extract the solution in terms of the individual sets of variables, constraints and 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 is 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.

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_ipopt, 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 (shown in Table 4-3) optional
`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), GLPK (LP only), BPMPD, MIPS `'BPMPD'` – BPMPD* `'CLP'` – CLP* `'CPLEX'` – CPLEX* `'GLPK'` – GLPK* *(LP only)* `'GUROBI'` – Gurobi* `'IPOPT'` – IPOPT* `'MIPS'` – MIPS, **M**ATPOWER **I**nterior **P**oint **S**olver, primal/dual interior point method `'MOSEK'` – MOSEK* `'OT'` – MATLAB Opt Toolbox, `quadprog`, `linprog`
`verbose`	1	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*
`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), GLPK (MILP only) `'CPLEX'` – CPLEX* `'GLPK'` – GLPK* *(LP only)* `'GUROBI'` – Gurobi* `'MOSEK'` – MOSEK* `'OT'` – MATLAB Opt Toolbox, `intlinprog`
`verbose`	1	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*
`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_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 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:

$$\min_x f(x) \tag{4.6}$$

subject to

$$g(x) = 0 \tag{4.7}$$
$$h(x) \le 0 \tag{4.8}$$
$$l \le Ax \le u \tag{4.9}$$
$$x_{\min} \le x \le x_{\max} \tag{4.10}$$

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,

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

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

name	description
`f_fcn`	Handle to a function that evaluates the objective function, its gradients and Hessian‡ for a given value of x. Calling syntax for this function: `[f, df, d2f] = f_fcn(x)`
`x0`	Starting value of optimization vector x.
`A`, `l`, `u`	Define the optional linear constraints $l \leq Ax \leq u$. Default values for the elements of `l` and `u` are `-Inf` and `Inf`, respectively.
`xmin`, `xmax`	Optional lower and upper bounds on the x variables, defaults are `-Inf` and `Inf`, respectively.
`gh_fcn`	Handle to function that evaluates the optional nonlinear constraints and their gradients for a given value of x. Calling syntax for this function is: `[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. The calling syntax for this function is: `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 structure with fields, all of which are also optional, described in Table 4-8.
`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`.

Table 4-7: 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 4-8: Options for `nlps_master`

name	default	description
`alg`	`'DEFAULT'`	determines which solver to use `'DEFAULT'` – automatic, current default is MIPS `'MIPS'` – MIPS `'FMINCON'` – MATLAB Opt Toolbox, `fmincon`* `'IPOPT'` – IPOPT* `'KNITRO'` – Artelys Knitro*
`verbose`	1	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.

The user-defined functions for evaluating the objective function, constraints and Hessian are identical to those required by MIPSj. That is, they 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

$$\mathcal{L}(x, \lambda, \mu, \sigma) = \sigma f(x) + \lambda^{\mathsf{T}} g(x) + \mu^{\mathsf{T}} h(x) \tag{4.11}$$

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.

4.3.1 NLP Example 1

The following code, included as nlps_master_ex1.m in <*MPOM*>lib/t, shows a simple example of using nlps_master to solve a 2-dimensional unconstrained optimization of Rosenbrock's "banana" function[14]

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2. \tag{4.12}$$

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.

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

```
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
```

4.3.2 NLP Example 2

The second example[15] solves the following 3-dimensional constrained optimization, printing the details of the solver's progress:

$$\min_x f(x) = -x_1 x_2 - x_2 x_3 \tag{4.13}$$

subject to

$$\begin{aligned} x_1^2 - x_2^2 + x_3^2 - 2 &\leq 0 \qquad (4.14) \\ x_1^2 + x_2^2 + x_3^2 - 10 &\leq 0. \qquad (4.15) \end{aligned}$$

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

First, create a function to evaluate the objective function and its gradients,[16]

```
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
```

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.

[16]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.

```
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 1.3.1, 20-Jun-2019
 (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  9.84879e-10  3.41587e-06  6.05875e-06
  8    -7.0710673 5.6761e-06           0  9.73527e-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.


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

4.4 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:

$$f(x) = 0 \tag{4.16}$$

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 4-9 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 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 4-9: Input Arguments for `nleqs_master`

name	description
`fcn`	Handle to a function that evaluates the function $f(x)$ and optionally its Jacobian $J(x)$ for a given value of x. Calling syntax for this function is: `f = fcn(x)`, or `[f, J] = fcn(x)` Whether `fcn` is required to return the Jacobian or not depends on the selected solver algorithm.
`x0`	Starting value of vector x.
`opt`†	Options structure with fields, all of which are also optional, described in Table 4-11.
`problem`	Alternative, single argument input struct with fields corresponding to arguments above.

† Optional.

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.

Table 4-10: 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.

4.4.1 NLEQ Example 1

The following code, included as `nleqs_master_ex1.m` in <*MPOM*>`lib/t`, shows a simple example of using `nleqs_master` to solve a 2-dimensional nonlinear function[17]

$$f(x) = \begin{bmatrix} x_1 + x_2 - 1 \\ -x_1^2 + x_2 + 5 \end{bmatrix} \tag{4.17}$$

First, create a function that will evaluate the function and its Jacobian 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
```

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

Table 4-11: 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`	1	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.

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.

```
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
```

4.4.2 NLEQ Example 2

The following code, included as `nleqs_master_ex2.m` in <*MPOM*>`lib/t`, shows another simple example of using `nleqs_master` to solve a 2-dimensional nonlinear function.[18] This example includes the update function required for Gauss-Seidel and the Jaco-

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

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

$$f(x) = \begin{bmatrix} x_1^2 + x_1 x_2 - 10 \\ x_2 + 3x_1 x_2^2 - 57 \end{bmatrix} \tag{4.18}$$

```
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
```

5 Optimization Model Class – `opt_model`

The `opt_model` class provides facilities for constructing an 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.

In this manual, and in the code, `om` is the name of the variable used by convention for the optimization 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 optimization variable 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.

5.1 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[19], `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

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

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 optimization vector is the $(n_u + n_v + n_w) \times 1$ vector

$$x = \begin{bmatrix} u \\ v \\ w \end{bmatrix}. \tag{5.1}$$

See Section 5.6 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 optimization 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 $x_2 \equiv \begin{bmatrix} u \\ w \end{bmatrix}$.

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 three types of constraints, doubly-bounded linear constraints, general nonlinear equality constraints, and general nonlinear inequality constraints.

5.2.1 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

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

where x here refers to either the full optimization 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.[20]

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

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

where x_1 and x_2 are as defined in Section 5.1.1 and x is the full optimization 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_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 5.6 for details on indexed named sets and the `idx_list` argument.

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

5.2.2 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

$$g(x) = 0, \text{ or} \tag{5.6}$$

$$g(x) \leq 0 \tag{5.7}$$

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 optimization 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,

$$g_1(x_1) \leq 0 \tag{5.8}$$
$$g_2(x_2) = 0 \tag{5.9}$$
$$g_3(x) \leq 0, \tag{5.10}$$

where x_1 and x_2 are as defined in Section 5.1.1 and x is the full optimization 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_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 5.6 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

```
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:

$$f(x) = \frac{1}{2} x^\mathsf{T} Q x + c^\mathsf{T} x + k \tag{5.11}$$

where x here refers to either the full optimization 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.[21]

Alternatively, `Q` can be an $n_x \times 1$ vector, in which case $f(x)$ and k are also $n_x \times 1$ vectors and the i-th element of $f(x)$ is given by

$$f_i(x) = \frac{1}{2} Q_i x_i^2 + c_i x_i + k_i. \tag{5.12}$$

If `Q` is empty, then $f(x)$ is also an $n_x \times 1$ vector, unless `k` is scalar and non-zero.

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

$$q_1(x_1) = \frac{1}{2} {x_1}^\mathsf{T} Q_1 x_1 + {c_1}^\mathsf{T} x_1 + k_1 \tag{5.13}$$

$$q_2(x_2) = \frac{1}{2} {x_2}^\mathsf{T} Q_2 x_2 + {c_2}^\mathsf{T} x_2 + k_2 \tag{5.14}$$

$$q_3(x) = \frac{1}{2} x^\mathsf{T} Q_3 x + {c_3}^\mathsf{T} x + k_3, \tag{5.15}$$

where x_1 and x_2 are as defined in Section 5.1.1 and x is the full optimization vector from (5.1). 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.

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 5.6 for details on indexed named sets and the `idx_list` argument.

[21] The `Q` matrix can be sparse.

5.3.2 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 $1 \times n_x$ 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 optimization 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 optimization 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_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 5.6 for details on indexed named sets and the `idx_list` argument.

5.4 Solving the Model

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

After all variables, constraints and costs have been added to the model, the 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`, or `nleqs_master`. Note that one of the equation solvers, `mplinsolve` or `nleqs_master` is chosen if the model has only equality constraints, with no costs and no inequality constraints.

For details on the return values and the input options struct `opt`, see the descriptions of the individual solver functions in Sections 4.1, 4.2, 4.3, and 4.4. 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 `opt.leq_opt.solver` and `opt.leq_opt.opt`, respectively.

5.5 Accessing the Model

5.5.1 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 5.1, then the contents of `vv` will be as shown in Table 5-1.

Table 5-1: 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 5-2. When called without input arguments, the indexing structs are simply returned in the order listed in the table.

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

Table 5-2: Example Indexing Data

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.

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 optimization variable x can be obtained as follows.

```
nx = om.getN('var');
nv = om.getN('var', 'v');
```

See Section 5.6 for details on indexed named sets and the `idx_list` argument.

describe_idx

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

Given a particular index (or set of indices) for the full set of variables or constraints of a particular type, the `describe_idx` method can be used to show which element of which particular named set the index corresponds to. This can be useful 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 optimization 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)'}
```

5.5.2 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 optimization 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 5.6 for details on indexed named sets and the `idx_list` argument.

5.5.3 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

$$l \leq Ax \leq u. \tag{5.16}$$

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 5.6 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_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 optimization 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 optimization 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)
```

5.5.4 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

$$f(x) = \frac{1}{2}x^{\mathsf{T}}Qx + c^{\mathsf{T}}x + k. \tag{5.17}$$

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

$$f_i(x) = \frac{1}{2}Q_i x_i^2 + c_i x_i + k_i, \tag{5.18}$$

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 5.6 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 optimization 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 5.6 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 optimization vector x, based on costs added by `add_nln_cost`.

Examples:

```
[f, df, d2f] = om.eval_quad_cost(x);
[f, df, d2f] = om.eval_quad_cost(x, 'nlncost2');
```

See Section 5.6 for details on indexed named sets and the `idx_list` argument.

5.6 Indexed Sets

A variable, constraint or cost set is typically identified simply by a `name`, but it is also possible to used 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.[22]

`init_indexed_name`

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

Examples:

```
[f, df, d2f] = om.init_indexed_name('var', 'y', {7});
[f, df, d2f] = 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
```

[22]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_`.[23]

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 5-1. 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}});
```

5.7 Miscellaneous Methods

5.7.1 Public Methods

`copy`

```
om2 = om.copy()
```

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

[23] 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.

`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()
```

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[24]

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

`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 optimization 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 optimization sub-vector specified by `varsets`.

`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 optimization 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.

5.7.2 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 5-2.

`init_set_types`

```
om.init_set_types()
```

Initializes the base data structures for each set type.

5.8 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 5-3.

The MATPOWER Index Manager base class initializes and manages the data that is common across all set types. Table 5-4 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 5-3: 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	identifies indices of a given set type E.g. variable 361 corresponds to w(68)
display_set	displays indexing for a particular set type
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	initializes dimensions for a particular indexed named set
Private Methods‡	
add_named_set	adds indexing information for new instance of a given set type
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

* This value is initialized automatically by the def_set_types method of the sub-class.
† For all, or alternatively, only for a named (and possibly indexed) subset.
‡ For internal use only.

Table 5-4: MATPOWER Index Manager (mp_idx_manager) Object Structure

name	description
obj	
.set_types	struct whose fields define the valid set types
.var	data for 'var' set type, e.g. variable sets that make up the full optimization 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} → z(2,3)
.<other-set-types>	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

5.9 Reference

5.9.1 Properties

The properties in `opt_model` consist of those inherited from the base class, plus one corresponding to each set type.

Table 5-5: `opt_model` Properties

name	description
`set_types`†	struct whose fields define the valid set types*
`var`‡	data for `'var'` set type, variables
`lin`‡	data for `'lin'` set type, linear 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 5-4 for details of the structure of this field. The only difference between set types is the structure of the `data` sub-field.

5.9.2 Methods

Table 5-6: `opt_model` Methods

name	description
Public Methods	
`add_lin_constraint`	add linear constraint set, see Section 5.2.1
`add_nln_constraint`	add general nonlinear constraint set, see Section 5.2.2
`add_nln_cost`	add general nonlinear cost set, see Section 5.3.2
`add_quad_cost`	add quadratic cost set, see Section 5.3.1
`add_var`	add variable set, see Section 5.1
`copy`†	makes a copy of an existing `opt_model` object
`describe_idx`†	identifies indices of a given set type, see Section 5.5.1
`display`	displays variable, constraint and cost sets, see Section 5.7.1
`display_set`†	displays indexing for a particular set type, called by `display`
`eval_nln_constraint`	builds full set of nonlinear equality or inequality constraints and their gradients for given value of x, see Section 5.5.3
`eval_nln_constraint_hess`	builds Hessian for full set of nonlinear equality or inequality constraints for given value of x, see Section 5.5.3
`eval_nln_cost`	evaluates nonlinear cost function and its derivatives‡ for given value of x, see Section 5.5.4
`eval_quad_cost`	evaluates quadratic cost function and its derivatives‡ for given value of x, see Section 5.5.4
`get`†	access (possibly nested) fields of the object
`get_idx`†	returns index structures for specified set types, see Section 5.5.1
`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`†	initializes dimensions for a particular indexed named set
`is_mixed_integer`	returns 1 if any of the variables are binary or integer, 0 otherwise
`params_lin_constraint`	assembles and returns parameters for linear constraints‡
`params_quad_cost`	assembles and returns parameters for quadratic costs‡
`params_var`	assembles and returns inital values, bounds, types for variables‡
`problem_type`	type of mathematical program represented by current model
`solve`	solves the model, see Section 5.4
`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`
*Private Methods**	
`add_named_set`§	adds information for new instance of a given set type
`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`.
‡ 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`.

6 Utility Functions

6.1 `have_fcn`

```
TorF = have_fcn(tag)
TorF = have_fcn(tag, toggle)
ver_str = have_fcn(tag, 'vstr')
ver_num = have_fcn(tag, 'vnum')
rdate   = have_fcn(tag, 'date')
info    = have_fcn(tag, 'all')
```

The `have_fcn` function provides a unified mechanism for testing for optional functionality, such as the presence of certain solvers, or to detect whether the code is running under MATLAB or Octave. Since its results are cached they allow for a very quick way to check frequently for functionality that may initially be a bit more costly to determine. For installed functionality, `have_fcn` also determines the installed version and release date, if possible. The optional second argument, when it is a string, defines which value is returned, as follows:

- *empty* – 1 if optional functionality is available, 0 if not available
- `'vstr'` – version number as a string (e.g. `'3.11.4'`)
- `'vnum'` – version number as numeric value (e.g. 3.011004)
- `'date'` – release date as a string (e.g. `'20-Jan-2015'`)
- `'all'` – struct with fields named `av` (for "availability"), `vstr`, `vnum` and `date`, and values corresponding to each of the above, respectively.

Alternatively, the optional functionality specified by `tag` can be toggled OFF or ON by calling `have_fcn` with a numeric second argument `toggle` with one of the following values:

- 0 – turn OFF the optional functionality
- 1 – turn ON the optional functionality (if available)
- -1 – toggle the ON/OFF state of the optional functionality

The valid values of the `tag` string input argument are listed in Table 6-1.

Table 6-1: Valid Tag Values for `have_fcn`

tag value	description
`bpmpd`	`bp`, BPMPD interior point LP/QP solver
`clp`	CLP, LP/QP solver, https://github.com/coin-or/Clp
`opti_clp`	version of CLP distributed with OPTI Toolbox, https://www.inverseproblem.co.nz/OPTI/
`cplex`	CPLEX, IBM ILOG CPLEX Optimizer
`fmincon`	`fmincon`, solver from Optimization Toolbox
`fmincon_ipm`	`fmincon` with interior point solver from Optimization Toolbox 4.x+
`fsolve`	`fsolve`, nonlinear equation solver from Optimization Toolbox
`glpk`	`glpk`, GNU Linear Programming Kit, LP/MILP solver
`gurobi`	`gurobi`, Gurobi solver, https://www.gurobi.com/
`intlinprog`	`intlinprog`, MILP solver from Optimization Toolbox 7.0 (R2014a)+
`ipopt`	IPOPT, NLP solver, https://github.com/coin-or/Ipopt
`knitro`	Artelys Knitro, NLP solver, https://www.artelys.com/solvers/knitro/
`knitromatlab`	Artelys Knitro, version 9.0.0+
`ktrlink`	Knitro, version prior to 9.0.0 (requires Optimization Toolbox)
`linprog`	`linprog`, LP solver from Optimization Toolbox
`linprog_ds`	`linprog` w/dual-simplex solver from Optimization Toolbox 7.1 (R2014b)+
`matlab`	MATLAB, as opposed to Octave
`mosek`	MOSEK, LP/QP solver, https://www.mosek.com/
`octave`	GNU Octave, as opposed to MATLAB
`optimoptions`	`optimoptions`, option setting funciton for Optimization Toolbox 6.3+
`pardiso`	PARDISO, **Par**allel Sparse **D**irect & **I**terative Linear **So**lver, https://pardiso-project.org
`quadprog`	`quadprog`, QP solver from Optimization Toolbox
`quadprog_ls`	`quadprog` with large-scale interior point convex solver from Optimization Toolbox 6.x+
`sdpt3`	SDPT3 SDP solver, https://github.com/sqlp/sdpt3
`sedumi`	SeDuMi SDP solver, http://sedumi.ie.lehigh.edu
`yalmip`	YALMIP modeling platform, https://yalmip.github.io
Functionality related to MATPOWER	
`e4st`	E4ST, http://www.e4st.com/
`minopf`	`minopf`, MINOPF, MINOS-based optimal power flow (OPF) solver
`most`	MOST, **M**ATPOWER **O**ptimal **S**cheduling **T**ool
`pdipmopf`	PDIPMOPF, primal-dual interior point method OPF solver
`scpdipmopf`	SCPDIPMOPF, step-controlled PDIPM OPF solver
`sdp_pf`	SDP_PF, applications of SDP relaxation of power flow equations
`smartmarket`	`runmarket` and friends, for running an energy auction
`syngrid`	SynGrid, Synthetic Grid Creation for MATPOWER
`tralmopf`	TRALMOPF, trust region based augmented Langrangian OPF solver

6.2 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.

6.3 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.

6.4 Matpower-related Functions

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

6.4.1 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.

6.4.2 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.

6.4.3 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[25] to the specific problem class, based on the following precedence: Gurobi, CPLEX, MOSEK, Optimization Toolbox, GLPK, BPMPD, MIPS.

[25] 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 is not an option for problems that combine the two (MIQP).

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
`  src/MP-Opt-Model-manual/`	
`    MP-Opt-Model-manual.tex`	LaTeX source for MP-Opt-Model User's Manual
`lib/`	MP-Opt-Model software (see Tables A-2, A-4, A-5 and A-6)
`  t/`	MP-Opt-Model tests (see Table A-7)

Table A-2: Solver Functions

name	description
`miqps_master`	Mixed-Integer Quadratic Program Solver wrapper function, provides a unified interface for various MIQP/MILP solvers
`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_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_master`	Nonlinear Equation Solver wrapper function, provides a unified interface for various nonlinear equation (NLEQ) solvers
`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_master`	Nonlinear Program Solver wrapper function, provides a unified interface for various NLP solvers
`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†
`qps_master`	Quadratic Program Solver wrapper function, provides a unified interface for various QP/LP solvers
`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_ipopt`	QP/LP solver API implementation for IPOPT-based solver†
`qps_mosek`	QP/LP solver API implementation for MOSEK (`mosekopt`)†
`qps_ot`	QP/LP solver API implementation for MATLAB Opt Toolbox's `quadprog`, `linprog`
	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 Options, etc.

name	description
`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†
`gurobiver`	prints version information for Gurobi/Gurobi_MEX
`ipopt_options`	default options for IPOPT solver†
`mosek_options`	default options for MOSEK solver†
`mosek_symbcon`	symbolic constants to use for MOSEK solver options†

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

Table A-4: Optimization Model Class

name	description
`@opt_model/`	optimization 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_named_set`†	adds a named subset of costs, constraints or variables 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 optimization variables to the model
`display`	called to display object when statement not ended with semicolon
`eval_nln_constraint`	returns full set of 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, lin/nln constraints, costs
`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 $\hat{x}$‡
`problem_type`	indicates type of mathematical program (e.g. LP, QP, MILP, MIQP, or NLP)
`solve`	solves the optimization 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 optimization variables for variable set list
`varsets_x`	assembles cell array of sub-vectors of opt vector $\hat{x}$ specified by variable set list
`nlp_consfcn`§	evaluates nonlinear constraints and gradients for `opt_model`
`nlp_costfcn`§	evaluates nonlinear costs, gradients, Hessian for `opt_model`
`nlp_hessfcn`§	evaluates nonlinear constraint Hessians for `opt_model`

† Private method for internal use only.

‡ For all, or alternatively, only for a named (and possibly indexed) subset.

§ Ideally should be implemented as a method of the `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.

Table A-5: 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	identifies indices of a given set type E.g. variable 361 corresponds to Pg(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
valid_named_set_type†	returns label for given named set type if valid, empty otherwise

† Private method for internal use only.
‡ For all, or alternatively, only for a named (and possibly indexed) subset.

Table A-6: 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
nested_struct_copy	copies the contents of nested structs

Table A-7: MP-Opt-Model Examples & Tests

name	description
lib/t/	MP-Opt-Model examples & tests
nleqs_master_ex1	code for NLEQ Example 1 (see Section 4.4.1) for nleqs_master
nleqs_master_ex2	code for NLEQ Example 2 (see Section 4.4.2) for nleqs_master
nlps_master_ex1	code for NLP Example 1 (see Section 4.3.1) for nlps_master
nlps_master_ex2	code for NLP Example 2 (see Section 4.3.2) for nlps_master
qp_ex1	code for QP Example from Section 2.3
test_mp_opt_model	runs full MP-Opt-Model test suite
t_have_fcn	runs tests for have_fcn
t_miqps_master	runs tests of MILP/MIQP solvers via miqps_master
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 vianlps_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_qps	runs tests of LP/QP solvers via om.solve()
t_opt_model	runs tests for opt_model objects
t_qps_master	runs tests of LP/QP solvers viaqps_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 path.

B.1 BPMPD_MEX – MEX interface for BPMPD

BPMPD_MEX [8,9] 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 [10] (**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[26]. For Microsoft

[26] 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 `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.

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[27] [11]

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 [12] (**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[28]. 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[29] [11].

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 com-

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

[28]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.

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

mon 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, MILP and MIQP Solvers

Gurobi [13] 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 Ipopt – Interior Point Optimizer

Ipopt [14] (**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 [15, 16], are available from the PARDISO project[30]. For Microsoft Windows users, a pre-compiled MEX version of Ipopt (and numerous

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

other solvers, such as CLP and GLPK) are easily installable as part of the OPTI Toolbox[31] [11].

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.7 Artelys Knitro – Non-Linear Programming Solver

Artelys Knitro [17] is a general purpose optimization solver specializing in nonlinear problems, available from Artelys. As of version 9, Knitro includes a native MATLAB interface, `knitromatlab`[32]. 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 function, `knitromatlab` or `ktrlink`, can be used by MP-Opt-Model to solve general NLP problems via MP-Opt-Model's common NLP solver interface `nlps_master` with the algorithm option set to `'KNITRO'`, or by calling `nlps_knitro` directly.

B.8 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.

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

[32] Earlier versions required the MATLAB Optimization Toolbox from The MathWorks, which included an interface to the Knitro libraries called `ktrlink`, but the libraries themselves still had to be acquired directly from Ziena Optimization, LLC (subsequently acquired by Artelys).

B.9 Optimization Toolbox – LP, QP, NLP, NLEQ and MILP Solvers

MATLAB's Optimization Toolbox [18, 19], 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.

Appendix C 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`.

C.1 Version 0.7 – Jun 20, 2019

This release history begins with the code that was part of the MATPOWER 7.0 release.

C.2 Version 0.8 – Apr 29, 2020 *(not released publicly)*

This version consists of functionality moved directly from MATPOWER.[33] 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.*

[33] 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

- 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 sub-class.

- `opf_consfcn()` → `nlp_consfcn()`
- `opf_costfcn()` → `nlp_costfcn()`
- `opf_hessfcn()` → `nlp_hessfcn()`

C.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.[34]

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.

C.4 Version 2.0 – released Jul 8, 2020

The MP-Opt-Model 2.0 User's Manual is available online.[35]

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.

[34] https://matpower.org/docs/MP-Opt-Model-manual-1.0.pdf
[35] 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.

C.5 Version 2.1 – released Aug 25, 2020

The MP-Opt-Model 2.1 User's Manual is available online.[36]

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.

[36] https://matpower.org/docs/MP-Opt-Model-manual-2.1.pdf

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 (2019). MATPOWER [Software]. Available: https://matpower.org doi: 10.5281/zenodo.3236535 1.1

[3] John W. Eaton, David Bateman, Søren Hauberg, Rik Wehbring (2015). *GNU Octave version 4.0.0 manual: a high-level interactive language for numerical computations.* Available: https://www.gnu.org/software/octave/doc/interpreter/. 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. 2020. [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. 2019. [Online]. Available: https://matpower.org/docs/MIPS-manual.pdf doi: 10.5281/zenodo.3236506 2.1

[8] BPMPD_MEX. [Online]. Available: http://www.pserc.cornell.edu/bpmpd/. B.1

[9] 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

[10] COIN-OR Linear Programming (CLP) Solver. [Online]. Available: https://github.com/coin-or/Clp. B.2

[11] 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. B.2, B.4, B.6

[12] GLPK. [Online]. Available: `https://www.gnu.org/software/glpk/`. B.4

[13] Gurobi Optimization, Inc., "Gurobi Optimizer Reference Manual," 2016. [Online]. Available: `https://www.gurobi.com/`. B.5

[14] 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.6

[15] 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.6

[16] 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.6

[17] 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.7

[18] *Optimization Toolbox*, The MathWorks, Inc. [Online]. Available: `https://www.mathworks.com/products/optimization/`. B.9

[19] *Optimization Toolbox Users's Guide*, The MathWorks, Inc., 2016. [Online]. Available: `https://www.mathworks.com/help/releases/R2016b/pdf_doc/optim/optim_tb.pdf`. B.9