

MP-Opt-Model User's Manual

Version 4.2

Ray D. Zimmerman

May 10, 2024

Contents

1	Introduction	8
1.1	Background	8
1.2	License and Terms of Use	9
1.3	Citing MP-Opt-Model	10
1.4	MP-Opt-Model Development	10
2	Getting Started	11
2.1	System Requirements	11
2.2	Installation	11
2.3	Sample Usage	12
2.4	Documentation	15
3	MP-Opt-Model – Overview	16
4	Solver Interface Functions	17
4.1	LP/QP Solvers – <code>qps_master</code>	17
4.1.1	QP Example	20
4.2	MILP/MIQP Solvers – <code>miqps_master</code>	21
4.2.1	MILP Example	23
4.3	NLP Solvers – <code>nlps_master</code>	23
4.3.1	NLP Example 1	26
4.3.2	NLP Example 2	27
4.4	Nonlinear Equation Solvers – <code>nleqs_master</code>	30
4.4.1	NLEQ Example 1	32
4.4.2	NLEQ Example 2	35
4.5	Parameterized Nonlinear Equation Solver – <code>pnes_master</code>	39
4.5.1	Parameterization	39
4.5.2	Predictor	40
4.5.3	Corrector	41
4.5.4	Step Length Control	41
4.5.5	Event Detection and Location	42
4.5.6	Callback Functions	42
4.5.7	<code>pnes_master</code>	46
4.5.8	PNE Example	51

5	Optimization Model Class – <code>opt_model</code>	54
5.1	Adding Variables	54
5.1.1	Variable Subsets	55
5.2	Adding Constraints	56
5.2.1	Linear Constraints	56
5.2.2	General Nonlinear Constraints	57
5.3	Adding Costs	58
5.3.1	Quadratic Costs	59
5.3.2	General Nonlinear Costs	60
5.4	Solving the Model	61
5.5	Accessing the Model	63
5.5.1	Indexing	63
5.5.2	Variables	68
5.5.3	Constraints	69
5.5.4	Costs	71
5.5.5	Model Solution	73
5.6	Modifying the Model	78
5.7	Indexed Sets	79
5.8	Miscellaneous Methods	80
5.8.1	Public Methods	80
5.8.2	Private Methods	83
5.9	MATPOWER Index Manager Base Class – <code>mp_idx_manager</code>	83
5.10	Reference	86
5.10.1	Properties	86
5.10.2	Methods	86
6	Utility Functions	88
6.1	<code>convert_lin_constraint</code>	88
6.2	<code>convert_lin_constraint_multipliers</code>	88
6.3	<code>have_fcn</code>	89
6.4	<code>mpomver</code>	89
6.5	<code>nested_struct_copy</code>	89
6.6	Private Feature Detection Functions	89
6.6.1	<code>have_feature_bpmpd</code>	90
6.6.2	<code>have_feature_catchme</code>	90
6.6.3	<code>have_feature_clp</code>	90
6.6.4	<code>have_feature_opti_clp</code>	90
6.6.5	<code>have_feature_cplex</code>	90

6.6.6	have_feature_evalc	90
6.6.7	have_feature_fmincon	90
6.6.8	have_feature_fmincon_ipm	91
6.6.9	have_feature_fsolve	91
6.6.10	have_feature_glpk	91
6.6.11	have_feature_gurobi	91
6.6.12	have_feature_intlinprog	91
6.6.13	have_feature_ipopt	91
6.6.14	have_feature_ipopt_auxdata	91
6.6.15	have_feature_isequaln	92
6.6.16	have_feature_knitro	92
6.6.17	have_feature_knitromatlab	92
6.6.18	have_feature_ktrlink	92
6.6.19	have_feature_linprog	92
6.6.20	have_feature_linprog_ds	92
6.6.21	have_feature_mosek	92
6.6.22	have_feature_optim	93
6.6.23	have_feature_optimoptions	93
6.6.24	have_feature_osqp	93
6.6.25	have_feature_quadprog	93
6.6.26	have_feature_quadprog_ls	93
6.6.27	have_feature_sdpt3	93
6.6.28	have_feature_sedumi	93
6.6.29	have_feature_yalmip	94
6.7	MATPOWER-related Functions	94
6.7.1	mpopt2nleqopt	94
6.7.2	mpopt2nlpopt	94
6.7.3	mpopt2qpopt	94
6.7.4	mpopt2pneopt	95
7	Acknowledgments	96
	Appendix A MP-Opt-Model Files, Functions and Classes	97
	Appendix B Optional Packages	104
B.1	BPMPD_MEX – MEX interface for BPMPD	104
B.2	CLP – COIN-OR Linear Programming	104
B.3	CPLEX – High-performance LP, QP, MILP and MIQP Solvers	105
B.4	GLPK – GNU Linear Programming Kit	105

B.5	Gurobi – High-performance LP, QP, MILP and MIQP Solvers	106
B.6	IPOPT – Interior Point Optimizer	106
B.7	Artelys Knitro – Non-Linear Programming Solver	107
B.8	MOSEK – High-performance LP, QP, MILP and MIQP Solvers . . .	107
B.9	Optimization Toolbox – LP, QP, NLP, NLEQ and MILP Solvers . . .	108
B.10	OSQP – Operator Splitting Quadratic Program Solver	108
Appendix C Release History		109
C.1	Version 0.7 – Jun 20, 2019	109
C.2	Version 0.8 – Apr 29, 2020 (<i>not released publicly</i>)	109
C.3	Version 1.0 – released May 8, 2020	111
C.4	Version 2.0 – released Jul 8, 2020	111
C.5	Version 2.1 – released Aug 25, 2020	113
C.6	Version 3.0 – released Oct 8, 2020	114
C.7	Version 4.0 – released Oct 18, 2021	115
C.8	Version 4.1 – released Dec 13, 2022	117
C.9	Version 4.2 – released May 10, 2024	117
References		120

List of Tables

4-1	Input Arguments for <code>qps_master</code>	18
4-2	Output Arguments for <code>qps_master</code>	18
4-3	Options for <code>qps_master</code>	19
4-4	Input Arguments for <code>miqps_master</code>	21
4-5	Options for <code>miqps_master</code>	22
4-6	Input Arguments for <code>nlps_master</code>	24
4-7	Output Arguments for <code>nlps_master</code>	25
4-8	Options for <code>nlps_master</code>	25
4-9	Input Arguments for <code>nleqs_master</code>	31
4-10	Output Arguments for <code>nleqs_master</code>	32
4-11	Options for <code>nleqs_master</code>	33
4-12	Callback Input Arguments	43
4-13	Callback Output Arguments	44
4-14	Fields of Continuation State Struct	44
4-15	Input Arguments for <code>pnes_master</code>	46
4-16	Output Arguments for <code>pnes_master</code>	47
4-17	Options for <code>pnes_master</code>	48
4-18	Plot Options for <code>pnes_master</code>	49
4-19	Warm-start Data for <code>pnes_master</code>	50
5-1	Options for <code>solve</code>	62
5-2	Values for <code>alg</code> Option to <code>solve</code>	63
5-3	Example Indexing Data	64
5-4	Valid Set Types	65
5-5	Model Solution	74
5-6	Inputs for <code>get_soln</code>	75
5-7	Values of <code>tags</code> input to <code>get_soln</code>	76
5-8	Output of <code>parse_soln</code>	77
5-9	Inputs for <code>set_params</code>	78
5-10	MATPOWER Index Manager (<code>mp_idx_manager</code>) Properties and Methods	84
5-11	MATPOWER Index Manager (<code>mp_idx_manager</code>) Object Structure	85
5-12	<code>opt_model</code> Properties	86
5-13	<code>opt_model</code> Methods	87
A-1	MP-Opt-Model Files and Functions	97
A-2	Solver Functions	98
A-3	PNE Implementation Functions	99
A-4	Solver Options, etc.	99

A-5	Optimization Model Class	100
A-6	MATPOWER Index Manager Class	101
A-7	Utility Functions	101
A-8	Feature Detection Functions	102
A-9	MP-Opt-Model Examples & Tests	103

1 Introduction

1.1 Background

[MP-Opt-Model](#) is a package of MATLAB language M-files¹ 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 PSERC² at Cornell University, along with significant contributions from others, as part of the [MATPOWER](#) [1,2] project.

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

¹Also 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) 2004-2024, 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 4.2, use:

R. D. Zimmerman. MP-Opt-Model User’s Manual, Version 4.2. 2024. [Online]. Available: <https://matpower.org/docs/MP-Opt-Model-manual-4.2.pdf>
doi: 10.5281/zenodo.11177079

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 4.2 you will need:

- MATLAB[®] version 7.9 (R2009b) or later³, or
- GNU Octave version 4.0 or later⁴
- MIPS, MATPOWER Interior Point Solver [6, 7]⁵
- MP-Test⁶

For the hardware requirements, please refer to the system requirements for the version of MATLAB⁷ or Octave that you are using.

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

2.2 Installation

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

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

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

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

⁴GNU Octave [3] is free software, available online at <https://www.gnu.org/software/octave/>.

⁵MIPS is available at <https://github.com/MATPOWER/mips>.

⁶MP-Test is available at <https://github.com/MATPOWER/mpptest>.

⁷https://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/Octave prompt, type `test_mp_opt_model` to run the test suite and verify that MP-Opt-Model is properly installed and functioning.⁹ The result should resemble the following:

```
>> test_mp_opt_model
t_have_fcn.....ok
t_nested_struct_copy...ok
t_nleqs_master.....ok (30 of 150 skipped)
t_pnes_master.....ok
t_qps_master.....ok (100 of 432 skipped)
t_miqps_master.....ok (78 of 318 skipped)
t_nlps_master.....ok
t_opt_model.....ok
t_om_solve_leqs.....ok
t_om_solve_nleqs.....ok (36 of 195 skipped)
t_om_solve_pne.....ok
t_om_solve_qps.....ok (81 of 388 skipped)
t_om_solve_miqps.....ok (20 of 131 skipped)
t_om_solve_nlps.....ok
All tests successful (3461 passed, 345 skipped of 3806)
Elapsed time 2.28 seconds.
```

2.3 Sample Usage

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

$$\min_{y,z} \frac{1}{2} \begin{bmatrix} y^T & z^T \end{bmatrix} Q \begin{bmatrix} y \\ z \end{bmatrix} \quad (2.1)$$

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

subject to

$$A_1 \begin{bmatrix} y \\ z \end{bmatrix} = b_1 \quad (2.2)$$

$$A_2 y \leq u_2 \quad (2.3)$$

$$y \geq y_{\min} \quad (2.4)$$

$$z \leq z_{\max} \quad (2.5)$$

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

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

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

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

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

Below, we will show two approaches to construct and solve the problem. The first method, based on the 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, [], zmax);
om.add_lin_constraint('lincon1', A1, b1, b1);
om.add_lin_constraint('lincon2', A2, [], u2, {'y'});
om.add_quad_cost('cost', Q, []);

%% 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; -Inf(2,1)];
xmax = [ Inf(2,1); zmax];
x0 = [y0; z0];
A = [ A1; A2 0 0];
l = [ b1; -Inf ];
u = [ b1; u2 ];

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

```

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

```

f = 1.875      exitflag = 1

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

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

```

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

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

2.4 Documentation

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

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-Model¹⁰ 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)
- linear equations (LEQ)
- nonlinear equations (NLEQ)
- parameterized nonlinear equations (PNE)

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

Finally, MP-Opt-Model includes a utility 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.

¹⁰The 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:

$$\min_x \frac{1}{2}x^\top Hx + c^\top x \quad (4.1)$$

subject to

$$l \leq Ax \leq u \quad (4.2)$$

$$x_{\min} \leq x \leq x_{\max}. \quad (4.3)$$

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
<code>H</code>	(possibly sparse) matrix H of quadratic cost coefficients
<code>c</code>	column vector c of linear cost coefficients
<code>A</code>	(possibly sparse) matrix A of linear constraint coefficients
<code>l</code>	column vector l of lower bounds on Ax , defaults to $-\infty$
<code>u</code>	column vector u of upper bounds on Ax , defaults to $+\infty$
<code>xmin</code>	column vector x_{\min} of lower bounds on x , defaults to $-\infty$
<code>xmax</code>	column vector x_{\max} of upper bounds on x , defaults to $+\infty$
<code>x0</code>	optional starting value of optimization vector x (<i>ignored by some solvers</i>)
<code>opt</code>	optional options struct (all fields optional), see Table 4-3 for details
<code>problem</code>	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
<code>x</code>	solution vector x
<code>f</code>	final objective function value $f(x) = \frac{1}{2}x^T Hx + c^T x$
<code>exitflag</code>	exit flag 1 – converged successfully ≤ 0 – solver-specific failure code
<code>output</code>	output struct with the following fields: <code>alg</code> – algorithm code of solver used (<i>others</i>) – solver-specific fields
<code>lambda</code>	struct containing the Lagrange and Kuhn-Tucker multipliers on the constraints, with fields: <code>mu.l</code> – lower (left-hand) limit on linear constraints <code>mu.u</code> – upper (right-hand) limit on linear constraints <code>lower</code> – lower bound on optimization variables <code>upper</code> – upper bound on optimization variables

Table 4-3: Options for `qps_master`

name	default	description
<code>alg</code>	'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* (<i>LP only</i>) 'GUROBI' – Gurobi* 'ILOPT' – ILOPT* 'MIPS' – MIPS, MATPOWER Interior Point Solver 'MOSEK' – MOSEK* 'OT' – MATLAB Opt Toolbox, <code>quadprog</code> , <code>linprog</code>
<code>verbose</code>	0	amount of progress info to be printed 0 – print no progress info 1 – print a little progress info 2 – print a lot of progress info 3 – print all progress info
<code>bp_opt</code>	<i>empty</i>	options vector for <code>bp</code> *
<code>clp_opt</code>	<i>empty</i>	options vector for <code>CLP</code> *
<code>cplex_opt</code>	<i>empty</i>	options struct for <code>CPLEX</code> *
<code>glpk_opt</code>	<i>empty</i>	options struct for <code>GLPK</code> *
<code>grb_opt</code>	<i>empty</i>	options struct for <code>Gurobi</code> *
<code>ipopt_opt</code>	<i>empty</i>	options struct for <code>ILOPT</code> *
<code>linprog_opt</code>	<i>empty</i>	options struct for <code>linprog</code> *
<code>mips_opt</code>	<i>empty</i>	options struct for <code>MIPS</code>
<code>mosek_opt</code>	<i>empty</i>	options struct for <code>MOSEK</code> *
<code>quadprog_opt</code>	<i>empty</i>	options struct for <code>quadprog</code> *

* 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 problem¹¹ 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`.

¹¹From <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 solver 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,

$$x_i \in \mathbb{Z}, \quad \forall i \in \mathcal{I} \quad (4.4)$$

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

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
<i>all <code>qps_master</code> input args from Table 4-1, with the following additions/modifications</i>	
<code>vtype</code>	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
<code>alg</code>	'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*(<i>LP only</i>) 'GUROBI' – Gurobi* 'MOSEK' – MOSEK* 'OT' – MATLAB Opt Toolbox, <code>intlinprog</code>
<code>verbose</code>	0	amount of progress info to be printed 0 – print no progress info 1 – print a little progress info 2 – print a lot of progress info 3 – print all progress info
<code>skip_prices</code>	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
<code>price_stage_warn_tol</code>	10^{-7}	tolerance on the objective function value and primal variable relative mismatch required to avoid mismatch warning message
<code>cplex_opt</code>	<i>empty</i>	options struct for CPLEX*
<code>glpk_opt</code>	<i>empty</i>	options struct for GLPK*
<code>grb_opt</code>	<i>empty</i>	options struct for Gurobi*
<code>intlinprog_opt</code>	<i>empty</i>	options struct for <code>intlinprog</code> *
<code>mosek_opt</code>	<i>empty</i>	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 problem¹² 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 nonlinear programming solver interface for general nonlinear programming (NLP) problems, that is, problems of the form:

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

subject to

$$g(x) = 0 \tag{4.7}$$

$$h(x) \leq 0 \tag{4.8}$$

$$l \leq Ax \leq u \tag{4.9}$$

$$x_{\min} \leq x \leq x_{\max} \tag{4.10}$$

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

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);
```

¹²From 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
<code>f_fcn</code>	handle to function that evaluates the objective function, its gradients and Hessian [‡] for a given value of x , with calling syntax: $[f, df, d2f] = f_fcn(x)$
<code>x0</code>	starting value of optimization vector x
<code>A, l, u</code>	define optional linear constraints $l \leq Ax \leq u$, where default values for elements of <code>l</code> and <code>u</code> are <code>-Inf</code> and <code>Inf</code> , respectively.
<code>xmin, xmax</code>	optional lower and upper bounds on x , with defaults <code>-Inf</code> and <code>Inf</code> , respectively
<code>gh_fcn</code>	handle to function that evaluates the optional nonlinear constraints and their gradients for a given value of x , with calling syntax: $[h, g, dh, dg] = gh_fcn(x)$ where the columns of <code>dh</code> and <code>dg</code> are the gradients of the corresponding elements of <code>h</code> and <code>g</code> , i.e. <code>dh</code> and <code>dg</code> are transposes of the Jacobians of <code>h</code> and <code>g</code> , respectively
<code>hess_fcn</code>	handle to function that computes the Hessian [‡] of the Lagrangian for given values of x , λ and μ , where λ and μ are the multipliers on the equality and inequality constraints, g and h , respectively, with calling syntax: $Lxx = hess_fcn(x, lam, cost_mult),$ where $\lambda = lam.eqnonlin$, $\mu = lam.ineqnonlin$ and <code>cost_mult</code> is a parameter used to scale the objective function
<code>opt</code>	optional options struct (all fields optional), see Table 4-8 for details
<code>problem</code>	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`.

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

Table 4-7: Output Arguments for `nlp_solver`

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

Table 4-8: Options for `nlp_solver`

name	default	description
<code>alg</code>	'DEFAULT'	determines which solver to use 'DEFAULT' – automatic, current default is MIPS 'MIPS' – MIPS, MATPOWER Interior Point Solver 'FMINCON' – MATLAB Opt Toolbox, <code>fmincon</code> * 'IPOPT' – IPOPT* 'KNITRO' – Artelys Knitro*
<code>verbose</code>	0	amount of progress info to be printed 0 – print no progress info 1 – print a little progress info 2 – print a lot of progress info
<code>mips_opt</code>	<i>empty</i>	options struct for MIPS
<code>fmincon_opt</code>	<i>empty</i>	options struct for <code>fmincon</code> *
<code>ipopt_opt</code>	<i>empty</i>	options struct for IPOPT*
<code>knitro_opt</code>	<i>empty</i>	options struct for Artelys Knitro*

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

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

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

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” function¹³

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

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 parameter `a`.

¹³https://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 nargin > 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 nargin > 2      %% Hessian is required
        d2f = 4*a*[ 3*x(1)^2 - x(2) + 1/(2*a),  -x(1);
                    -x(1)                        1/2 ];
    end
end
end

```

Then, create a handle to the function, defining the value of the parameter 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 example¹⁴ solves the following 3-dimensional constrained optimization, printing the details of the solver's progress:

$$\min_x f(x) = -x_1x_2 - x_2x_3 \quad (4.13)$$

subject to

$$x_1^2 - x_2^2 + x_3^2 - 2 \leq 0 \quad (4.14)$$

$$x_1^2 + x_2^2 + x_3^2 - 10 \leq 0. \quad (4.15)$$

¹⁴From https://en.wikipedia.org/wiki/Nonlinear_programming#3-dimensional_example.

First, create a function to evaluate the objective function and its gradients,¹⁵

```
function [f, df, d2f] = f2(x)
f = -x(1)*x(2) - x(2)*x(3);
if nargin > 1          %% gradient is required
    df = -[x(2); x(1)+x(3); x(2)];
    if nargin > 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.

¹⁵Since 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 1.5.1, 10-May-2024
(using built-in linear solver)

```

it	objective	step size	feascond	gradcond	compcnd	costcnd
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')

Iter F-count      f(x)  Feasibility  First-order  Norm of
                    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 nonlinear equation solver interface for general nonlinear equations (NLEQ), that is, problems of the form:

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

where $f: \mathbb{R}^n \rightarrow \mathbb{R}^n$.

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
<code>fcn</code>	handle to function that evaluates the function $f(x)$ and optionally its Jacobian $J(x)$ for a given value of x , with calling syntax: $f = fcn(x)$, or $[f, J] = fcn(x)$ where selected solver algorithm determines whether <code>fcn</code> is required to return the Jacobian or not
<code>x0</code>	starting value of vector x
<code>opt</code>	optional options struct (all fields optional), see Table 4-11 for details
<code>problem</code>	alternative, single argument input struct with fields corresponding to arguments above

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
<code>x</code>	solution vector
<code>f</code>	final function value, $f(x)$
<code>exitflag</code>	exit flag 1 – converged successfully ≤ 0 – solver-specific failure code
<code>output</code>	output struct with the following fields: <code>alg</code> – algorithm code of solver used (<i>others</i>) – solver-specific fields
<code>jac</code>	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 function¹⁶

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

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

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

¹⁶<https://www.chilimath.com/lessons/advanced-algebra/systems-non-linear-equations/>

Table 4-11: Options for `nleqs_master`

name	default	description
<code>alg</code>	'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, <code>fsolve</code> * 'GS' – Gauss-Seidel method‡
<code>verbose</code>	0	amount of progress info to be printed 0 – print no progress info 1 – print a little progress info 2 – print a lot of progress info
<code>max_it</code>	0	maximum number of iterations§
<code>tol</code>	0	termination tolerance on $f(x)$ §
<code>core_sp</code>	<i>empty</i>	solver parameters struct for <code>nleqs_core</code> ¶
<code>fd_opt</code>	<i>empty</i>	options struct for fast-decoupled Newton's method, <code>nleqs_fd_newton</code> †
<code>fsolve_opt</code>	<i>empty</i>	options struct for <code>fsolve</code> *
<code>gs_opt</code>	<i>empty</i>	options struct for Gauss-Seidel method, <code>nleqs_gauss_seidel</code> ‡
<code>newton_opt</code>	<i>empty</i>	options struct for Newton's method, <code>nleqs_newton</code>

* 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')
```

Iteration	Func-count	f(x)	Norm of step	First-order optimality	Trust-region 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.¹⁷ This example includes the update function required for Gauss-Seidel and the Jaco-

¹⁷From Christi Patton Luks, <https://www.youtube.com/watch?v=pJG4yhtgerg>

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

$$f(x) = \begin{bmatrix} x_1^2 + x_1x_2 - 10 \\ x_2 + 3x_1x_2^2 - 57 \end{bmatrix} \quad (4.18)$$

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

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

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

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

Fast-decoupled Newton example results:

```
>> nleqs_master_ex2('FD')

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

Fast-decoupled Newton's method converged in 13 P- and 13 Q-iterations.

exitflag = 1

x =
     2
     3

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

Gauss-Seidel example results:

```
>> nleqs_master_ex2('GS')
```

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

Gauss-Seidel method converged in 20 iterations.

exitflag = 1

x =

2
3

f =

-7.51313e-09
4.48558e-09

4.5 Parameterized Nonlinear Equation Solver – `pnes_master`

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

$$f(x) = 0, \tag{4.19}$$

where $f: \mathbb{R}^{n+1} \rightarrow \mathbb{R}^n$.

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

$$x = \begin{bmatrix} y \\ \lambda \end{bmatrix} \tag{4.20}$$

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

$$f(x) = (1 - \lambda)g_0(y) + \lambda g(y), \tag{4.21}$$

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

Currently MP-Opt-Model includes only a single solver implementation for PNE problems based on a numerical continuation method commonly known as a predictor-corrector method [8]. This method involves adding another equation to the system which identifies the location of the current solution with respect to the previous or next solution. The continuation process can be diagrammatically shown by (4.22).

$$x^j \xrightarrow{\text{Predictor}} \hat{x}^{j+1} \xrightarrow{\text{Corrector}} x^{j+1} \tag{4.22}$$

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

4.5.1 Parameterization

The values of x along the solution curve can be parameterized in a number of ways [9, 10]. Parameterization is a mathematical way of identifying each solution so that

the *next* solution or *previous* solution can be quantified. MP-Opt-Model includes three parameterization scheme options to quantify this relationship, detailed below, where σ is the continuation step size parameter and λ is the last element of x .

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

$$p^j(x) = \lambda - \lambda^j - \sigma^j = 0 \quad (4.23)$$

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

$$p^j(x) = \sum_i (x_i - x_i^j)^2 - (\sigma^j)^2 = 0 \quad (4.24)$$

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

$$p^j(x) = (x - x^j)^\top \bar{z}^j - \sigma^j = 0, \quad (4.25)$$

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

4.5.2 Predictor

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

$$\begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial p^{j-1}}{\partial x} \end{bmatrix} z^j = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \quad (4.26)$$

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

The resulting tangent vector is then normalized

$$\bar{z}^j = \frac{z^j}{\|z^j\|_2} \quad (4.27)$$

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

$$\hat{x}^{j+1} = x^j + \sigma^j \bar{z}^j, \quad (4.28)$$

where σ^j is the continuation step size.

4.5.3 Corrector

The corrector stage at step j finds the next solution x^{j+1} by correcting the approximation \hat{x}^{j+1} estimated by the predictor. A method such as Newton's method is used to find the next solution by solving the $n + 1$ dimensional system in (4.29), where one of (4.23)–(4.25) has been added as an additional constraint to the parameterized nonlinear equations of (4.19).

$$\begin{bmatrix} f(x) \\ p^j(x) \end{bmatrix} = 0 \quad (4.29)$$

The corrector in MP-Opt-Model uses `nleqs_master` with its variety of available solvers to solve (4.29) for each new solution point on the curve.

4.5.4 Step Length Control

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

$$\sigma^{j+1} = \alpha^j \sigma^j, \quad \sigma_{\min} \leq \sigma^{j+1} \leq \sigma_{\max} \quad (4.30)$$

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

$$\alpha^j = 1 + \beta \left(\frac{\epsilon}{\gamma^j} - 1 \right), \quad \alpha^j \leq 2, \quad (4.31)$$

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

$$\gamma^j = \|x^{j+1} - \hat{x}^{j+1}\|_{\infty}. \quad (4.32)$$

4.5.5 Event Detection and Location

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

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

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

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

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

4.5.6 Callback Functions

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

The prototype for a `pnes_master` callback function is

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

and the input and output arguments are described in Tables 4-12 through 4-14 and in the help for `pne_callback_default`. Each registered callback function is called in three different contexts, distinguished by the value of the first argument `k` as follows:

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

Table 4-12: Callback Input Arguments

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

^{*} See Table 4-14 for details of the continuation state.

[†] See Table 4-19 for details.

[‡] See `pne_detect_events` for details of the `events` field.

Table 4-13: Callback Output Arguments

name	description
<i>All are updated versions of the corresponding input arguments, see Table 4-12 for more details.</i>	
<code>cx</code>	current continuation state,* update values in <code>cx</code> such as <code>this_step</code> or <code>this_parm</code> if <code>s.rollback</code> is true
<code>nx</code>	next continuation state,* update values in this state if <code>s.rollback</code> is false
<code>s</code>	container struct with various flags, etc, with fields:
<code>.done</code>	callback may set this to request termination
<code>.done_msg</code>	callback may assign the reason for termination
<code>.warmstart</code>	callback may create this field to prepare for a subsequent warm-started call to <code>pnes_master</code> [†]
<code>.rollback</code>	callback can request a rollback step, even if it was not indicated by an event function [‡]
<code>.events</code>	<code>msg</code> field for a given event may be updated [§]
<code>.results</code>	updated version of results struct whose fields are to be included in the <code>output</code> struct returned by <code>pnes_master</code>

* See Table 4-14 for details of the continuation state.

† See Table 4-19 for details.

‡ In this case, the callback should also modify the step size or parameterization to be used for the re-try, by setting the `this_step` or `this_parm` fields in `cx`.

§ See `pne_detect_events` for details of the `events` field.

Table 4-14: Fields of Continuation State Struct

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

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

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

- `fcn` - function handle to the callback function
- `priority` - numerical value specifying callback priority,¹⁸ default = 20

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

¹⁸See `pne_register_callbacks` for details.

4.5.7 pnes_master

This function can be used to trace the parameterized solution curve with any of the available solvers¹⁹ by calling it as follows,

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

where the input and output arguments are described in Tables 4-15 and 4-16, respectively. Alternatively, the input arguments can be packaged as fields in a `problem` struct and passed in as a single argument, where the `opt` field is optional.

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

Table 4-15: Input Arguments for `pnes_master`

name	description
<code>fcn</code>	handle to function that evaluates the function $f(x)$ and its Jacobian $J(x)$ for a given value of x , with calling syntax: $f = \text{fcn}(x)$, or $[f, J] = \text{fcn}(x)$ where f is $n \times 1$, x is $(n + 1) \times 1$, and J is $n \times (n + 1)$.
<code>x0</code>	starting value of vector x
<code>opt</code>	optional options struct (all fields also optional), see Table 4-17 for details
<code>problem</code>	alternative, single argument input struct with fields corresponding to arguments above

¹⁹The current implementation includes only a single solver based on a predictor-corrector continuation method.

Table 4-16: Output Arguments for `pnes_master`[†]

name	description
<code>x</code>	solution vector
<code>f</code>	final function value, $f(x)$
<code>exitflag</code>	exit flag 1 – converged successfully ≤ 0 – solver-specific failure code
<code>output</code>	output struct with the following fields:
<code>corrector</code>	output return value from <code>nleqs_master</code> from final corrector run, see Table 4-10 for details
<code>iterations</code>	N , total number of continuation steps performed
<code>events</code>	struct array of size n_e of events detected, with the following fields:
<code>k</code>	continuation step at which event was located
<code>name</code>	name of detected event
<code>idx</code>	index(es) of critical element(s) in corresponding event function
<code>msg</code>	descriptive text detailing the event
<code>done_msg</code>	message describing cause of continuation termination
<code>steps</code>	$(N + 1)$ row vector of stepsizes taken at each continuation step
<code>lam_hat</code>	$(N + 1)$ row vector of $\hat{\lambda}$ values from prediction steps
<code>lam</code>	$(N + 1)$ row vector of λ values from correction steps
<code>max_lam</code>	maximum value of parameter λ (from <code>output.lam</code>)
<code>warmstart</code>	optional output with information needed for warm-starting an updated continuation problem, see Table 4-19 for details
<code>x_hat</code> [‡]	$n \times (N + 1)$ matrix of solution values from prediction steps
<code>x</code> [‡]	$n \times (N + 1)$ matrix of solution values from correction steps
(<i>others</i>)	depends on <code>opt.output_fcn</code> , a custom output function can add arbitrary fields to <code>output</code>
<code>jac</code>	final value of Jacobian matrix

[†] All output arguments are optional.

[‡] This field is created by the default output function and may not be present if using a custom output function defined by `opt.output_fcn`.

Table 4-17: Options for `pnes_master`

name	default	description
<code>alg</code>	'DEFAULT'	determines which solver to use*
<code>verbose</code>	0	amount of progress info to be printed 0 – print no progress info 1–5 – print increasing level of progress info
<code>nleqs_opt</code>	<i>empty</i>	options struct for <code>nleqs_master</code> used for corrector stage, see Table 4-11 for details
<code>solve_base</code>	1	0/1 flag that determines whether or not to run a corrector stage for initial solution point, <code>x0</code>
<code>parameterization</code>	3	choice of parameterization 1 – natural 2 – arc length 3 – pseudo arc length
<code>stop_at</code>	'NOSE'	determines stopping criterion 'NOSE' – stop when limit or nose point is reached 'FULL' – trace full continuation curve λ_{stop} – numeric, stop upon reaching target λ value λ_{stop}
<code>max_it</code>	2000	maximum number of continuation steps
<code>step</code>	0.05	continuation step size
<code>adapt_step</code>	0	toggle adaptive step size feature 0 – adaptive step size disabled 1 – adaptive step size enabled
<code>adapt_step_damping</code>	0.7	damping factor β from (4.31) for adaptive step sizing
<code>adapt_step_tol</code>	10^{-3}	tolerance ϵ from (4.31) for adaptive step sizing
<code>adapt_step_ws</code>	1	scale factor for default initial step size when warm-starting with adaptive step size enabled
<code>step_min</code>	10^{-4}	minimum allowed continuation step size, σ_{min} from (4.30)
<code>step_max</code>	0.2	maximum allowed continuation step size, σ_{max} from (4.30)
<code>default_event_tol</code>	10^{-3}	default tolerance for event functions
<code>target_lam_tol</code>	0	tolerance for target λ detection [†]
<code>nose_tol</code>	0	tolerance for nose point detection [†]
<code>events</code>	<i>empty</i>	cell array of specs for user-defined event functions [‡]
<code>callbacks</code>	<i>empty</i>	cell array of specs for user-defined callback functions [§]
<code>output_fcn</code>	<i>empty</i>	custom output function called by <code>pne_callback_default()</code>
<code>plot</code>	–	struct of options to control plotting of continuation curve by <code>pne_callback_default()</code> , see Table 4-18 for details
<code>warmstart</code>	<i>empty</i>	struct with information needed for warm-starting a continuation problem, see Table 4-19 for details

* Currently 'DEFAULT' is the only option.

† A value of 0 means use the value of `default_event_tol`.‡ Passed as `my_events` arg to `pne_register_events()`. For details see `help pne_register_events`.§ Passed as `my_callbacks` arg to `pne_register_callbacks()`. For details see `help pne_register_callbacks` and `help pne_callback_default`.

Table 4-18: Plot Options for `pnes_master`*

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

* Defines the fields for optional input `opt.plot`.

[†] Relevant field is indicated by the value of `opt.plot.xname`.

[‡] Relevant field is indicated by the value of `opt.plot.yname`.

[§] Can use `%d` as placeholder for index `idx` of quantity to plot.

Table 4-19: Warm-start Data for `pnes_master`*

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

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

4.5.8 PNE Example

The following code is a simplified version of the example included as `pne_ex1.m` in `<MPOM>lib/t`. It illustrates the use of `pnes_master` to solve a 2-dimensional parameterized nonlinear function of 3 variables.²⁰ Recall that x_3 , the last element of x , corresponds to the parameter λ .

$$f(x) = \begin{bmatrix} x_1 + x_2 + 6x_3 - 1 \\ -x_1^2 + x_2 + 5 \end{bmatrix} \quad (4.33)$$

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

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

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

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

x =

    2.0000
   -1.0000
    0
```

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

²⁰Based on a similar problem from <https://www.chilimath.com/lessons/advanced-algebra/systems-non-linear-equations/>.

```

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

```

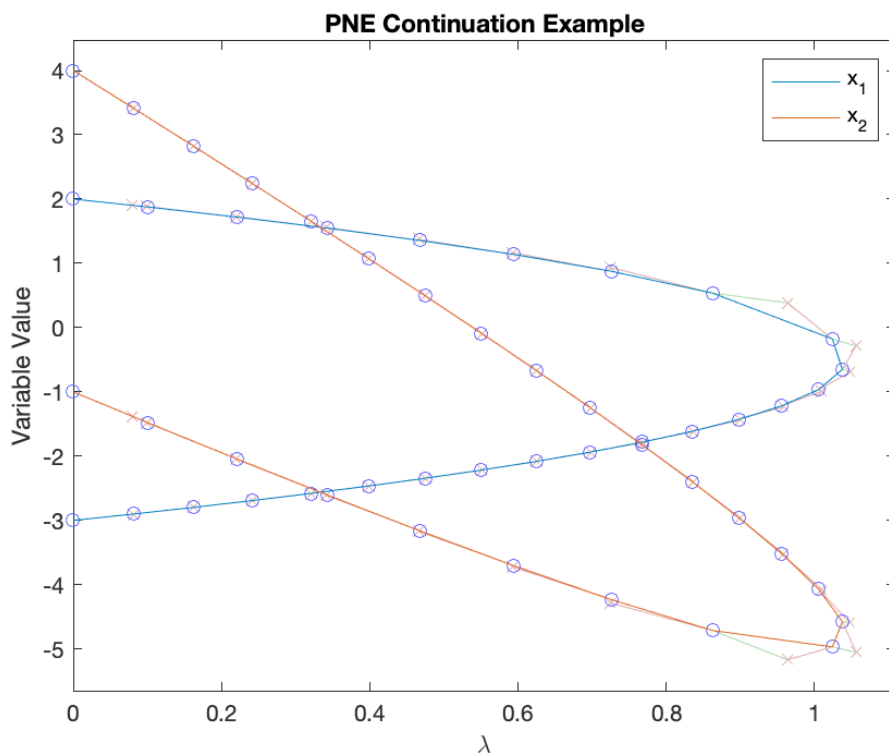


Figure 4-1: Continuation Curve for PNE Example

```

>> pne_ex1

MP-Opt-Model Version 4.2, 10-May-2024 -- Predictor/Corrector Continuation Method
step  0  :                               lambda = 0.000,  6 corrector steps
step  1  : PAL stepsize = 0.6             lambda = 0.081  2 corrector steps
step  2  : PAL stepsize = 0.6             lambda = 0.162  2 corrector steps
step  3  : PAL stepsize = 0.6             lambda = 0.241  2 corrector steps
step  4  : PAL stepsize = 0.6             lambda = 0.320  2 corrector steps
step  5  : PAL stepsize = 0.6             lambda = 0.398  2 corrector steps
step  6  : PAL stepsize = 0.6             lambda = 0.475  2 corrector steps
step  7  : PAL stepsize = 0.6             lambda = 0.551  2 corrector steps
step  8  : PAL stepsize = 0.6             lambda = 0.625  2 corrector steps
step  9  : PAL stepsize = 0.6             lambda = 0.697  2 corrector steps
step 10  : PAL stepsize = 0.6             lambda = 0.767  2 corrector steps
step 11  : PAL stepsize = 0.6             lambda = 0.835  2 corrector steps
step 12  : PAL stepsize = 0.6             lambda = 0.898  2 corrector steps
step 13  : PAL stepsize = 0.6             lambda = 0.956  2 corrector steps
step 14  : PAL stepsize = 0.6             lambda = 1.005  3 corrector steps
step 15  : PAL stepsize = 0.6             lambda = 1.038  3 corrector steps
step 16  : PAL stepsize = 0.6             lambda = 1.024  3 corrector steps
step 17  : PAL stepsize = 0.6             lambda = 0.863  3 corrector steps
step 18  : PAL stepsize = 0.6             lambda = 0.726  3 corrector steps
step 19  : PAL stepsize = 0.6             lambda = 0.595  3 corrector steps
step 20  : PAL stepsize = 0.6             lambda = 0.468  2 corrector steps
step 21  : PAL stepsize = 0.6             lambda = 0.343  2 corrector steps
step 22  : PAL stepsize = 0.6             lambda = 0.221  2 corrector steps
step 23  : PAL stepsize = 0.6             lambda = 0.100  2 corrector steps
step 24a : PAL stepsize = 0.6             lambda = -0.019 2 corrector steps ^ ROLLBACK
step 24  : NAT stepsize = 0.1             lambda = 0.000  3 corrector steps
CONTINUATION TERMINATION: Traced full continuation curve in 24 continuation steps

exitflag = 1
output.max_lam = 1.03783

x =
  2
 -1
  0

f =
      0
-6.4837e-13

jac =
  1  1  6
 -4  1  0

```

5 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`, `nlp_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, v1);  
om.add_var(name, N, v0, v1, vu);  
om.add_var(name, N, v0, v1, 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 set²¹, `N` is the number n of variables in the set, `v0` is the initial value of the variables, `v1` 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

²¹A set name must be a valid field name for a struct.

The inputs `v0`, `v1` 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`, `v1`, `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 5.7 for details on indexed named sets and the `idx_list` argument.

5.1.1 Variable Subsets

A key feature of MP-Opt-Model is that each set of constraints or costs can be defined in terms of the relevant variables only, as opposed to the entire 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

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

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 .²²

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

$$l_1 \leq A_1 x_1 \leq u_1 \tag{5.3}$$

$$l_2 \leq A_2 x_2 \tag{5.4}$$

$$A_3 x \leq u_3, \tag{5.5}$$

where x_1 and x_2 are as defined in Section 5.1.1 and x is the full 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 5.7 for details on indexed named sets and the `idx_list` argument.

²²The 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 $J^T(x)\lambda$ 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 **nlcon1**, **nlcon2**, and **nlcon3**, 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('nlcon1', ng1, 0, fcn1, hess1 {'v'});
om.add_nln_constraint('nlcon2', ng2, 1, fcn2, hess2, {'u', 'w'});
om.add_nln_constraint('nlcon3', ng3, 0, fcn3, hess3);

```

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

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

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

5.3 Adding Costs

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

5.3.1 Quadratic Costs

```
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:

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

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 .²³

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

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

where $k_i = k$ for all i if k is scalar.

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

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

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

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

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

²³The 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 $n_x \times 1$ gradient of f , and `d2f` is the $n_x \times n_x$ Hessian H , where n_x is the number of elements in x .

The first input argument `x` takes one of two forms. If the constraint set is added with `varsets` empty or missing, then `x` will be the full 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` \rightarrow `x` = $\{v\}$
- `my_cost_fcn2` \rightarrow `x` = $\{u, w\}$
- `my_cost_fcn3` \rightarrow `x` = $[u; v; w]$

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

5.4 Solving the Model

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

After all variables, constraints and costs have been added to the model, the 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`, `nlp_master`, `nleqs_master`, or `pnes_master`. Note that one of the equation solvers is chosen if the model has no costs and no inequality constraints. In this case, if the number of variables is equal to the number of equality constraints, `mplinsolve` or `nleqs_master` is selected. If the number of variables is one more than the number of constraints `pnes_master` is chosen.

The results are stored in the `soln` field (see Section 5.5.5) of the MP-Opt-Model object and can be returned in the optional output arguments. The input options struct `opt`, summarized in Tables 5-1 and 5-2, is optional, as are all of its fields. For details on the return values see the descriptions of the individual solver functions in Sections 4.1, 4.2, 4.3, 4.4, and 4.5. For linear equations, the `solver` and `opt` arguments for `mplinsolve`, described in Section 4.1 of the [MIPS User's Manual](#), can be provided in the respective fields of `opt.leq.opt`.

Table 5-1: Options for solve

name	default	description
<code>alg</code>	'DEFAULT'	determines which solver to use, see Table 5-2
<code>verbose</code>	1	amount of progress info to be printed 0 – print no progress info 1–5 – print increasing level of progress info
<code>parse_soln</code>	0	flag that specifies whether or not to call the <code>parse_soln</code> method and place the return values in <code>om.soln</code>
<code>x0</code>	<i>empty</i>	optional initial value of x , overrides value stored in model, (<i>ignored by some solvers</i>)
<i>Additional Options for Specific Problem Types</i>		
LP/QP		see Table 4-3
MILP/MIQP		see Table 4-5
NLP		see Table 4-8
LEQ		see Section 4.1 of the MIPS User's Manual
<code>leq_opt.solver</code>	''	see <code>help mplinsolve</code> , input argument <code>solver</code>
<code>leq_opt.opt</code>	<i>empty</i>	see <code>help mplinsolve</code> , input argument <code>opt</code>
NLEQ		see Table 4-11
PNE		see Table 4-17

Table 5-2: Values for `alg` Option to solve

alg value	problem type(s)	description
'DEFAULT'	<i>all</i>	automatic, depends on problem type, uses first available of:
	LP	Gurobi, CPLEX, MOSEK, <code>linprog</code> , [¶] GLPK, BPMPD, MIPS
	QP	Gurobi, CPLEX, MOSEK, <code>quadprog</code> , [¶] BPMPD, MIPS
	MILP	Gurobi, CPLEX, MOSEK, <code>intlinprog</code> , GLPK
	MIQP	Gurobi, CPLEX, MOSEK
	NLP	MIPS
	MINLP	Artelys Knitro (not yet implemented)
	LEQ	built-in backslash operator
	NLEQ	Newton's method
	PNE	predictor/corrector continuation method
'BPMPD'	LP, QP	BPMPD [*]
'CLP'	LP, QP	CLP [*]
'CPLEX'	LP, QP, MILP, MIQP	CPLEX [*]
'FD'	NLEQ	fast-decoupled Newton's method [†]
'FMINCON'	NLP	MATLAB Opt Toolbox, <code>fmincon</code> [*]
'FSOLVE'	NLEQ	MATLAB Opt Toolbox, <code>fsolve</code> [§]
'GLPK'	LP, MILP	GLPK [*] (<i>LP only</i>)
'GS'	NLEQ	Gauss-Seidel method [‡]
'GUROBI'	LP, QP, MILP, MIQP	Gurobi [*]
'IPOPT'	LP, QP, NLP	IPOPT [*]
'KNITRO'	NLP, MINLP	Artelys Knitro [*]
'MIPS'	LP, QP, NLP	MIPS, MATPOWER Interior Point Solver
'MOSEK'	LP, QP, MILP, MIQP	MOSEK [*]
'NEWTON'	NLEQ	Newton's method
'OSQP'	LP, QP	OSQP [*]
'OT'	LP, QP, MILP	MATLAB Opt Toolbox, <code>quadprog</code> , <code>linprog</code> , <code>intlinprog</code>

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

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

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

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

[¶] If running on MATLAB.

5.5 Accessing the Model

5.5.1 Indexing

For each 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 5-3.

Table 5-3: Example Indexing Data

field	value	description
<code>vv.N.u</code>	n_u	number of u variables
<code>vv.N.v</code>	n_v	number of v variables
<code>vv.N.w</code>	n_w	number of w variables
<code>vv.i1.u</code>	1	starting index of u in full x
<code>vv.i1.v</code>	$n_u + 1$	starting index of v in full x
<code>vv.i1.w</code>	$n_u + n_v + 1$	starting index of w in full x
<code>vv.iN.u</code>	n_u	ending index of u in full x
<code>vv.iN.v</code>	$n_u + n_v$	ending index of v in full x
<code>vv.iN.w</code>	$n_u + n_v + n_w$	ending index of w in full x

`get_idx`

```
[idx1, idx2, ...] = om.get_idx(set_type1, set_type2, ...);
vv = om.get_idx('var');
[l1, nne, nni] = om.get_idx('lin', 'nle', 'nli');

vv = om.get_idx()
[vv, l1] = om.get_idx()
[vv, l1, nne] = om.get_idx()
[vv, l1, nne, nni] = om.get_idx()
[vv, l1, nne, nni, qq] = om.get_idx()
[vv, l1, 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 5-4. When called without input arguments, the indexing structs are simply returned in the order listed in the table.

Table 5-4: Valid Set Types

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

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

For the example model built in Sections 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 `nlncn3` constraints with the following code.

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

getN

```
N = om.getN(set_type)
N = om.getN(set_type, name)
N = om.getN(set_type, name, idx_list)
```

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

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

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

set_type_idx_map

```
s = om.set_type_idx_map(set_type, idxs)
s = om.set_type_idx_map(set_type)
s = om.set_type_idx_map(set_type, idxs)
```

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

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

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

```

>> lin38 = om.set_type_idx_map('lin', 38)

lin38 =

  struct with fields:

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

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

var15 =

  struct with fields:

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

>> var23 = s(2)

var23 =

  struct with fields:

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

```

describe_idx

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

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

Consider an example in which element 38 of the linear constraints corresponds to the 11th row of **lincon3** and elements 15 and 23 of the 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, v1, vu] = om.params_var()
[v0, v1, vu] = om.params_var(name)
[v0, v1, vu] = om.params_var(name, idx_list)
[v0, v1, vu, vt] = params_var(...)

```

The `params_var` method returns the initial value `v0`, lower bound `v1` 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 5.7 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

$$l \leq Ax \leq u. \quad (5.16)$$

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

params_nln_constraint

```
N = om.params_nln_constraint(iseq, name)
N = om.params_nln_constraint(iseq, name, idx_list)
[N, fcn] = om.params_nln_constraint(...)
[N, fcn, hess] = om.params_nln_constraint(...)
[N, fcn, hess, vs] = om.params_nln_constraint(...)
[N, fcn, hess, vs, include] = om.params_nln_constraint(...)
```

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

indexed named sets specified by `name` and `idx_list`. The `iseq` input should be set to 1 for equality constraints and to 0 for inequality constraints.

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

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

`eval_lin_constraint`

```
Ax_u = om.eval_lin_constraint(x)
Ax_u = om.eval_lin_constraint(x, name)
Ax_u = om.eval_lin_constraint(x, name, idx_list)
[Ax_u, l_Ax] = om.eval_lin_constraint(...)
[Ax_u, l_Ax, A] = om.eval_lin_constraint(...)
```

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

Examples:

```
[Ax_u, l_Ax, A] = om.eval_lin_constraint(x);
```

`eval_nln_constraint`

```
g = om.eval_nln_constraint(x, iseq)
g = om.eval_nln_constraint(x, iseq, name)
g = om.eval_nln_constraint(x, iseq, name, idx_list)
[g, dg] = om.eval_nln_constraint(...)
```

Builds the nonlinear equality constraints $g(x)$ or inequality constraints $h(x)$ and optionally their gradients for the full set of constraints or an individual named subset for a given value of the 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

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

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

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

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

`params_nln_cost`

```
[N, fcn] = om.params_nln_cost(name)
[N, fcn] = om.params_nln_cost(name, idx_list)
[N, fcn, vs] = om.params_nln_cost(...)
```

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

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

`eval_quad_cost`

```
f = om.eval_quad_cost(x ...)
[f, df] = om.eval_quad_cost(x ...)
[f, df, d2f] = om.eval_quad_cost(x ...)
[f, df, d2f] = om.eval_quad_cost(x, name)
[f, df, d2f] = om.eval_quad_cost(x, name, idx_list)
```

The `eval_quad_cost` method evaluates the cost function and its derivatives for an individual named set or the full set of quadratic costs for a given value of the 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 5.7 for details on indexed named sets and the `idx_list` argument.

`eval_nln_cost`

```
f = om.eval_nln_cost(x)
[f, df] = om.eval_nln_cost(x)
[f, df, d2f] = om.eval_nln_cost(x)
[f, df, d2f] = om.eval_nln_cost(x, name)
[f, df, d2f] = om.eval_nln_cost(x, name, idx_list)
```

The `eval_nln_cost` method evaluates the cost function and its derivatives for an individual named set or the full set of general nonlinear costs for a given value of the 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 5.7 for details on indexed named sets and the `idx_list` argument.

5.5.5 Model Solution

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

`is_solved`

```
TorF = om.is_solved()
```

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

`get_soln`

```
vals = om.get_soln(set_type, name)
vals = om.get_soln(set_type, name, idx)
vals = om.get_soln(set_type, tags, name)
vals = om.get_soln(set_type, tags, name, idx)
```

The `get_soln` method can be used to extract solved results for a given named set of variables, constraints or costs. The input arguments for `get_soln` are summarized in Table 5-6 and Table 5-7. The variable number of output arguments correspond to the `tags` input. If `tags` is empty or not specified, the calling context will define the number of outputs, returned in order of default tags for the specified `set_type`.

Table 5-5: Model Solution

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

* Objective function value for optimization problems, constraint function value for sets of equations.

[†] Only available after calling `parse_soln(true)` or calling `solve()` with the `opt.parse_soln` option set to 1.

[‡] See Table 5-8 for details.

Examples:

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

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

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

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

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

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

Table 5-6: Inputs for `get_soln`

name	default	description
set_type	<i>required</i>	one of the following, specifying the type of set 'var' – variables 'lin' – linear constraints 'nle' – nonlinear equality constraints 'nli' – nonlinear inequality constraints 'nlc' – nonlinear costs 'qdc' – quadratic costs
tags	<i>depends</i>	char array or cell array of char arrays specifying the desired output(s) [†]
name	<i>required</i>	char array specifying the name of the set
idx	<i>empty</i>	cell array specifying the indices of the set

[†] Valid values and defaults for `tags` depend on `set_type`. See Table 5-7 for details.

Table 5-7: Values of `tags` input to `get_soln`

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

[†] For LEQ problems only.

[‡] For 'qdc', $f(x)$ can return be a vector.

parse_soln

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

The `parse_soln` method returns a struct of parsed solution vector and shadow price values for each named set of variables and constraints. The returned `ps` (parsed solution) struct has the format shown in Table 5-8, where each of the terminal elements is a struct with fields corresponding to the respective named sets.

Table 5-8: Output of `parse_soln`

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

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

If the optional `stash` input argument is present and true, the fields of the return struct are copied to `om.soln` where they are available with others listed in Table 5-5.

has_parsed_soln

```
TorF = om.has_parsed_soln()
```

The `has_parsed_soln` method returns 1 if the model has a parsed solution available in `om.soln`, 0 otherwise.

5.6 Modifying the Model

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

`set_params`

```
om.set_params(set_type, name, params, vals)
om.set_params(set_type, name, idx, params, vals)
```

The `set_params` method, inputs summarized in Table 5-9, can be used to modify any of the parameters associated with an existing variable, cost or constraint set.

Examples:

```
om.set_params('var', 'Pg', 'v0', Pg0);
om.set_params('lin', 'y', {2,3}, {'l', 'u'}, {1, u});
om.set_params('nle', 'Pmis', 'all', {N, @fcn, @hess, vs});
```

Table 5-9: Inputs for `set_params`

name	description
<code>set_type</code>	one of the following, specifying the type of set, with the corresponding valid parameter names ' <code>var</code> ' – variables: $N, v_0^\dagger, v_1^\dagger, v_u^\dagger, v_t^\dagger$ ' <code>lin</code> ' – linear constraints: A, l, u, v_s^\dagger ' <code>nle</code> ' – nonlinear equality constraints: $N, fcn, hess, v_s^\dagger$ ' <code>nli</code> ' – nonlinear inequality constraints: $N, fcn, hess, v_s^\dagger$ ' <code>nlc</code> ' – nonlinear costs: N, fcn, v_s^\dagger ' <code>qdc</code> ' – quadratic costs: $Q, c^\dagger, k^\dagger, v_s^\dagger$
<code>name</code>	char array specifying the name of the set
<code>idx[‡]</code>	cell array specifying the indices of the set
<code>params</code>	one of the following: ' <code>all</code> ' – indicates that <code>vals</code> is a cell array of values whose elements correspond to the input parameters of the respective <code>add_*</code> method <i>char array</i> – name of parameter to modify <i>cell array</i> – names of parameters to modify
<code>vals</code>	new value or cell array of new values corresponding the parameter name(s) specified in <code>params</code>

[†] Optional when `params = 'all'`.

[‡] The `idx` argument is optional.

5.7 Indexed Sets

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

In this case, we can 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.²⁴

`init_indexed_name`

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

Examples:

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

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

Examples:

```
for d = 1:7  
    om.add_var('y', {d}, ny(d), y0{d}, y1{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}, z1{d, h}, zu{d, h});  
    end  
end
```

²⁴The 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_`.²⁵

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 5-3. 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}});
```

5.8 Miscellaneous Methods

5.8.1 Public Methods

`copy`

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

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

²⁵Currently, `eval_nln_constraint` and `eval_nln_constraint_hess` are only implemented for the full aggregate set of constraints and do not yet support evaluation of individual constraint sets.

display

```
om
```

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

display_soln

```
om.display_soln()  
om.display_soln(set_type)  
om.display_soln(set_type, name)  
om.display_soln(set_type, name, idx)
```

The `display_soln` method displays the model solution, including values, bounds and shadow prices for variables and linear constraints, values and shadow prices for nonlinear constraints, and individual cost components. Results are displayed for each `set_type` or specified `set_type` and for each named/indexed set or a specified `name/idx`.

get_userdata

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

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

is_mixed_integer

```
TorF = om.is_mixed_integer()
```

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

problem_type

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

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

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

The `prob_type` string is one of the following:

- 'LEQ' – linear equation
- 'NLEQ' – nonlinear equation
- 'LP' – linear program
- 'QP' – quadratic program
- 'NLP' – nonlinear program
- 'MILP' – mixed-integer linear program
- 'MIQP' – mixed-integer quadratic program
- 'MINLP' – mixed-integer nonlinear program²⁶

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

`varsets_cell2struct`

```
varsets = om.varsets_cell2struct(varsets)
```

Converts variable subset `varsets` from a cell array to a struct array, if necessary.

`varsets_idx`

```
k = om.varsets_idx(varsets)
```

Returns a vector of indices into the full 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`.

²⁶MP-Opt-Model does not yet implement solving MINLP problems.

`varsets_x`

```
x = om.varsets_x(x, varsets)
x = om.varsets_x(x, varsets, 'vector')
```

Returns a cell array of sub-vectors of x specified by `varsets`, or the full 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.

5.8.2 Private Methods

`def_set_types`

```
om.def_set_types()
```

The `def_set_types` method is a *private* method that assigns a struct to the `set_types` property of the object. The fields of the struct correspond to the valid set types listed in Table 5-4 and the values are labels used by the `display` method.

`init_set_types`

```
om.init_set_types()
```

Initializes the base data structures for each set type.

5.9 MATPOWER Index Manager Base Class – `mp_idx_manager`

Most of the functionality of the `opt_model` class related to managing the indexing of the various set types is inherited from the MATPOWER Index Manager base class named `mp_idx_manager`. The properties and methods implemented in this base class and inherited or overridden by `opt_model` are listed in Table 5-10.

The MATPOWER Index Manager base class initializes and manages the data that is common across all set types. Table 5-11 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 5-10: MATPOWER Index Manager (`mp_idx_manager`) Properties and Methods

name	description
<i>Properties</i>	
<code>set_types</code>	struct whose fields define the valid set types*
<code>userdata</code>	struct for storing arbitrary user-defined data
<i>Public Methods</i>	
<code>mp_idx_manager</code>	constructor for <code>mp_idx_manager</code> class
<code>copy</code>	makes a copy of an existing <code>mp_idx_manager</code> object
<code>describe_idx</code>	describes indices of a given set type, e.g. variable 361 corresponds to <code>w(68)</code> , see Section 5.5.1
<code>display_set</code>	displays indexing for a particular set type, typically called by <code>display</code>
<code>get</code>	access (possibly nested) fields of the object
<code>get_idx</code>	returns index structure(s) for specified set type(s), with starting/ending indices and number of elements for each named (and optionally indexed) block
<code>get_userdata</code>	retrieves values of user data stored in the object
<code>getN</code>	returns the number of elements of any given set type [†]
<code>init_indexed_name</code>	initializes dimensions for a particular indexed named set
<code>set_type_idx_map</code>	maps indices of a given set type, e.g. variable 361 corresponds to <code>w(68)</code> , see Section 5.5.1
<i>Private Methods[‡]</i>	
<code>add_named_set</code>	adds indexing information for new instance of a given set type
<code>init_set_types</code>	initializes the data structures for each set type
<code>valid_named_set_type</code>	returns label for given named set type if valid, empty otherwise

* This value is initialized automatically by the `def_set_types` method of the subclass.

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

[‡] For internal use only.

Table 5-11: MATPOWER Index Manager (`mp_idx_manager`) Object Structure

name	description
<code>obj</code>	
<code>.set_types</code>	struct whose fields define the valid set types
<code>.var</code>	data for 'var' set type, e.g. variable sets that make up the full optimization variable x
<code>.idx</code>	
<code>.i1</code>	starting index within x
<code>.iN</code>	ending index within x
<code>.N</code>	number of elements in this variable set
<code>.N</code>	total number of elements in x
<code>.NS</code>	number of variable sets or named blocks
<code>.data</code>	additional set-type-specific data for each block [†]
<code>.order</code>	struct array of names/indices for variable blocks in the order they appear in x
<code>.name</code>	name of the block, e.g. z
<code>.idx</code>	indices for name, $\{2,3\} \rightarrow z(2,3)$
<code>.<other-set-types></code>	with structure identical to <code>var</code>
<code>.userdata</code>	struct for storing arbitrary user-defined data

[†] For the 'var' set type in `opt_model`, this is a struct with fields `v0`, `v1`, `vu`, and `vt` for storing initial value, lower and upper bounds, and variable type. For other set types

5.10 Reference

5.10.1 Properties

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

Table 5-12: `opt_model` Properties

name	description
<code>set_types</code> [†]	struct whose fields define the valid set types [*]
<code>prob_type</code>	used to cache return value of <code>problem_type</code> method
<code>var</code> [‡]	data for 'var' set type, variables
<code>lin</code> [‡]	data for 'lin' set type, linear constraints
<code>nle</code> [‡]	data for 'nle' set type, nonlinear equality constraints
<code>nli</code> [‡]	data for 'nli' set type, nonlinear inequality constraints
<code>qdc</code> [‡]	data for 'qdc' set type, quadratic costs
<code>nlc</code> [‡]	data for 'nlc' set type, general nonlinear costs
<code>userdata</code> [†]	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 5-11 for details of the structure of this field. The only difference between set types is the structure of the `data` sub-field.

5.10.2 Methods

Table 5-13: `opt_model` Methods

name	description
<i>Public Methods</i>	
<code>add_lin_constraint</code>	add linear constraint set, see Section 5.2.1
<code>add_nln_constraint</code>	add general nonlinear constraint set, see Section 5.2.2
<code>add_nln_cost</code>	add general nonlinear cost set, see Section 5.3.2
<code>add_quad_cost</code>	add quadratic cost set, see Section 5.3.1
<code>add_var</code>	add variable set, see Section 5.1
<code>display</code>	displays variable, constraint and cost sets, see Section 5.8.1
<code>display_soln</code>	displays model solution, see Section 5.8.1
<code>eval_lin_constraint</code>	computes linear constraint values, see Section 5.5.3
<code>eval_nln_constraint</code>	builds full set of nonlinear equality or inequality constraints and their gradients, see Section 5.5.3
<code>eval_nln_constraint_hess</code>	builds Hessian for full set of nonlinear equality or inequality constraints, see Section 5.5.3
<code>eval_nln_cost</code>	evaluates nonlinear cost function and its derivatives [‡] see Section 5.5.4
<code>eval_quad_cost</code>	evaluates quadratic cost function and its derivatives [‡] see Section 5.5.4
<code>get_soln</code>	returns named/indexed results for solved model
<code>has_parsed_soln</code>	returns 1 if model has a parsed solution available, 0 otherwise
<code>is_mixed_integer</code>	returns 1 if any of the variables are binary or integer, 0 otherwise
<code>is_solved</code>	returns 1 if model has been solved, 0 otherwise
<code>params_lin_constraint</code>	assembles and returns parameters for linear constraints [‡]
<code>params_nln_constraint</code>	assembles and returns parameters for nonlinear constraints [‡]
<code>params_nln_cost</code>	assembles and returns parameters for general nonlinear costs [‡]
<code>params_quad_cost</code>	assembles and returns parameters for quadratic costs [‡]
<code>params_var</code>	assembles and returns initial values, bounds, types for variables [‡]
<code>parse_soln</code>	returns struct of all named solution vectors and shadow prices
<code>problem_type</code>	type of mathematical program represented by current model
<code>solve</code>	solves the model, see Section 5.4
<code>varsets_cell2struct</code>	converts variable subset <code>varsets</code> from cell array to struct array
<code>varsets_idx</code>	returns vector of indices into x corresponding to <code>varsets</code>
<code>varsets_len</code>	returns number of elements in sub-vector specified by <code>varsets</code>
<code>varsets_x</code>	returns cell array of sub-vectors of x specified by <code>varsets</code>
<i>Inherited Public Methods</i> [†]	
<code>copy</code> , <code>describe_idx</code> , <code>display_set</code> , <code>get</code> , <code>get_idx</code> , <code>get_userdata</code> , <code>getN</code> , <code>init_indexed_name</code> , <code>set_type_idx_map</code>	
<i>Private Methods</i> [*]	
<code>add_named_set</code> [§]	adds information for new instance of a given set type
<code>def_set_types</code>	initializes the <code>set_types</code> property
<code>init_set_types</code> [§]	initializes the data structures for each set type
<code>valid_named_set_type</code> [†]	returns label for given named set type if valid, empty otherwise

* For internal use only.

† Inherited from MATPOWER Index Manager base class `mp_idx_manager`, see Table 5-10.

‡ 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 `convert_lin_constraint`

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

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

$$l \leq Ax \leq u \quad (6.1)$$

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

$$A_e x = b_e \quad (6.2)$$

$$A_i x \leq b_i \quad (6.3)$$

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

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

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

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

6.2 `convert_lin_constraint_multipliers`

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

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

6.3 `have_fcn`

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

6.4 `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.

6.5 `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 controlled via the optional options struct `opt`, including the possible checking of valid field names.

6.6 Private Feature Detection Functions

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

6.6.1 `have_feature_bpmpd`

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

6.6.2 `have_feature_catchme`

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

6.6.3 `have_feature_clp`

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

6.6.4 `have_feature_opti_clp`

This function implements the `'opti_clp'` tag for `have_feature` to detect the version of CLP distributed with OPTI Toolbox²⁷ [15]. See also Appendix B.2.

6.6.5 `have_feature_cplex`

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

6.6.6 `have_feature_evalc`

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

6.6.7 `have_feature_fmincon`

This function implements the `'fmincon'` tag for `have_feature` to detect availability/version of `fmincon`, solver from the MATLAB Optimization Toolbox. See also Appendix B.9.

²⁷The OPTI Toolbox is available from <https://www.inverseproblem.co.nz/OPTI/>.

6.6.8 `have_feature_fmincon_ipm`

This function implements the `'fmincon_ipm'` tag for `have_feature` to detect availability/version of `fmincon` with interior point solver from the MATLAB Optimization Toolbox 4.x and later. See also Appendix [B.9](#).

6.6.9 `have_feature_fsolve`

This function implements the `'fsolve'` tag for `have_feature` to detect availability/version of `fsolve`, nonlinear equation solver from the MATLAB Optimization Toolbox or GNU Octave. See also Appendix [B.9](#).

6.6.10 `have_feature_glpk`

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

6.6.11 `have_feature_gurobi`

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

6.6.12 `have_feature_intlinprog`

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

6.6.13 `have_feature_ipopt`

This function implements the `'ipopt'` tag for `have_feature` to detect availability/version of IPOPT, a nonlinear programming solver from COIN-OR. See also Appendix [B.6](#).

6.6.14 `have_feature_ipopt_auxdata`

This function implements the `'ipopt_auxdata'` tag for `have_feature` to detect support for `ipopt_auxdata()`, required by IPOPT 3.11 and later. See also Appendix [B.6](#).

6.6.15 `have_feature_isequaln`

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

6.6.16 `have_feature_knitro`

This function implements the 'knitro' tag for `have_feature` to detect availability/version of Artelys Knitro, a nonlinear programming solver. See also Appendix [B.7](#).

6.6.17 `have_feature_knitromatlab`

This function implements the 'knitromatlab' tag for `have_feature` to detect availability/version of Artelys Knitro 9.0.0 and later. See also Appendix [B.7](#).

6.6.18 `have_feature_ktrlink`

This function implements the 'ktrlink' tag for `have_feature` to detect availability/version of Artelys Knitro prior to version 9.0.0, which required the MATLAB Optimization Toolbox. See also Appendix [B.7](#).

6.6.19 `have_feature_linprog`

This function implements the 'linprog' tag for `have_feature` to detect availability/version of `linprog`, LP solver from the MATLAB Optimization Toolbox. See also Appendix [B.9](#).

6.6.20 `have_feature_linprog_ds`

This function implements the 'linprog_ds' tag for `have_feature` to detect availability/version of `linprog` with support for the dual simplex method, from the MATLAB Optimization Toolbox 7.1 (R2014b) and later. See also Appendix [B.9](#).

6.6.21 `have_feature_mosek`

This function implements the 'mosek' tag for `have_feature` to detect availability/version of MOSEK, LP/QP/MILP/MIQP solver. See also Appendix [B.8](#).

6.6.22 `have_feature_optim`

This function implements the 'optim' tag for `have_feature` to detect availability/version of the Optimization Toolbox. See also Appendix B.9.

6.6.23 `have_feature_optimoptions`

This function implements the 'optimoptions' tag for `have_feature` to detect support for `optimoptions`, option setting function for the MATLAB Optimization Toolbox 6.3 and later. See also Appendix B.9.

6.6.24 `have_feature_osqp`

This function implements the 'osqp' tag for `have_feature` to detect availability/version of OSQP, Operator Splitting Quadratic Program solver. See also Appendix B.10.

6.6.25 `have_feature_quadprog`

This function implements the 'quadprog' tag for `have_feature` to detect detect availability/version of `quadprog`, QP solver from the MATLAB Optimization Toolbox. See also Appendix B.9.

6.6.26 `have_feature_quadprog_ls`

This function implements the 'quadprog_ls' tag for `have_feature` to detect availability/version of `quadprog` with support for the large-scale interior point convex solver, from the MATLAB Optimization Toolbox 6.x and later. See also Appendix B.9.

6.6.27 `have_feature_sdpt3`

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

6.6.28 `have_feature_sedumi`

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

6.6.29 `have_feature_yalmip`

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

6.7 MATPOWER-related Functions

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

6.7.1 `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.

6.7.2 `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.

6.7.3 `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 applicable²⁸ to the specific problem class, based on the following precedence: Gurobi, CPLEX, MOSEK, Optimization Toolbox, GLPK, BPMPD, MIPS.

6.7.4 mpopt2pneopt

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

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

²⁸GLPK 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).

7 Acknowledgments

The authors would like to acknowledge the support of the research grants and contracts that have contributed directly and indirectly to the development of MP-Opt-Model. This includes funding from the Power Systems Engineering Research Center (PSERC), the U.S. Department of Energy,²⁹ and the National Science Foundation.³⁰

The authors would also like to explicitly thank and acknowledge Shirirang Abhyankar and Alexander Flueck for their contributions to the continuation power flow code and documentation in [MATPOWER](#) upon which the predictor-corrector continuation method for parameterized nonlinear equations in MP-Opt-Model is based.

²⁹Supported in part by the Consortium for Electric Reliability Technology Solutions (CERTS) and the Office of Electricity Delivery and Energy Reliability, Transmission Reliability Program of the U.S. Department of Energy under the National Energy Technology Laboratory Cooperative Agreement No. DE-FC26-09NT43321.

³⁰This material is based upon work supported in part by the National Science Foundation under Grant Nos. 0532744, 1642341 and 1931421. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

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

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

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

name	description
<code>AUTHORS</code>	list of authors and contributors
<code>CHANGES</code>	MP-Opt-Model change history
<code>CITATION</code>	info on how to cite MP-Opt-Model
<code>CONTRIBUTING.md</code>	notes on how to contribute to the MP-Opt-Model project
<code>LICENSE</code>	MP-Opt-Model license (3-clause BSD license)
<code>README.md</code>	basic introduction to MP-Opt-Model
<code>docs/</code>	
<code>MP-Opt-Model-manual.pdf</code>	MP-Opt-Model User's Manual
<code>src/MP-Opt-Model-manual/</code>	
<code>MP-Opt-Model-manual.tex</code>	LaTeX source for MP-Opt-Model User's Manual
<code>lib/</code>	MP-Opt-Model software (see Tables A-2 , A-5 , A-6 and A-7)
<code>t/</code>	MP-Opt-Model tests (see Table A-9)

Table A-2: Solver Functions

name	description
<code>miqps_master</code>	Mixed-Integer Quadratic Program Solver wrapper function, provides a unified interface for various MIQP/MILP solvers
<code>miqps_cplex</code>	MIQP/MILP solver API implementation for CPLEX (<code>cplexmiqp</code> and <code>cplexmilp</code>) [†]
<code>miqps_glpk</code>	MILP solver API implementation for GLPK [†]
<code>miqps_gurobi</code>	MIQP/MILP solver API implementation for Gurobi [†]
<code>miqps_mosek</code>	MIQP/MILP solver API implementation for MOSEK (<code>mosekopt</code>) [†]
<code>miqps_ot</code>	QP/MILP solver API implementation for MATLAB Opt Toolbox's <code>intlinprog</code> , <code>quadprog</code> , <code>linprog</code>
<code>nleqs_master</code>	Nonlinear Equation Solver wrapper function, provides a unified interface for various nonlinear equation (NLEQ) solvers
<code>nleqs_core</code>	core NLEQ solver API implementation with arbitrary update function, used to implement <code>nleqs_gauss_seidel</code> and <code>nleqs_newton</code>
<code>nleqs_fd_newton</code>	NLEQ solver API implementation for fast-decoupled Newton's method solver
<code>nleqs_fsolve</code>	NLEQ solver API implementation for <code>fsolve</code>
<code>nleqs_gauss_seidel</code>	NLEQ solver API implementation for Gauss-Seidel method solver
<code>nleqs_newton</code>	NLEQ solver API implementation for Newton's method solver
<code>nlps_master</code>	Nonlinear Program Solver wrapper function, provides a unified interface for various NLP solvers
<code>nlps_fmincon</code>	NLP solver API implementation for MATLAB Opt Toolbox's <code>fmincon</code>
<code>nlps_ipopt</code>	NLP solver API implementation for IPOPT-based solver [†]
<code>nlps_knitro</code>	NLP solver API implementation for Artelys Knitro-based solver [†]
<code>qps_master</code>	Quadratic Program Solver wrapper function, provides a unified interface for various QP/LP solvers
<code>qps_bpmpd</code>	QP/LP solver API implementation for BPMPD_MEX [†]
<code>qps_clp</code>	QP/LP solver API implementation for CLP [†]
<code>qps_cplex</code>	QP/LP solver API implementation for CPLEX (<code>cplexqp</code> and <code>cplexlp</code>) [†]
<code>qps_glpk</code>	QP/LP solver API implementation for GLPK [†]
<code>qps_gurobi</code>	QP/LP solver API implementation for Gurobi [†]
<code>qps_ipopt</code>	QP/LP solver API implementation for IPOPT-based solver [†]
<code>qps_mosek</code>	QP/LP solver API implementation for MOSEK (<code>mosekopt</code>) [†]
<code>qps_osqp</code>	QP/LP solver API implementation for OSQP [†]
<code>qps_ot</code>	QP/LP solver API implementation for MATLAB Opt Toolbox's <code>quadprog</code> , <code>linprog</code>
<code>pnes_master</code>	Parameterized Nonlinear Equation Solver wrapper function, provides a unified interface for parameterized nonlinear equation (PNE) solvers
<i>deprecated functions</i>	
<code>miqps_matpower</code>	use <code>miqps_master</code> instead
<code>qps_matpower</code>	use <code>qps_master</code> instead

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

Table A-3: PNE Implementation Functions*

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

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

Table A-4: Solver Options, etc.

name	description
<code>clp_options</code>	default options for CLP solver [†]
<code>cplex_options</code>	default options for CPLEX solver [†]
<code>glpk_options</code>	default options for GLPK solver [†]
<code>gurobi_options</code>	default options for Gurobi solver [†]
<code>gurobiver</code>	prints version information for Gurobi/Gurobi.MEX
<code>ipopt_options</code>	default options for IPOPT solver [†]
<code>mosek_options</code>	default options for MOSEK solver [†]
<code>mosek_symbcon</code>	symbolic constants to use for MOSEK solver options [†]
<code>osqp_options</code>	default options for OSQP solver [†]
<code>osqpver</code>	prints version information for OSQP

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

Table A-5: Optimization Model Class

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

[†] 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-6: MATPOWER Index Manager Class

name	description
<code>@mp_idx_manager/</code>	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).
<code>mp_idx_manager</code>	constructor for the <code>mp_idx_manager</code> class
<code>add_named_set[†]</code>	add named subset of a particular type to the object
<code>describe_idx</code>	describes indices of a given set type, e.g. variable 361 corresponds to <code>w(68)</code>
<code>get_idx</code>	returns index structure(s) for specified set type(s), with starting/ending indices and number of elements for each named (and optionally indexed) block
<code>get_userdata</code>	retrieves values of user data stored in the object
<code>get</code>	returns the value of a field of the object
<code>getN</code>	returns the number of elements of any given set type [‡]
<code>init_indexed_name</code>	initializes dimensions for a particular indexed named set
<code>set_type_idx_map</code>	maps indices of a given set type, e.g. variable 361 corresponds to <code>w(68)</code>
<code>valid_named_set_type[†]</code>	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-7: Utility Functions

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

* Deprecated. Please use `have_feature` from [MP-Test](#) instead.

Table A-8: Feature Detection Functions *

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

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

Table A-9: 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
pne_ex1	code for PNE Example (see Section 4.5.8) for pnes_master
test_mp_opt_model	runs full MP-Opt-Model test suite
t_have_fcn	runs tests for (deprecated) have_fcn
t_miqps_master	runs tests of MILP/MIQP solvers via miqps_master
t_nested_struct_copy	runs tests for nested_struct_copy
t_nleqs_master	runs tests of NLEQ solvers via nleqs_master
t_nlps_master	runs tests of NLP solvers via nlps_master
t_om_solve_leqs	runs tests of LEQ solvers via om.solve()
t_om_solve_miqps	runs tests of MILP/MIQP solvers via om.solve()
t_om_solve_nleqs	runs tests of NLEQ solvers via om.solve()
t_om_solve_nlps	runs tests of NLP solvers via om.solve()
t_om_solve_pne	runs tests of PNE solvers via om.solve()
t_om_solve_qps	runs tests of LP/QP solvers via om.solve()
t_opt_model	runs tests for opt_model objects
t_pnes_master	runs tests of PNE solvers via pnes_master
t_qps_master	runs tests of LP/QP solvers via qps_master

Appendix B Optional Packages

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

B.1 BPMPD_MEX – MEX interface for BPMPD

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

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

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

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

B.2 CLP – COIN-OR Linear Programming

The CLP [14] (COIN-OR Linear Programming) 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 required³¹. For Microsoft

³¹According to David Gleich at http://web.stanford.edu/~dgleich/notebook/2009/03/coinor_clop_for_matlab.html, there was a MATLAB MEX interface to CLP written by Johan Lofberg and available (at some point in the past) at <http://control.ee.ethz.ch/~joloef/mexclp.zip>. Unfortunately, at the time of this writing, it seems it is no longer available there, but Davide Barcelli makes some precompiled MEX files for some platforms available here

Windows users, a pre-compiled MEX version of CLP (and numerous other solvers, such as GLPK and IPOPT) are easily installable as part of the OPTI Toolbox³² [15]

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

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

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

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

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

B.4 GLPK – GNU Linear Programming Kit

The GLPK [16] (GNU Linear Programming Kit) 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 required³³. 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 Toolbox³⁴ [15].

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

³²The OPTI Toolbox is available from <https://www.inverseproblem.co.nz/OPTI/>.

³³The <http://glpkmx.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.

³⁴The OPTI Toolbox is available from <https://www.inverseproblem.co.nz/OPTI/>.

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

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

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

Although Gurobi is a commercial package, at the time of this writing there 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 [18] (Interior Point OPTimizer, pronounced I-P-Opt) is a software package for large-scale nonlinear optimization. It 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 [19, 20], are available from the PARDISO project³⁵. For Microsoft Windows users, a pre-compiled MEX version of IPOPT (and numerous other solvers, such as CLP and GLPK) are easily installable as part of the OPTI Toolbox³⁶ [15].

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

B.7 Artelys Knitro – Non-Linear Programming Solver

Artelys Knitro [21] is a general purpose optimization solver specializing in nonlinear problems, available from Artelys. As of version 9, Knitro includes a native MATLAB interface, `knitromatlab`³⁷. 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 `nlp_master` with the algorithm option set to 'KNITRO', or by calling `nlp_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

³⁵See <https://pardiso-project.org/> for the download links.

³⁶The OPTI Toolbox is available from <https://www.inverseproblem.co.nz/OPTI/>.

³⁷Earlier 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).

`qps_master`, or MILP and MIQP problems via `miqps_master`, with the algorithm option set to 'MOSEK', or by calling `qps_mosek` or `miqps_mosek` directly.

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

MATLAB's Optimization Toolbox [22, 23], 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 `nlp_master` with the algorithm option set to 'FMINCON', or by calling `nlp_fmincon` directly. It can also be used to solve general NLEQ problems via MP-Opt-Model's common NLEQ solver interface `nleqs_master` with the algorithm option set to 'FSOLVE', or by calling `nleqs_fsolve` directly.

B.10 OSQP – Operator Splitting Quadratic Program Solver

OSQP [24] is a numerical optimization package for solving convex quadratic programming problems. It uses a custom ADMM-based first-order method requiring only a single matrix factorization in the setup phase. More information is available at <https://osqp.org>.

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

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

Appendix C Release History

The full release history can be found in `CHANGES.md` or [online](https://github.com/MATPOWER/mp-opt-model/blob/master/CHANGES.md) 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](#).³⁸ There is no User’s Manual yet.

New Features

- New unified interface `nlp_solver_master()` for nonlinear programming solvers [MIPS](#), `fmincon`, IPOPT and Artelys Knitro.
- New functions:
 - `mopt2nlpopt()` creates or modifies an options struct for `nlp_solver_master()` from a MATPOWER options struct.
 - `nlp_solver_fmincon()` provides implementation of unified nonlinear programming solver interface for `fmincon`.
 - `nlp_solver_ipopt()` provides implementation of unified nonlinear programming solver interface interface for IPOPT.
 - `nlp_solver_knitro()` provides implementation of unified nonlinear programming solver interface interface for IPOPT.
 - `nlp_solver_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.*

³⁸From the current `master` branch in the [MATPOWER GitHub repository](#) at the time.

- Significant performance improvement for CPLEX on small problems by eliminating call to `cploptimset()`, 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 originally in `ktropf_solver()` (code now moved to `nlps_knitro()`) where Artelys Knitro was still using `fmincon` options.

Incompatible Changes

- Modify order of default output arguments of `opt_model.get_idx()` (again), removing the one related to legacy costs.

- MP-Opt-Model has renamed the following functions and modified the order of their input args so that the MP-Opt-Model object appears first. Ideally, these would be defined as methods of the `opt_model` class, but Octave 4.2 and earlier is not able to find them via a function handle (as used in the `solve()` method) if they are inherited by a subclass.

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

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.³⁹

New Documentation

- Add [MP-Opt-Model User’s Manual](#) with L^AT_EX 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.⁴⁰

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.

³⁹<https://matpower.org/docs/MP-Opt-Model-manual-1.0.pdf>

⁴⁰<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.⁴¹

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.

⁴¹<https://matpower.org/docs/MP-Opt-Model-manual-2.1.pdf>

C.6 Version 3.0 – released Oct 8, 2020

The [MP-Opt-Model 3.0 User’s Manual](#) is available online.⁴²

New Features

- Support for [OSQP](https://osqp.org) solver for LP and QP problems (<https://osqp.org>).
- Support for modifying parameters of an existing MP-Opt-Model object.
- Support for extracting specific named/indexed variables, costs, constraint values and shadow prices, etc. from a solved MP-Opt-Model object.
- Results of the `solve()` method saved to the `soln` field of the MP-Opt-Model object.
- Allow `v0`, `v1`, and `vu` inputs to `opt_model.add_var()` method, and `l` and `u` inputs to `opt_model.add_lin_constraint()` to be scalars that get expanded automatically to the appropriate vector dimension.
- New functions:
 - `opt_model.set_params()` method modifies parameters for a given named set of existing variables, costs, or constraints of an MP-Opt-Model object.
 - `opt_model.get_soln()` method extracts solved results for a given named set of variables, constraints or costs.
 - `opt_model.parse_soln()` method returns a complete set of solution vector and shadow price values for a solved model.
 - `opt_model.eval_lin_constraint()` method computes the constraint values for the full set or an individual named subset of linear constraints.
 - `qps_osqp()` provides standardized interface for using [OSQP](https://osqp.org) to solve LP/QP problems
 - `osqp_options()` initializes options for [OSQP](https://osqp.org) solver
 - `osqpver()` returns/displays version information for [OSQP](https://osqp.org)
 - ... plus 29 individual feature detection functions for `have_feature()`, see [Table A-8](#) for details.

⁴²<https://matpower.org/docs/MP-Opt-Model-manual-3.0.pdf>

Bugs Fixed

- Starting point supplied to `solve()` via `opt.x0` is no longer ignored for nonlinear equations.
- Calling `params_var()` method with empty `idx` no longer results in fatal error.
- For `opt_model`, incorrect evaluation of constant term has been fixed for vector valued quadratic costs with constant term supplied as a vector.

Other Changes

- Simplified logic to determine whether a quadratic cost for an MP-Opt-Model object is vector vs. scalar valued. If the quadratic coefficient is supplied as a matrix, the cost is scalar valued, otherwise it is vector valued.
- Deprecated `have_fcn()` and made it a simple wrapper around the new modular and extensible `have_feature()`, which has now been moved to [MP-Test](#).⁴³

C.7 Version 4.0 – released Oct 18, 2021

The [MP-Opt-Model 4.0 User’s Manual](#) is available online.⁴⁴

New Features

- Support for new class of problems – parameterized nonlinear equations (PNE). Either create a model with only equality constraints (no inequalities or costs) and with number of variables equal to 1 more than number of constraints, *or* call `pnes_master()` directly. See Section 4.5 of User’s Manual for details.
 - Predictor/corrector numerical continuation method for tracing solution curves for PNE problems.
 - Plotting of solution curves.
 - User-defined event functions and callback functions.
 - Warm-start capabilities.

Thanks to Shrirang Abhyankar and Alexander Flueck for contributions to this feature, which is based on the continuation power flow code in MATPOWER 7.1.

⁴³[MP-Test](https://github.com/MATPOWER/mpctest) is available at <https://github.com/MATPOWER/mpctest>.

⁴⁴<https://matpower.org/docs/MP-Opt-Model-manual-4.0.pdf>

- Optional threshold for detecting failure of LEQ solve, by setting the `leq_opt.thresh` option. If the absolute value of any element of the solution vector exceeds the threshold, `exitflag` is set to 0, indicating failure.
- New functions:
 - `pnes_master()` provides unified interface for parameterized nonlinear equation (PNE) solvers.
 - `pne_callback_default()` collects PNE results and optionally plots solution curve.
 - `pne_callback_nose()` handles event signaling a nose point or limit has been reached.
 - `pne_callback_target_lam()` handles event signaling a target value of parameter λ has been reached.
 - `pne_detect_events()` detects events from event function values.
 - `pne_detected_event()` returns detected event details for events with a particular name.
 - `pne_event_nose()` detects the limit or nose point.
 - `pne_event_target_lam()` detects a target λ value.
 - `pne_pfcn_arc_length()` implements arc length parameterization.
 - `pne_pfcn_natural()` implements natural parameterization.
 - `pne_pfcn_pseudo_arc_length()` implements pseudo arc length parameterization.
 - `pne_register_callbacks()` registers callback functions.
 - `pne_register_events()` registers event functions.
 - `mp_idx_manager.set_type_idx_map()` method returns information about mapping of indices for a given set type back to the corresponding named (and possibly indexed) sets.
 - `mpopt2pneopt()` creates or modifies an options struct for `pnes_master()` from a MATPOWER options struct.

Bugs Fixed

- Calling the `problem_type()` or `is_mixed_integer()` method on an empty model no longer causes a fatal error.

Other Changes

- Labels from the `set_types` property are now used as headers for `opt_model.display()` to simplify things facilitate use by subclasses.
- Refactored `describe_idx` into a new method, `set_type_idx_map`, that returns in information in a programmatically usable form, and an updated `describe_idx` that calls the new method, then formats the results in the expected char array(s).

C.8 Version 4.1 – released Dec 13, 2022

The [MP-Opt-Model 4.1 User’s Manual](#) is available online.⁴⁵

New Features

- Add support to `qps_master()` for calling `qps_<my_solver>()` by setting `opt.alg` to '`<MY_SOLVER>`' to allow for handling custom LP/QP solvers.

Other Changes

- Update for compatibility with Artelys Knitro 13.1 and later.
- Add elapsed time in seconds to results of the `solve()` method of `opt_model`, returned in `om.soln.output.et`.
- Add `runtime` field to output argument of `qps_glpk()` and `qps_mosek()`.

C.9 Version 4.2 – released May 10, 2024

The [MP-Opt-Model 4.2 User’s Manual](#) is available online.⁴⁶

New Features

- Option for `opt_model.add_lin_constraint()` to provide/store the transpose of the A matrix instead of the original. This can potentially save significant memory for sparse matrices with many more columns than rows. E.g. storage constraints in [MOST](#) for 8760 hour planning horizon.

⁴⁵<https://matpower.org/docs/MP-Opt-Model-manual-4.1.pdf>

⁴⁶<https://matpower.org/docs/MP-Opt-Model-manual-4.2.pdf>

- Add support to `nlp_solver_master()` for calling `nlp_solver_<my_solver>()` by setting `opt.alg` to '`<MY_SOLVER>`' to allow for handling custom NLP solvers.
- Add support to `miqp_solver_master()` for calling `miqp_solver_<my_solver>()` by setting `opt.alg` to '`<MY_SOLVER>`' to allow for handling custom MILP/MIQP solvers.
- Add to the `parse_soln()` method of `opt_model` an optional `stash` input argument that, if present and true, causes the parsed solution to be stored back in the object, as the `solve()` method was already doing when `opt.parse_soln` is true.
- New Sphinx-based Reference documentation.⁴⁷
- New functions:
 - `convert_lin_constraint()` converts linear constraints from a single set of doubly-bounded inequality constraints to separate sets of equality and upper-bounded inequality constraints.
 - `convert_lin_constraint_multipliers()` converts multipliers on linear constraints from separate sets for equality and upper-bounded inequality constraints to those for doubly-bounded inequality constraints.
- New `opt_model` methods:
 - `is_solved()` indicates whether the model has been solved.
 - `has_parsed_soln()` indicates whether a parsed solution is available in the model.
 - `display_soln()` display the results of a solved model, including values, bounds and shadow prices for variables and linear constraints, values and shadow prices for nonlinear constraints, and individual cost components.

Bugs Fixed

- Clear cached parameters after updating linear constraints or quadratic costs via `opt_model.set_params()`.
- In `miqp_solver_mosek()` the lower and upper bounds of binary variables got overwritten with 0 and 1, respectively, effectively relaxing any potentially tighter bounds provided as input.

⁴⁷<https://matpower.org/doc/mpom/>

- Fix false positive in `have_feature_fsolve` in case where the file is present, but without a valid license.

Other Changes

- Update for compatibility with MATLAB R2023a (Optimization Toolbox 9.5) and later, which removed `x0` as a valid input to `linprog`.
- Update `have_feature_ipopt()` to recognize IPOPT MEX installations from Enrico Bertolazzi's [mexIPOPT](#)⁴⁸, which include MEX files that have been renamed to architecture-specific names along with an `ipopt.m` wrapper function to call the appropriate one. *Thanks to Carlos Murillo-Sánchez.*
Note: *While MP-Opt-Model no longer requires this, my recommendation is still to simply rename the MEX file to `ipopt.<mexext>`, with the appropriate architecture-specific extension, and delete the unnecessary `ipopt.m` entirely.*
- Always skip price computation stage in `miqps_<solver>()` functions for pure (as opposed to mixed) integer problems.
- Add caching of aggregate output parameters in `opt_model.params_var()`.

⁴⁸[mexIPOPT](https://github.com/ebertolazzi/mexIPOPT) is available at <https://github.com/ebertolazzi/mexIPOPT>.

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](https://doi.org/10.1109/TPWRS.2010.2051168) 1.1
- [2] R. D. Zimmerman, C. E. Murillo-Sánchez (2022). MATPOWER [Software]. Available: <https://matpower.org>
doi: [10.5281/zenodo.3236535](https://doi.org/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. 2024. [Online]. Available: <https://matpower.org/docs/MP-Opt-Model-manual.pdf>
doi: [10.5281/zenodo.3818002](https://doi.org/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](https://doi.org/10.1109/TPWRS.2007.901301) 2.1
- [7] R. D. Zimmerman, H. Wang. MATPOWER Interior Point Solver (MIPS) User’s Manual. 2024. [Online]. Available: <https://matpower.org/docs/MIPS-manual.pdf>
doi: [10.5281/zenodo.3236506](https://doi.org/10.5281/zenodo.3236506) 2.1
- [8] E. L. Allgower, K. Georg, *Introduction to Numerical Continuation Methods*, Society for Industrial and Applied Mathematics, 2003. doi: [10.1137/1.9780898719154](https://doi.org/10.1137/1.9780898719154) 4.5
- [9] H.-D. Chiang, A. Flueck, K. Shah, and N. Balu, “CPFLOW: A Practical Tool for Tracing Power System Steady-State Stationary Behavior Due to Load and Generation Variations,” *Power Systems, IEEE Transactions on*, vol. 10, no. 2, pp. 623–634, May 1995. 4.5.1

- [10] S. H. Li and H. D. Chiang, “Nonlinear Predictors and Hybrid Corrector for Fast Continuation Power Flow”, *Generation, Transmission Distribution, IET*, 2(3):341–354, 2008. [4.5.1](#)
- [11] H. Mori and S. Yamada, “Continuation Power Flow with the Nonlinear Predictor of the Lagrange’s Polynomial Interpolation Formula, ” In *Transmission and Distribution Conference and Exhibition 2002: Asia Pacific. IEEE/PES*, vol. 2, pp. 1133–1138, Oct 6–10, 2002. [4.5.1](#)
- [12] BPMPD_MEX. [Online]. Available: <http://www.pserc.cornell.edu/bpmpd/>. [B.1](#)
- [13] C. Mészáros, *The Efficient Implementation of Interior Point Methods for Linear Programming and their Applications*, Ph.D. thesis, Eötvös Loránd University of Sciences, Budapest, Hungary, 1996. [B.1](#)
- [14] COIN-OR Linear Programming (CLP) Solver. [Online]. Available: <https://github.com/coin-or/Clp>. [B.2](#)
- [15] J. Currie and D. I. Wilson, “OPTI: Lowering the Barrier Between Open Source Optimizers and the Industrial MATLAB User,” *Foundations of Computer-Aided Process Operations*, Georgia, USA, 2012. [6.6.4](#), [B.2](#), [B.4](#), [B.6](#)
- [16] GLPK. [Online]. Available: <https://www.gnu.org/software/glpk/>. [B.4](#)
- [17] Gurobi Optimization, Inc., “Gurobi Optimizer Reference Manual,” 2016. [Online]. Available: <https://www.gurobi.com/>. [B.5](#)
- [18] A. Wächter and L. T. Biegler, “On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming,” *Mathematical Programming*, 106(1):25—57, 2006. [B.6](#)
- [19] 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](#)
- [20] 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](#)

- [21] 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](https://doi.org/10.1007/0-387-30065-1_4) B.7
- [22] *Optimization Toolbox*, The MathWorks, Inc. [Online]. Available: <https://www.mathworks.com/products/optimization/>. B.9
- [23] *Optimization Toolbox Users’s Guide*, The MathWorks, Inc., 2024. [Online]. Available: https://www.mathworks.com/help/pdf_doc/optim/optim.pdf. B.9
- [24] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd, S., “OSQP: An Operator Splitting Solver for Quadratic Programs”, *Mathematical Programming Computation*, 2020. doi: [10.1007/s12532-020-00179-2](https://doi.org/10.1007/s12532-020-00179-2) B.10