MATPOWER 5.0
User's Manual

Ray D. Zimmerman   Carlos E. Murillo-Sánchez

December 17, 2014

© 2010, 2011, 2012, 2013, 2014 Power Systems Engineering Research Center (PSERC)
All Rights Reserved

Contents

1 Introduction **9**
1.1 Background 9
1.2 License and Terms of Use 9
1.3 Citing MATPOWER 10

2 Getting Started **11**
2.1 System Requirements 11
2.2 Installation 11
2.3 Running a Simulation 13
2.3.1 Preparing Case Input Data 14
2.3.2 Solving the Case 14
2.3.3 Accessing the Results 15
2.3.4 Setting Options 15
2.4 Documentation 16

3 Modeling **18**
3.1 Data Formats 18
3.2 Branches 18
3.3 Generators 20
3.4 Loads 20
3.5 Shunt Elements 20
3.6 Network Equations 21
3.7 DC Modeling 21

4 Power Flow **25**
4.1 AC Power Flow 25
4.2 DC Power Flow 27
4.3 `runpf` 27
4.4 Linear Shift Factors 29

5 Continuation Power Flow **32**
5.1 Parameterization 33
5.2 Predictor 33
5.3 Corrector 34
5.4 Step length control 34
5.5 `runcpf` 35

6 Optimal Power Flow **41**
6.1 Standard AC OPF . . . 41
6.2 Standard DC OPF . . . 42
6.3 Extended OPF Formulation . . . 43
6.3.1 User-defined Costs . . . 43
6.3.2 User-defined Constraints . . . 44
6.3.3 User-defined Variables . . . 46
6.4 Standard Extensions . . . 46
6.4.1 Piecewise Linear Costs . . . 46
6.4.2 Dispatchable Loads . . . 47
6.4.3 Generator Capability Curves . . . 49
6.4.4 Branch Angle Difference Limits . . . 49
6.5 Solvers . . . 50
6.6 `runopf` . . . 51

7 Extending the OPF **56**
7.1 Direct Specification . . . 56
7.2 Callback Functions . . . 57
7.2.1 `ext2int` Callback . . . 58
7.2.2 `formulation` Callback . . . 60
7.2.3 `int2ext` Callback . . . 62
7.2.4 `printpf` Callback . . . 64
7.2.5 `savecase` Callback . . . 67
7.3 Registering the Callbacks . . . 69
7.4 Summary . . . 71
7.5 Example Extensions . . . 71
7.5.1 Fixed Zonal Reserves . . . 71
7.5.2 Interface Flow Limits . . . 72
7.5.3 DC Transmission Lines . . . 73
7.5.4 DC OPF Branch Flow Soft Limits . . . 76

8 Unit De-commitment Algorithm **79**

9 Miscellaneous Matpower Functions **81**
9.1 Input/Output Functions . . . 81
9.1.1 `loadcase` . . . 81
9.1.2 `savecase` . . . 81
9.1.3 `cdf2mpc` . . . 82
9.1.4 `psse2mpc` . . . 82

9.2 System Information . . . 82
9.2.1 case_info . . . 82
9.2.2 compare_case . . . 83
9.2.3 find_islands . . . 83
9.2.4 get_losses . . . 83
9.2.5 margcost . . . 84
9.2.6 isload . . . 84
9.2.7 printpf . . . 84
9.2.8 total_load . . . 85
9.2.9 totcost . . . 85
9.3 Modifying a Case . . . 85
9.3.1 extract_islands . . . 85
9.3.2 load2disp . . . 86
9.3.3 modcost . . . 86
9.3.4 scale_load . . . 86
9.4 Conversion between External and Internal Numbering . . . 87
9.4.1 ext2int, int2ext . . . 87
9.4.2 e2i_data, i2e_data . . . 87
9.4.3 e2i_field, i2e_field . . . 88
9.5 Forming Standard Power Systems Matrices . . . 88
9.5.1 makeB . . . 88
9.5.2 makeBdc . . . 89
9.5.3 makeJac . . . 89
9.5.4 makeYbus . . . 89
9.6 Miscellaneous . . . 89
9.6.1 define_constants . . . 89
9.6.2 have_fcn . . . 90
9.6.3 mpver . . . 90
9.6.4 nested_struct_copy . . . 90

10 Acknowledgments 91

Appendix A MIPS – MATLAB Interior Point Solver 92
A.1 Example 1 . . . 94
A.2 Example 2 . . . 96
A.3 Quadratic Programming Solver . . . 98
A.4 Primal-Dual Interior Point Algorithm . . . 99
A.4.1 Notation . . . 99

A.4.2 Problem Formulation and Lagrangian . . . 100
A.4.3 First Order Optimality Conditions . . . 101
A.4.4 Newton Step . . . 102

Appendix B Data File Format **105**

Appendix C MATPOWER **Options** **111**
C.1 Mapping of Old-Style Options to New-Style Options . . . 125

Appendix D MATPOWER **Files and Functions** **129**
D.1 Documentation Files . . . 129
D.2 MATPOWER Functions . . . 129
D.3 Example MATPOWER Cases . . . 137
D.4 Automated Test Suite . . . 138

Appendix E Extras Directory **142**

Appendix F "Smart Market" Code **143**
F.1 Handling Supply Shortfall . . . 145
F.2 Example . . . 145
F.3 Smartmarket Files and Functions . . . 149

Appendix G Optional Packages **151**
G.1 BPMPD_MEX – MEX interface for BPMPD . . . 151
G.2 CPLEX – High-performance LP and QP Solvers . . . 151
G.3 GLPK – GNU Linear Programming Kit . . . 152
G.4 Gurobi – High-performance LP and QP Solvers . . . 153
G.5 IPOPT – Interior Point Optimizer . . . 153
G.6 KNITRO – Non-Linear Programming Solver . . . 154
G.7 MINOPF – AC OPF Solver Based on MINOS . . . 155
G.8 MOSEK – High-performance LP and QP Solvers . . . 155
G.9 SDP_PF – Applications of a Semidefinite Programming Relaxation of the Power Flow Equations . . . 155
G.10 TSPOPF – Three AC OPF Solvers by H. Wang . . . 156

References **157**

List of Figures

3-1 Branch Model 19
5-1 Nose Curve of Voltage Magnitude at Bus 9 39
6-1 Relationship of w_i to r_i for $d_i = 1$ (linear option) 45
6-2 Relationship of w_i to r_i for $d_i = 2$ (quadratic option) 45
6-3 Constrained Cost Variable 47
6-4 Marginal Benefit or Bid Function 48
6-5 Total Cost Function for Negative Injection 48
6-6 Generator P-Q Capability Curve 50
7-1 Adding Constraints Across Subsets of Variables 61
7-2 DC Line Model 74
7-3 Equivalent "Dummy" Generators 74
7-4 Feasible Region for Branch Flow Violation Constraints 76

List of Tables

4-1 Power Flow Results 28
4-2 Power Flow Options 28
4-3 Power Flow Output Options 29
5-1 Continuation Power Flow Results 35
5-2 Continuation Power Flow Options 36
5-3 Continuation Power Flow Callback Arguments 37
6-1 Optimal Power Flow Results 52
6-2 Optimal Power Flow Solver Options 53
6-3 Other OPF Options 54
6-4 OPF Output Options 54
7-1 Names Used by Implementation of OPF with Reserves 59
7-2 Results for User-Defined Variables, Constraints and Costs 63
7-3 Callback Functions 71
7-4 Input Data Structures for Interface Flow Limits 73
7-5 Output Data Structures for Interface Flow Limits 73
7-6 Input Data Structures for DC OPF Branch Flow Soft Limits 77
7-7 Output Data Structures for DC OPF Branch Flow Soft Limits 77
A-1 Input Arguments for `mips` 93
A-2 Output Arguments for `mips` 94
A-3 Options for `mips` 95
B-1 Bus Data (`mpc.bus`) 106

B-2 Generator Data (mpc.gen) . . . 107
B-3 Branch Data (mpc.branch) . . . 108
B-4 Generator Cost Data (mpc.gencost) . . . 109
B-5 DC Line Data (mpc.dcline) . . . 110
C-1 Top-Level Options . . . 113
C-2 Power Flow Options . . . 114
C-3 Continuation Power Flow Options . . . 115
C-4 OPF Solver Options . . . 116
C-5 General OPF Options . . . 117
C-6 Power Flow and OPF Output Options . . . 118
C-7 OPF Options for MIPS . . . 119
C-8 OPF Options for CPLEX . . . 119
C-9 OPF Options for fmincon . . . 120
C-10 OPF Options for GLPK . . . 120
C-11 OPF Options for Gurobi . . . 121
C-12 OPF Options for Ipopt . . . 121
C-13 OPF Options for KNITRO . . . 122
C-14 OPF Options for MINOPF . . . 122
C-15 OPF Options for MOSEK . . . 123
C-16 OPF Options for PDIPM . . . 124
C-17 OPF Options for TRALM . . . 124
C-18 Old-Style to New-Style Option Mapping . . . 125
D-1 Matpower Documentation Files . . . 129
D-2 Top-Level Simulation Functions . . . 130
D-3 Input/Output Functions . . . 130
D-4 Data Conversion Functions . . . 130
D-5 Power Flow Functions . . . 131
D-6 Continuation Power Flow Functions . . . 131
D-7 OPF and Wrapper Functions . . . 131
D-8 OPF Model Objects . . . 132
D-9 OPF Solver Functions . . . 132
D-10 Other OPF Functions . . . 133
D-11 OPF User Callback Functions . . . 133
D-12 Power Flow Derivative Functions . . . 134
D-13 NLP, LP & QP Solver Functions . . . 134
D-14 Matrix Building Functions . . . 135
D-15 Utility Functions . . . 136
D-16 Example Cases . . . 137

D-17 Automated Test Utility Functions 138
D-18 Test Data 139
D-19 Miscellaneous MATPOWER Tests 140
D-20 MATPOWER OPF Tests 141
F-1 Auction Types 144
F-2 Generator Offers 146
F-3 Load Bids 146
F-4 Generator Sales 149
F-5 Load Purchases 149
F-6 Smartmarket Files and Functions 150

1 Introduction

1.1 Background

Matpower is a package of Matlab® M-files for solving power flow and optimal power flow problems. It is intended as a simulation tool for researchers and educators that is easy to use and modify. Matpower is designed to give the best performance possible while keeping the code simple to understand and modify. The Matpower home page can be found at:

http://www.pserc.cornell.edu/matpower/

Matpower was initially developed by Ray D. Zimmerman, Carlos E. Murillo-Sánchez and Deqiang Gan of PSERC[1] at Cornell University under the direction of Robert J. Thomas. The initial need for Matlab-based power flow and optimal power flow code was born out of the computational requirements of the PowerWeb project[2]. Many others have contributed to Matpower over the years and it continues to be developed and maintained under the direction of Ray Zimmerman.

1.2 License and Terms of Use

Beginning with version 4, the code in Matpower is distributed under the GNU General Public License (GPL) [1] with an exception added to clarify our intention to allow Matpower to interface with Matlab as well as any other Matlab code or MEX-files a user may have installed, regardless of their licensing terms. The full text of the GPL can be found in the `COPYING` file at the top level of the distribution or at http://www.gnu.org/licenses/gpl-3.0.txt.

The text of the license notice that appears with the copyright in each of the code files reads:

---

[1] http://www.pserc.cornell.edu/

[2] http://www.pserc.cornell.edu/powerweb/

```
MATPOWER is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published
by the Free Software Foundation, either version 3 of the License,
or (at your option) any later version.

MATPOWER is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with MATPOWER. If not, see <http://www.gnu.org/licenses/>.

Additional permission under GNU GPL version 3 section 7

If you modify MATPOWER, or any covered work, to interface with
other modules (such as MATLAB code and MEX-files) available in a
MATLAB(R) or comparable environment containing parts covered
under other licensing terms, the licensors of MATPOWER grant
you additional permission to convey the resulting work.
```

Please note that the MATPOWER case files distributed with MATPOWER are not covered by the GPL. In most cases, the data has either been included with permission or has been converted from data available from a public source.

1.3 Citing MATPOWER

While not required by the terms of the license, we do request that publications derived from the use of MATPOWER explicitly acknowledge that fact by citing reference [2].

> 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.

2 Getting Started

2.1 System Requirements

To use MATPOWER 5.0 you will need:

- MATLAB® version 7 (R14) or later[3], or
- GNU Octave version 3.4 or later[4]

For the hardware requirements, please refer to the system requirements for the version of MATLAB[5] or Octave that you are using. If the MATLAB Optimization Toolbox is installed as well, MATPOWER enables an option to use it to solve optimal power flow problems, though this option is not recommended for most applications.

In this manual, references to MATLAB usually apply to Octave as well. At the time of writing, none of the optional MEX-based MATPOWER packages have been built for Octave.

2.2 Installation

Installation and use of MATPOWER requires familiarity with the basic operation of MATLAB, including setting up your MATLAB path.

Step 1: Follow the download instructions on the MATPOWER home page[6]. You should end up with a file named `matpowerXXX.zip`, where `XXX` depends on the version of MATPOWER.

Step 2: Unzip the downloaded file. Move the resulting `matpowerXXX` directory to the location of your choice. These files should not need to be modified, so it is recommended that they be kept separate from your own code. We will use `$MATPOWER` to denote the path to this directory.

---

[3] MATLAB is available from The MathWorks, Inc. (http://www.mathworks.com/). MATPOWER 4 required MATLAB 6.5 (R13), MATPOWER 3.2 required MATLAB 6 (R12), MATPOWER 3.0 required MATLAB 5 and MATPOWER 2.0 and earlier required only MATLAB 4. MATLAB is a registered trademark of The MathWorks, Inc.

[4] GNU Octave is free software, available online at http://www.gnu.org/software/octave/. Some parts of MATPOWER 5.0 may work on earlier versions of Octave, but it has not been tested on versions prior to 3.4, and `psse2mpc` currently requires 3.8 or newer.

[5] http://www.mathworks.com/support/sysreq/previous_releases.html

[6] http://www.pserc.cornell.edu/matpower/

Step 3: Add the following directories to your MATLAB path:

- `$MATPOWER` – core MATPOWER functions
- `$MATPOWER/t` – test scripts for MATPOWER
- (optional) sub-directories of `$MATPOWER/extras` – additional functionality and contributed code (see Appendix E for details).

Step 4: At the MATLAB prompt, type `test_matpower` to run the test suite and verify that MATPOWER is properly installed and functioning. The result should resemble the following, possibly including extra tests, depending on the availablility of optional packages, solvers and extras.

```
>> test_matpower
t_nested_struct_copy....ok
t_mpoption..............ok
t_loadcase..............ok
t_ext2int2ext...........ok
t_jacobian..............ok
t_hessian...............ok
t_margcost..............ok
t_totcost...............ok
t_modcost...............ok
t_hasPQcap..............ok
t_mips..................ok
t_qps_matpower..........ok (252 of 324 skipped)
t_pf....................ok
t_cpf...................ok
t_islands...............ok
t_opf_model.............ok
t_opf_mips..............ok
t_opf_mips_sc...........ok
t_opf_dc_mips...........ok
t_opf_dc_mips_sc........ok
t_opf_userfcns..........ok
t_opf_softlims..........ok
t_runopf_w_res..........ok
t_dcline................ok
t_get_losses............ok
t_makePTDF..............ok
t_makeLODF..............ok
t_printpf...............ok
t_total_load............ok
t_scale_load............ok
t_psse..................ok
All tests successful (2739 passed, 252 skipped of 2991)
Elapsed time 7.34 seconds.
```

2.3 Running a Simulation

The primary functionality of MATPOWER is to solve power flow and optimal power flow (OPF) problems. This involves (1) preparing the input data defining the all of the relevant power system parameters, (2) invoking the function to run the simulation and (3) viewing and accessing the results that are printed to the screen and/or saved in output data structures or files.

2.3.1 Preparing Case Input Data

The input data for the case to be simulated are specified in a set of data matrices packaged as the fields of a MATLAB struct, referred to as a "MATPOWER case" struct and conventionally denoted by the variable `mpc`. This struct is typically defined in a case file, either a function M-file whose return value is the `mpc` struct or a MAT-file that defines a variable named `mpc` when loaded[7]. The main simulation routines, whose names begin with `run` (e.g. `runpf`, `runopf`), accept either a file name or a MATPOWER case struct as an input.

Use `loadcase` to load the data from a case file into a struct if you want to make modifications to the data before passing it to the simulation.

```
>> mpc = loadcase(casefilename);
```

See also `savecase` for writing a MATPOWER case struct to a case file.

The structure of the MATPOWER case data is described a bit further in Section 3.1 and the full details are documented in Appendix B and can be accessed at any time via the command `help caseformat`. The MATPOWER distribution also includes many example case files listed in Table D-16.

2.3.2 Solving the Case

The solver is invoked by calling one of the main simulation functions, such as `runpf` or `runopf`, passing in a case file name or a case struct as the first argument. For example, to run a simple Newton power flow with default options on the 9-bus system defined in `case9.m`, at the MATLAB prompt, type:

```
>> runpf('case9');
```

If, on the other hand, you wanted to load the 30-bus system data from `case30.m`, increase its real power demand at bus 2 to 30 MW, then run an AC optimal power flow with default options, this could be accomplished as follows:

```
>> define_constants;
>> mpc = loadcase('case30');
>> mpc.bus(2, PD) = 30;
>> runopf(mpc);
```

[7]This describes version 2 of the MATPOWER case format, which is used internally and is the default. The version 1 format, now deprecated, but still accessible via the `loadcase` and `savecase` functions, defines the data matrices as individual variables rather than fields of a struct, and some do not include all of the columns defined in version 2.

The define_constants in the first line is simply a convenience script that defines a number of variables to serve as named column indices for the data matrices. In this example, it allows us to access the "real power demand" column of the bus matrix using the name PD without having to remember that it is the 3rd column.

Other top-level simulation functions are available for running DC versions of power flow and OPF, for running an OPF with the option for MATPOWER to shut down (decommit) expensive generators, etc. These functions are listed in Table D-2 in Appendix D.

2.3.3 Accessing the Results

By default, the results of the simulation are pretty-printed to the screen, displaying a system summary, bus data, branch data and, for the OPF, binding constraint information. The bus data includes the voltage, angle and total generation and load at each bus. It also includes nodal prices in the case of the OPF. The branch data shows the flows and losses in each branch. These pretty-printed results can be saved to a file by providing a filename as the optional 3rd argument to the simulation function.

The solution is also stored in a results struct available as an optional return value from the simulation functions. This results struct is a superset of the MATPOWER case struct mpc, with additional columns added to some of the existing data fields and additional fields. The following example shows how simple it is, after running a DC OPF on the 118-bus system in case118.m, to access the final objective function value, the real power output of generator 6 and the power flow in branch 51.

```
>> define_constants;
>> results = rundcopf('case118');
>> final_objective = results.f;
>> gen6_output     = results.gen(6, PG);
>> branch51_flow   = results.branch(51, PF);
```

Full documentation for the content of the results struct can be found in Sections 4.3 and 6.6.

2.3.4 Setting Options

MATPOWER has many options for selecting among the available solution algorithms, controlling the behavior of the algorithms and determining the details of the pretty-printed output. These options are passed to the simulation routines as a MATPOWER options struct. The fields of the struct have names that can be used to set the

corresponding value via the `mpoption` function. Calling `mpoption` with no arguments returns the default options struct, the struct used if none is explicitly supplied. Calling it with a set of name and value pairs modifies the default vector.

For example, the following code runs a power flow on the 300-bus example in `case300.m` using the fast-decoupled (XB version) algorithm, with verbose printing of the algorithm progress, but suppressing all of the pretty-printed output.

```
>> mpopt = mpoption('pf.alg', 'FDXB', 'verbose', 2, 'out.all', 0);
>> results = runpf('case300', mpopt);
```

To modify an existing options struct, for example, to turn the verbose option off and re-run with the remaining options unchanged, simply pass the existing options as the first argument to `mpoption`.

```
>> mpopt = mpoption(mpopt, 'verbose', 0);
>> results = runpf('case300', mpopt);
```

See Appendix C or type:

```
>> help mpoption
```

for more information on MATPOWER's options.

2.4 Documentation

There are two primary sources of documentation for MATPOWER. The first is this manual, which gives an overview of MATPOWER's capabilities and structure and describes the modeling and formulations behind the code. It can be found in your MATPOWER distribution at `$MATPOWER/docs/manual.pdf`.

The second is the built-in `help` command. As with MATLAB's built-in functions and toolbox routines, you can type `help` followed by the name of a command or M-file to get help on that particular function. Nearly all of MATPOWER's M-files have such documentation and this should be considered the main reference for the calling options for each individual function. See Appendix D for a list of MATPOWER functions.

As an example, the help for `runopf` looks like:

```
>> help runopf
 RUNOPF  Runs an optimal power flow.
    [RESULTS, SUCCESS] = RUNOPF(CASEDATA, MPOPT, FNAME, SOLVEDCASE)

    Runs an optimal power flow (AC OPF by default), optionally returning
    a RESULTS struct and SUCCESS flag.

    Inputs (all are optional):
        CASEDATA : either a MATPOWER case struct or a string containing
            the name of the file with the case data (default is 'case9')
            (see also CASEFORMAT and LOADCASE)
        MPOPT : MATPOWER options struct to override default options
            can be used to specify the solution algorithm, output options
            termination tolerances, and more (see also MPOPTION).
        FNAME : name of a file to which the pretty-printed output will
            be appended
        SOLVEDCASE : name of file to which the solved case will be saved
            in MATPOWER case format (M-file will be assumed unless the
            specified name ends with '.mat')

    Outputs (all are optional):
        RESULTS : results struct, with the following fields:
            (all fields from the input MATPOWER case, i.e. bus, branch,
                gen, etc., but with solved voltages, power flows, etc.)
            order - info used in external <-> internal data conversion
            et - elapsed time in seconds
            success - success flag, 1 = succeeded, 0 = failed
            (additional OPF fields, see OPF for details)
        SUCCESS : the success flag can additionally be returned as
            a second output argument

    Calling syntax options:
        results = runopf;
        results = runopf(casedata);
        results = runopf(casedata, mpopt);
        results = runopf(casedata, mpopt, fname);
        results = runopf(casedata, mpopt, fname, solvedcase);
        [results, success] = runopf(...);

        Alternatively, for compatibility with previous versions of MATPOWER,
        some of the results can be returned as individual output arguments:

        [baseMVA, bus, gen, gencost, branch, f, success, et] = runopf(...);

    Example:
        results = runopf('case30');

    See also RUNDCOPF, RUNUOPF.
```

3 Modeling

MATPOWER employs all of the standard steady-state models typically used for power flow analysis. The AC models are described first, then the simplified DC models. Internally, the magnitudes of all values are expressed in per unit and angles of complex quantities are expressed in radians. Internally, all off-line generators and branches are removed before forming the models used to solve the power flow or optimal power flow problem. All buses are numbered consecutively, beginning at 1, and generators are reordered by bus number. Conversions to and from this internal indexing is done by the functions `ext2int` and `int2ext`. The notation in this section, as well as Sections 4 and 6, is based on this internal numbering, with all generators and branches assumed to be in-service. Due to the strengths of the MATLAB programming language in handling matrices and vectors, the models and equations are presented here in matrix and vector form.

3.1 Data Formats

The data files used by MATPOWER are MATLAB M-files or MAT-files which define and return a single MATLAB struct. The M-file format is plain text that can be edited using any standard text editor. The fields of the struct are `baseMVA`, `bus`, `branch`, `gen` and optionally `gencost`, where `baseMVA` is a scalar and the rest are matrices. In the matrices, each row corresponds to a single bus, branch, or generator. The columns are similar to the columns in the standard IEEE CDF and PTI formats. The number of rows in `bus`, `branch` and `gen` are n_b, n_l and n_g, respectively. If present, `gencost` has either n_g or $2n_g$ rows, depending on whether it includes costs for reactive power or just real power. Full details of the MATPOWER case format are documented in Appendix B and can be accessed from the MATLAB command line by typing `help caseformat`.

3.2 Branches

All transmission lines[8], transformers and phase shifters are modeled with a common branch model, consisting of a standard π transmission line model, with series impedance $z_s = r_s + jx_s$ and total charging susceptance b_c, in series with an ideal phase shifting transformer. The transformer, whose tap ratio has magnitude τ and

[8] This does not include DC transmission lines. For more information the handling of DC transmission lines in MATPOWER, see Section 7.5.3.

phase shift angle θ_{shift}, is located at the *from* end of the branch, as shown in Figure 3-1. The parameters r_s, x_s, b_c, τ and θ_{shift} are specified directly in columns `BR_R` (3), `BR_X` (4), `BR_B` (5), `TAP` (9) and `SHIFT` (10), respectively, of the corresponding row of the `branch` matrix.

The complex current injections i_f and i_t at the *from* and *to* ends of the branch, respectively, can be expressed in terms of the 2×2 branch admittance matrix Y_{br} and the respective terminal voltages v_f and v_t

$$\begin{bmatrix} i_f \\ i_t \end{bmatrix} = Y_{br} \begin{bmatrix} v_f \\ v_t \end{bmatrix}. \tag{3.1}$$

With the series admittance element in the π model denoted by $y_s = 1/z_s$, the branch admittance matrix can be written

$$Y_{br} = \begin{bmatrix} \left(y_s + j\frac{b_c}{2}\right)\frac{1}{\tau^2} & -y_s\frac{1}{\tau e^{-j\theta_{\text{shift}}}} \\ -y_s\frac{1}{\tau e^{j\theta_{\text{shift}}}} & y_s + j\frac{b_c}{2} \end{bmatrix}. \tag{3.2}$$

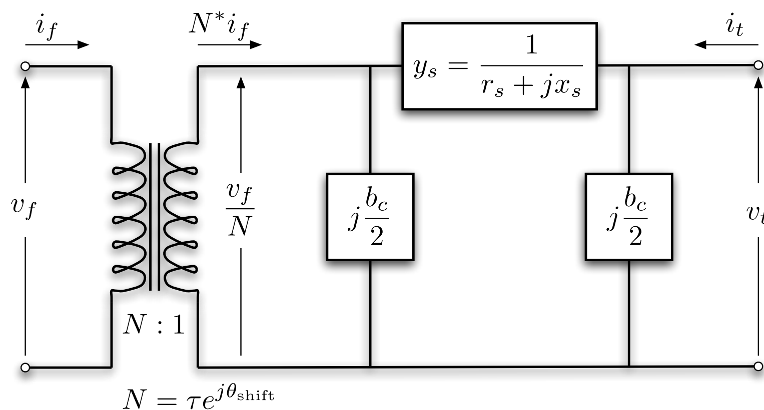

Figure 3-1: Branch Model

If the four elements of this matrix for branch i are labeled as follows:

$$Y_{br}^i = \begin{bmatrix} y_{ff}^i & y_{ft}^i \\ y_{tf}^i & y_{tt}^i \end{bmatrix} \tag{3.3}$$

then four $n_l \times 1$ vectors Y_{ff}, Y_{ft}, Y_{tf} and Y_{tt} can be constructed, where the i-th element of each comes from the corresponding element of Y_{br}^i. Furthermore, the $n_l \times n_b$ sparse connection matrices C_f and C_t used in building the system admittance matrices can be defined as follows. The $(i,j)^{\text{th}}$ element of C_f and the $(i,k)^{\text{th}}$ element of C_t are equal to 1 for each branch i, where branch i connects from bus j to bus k. All other elements of C_f and C_t are zero.

3.3 Generators

A generator is modeled as a complex power injection at a specific bus. For generator i, the injection is

$$s_g^i = p_g^i + jq_g^i. \tag{3.4}$$

Let $S_g = P_g + jQ_g$ be the $n_g \times 1$ vector of these generator injections. The MW and MVAr equivalents (before conversion to p.u.) of p_g^i and q_g^i are specified in columns `PG` (2) and `QG` (3), respectively of row i of the `gen` matrix. A sparse $n_b \times n_g$ generator connection matrix C_g can be defined such that its $(i, j)^{\text{th}}$ element is 1 if generator j is located at bus i and 0 otherwise. The $n_b \times 1$ vector of all bus injections from generators can then be expressed as

$$S_{g,\text{bus}} = C_g \cdot S_g. \tag{3.5}$$

3.4 Loads

Constant power loads are modeled as a specified quantity of real and reactive power consumed at a bus. For bus i, the load is

$$s_d^i = p_d^i + jq_d^i \tag{3.6}$$

and $S_d = P_d + jQ_d$ denotes the $n_b \times 1$ vector of complex loads at all buses. The MW and MVAr equivalents (before conversion to p.u.) of p_d^i and q_d^i are specified in columns `PD` (3) and `QD` (4), respectively of row i of the `bus` matrix.

Constant impedance and constant current loads are not implemented directly, but the constant impedance portions can be modeled as a shunt element described below. Dispatchable loads are modeled as negative generators and appear as negative values in S_g.

3.5 Shunt Elements

A shunt connected element such as a capacitor or inductor is modeled as a fixed impedance to ground at a bus. The admittance of the shunt element at bus i is given as

$$y_{sh}^i = g_{sh}^i + jb_{sh}^i \tag{3.7}$$

and $Y_{sh} = G_{sh} + jB_{sh}$ denotes the $n_b \times 1$ vector of shunt admittances at all buses. The parameters g_{sh}^i and b_{sh}^i are specified in columns `GS` (5) and `BS` (6), respectively, of row i of the `bus` matrix as equivalent MW (consumed) and MVAr (injected) at a nominal voltage magnitude of 1.0 p.u and angle of zero.

3.6 Network Equations

For a network with n_b buses, all constant impedance elements of the model are incorporated into a complex $n_b \times n_b$ bus admittance matrix $Y_{\rm bus}$ that relates the complex nodal current injections $I_{\rm bus}$ to the complex node voltages V:

$$I_{\rm bus} = Y_{\rm bus} V. \tag{3.8}$$

Similarly, for a network with n_l branches, the $n_l \times n_b$ system branch admittance matrices Y_f and Y_t relate the bus voltages to the $n_l \times 1$ vectors I_f and I_t of branch currents at the *from* and *to* ends of all branches, respectively:

$$I_f = Y_f V \tag{3.9}$$
$$I_t = Y_t V. \tag{3.10}$$

If $[\,\cdot\,]$ is used to denote an operator that takes an $n \times 1$ vector and creates the corresponding $n \times n$ diagonal matrix with the vector elements on the diagonal, these system admittance matrices can be formed as follows:

$$Y_f = [Y_{ff}]\, C_f + [Y_{ft}]\, C_t \tag{3.11}$$
$$Y_t = [Y_{tf}]\, C_f + [Y_{tt}]\, C_t \tag{3.12}$$
$$Y_{\rm bus} = C_f^{\mathsf{T}} Y_f + C_t^{\mathsf{T}} Y_t + [Y_{sh}]\,. \tag{3.13}$$

The current injections of (3.8)–(3.10) can be used to compute the corresponding complex power injections as functions of the complex bus voltages V:

$$S_{\rm bus}(V) = [V]\, I_{\rm bus}^* = [V]\, Y_{\rm bus}^* V^* \tag{3.14}$$
$$S_f(V) = [C_f V]\, I_f^* = [C_f V]\, Y_f^* V^* \tag{3.15}$$
$$S_t(V) = [C_t V]\, I_t^* = [C_t V]\, Y_t^* V^*. \tag{3.16}$$

The nodal bus injections are then matched to the injections from loads and generators to form the AC nodal power balance equations, expressed as a function of the complex bus voltages and generator injections in complex matrix form as

$$g_S(V, S_g) = S_{\rm bus}(V) + S_d - C_g S_g = 0. \tag{3.17}$$

3.7 DC Modeling

The DC formulation [8] is based on the same parameters, but with the following three additional simplifying assumptions.

- Branches can be considered lossless. In particular, branch resistances r_s and charging capacitances b_c are negligible:

$$y_s = \frac{1}{r_s + jx_s} \approx \frac{1}{jx_s}, \quad b_c \approx 0. \tag{3.18}$$

- All bus voltage magnitudes are close to 1 p.u.

$$v_i \approx e^{j\theta_i}. \tag{3.19}$$

- Voltage angle differences across branches are small enough that

$$\sin(\theta_f - \theta_t - \theta_{\text{shift}}) \approx \theta_f - \theta_t - \theta_{\text{shift}}. \tag{3.20}$$

Substituting the first set of assumptions regarding branch parameters from (3.18), the branch admittance matrix in (3.2) approximates to

$$Y_{br} \approx \frac{1}{jx_s} \begin{bmatrix} \frac{1}{\tau^2} & -\frac{1}{\tau e^{-j\theta_{\text{shift}}}} \\ -\frac{1}{\tau e^{j\theta_{\text{shift}}}} & 1 \end{bmatrix}. \tag{3.21}$$

Combining this and the second assumption with (3.1) yields the following approximation for i_f:

$$\begin{aligned} i_f &\approx \frac{1}{jx_s}(\frac{1}{\tau^2}e^{j\theta_f} - \frac{1}{\tau e^{-j\theta_{\text{shift}}}}e^{j\theta_t}) \\ &= \frac{1}{jx_s\tau}(\frac{1}{\tau}e^{j\theta_f} - e^{j(\theta_t+\theta_{\text{shift}})}). \end{aligned} \tag{3.22}$$

The approximate real power flow is then derived as follows, first applying (3.19) and (3.22), then extracting the real part and applying (3.20).

$$\begin{aligned} p_f &= \Re\left\{s_f\right\} \\ &= \Re\left\{v_f \cdot i_f^*\right\} \\ &\approx \Re\left\{e^{j\theta_f} \cdot \frac{j}{x_s\tau}(\frac{1}{\tau}e^{-j\theta_f} - e^{-j(\theta_t+\theta_{\text{shift}})})\right\} \\ &= \Re\left\{\frac{j}{x_s\tau}\left(\frac{1}{\tau} - e^{j(\theta_f-\theta_t-\theta_{\text{shift}})}\right)\right\} \\ &= \Re\left\{\frac{1}{x_s\tau}\left[\sin(\theta_f - \theta_t - \theta_{\text{shift}}) + j\left(\frac{1}{\tau} - \cos(\theta_f - \theta_t - \theta_{\text{shift}})\right)\right]\right\} \\ &\approx \frac{1}{x_s\tau}(\theta_f - \theta_t - \theta_{\text{shift}}) \end{aligned} \tag{3.23}$$

As expected, given the lossless assumption, a similar derivation for the power injection at the *to* end of the line leads to leads to $p_t = -p_f$.

The relationship between the real power flows and voltage angles for an individual branch i can then be summarized as

$$\begin{bmatrix} p_f \\ p_t \end{bmatrix} = B_{br}^i \begin{bmatrix} \theta_f \\ \theta_t \end{bmatrix} + P_{\text{shift}}^i \tag{3.24}$$

where

$$B_{br}^i = b_i \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix},$$
$$P_{\text{shift}}^i = \theta_{\text{shift}}^i b_i \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

and b_i is defined in terms of the series reactance x_s^i and tap ratio τ^i for branch i as

$$b_i = \frac{1}{x_s^i \tau^i}.$$

For a shunt element at bus i, the amount of complex power consumed is

$$\begin{aligned} s_{sh}^i &= v_i (y_{sh}^i v_i)^* \\ &\approx e^{j\theta_i}(g_{sh}^i - jb_{sh}^i) e^{-j\theta_i} \\ &= g_{sh}^i - jb_{sh}^i. \end{aligned} \tag{3.25}$$

So the vector of real power consumed by shunt elements at all buses can be approximated by

$$P_{sh} \approx G_{sh}. \tag{3.26}$$

With a DC model, the linear network equations relate real power to bus voltage angles, versus complex currents to complex bus voltages in the AC case. Let the $n_l \times 1$ vector B_{ff} be constructed similar to Y_{ff}, where the i-th element is b_i and let $P_{f,\text{shift}}$ be the $n_l \times 1$ vector whose i-th element is equal to $-\theta_{\text{shift}}^i b_i$. Then the nodal real power injections can be expressed as a linear function of Θ, the $n_b \times 1$ vector of bus voltage angles

$$P_{\text{bus}}(\Theta) = B_{\text{bus}} \Theta + P_{\text{bus,shift}} \tag{3.27}$$

where

$$P_{\text{bus,shift}} = (C_f - C_t)^\mathsf{T} P_{f,\text{shift}}. \tag{3.28}$$

Similarly, the branch flows at the *from* ends of each branch are linear functions of the bus voltage angles

$$P_f(\Theta) = B_f \Theta + P_{f,\text{shift}} \tag{3.29}$$

and, due to the lossless assumption, the flows at the *to* ends are given by $P_t = -P_f$. The construction of the system B matrices is analogous to the system Y matrices for the AC model:

$$B_f = [B_{ff}]\,(C_f - C_t) \tag{3.30}$$

$$B_{\text{bus}} = (C_f - C_t)^{\mathsf{T}} B_f. \tag{3.31}$$

The DC nodal power balance equations for the system can be expressed in matrix form as

$$g_P(\Theta, P_g) = B_{\text{bus}} \Theta + P_{\text{bus,shift}} + P_d + G_{sh} - C_g P_g = 0 \tag{3.32}$$

4 Power Flow

The standard power flow or loadflow problem involves solving for the set of voltages and flows in a network corresponding to a specified pattern of load and generation. MATPOWER includes solvers for both AC and DC power flow problems, both of which involve solving a set of equations of the form

$$g(x) = 0, \tag{4.1}$$

constructed by expressing a subset of the nodal power balance equations as functions of unknown voltage quantities.

All of MATPOWER's solvers exploit the sparsity of the problem and, except for Gauss-Seidel, scale well to very large systems. Currently, none of them include any automatic updating of transformer taps or other techniques to attempt to satisfy typical optimal power flow constraints, such as generator, voltage or branch flow limits.

4.1 AC Power Flow

In MATPOWER, by convention, a single generator bus is typically chosen as a reference bus to serve the roles of both a voltage angle reference and a real power slack. The voltage angle at the reference bus has a known value, but the real power generation at the slack bus is taken as unknown to avoid overspecifying the problem. The remaining generator buses are typically classified as PV buses, with the values of voltage magnitude and generator real power injection given. These are specified in the `VG` (6) and `PG` (3) columns of the `gen` matrix, respectively. Since the loads P_d and Q_d are also given, all non-generator buses are classified as PQ buses, with real and reactive injections fully specified, taken from the `PD` (3) and `QD` (4) columns of the `bus` matrix. Let $\mathcal{I}_{\text{ref}}$, $\mathcal{I}_{\text{PV}}$ and $\mathcal{I}_{\text{PQ}}$ denote the sets of bus indices of the reference bus, PV buses and PQ buses, respectively. The bus type classification is specified in the MATPOWER case file in the `BUS_TYPE` column (2) of the `bus` matrix. Any isolated buses must be identified as such in this column as well.

In the traditional formulation of the AC power flow problem, the power balance equation in (3.17) is split into its real and reactive components, expressed as functions of the voltage angles Θ and magnitudes V_m and generator injections P_g and Q_g, where the load injections are assumed constant and given:

$$g_P(\Theta, V_m, P_g) = P_{\text{bus}}(\Theta, V_m) + P_d - C_g P_g \ = 0 \tag{4.2}$$

$$g_Q(\Theta, V_m, Q_g) = Q_{\text{bus}}(\Theta, V_m) + Q_d - C_g Q_g = 0. \tag{4.3}$$

For the AC power flow problem, the function $g(x)$ from (4.1) is formed by taking the left-hand side of the real power balance equations (4.2) for all non-slack buses and the reactive power balance equations (4.3) for all PQ buses and plugging in the reference angle, the loads and the known generator injections and voltage magnitudes:

$$g(x) = \begin{bmatrix} g_P^{\{i\}}(\Theta, V_m, P_g) \\ g_Q^{\{j\}}(\Theta, V_m, Q_g) \end{bmatrix} \quad \begin{array}{l} \forall i \in \mathcal{I}_{\mathrm{PV}} \cup \mathcal{I}_{\mathrm{PQ}} \\ \forall j \in \mathcal{I}_{\mathrm{PQ}}. \end{array} \tag{4.4}$$

The vector x consists of the remaining unknown voltage quantities, namely the voltage angles at all non-reference buses and the voltage magnitudes at PQ buses:

$$x = \begin{bmatrix} \theta_{\{i\}} \\ v_m^{\{j\}} \end{bmatrix} \quad \begin{array}{l} \forall i \notin \mathcal{I}_{\mathrm{ref}} \\ \forall j \in \mathcal{I}_{\mathrm{PQ}}. \end{array} \tag{4.5}$$

This yields a system of nonlinear equations with $n_{pv} + 2n_{pq}$ equations and unknowns, where n_{pv} and n_{pq} are the number of PV and PQ buses, respectively. After solving for x, the remaining real power balance equation can be used to compute the generator real power injection at the slack bus. Similarly, the remaining $n_{pv} + 1$ reactive power balance equations yield the generator reactive power injections.

Matpower includes four different algorithms for solving the AC power flow problem. The default solver is based on a standard Newton's method [4] using a polar form and a full Jacobian updated at each iteration. Each Newton step involves computing the mismatch $g(x)$, forming the Jacobian based on the sensitivities of these mismatches to changes in x and solving for an updated value of x by factorizing this Jacobian. This method is described in detail in many textbooks.

Also included are solvers based on variations of the fast-decoupled method [5], specifically, the XB and BX methods described in [6]. These solvers greatly reduce the amount of computation per iteration, by updating the voltage magnitudes and angles separately based on constant approximate Jacobians which are factored only once at the beginning of the solution process. These per-iteration savings, however, come at the cost of more iterations.

The fourth algorithm is the standard Gauss-Seidel method from Glimm and Stagg [7]. It has numerous disadvantages relative to the Newton method and is included primarily for academic interest.

By default, the AC power flow solvers simply solve the problem described above, ignoring any generator limits, branch flow limits, voltage magnitude limits, etc. However, there is an option (`pf.enforce_q_lims`) that allows for the generator reactive power limits to be respected at the expense of the voltage setpoint. This is done in a rather brute force fashion by adding an outer loop around the AC power flow

solution. If any generator has a violated reactive power limit, its reactive injection is fixed at the limit, the corresponding bus is converted to a PQ bus and the power flow is solved again. This procedure is repeated until there are no more violations. Note that this option is based solely on the `QMAX` and `QMIN` parameters for the generator, from columns 4 and 5 of the `gen` matrix, and does not take into account the trapezoidal generator capability curves described in Section 6.4.3 and specifed in columns `PC1`–`QC2MAX` (11–16).

4.2 DC Power Flow

For the DC power flow problem [8], the vector x consists of the set of voltage angles at non-reference buses

$$x = \left[\theta_{\{i\}}\right], \quad \forall i \notin \mathcal{I}_{\text{ref}} \tag{4.6}$$

and (4.1) takes the form

$$B_{dc}x - P_{dc} = 0 \tag{4.7}$$

where B_{dc} is the $(n_b - 1) \times (n_b - 1)$ matrix obtained by simply eliminating from B_{bus} the row and column corresponding to the slack bus and reference angle, respectively. Given that the generator injections P_g are specified at all but the slack bus, P_{dc} can be formed directly from the non-slack rows of the last four terms of (3.32).

The voltage angles in x are computed by a direct solution of the set of linear equations. The branch flows and slack bus generator injection are then calculated directly from the bus voltage angles via (3.29) and the appropriate row in (3.32), respectively.

4.3 `runpf`

In Matpower, a power flow is executed by calling `runpf` with a case struct or case file name as the first argument (`casedata`). In addition to printing output to the screen, which it does by default, `runpf` optionally returns the solution in a `results` struct.

```
>> results = runpf(casedata);
```

The `results` struct is a superset of the input Matpower case struct `mpc`, with some additional fields as well as additional columns in some of the existing data fields. The solution values are stored as shown in Table 4-1.

Additional optional input arguments can be used to set options (`mpopt`) and provide file names for saving the pretty printed output (`fname`) or the solved case data (`solvedcase`).

Table 4-1: Power Flow Results

name	description
`results.success`	success flag, 1 = succeeded, 0 = failed
`results.et`	computation time required for solution
`results.order`	see `ext2int` help for details on this field
`results.bus(:, VM)`†	bus voltage magnitudes
`results.bus(:, VA)`	bus voltage angles
`results.gen(:, PG)`	generator real power injections
`results.gen(:, QG)`†	generator reactive power injections
`results.branch(:, PF)`	real power injected into "from" end of branch
`results.branch(:, PT)`	real power injected into "to" end of branch
`results.branch(:, QF)`†	reactive power injected into "from" end of branch
`results.branch(:, QT)`†	reactive power injected into "to" end of branch

† AC power flow only.

Table 4-2: Power Flow Options

name	default	description
`model`	`'AC'`	AC vs. DC modeling for power flow and OPF formulation `'AC'` – use AC formulation and corresponding alg options `'DC'` – use DC formulation and corresponding alg options
`pf.alg`	`'NR'`	AC power flow algorithm: `'NR'` – Newtons's method `'FDXB'` – Fast-Decoupled (XB version) `'FDBX'` – Fast-Decouple (BX version) `'GS'` – Gauss-Seidel
`pf.tol`	10^{-8}	termination tolerance on per unit P and Q dispatch
`pf.nr.max_it`	10	maximum number of iterations for Newton's method
`pf.fd.max_it`	30	maximum number of iterations for fast-decoupled method
`pf.gs.max_it`	1000	maximum number of iterations for Gauss-Seidel method
`pf.enforce_q_lims`	0	enforce gen reactive power limits at expense of $\|V_m\|$ 0 – do *not* enforce limits 1 – enforce limits, simultaneous bus type conversion 2 – enforce limits, one-at-a-time bus type conversion

```
>> results = runpf(casedata, mpopt, fname, solvedcase);
```

The options that control the power flow simulation are listed in Table 4-2 and those controlling the output printed to the screen in Table 4-3.

By default, `runpf` solves an AC power flow problem using a standard Newton's method solver. To run a DC power flow, the `model` option must be set to `'DC'`. For convenience, MATPOWER provides a function `rundcpf` which is simply a wrapper that sets the `model` option to `'DC'` before calling `runpf`.

Table 4-3: Power Flow Output Options

name	default	description
`verbose`	1	amount of progress info to be printed 0 – print no progress info 1 – print a little progress info 2 – print a lot of progress info 3 – print all progress info
`out.all`	-1	controls pretty-printing of results -1 – individual flags control what is printed 0 – do *not* print anything[†] 1 – print everything[†]
`out.sys_sum`	1	print system summary (0 or 1)
`out.area_sum`	0	print area summaries (0 or 1)
`out.bus`	1	print bus detail, includes per bus gen info (0 or 1)
`out.branch`	1	print branch detail (0 or 1)
`out.gen`	0	print generator detail (0 or 1)
`out.force`	0	print results even if success flag = 0 (0 or 1)
`out.suppress_detail`	-1	suppress all output but system summary -1 – suppress details for large systems (> 500 buses) 0 – do *not* suppress any output specified by other flags 1 – suppress all output except system summary section[†]

[†] Overrides individual flags, but (in the case of `out.suppress_detail`) not `out.all` = 1.

Internally, the `runpf` function does a number of conversions to the problem data before calling the appropriate solver routine for the selected power flow algorithm. This external-to-internal format conversion is performed by the `ext2int` function, described in more detail in Section 7.2.1, and includes the elimination of out-of-service equipment, the consecutive renumbering of buses and the reordering of generators by increasing bus number. All computations are done using this internal indexing. When the simulation has completed, the data is converted back to external format by `int2ext` before the results are printed and returned.

4.4 Linear Shift Factors

The DC power flow model can also be used to compute the sensitivities of branch flows to changes in nodal real power injections, sometimes called injection shift factors

(ISF) or generation shift factors [8]. These $n_l \times n_b$ sensitivity matrices, also called power transfer distribution factors or PTDFs, carry an implicit assumption about the slack distribution. If H is used to denote a PTDF matrix, then the element in row i and column j, h_{ij}, represents the change in the real power flow in branch i given a unit increase in the power injected at bus j, *with the assumption* that the additional unit of power is extracted according to some specified slack distribution:

$$\Delta P_f = H \Delta P_{\text{bus}}. \tag{4.8}$$

This slack distribution can be expressed as an $n_b \times 1$ vector w of non-negative weights whose elements sum to 1. Each element specifies the proportion of the slack taken up at each bus. For the special case of a single slack bus k, w is equal to the vector e_k. The corresponding PTDF matrix H_k can be constructed by first creating the $n_l \times (n_b - 1)$ matrix

$$\widetilde{H}_k = \widetilde{B}_f \cdot B_{dc}^{-1} \tag{4.9}$$

then inserting a column of zeros at column k. Here $\widetilde{B}_f$ and B_{dc} are obtained from B_f and B_{bus}, respectively, by eliminating their reference bus columns and, in the case of B_{dc}, removing row k corresponding to the slack bus.

The PTDF matrix H_w, corresponding to a general slack distribution w, can be obtained from any other PTDF, such as H_k, by subtracting $H_k \cdot w$ from each column, equivalent to the following simple matrix multiplication:

$$H_w = H_k(I - w \cdot 1^{\mathsf{T}}). \tag{4.10}$$

These same linear shift factors may also be used to compute sensitivities of branch flows to branch outages, known as line outage distribution factors or LODFs [9]. Given a PTDF matrix H_w, the corresponding $n_l \times n_l$ LODF matrix L can be constructed as follows, where l_{ij} is the element in row i and column j, representing the change in flow in branch i (as a fraction of the initial flow in branch j) for an outage of branch j.

First, let H represent the matrix of sensitivities of branch flows to branch endpoint injections, found by multplying the PTDF matrix by the node-branch incidence matrix:

$$H = H_w(C_f - C_t)^{\mathsf{T}}. \tag{4.11}$$

Here the individual elements h_{ij} represent the sensitivity of flow in branch i with respect to injections at branch j endpoints, corresponding to a simulated increase in flow in branch j. Then l_{ij} can be expressed as

$$l_{ij} = \begin{cases} \dfrac{h_{ij}}{1 - h_{jj}} & i \neq j \\ -1 & i = j. \end{cases} \tag{4.12}$$

Matpower includes functions for computing both the DC PTDF matrix and the corresponding LODF matrix for either a single slack bus k or a general slack distribution vector w. See the help for `makePTDF` and `makeLODF` for details.

5 Continuation Power Flow

Continuation methods or branch tracing methods are used to trace a curve given an initial point on the curve. These are also called predictor-corrector methods since they involve the prediction of the next solution point and correcting the prediction to get the next point on the curve.

Consider a system of n nonlinear equations $g(x) = 0, x \in \mathbb{R}^n$. By adding a continuation parameter λ and one more equation to the system, x can be traced by varying λ. The resulting system $f(x, \lambda) = 0$ has $n + 1$ dimensions. The additional equation is a parameterized equation which identifies the location of the current solution with respect to the previous or next solution.

The continuation process can be diagrammatically shown by (5.1).

$$\left(x^j, \lambda^j\right) \xrightarrow{Predictor} \left(\hat{x}^{j+1}, \hat{\lambda}^{j+1}\right) \xrightarrow{Corrector} \left(x^{j+1}, \lambda^{j+1}\right) \tag{5.1}$$

where, (x^j, λ^j) represents the current solution, $(\hat{x}^{j+1}, \hat{\lambda}^{j+1})$ is the predicted solution, and (x^{j+1}, λ^{j+1}) is the next solution on the curve.

Continuation methods are employed in power systems to determine steady state stability limits [10] in what is called a continuation power flow[9]. The limit is determined from a nose curve where the nose represents the maximum power transfer that the system can handle given a power transfer schedule. To determine the steady state loading limit, the basic power flow equations

$$g(x) = \begin{bmatrix} P(x) - P^{inj} \\ Q(x) - Q^{inj} \end{bmatrix} = 0, \tag{5.2}$$

are restructured as

$$f(x, \lambda) = g(x) - \lambda b = 0 \tag{5.3}$$

where $x \equiv (\Theta, V_m)$ and b is a vector of power transfer given by

$$b = \begin{bmatrix} P^{inj}_{target} - P^{inj}_{base} \\ Q^{inj}_{target} - Q^{inj}_{base} \end{bmatrix}. \tag{5.4}$$

The effects of the variation of loading or generation can be investigated using the continuation method by composing the b vector appropriately.

[9]Thanks to Shrirang Abhyankar, Rui Bo, and Alexander Flueck for contributions to MATPOWER's continuation power flow feature.

5.1 Parameterization

The values of (x, λ) along the solution curve can parameterized in a number of ways [11, 12]. Parameterization is a mathematical way of identifying each solution so that the *next* solution or *previous* solution can be quantified. MATPOWER includes three parameterization scheme options to quantify this relationship, detailed below, where σ is the continuation step size parameter.

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

$$p^j(x, \lambda) = \lambda - \lambda^j - \sigma = 0 \tag{5.5}$$

- **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.

$$p^j(x, \lambda) = \sum_i (x_i - x_i^j)^2 + (\lambda - \lambda^j)^2 - \sigma^2 = 0 \tag{5.6}$$

- **Pseudo arc length parameterization** [14] is MATPOWER's default parameterization scheme, where the next point (x, λ) on the solution curve is constrained to lie in the hyperplane running through the predicted solution $(\hat{x}^{j+1}, \hat{\lambda}^{j+1})$ orthogonal to the tangent line from the previous corrected solution (x^j, λ^j). This relationship can be quantified by the function

$$p^j(x, \lambda) = \bar{z}_j^{\mathsf{T}} \left(\begin{bmatrix} x \\ \lambda \end{bmatrix} - \begin{bmatrix} x^j \\ \lambda^j \end{bmatrix} \right) - \sigma = 0, \tag{5.7}$$

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

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. MATPOWER uses a tangent predictor for estimating the curve to the next solution. The tangent vector $z_j = \begin{bmatrix} dx & d\lambda \end{bmatrix}_j^{\mathsf{T}}$ at the current solution (x^j, λ^j) is found by solving the linear system

$$\begin{bmatrix} f_x & f_\lambda \\ p_x^{j-1} & p_\lambda^{j-1} \end{bmatrix} z_j = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{5.8}$$

The matrix on the left-hand side is simply the standard power flow Jacobian with an additional column and row added. The extra column f_λ is simply the negative of the power transfer vector b and the extra row, required to make the system non-singular and define the magnitude of z_j, is the derivative of the the parameterization function at the previous solution point p^{j-1}.

The resulting tangent vector is then normalized

$$\bar{z}_j = \frac{z_j}{||z_j||_2} \tag{5.9}$$

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

$$\begin{bmatrix} \hat{x}^{j+1} \\ \hat{\lambda}^{j+1} \end{bmatrix} = \begin{bmatrix} x^j \\ \lambda^j \end{bmatrix} + \sigma \bar{z}_j, \tag{5.10}$$

where σ is the continuation step size.

5.3 Corrector

The corrector stage finds the next solution (x^{j+1}, λ^{j+1}) by correcting the approximation estimated by the predictor $(\hat{x}^{j+1}, \hat{\lambda}^{j+1})$. Newton's method is used to find the next solution by solving the $n+1$ dimensional system in (5.11), where one of (5.5)–(5.7) has been added as an additional constraint to the parameterized power flow equations of (5.3).

$$\begin{bmatrix} f(x, \lambda) \\ p^j(x, \lambda) \end{bmatrix} = 0 \tag{5.11}$$

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. MATPOWER's continuation power flow can optionally use adaptive steps, where the step is varied based on an error estimation between the predicted and corrected solutions as given in (5.12).

$$\sigma^{n+1} = \sigma^n \frac{\epsilon_{\rm cpf}}{||\left(x^{j+1}, \lambda^{j+1}\right) - \left(\hat{x}^{j+1}, \hat{\lambda}^{j+1}\right)||_\infty} \qquad \sigma_{\rm min} \le \sigma^{n+1} \le \sigma_{\rm max} \tag{5.12}$$

5.5 runcpf

In MATPOWER, a continuation power flow is executed by calling `runcpf` with two MATPOWER cases (case structs or case file names) as the first two arguments, `basecasedata` and `targetcasedata`, respectively. The first contains the base loading/generation profile while the second contains the target loading/generation profile. In addition to printing output to the screen, which it does by default, `runcpf` optionally returns the solution in a results struct.

```
>> results = runcpf(basecasedata, targetcasedata);
```

Additional optional input arguments can be used to set options (`mpopt`) and provide file names for saving the pretty printed output (`fname`) or the solved case data (`solvedcase`).

```
>> results = runcpf(basecasedata, targetcasedata, mpopt, fname, solvedcase);
```

The `results` struct is a superset of the input MATPOWER case struct `mpc`, with some additional fields as well as additional columns in some of the existing data fields. In addition to the solution values included in the `results` for a simple power flow, shown in Table 4-1 in Section 4.3, the following additional continuation power flow solution values are stored in the `cpf` field as shown in Table 5-1.

Table 5-1: Continuation Power Flow Results

name	description
`results.cpf.iterations`	n_{steps}, number of continuation steps performed
`results.cpf.lam_c`	$1 \times n$ vector of λ values from correction steps†
`results.cpf.lam_p`	$1 \times n$ vector of λ values from prediction steps†
`results.cpf.max_lam`	maximum value of λ found in `results.cpf.lam_c`
`results.cpf.V_c`	$n_b \times n$ matrix of complex bus voltages from correction steps†
`results.cpf.V_p`	$n_b \times n$ matrix of complex bus voltages from prediction steps†

† n is one more than the number of continuation steps, i.e. $n_{\text{steps}} + 1$.

The options that control the continuation power flow simulation are listed in Table 5-2. All the power flow options for Newton's method (tolerance, maximum iterations) and for controlling the output on the screen (see Tables 4-2 and 4-3) are also available with the continuation power flow.

MATPOWER's continuation power flow also 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 PV nose curve.

Table 5-2: Continuation Power Flow Options

name	default	description
`cpf.parameterization`	3	choice of parameterization 1 — natural 2 — arc length 3 — pseudo arc length
`cpf.stop_at`	`'NOSE'`	determines stopping criterion `'NOSE'` — stop when nose point is reached `'FULL'` — trace full nose curve λ_{stop} — stop upon reaching target λ value λ_{stop}
`cpf.step`	0.05	continuation power flow step size
`cpf.adapt_step`	0	toggle adaptive step size feature 0 — adaptive step size disabled 1 — adaptive step size enabled
`cpf.error_tol`	10^{-3}	tolerance for the adaptive step controller
`cpf.step_min`	10^{-4}	minimum allowed step size
`cpf.step_max`	0.2	maximum allowed step size
`cpf.plot.level`	0	control plotting of nose curve 0 — do not plot nose curve 1 — plot when completed 2 — plot incrementally at each iteration 3 — same as 2, with `pause` at each iteration
`cpf.plot.bus`	*empty*	index of bus whose voltage is to be plotted
`cpf.user_callback`	*empty*	string or cell array of strings with names of user callback functions†
`cpf.user_callback_args`	*empty*	struct passed to user-defined callback functions†

† See `help cpf_default_callback` for details.

Any user-defined callback takes the same input and output arguments as those used by the `cpf_default_callback` function, which is always called and is used to collect the λ and V results from each predictor and corrector iteration. To register user callback function(s) so it(they) will be executed, the function name(s) is(are) supplied as a string (cell array of strings) and assigned to the `cpf.user_callback` option. If your callback requires additional arguments, they can be provided in the `cpf.user_callback_args` option.

The prototype for a CPF user callback function is

```
function [cb_state, results] = cpf_user_callback(...
    k, V_c, lam_c, V_p, lam_p, cb_data, cb_state, cb_args, results)
```

and the input and output arguments are described in Table 5-3 and in the help for `cpf_default_callback`. The CPF user callback functions are called in three different

Table 5-3: Continuation Power Flow Callback Arguments

name	description
Inputs	
`k`	continuation step iteration count
`v_c`	vector of complex bus voltages after k-th corrector step
`lam_c`	value of λ after k-th corrector step
`v_p`	vector of complex bus voltages after k-th predictor step
`lam_p`	value of λ after k-th predictor step
`cb_data`	struct containing potentially useful static data, with the following fields (all based on internal indexing):
`.mpc_base`	MATPOWER case struct of base state
`.mpc_target`	MATPOWER case struct of target state
`.Sxfr`	power transfer vector b from (5.4)
`.Ybus`	bus admittance matrix
`.Yf`	branch admittance matrix, "from" end of branches
`.Yt`	branch admittance matrix, "to" end of branches
`.pv`	list of indices of PV buses
`.pq`	list of indices of PQ buses
`.ref`	list of indices of reference buses
`.mpopt`	MATPOWER options struct
`cb_state`	user-defined struct containing any information the callback function would like to pass from one invokation to the next
`cb_args`	callback arguments struct specified in `cpf.user_callback_args`
`results`	initial value of output struct to be assigned to `cpf` field of results struct returned by `runcpf`
Ouputs	
`cb_state`	updated version of `cb_state` input argument
`results`	updated version of `results` input argument

contexts, distinguished as follows:

1. *initial* – called without `results` output arg, with input argument `k` $= 0$, after base power flow, before first CPF step.

2. *iterations* – called without `results` output arg, with input argument `k` > 0, at each iteration, after predictor-corrector step

3. *final* – called with `results` output arg, after exiting predictor-corrector loop, inputs identical to last iteration call

Example

The following is an example of running a continuation power flow using a version of the 9-bus system, looking at increasing all loads by a factor of 2.5. This example plots the nose curve shown in Figure 5-1.

```
define_constants;
mpopt = mpoption('out.all', 0, 'verbose', 2);
mpopt = mpoption(mpopt, 'cpf.stop_at', 'FULL', 'cpf.step', 0.2);
mpopt = mpoption(mpopt, 'cpf.plot.level', 2);
mpcb = loadcase(t_case9_pfv2);                        % load base case
mpct = mpcb;                                          % set up target case with
mpct.gen(:, [PG QG]) = mpcb.gen(:, [PG QG]) * 2.5;  % increased generation
mpct.bus(:, [PD QD]) = mpcb.bus(:, [PD QD]) * 2.5;  % and increased load
results = runcpf(mpcb, mpct, mpopt);
```

This should result in something like the following output to the screen.

```
MATPOWER Version 5.0, 17-Dec-2014 -- AC Continuation Power Flow
step   0 : lambda =  0.000,  4 Newton steps
step   1 : lambda =  0.181,  2 corrector Newton steps
step   2 : lambda =  0.359,  2 corrector Newton steps
step   3 : lambda =  0.530,  2 corrector Newton steps
step   4 : lambda =  0.693,  3 corrector Newton steps
step   5 : lambda =  0.839,  3 corrector Newton steps
step   6 : lambda =  0.952,  3 corrector Newton steps
step   7 : lambda =  0.988,  3 corrector Newton steps
step   8 : lambda =  0.899,  3 corrector Newton steps
step   9 : lambda =  0.776,  3 corrector Newton steps
step  10 : lambda =  0.654,  3 corrector Newton steps
step  11 : lambda =  0.533,  2 corrector Newton steps
step  12 : lambda =  0.413,  2 corrector Newton steps
step  13 : lambda =  0.294,  2 corrector Newton steps
step  14 : lambda =  0.176,  2 corrector Newton steps
step  15 : lambda =  0.000,  3 corrector Newton steps

Traced full continuation curve in 15 continuation steps
```

The results of the continuation power flow are then found in the `cpf` field of the returned `results` struct.

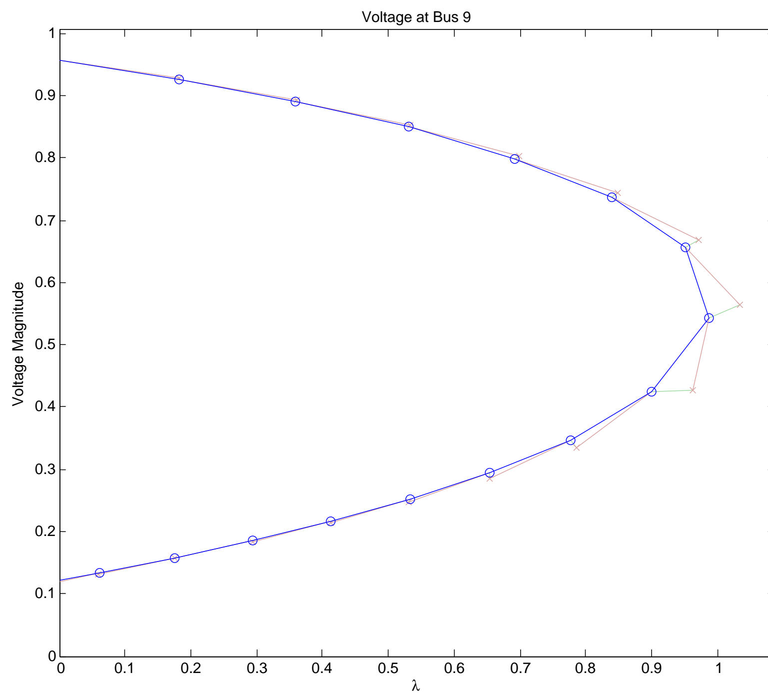

Figure 5-1: Nose Curve of Voltage Magnitude at Bus 9

```
>> results.cpf

ans =

           V_p: [9x16 double]
         lam_p: [1x16 double]
           V_c: [9x16 double]
         lam_c: [1x16 double]
    iterations: 15
       max_lam: 0.9876
```

6 Optimal Power Flow

MATPOWER includes code to solve both AC and DC versions of the optimal power flow problem. The standard version of each takes the following form:

$$\min_x f(x) \tag{6.1}$$

subject to

$$g(x) = 0 \tag{6.2}$$

$$h(x) \leq 0 \tag{6.3}$$

$$x_{\min} \leq x \leq x_{\max} \quad . \tag{6.4}$$

6.1 Standard AC OPF

The optimization vector x for the standard AC OPF problem consists of the $n_b \times 1$ vectors of voltage angles Θ and magnitudes V_m and the $n_g \times 1$ vectors of generator real and reactive power injections P_g and Q_g.

$$x = \begin{bmatrix} \Theta \\ V_m \\ P_g \\ Q_g \end{bmatrix} \tag{6.5}$$

The objective function (6.1) is simply a summation of individual polynomial cost functions f_P^i and f_Q^i of real and reactive power injections, respectively, for each generator:

$$\min_{\Theta, V_m, P_g, Q_g} \sum_{i=1}^{n_g} f_P^i(p_g^i) + f_Q^i(q_g^i). \tag{6.6}$$

The equality constraints in (6.2) are simply the full set of $2 \cdot n_b$ nonlinear real and reactive power balance equations from (4.2) and (4.3). The inequality constraints (6.3) consist of two sets of n_l branch flow limits as nonlinear functions of the bus voltage angles and magnitudes, one for the *from* end and one for the *to* end of each branch:

$$h_f(\Theta, V_m) = |F_f(\Theta, V_m)| - F_{\max} \leq 0 \tag{6.7}$$

$$h_t(\Theta, V_m) = |F_t(\Theta, V_m)| - F_{\max} \leq 0. \tag{6.8}$$

The flows are typically apparent power flows expressed in MVA, but can be real power or current flows, yielding the following three possible forms for the flow constraints:

$$F_f(\Theta, V_m) = \begin{cases} S_f(\Theta, V_m), & \text{apparent power} \\ P_f(\Theta, V_m), & \text{real power} \\ I_f(\Theta, V_m), & \text{current} \end{cases} \tag{6.9}$$

where I_f is defined in (3.9), S_f in (3.15), $P_f = \Re\{S_f\}$ and the vector of flow limits $F_{\max}$ has the appropriate units for the type of constraint. It is likewise for $F_t(\Theta, V_m)$. The values used by MATPOWER's OPF for the flow limits $F_{\max}$ are specified in the `RATE_A` column (6) of the `branch` matrix.

The variable limits (6.4) include an equality constraint on any reference bus angle and upper and lower limits on all bus voltage magnitudes and real and reactive generator injections:

$$\theta_i^{\text{ref}} \le \theta_i \le \theta_i^{\text{ref}}, \qquad i \in \mathcal{I}_{\text{ref}} \tag{6.10}$$

$$v_m^{i,\min} \le v_m^i \le v_m^{i,\max}, \qquad i = 1 \dots n_b \tag{6.11}$$

$$p_g^{i,\min} \le p_g^i \le p_g^{i,\max}, \qquad i = 1 \dots n_g \tag{6.12}$$

$$q_g^{i,\min} \le q_g^i \le q_g^{i,\max}, \qquad i = 1 \dots n_g. \tag{6.13}$$

The voltage reference angle θ_i^{ref} and voltage magnitude bounds $v_m^{i,\max}$ and $v_m^{i,\min}$ are specified in columns `VA` (9), `VMAX` (12) and `VMIN` (13), respectively, of row i of the `bus` matrix. Similarly, the generator bounds $q_g^{i,\max}$, $q_g^{i,\min}$, $p_g^{i,\max}$ and $p_g^{i,\min}$ are specfied in columns `QMAX` (4), `QMIN` (5), `PMAX` (9) and `PMIN` (10), respectively, of row i of the `gen` matrix.

6.2 Standard DC OPF

When using DC network modeling assumptions and limiting polynomial costs to second order, the standard OPF problem above can be simplified to a quadratic program, with linear constraints and a quadratic cost function. In this case, the voltage magnitudes and reactive powers are eliminated from the problem completely and real power flows are modeled as linear functions of the voltage angles. The optimization variable is

$$x = \begin{bmatrix} \Theta \\ P_g \end{bmatrix} \tag{6.14}$$

and the overall problem reduces to the following form.

$$\min_{\Theta, P_g} \sum_{i=1}^{n_g} f_P^i(p_g^i) \tag{6.15}$$

subject to

$$g_P(\Theta, P_g) = B_{\text{bus}}\Theta + P_{\text{bus,shift}} + P_d + G_{sh} - C_g P_g = 0 \tag{6.16}$$

$$h_f(\Theta) = \quad B_f\Theta + P_{f,\text{shift}} - F_{\max} \leq 0 \tag{6.17}$$

$$h_t(\Theta) = -B_f\Theta - P_{f,\text{shift}} - F_{\max} \leq 0 \tag{6.18}$$

$$\theta_i^{\text{ref}} \leq \theta_i \leq \theta_i^{\text{ref}}, \qquad i \in \mathcal{I}_{\text{ref}} \tag{6.19}$$

$$p_g^{i,\min} \leq p_g^i \leq p_g^{i,\max}, \qquad i = 1 \ldots n_g \tag{6.20}$$

6.3 Extended OPF Formulation

MATPOWER employs an extensible OPF structure [15] to allow the user to modify or augment the problem formulation without rewriting the portions that are shared with the standard OPF formulation. This is done through optional input parameters, preserving the ability to use pre-compiled solvers. The standard formulation is modified by introducing additional optional user-defined costs f_u, constraints, and variables z and can be written in the following form:

$$\min_{x,z} f(x) + f_u(x,z) \tag{6.21}$$

subject to

$$g(x) = 0 \tag{6.22}$$

$$h(x) \leq 0 \tag{6.23}$$

$$x_{\min} \leq x \leq x_{\max} \tag{6.24}$$

$$l \leq A \begin{bmatrix} x \\ z \end{bmatrix} \leq u \tag{6.25}$$

$$z_{\min} \leq z \leq z_{\max}. \tag{6.26}$$

Section 7 describes the mechanisms available to the user for taking advantage of the extensible formulation described here.

6.3.1 User-defined Costs

The user-defined cost function f_u is specified in terms of parameters H, C, N, $\hat{r}$, k, d and m. All of the parameters are $n_w \times 1$ vectors except the symmetric $n_w \times n_w$ matrix H and the $n_w \times (n_x + n_z)$ matrix N. The cost takes the form

$$f_u(x,z) = \frac{1}{2} w^{\mathsf{T}} H w + C^{\mathsf{T}} w \tag{6.27}$$

where w is defined in several steps as follows. First, a new vector u is created by applying a linear transformation N and shift $\hat{r}$ to the full set of optimization variables

$$r = N \begin{bmatrix} x \\ z \end{bmatrix}, \tag{6.28}$$

$$u = r - \hat{r}, \tag{6.29}$$

then a scaled function with a "dead zone" is applied to each element of u to produce the corresponding element of w.

$$w_i = \begin{cases} m_i f_{d_i}(u_i + k_i), & u_i < -k_i \\ 0, & -k_i \leq u_i \leq k_i \\ m_i f_{d_i}(u_i - k_i), & u_i > k_i \end{cases} \tag{6.30}$$

Here k_i specifies the size of the "dead zone", m_i is a simple scale factor and f_{d_i} is a pre-defined scalar function selected by the value of d_i. Currently, Matpower implements only linear and quadratic options:

$$f_{d_i}(\alpha) = \begin{cases} \alpha, & \text{if } d_i = 1 \\ \alpha^2, & \text{if } d_i = 2 \end{cases} \tag{6.31}$$

as illustrated in Figure 6-1 and Figure 6-2, respectively.

This form for f_u provides the flexibility to handle a wide range of costs, from simple linear functions of the optimization variables to scaled quadratic penalties on quantities, such as voltages, lying outside a desired range, to functions of linear combinations of variables, inspired by the requirements of price coordination terms found in the decomposition of large loosely coupled problems encountered in our own research.

Some limitations are imposed on the parameters in the case of the DC OPF since Matpower uses a generic quadratic programming (QP) solver for the optimization. In particular, $k_i = 0$ and $d_i = 1$ for all i, so the "dead zone" is not considered and only the linear option is available for f_{d_i}. As a result, for the DC case (6.30) simplifies to $w_i = m_i u_i$.

6.3.2 User-defined Constraints

The user-defined constraints (6.25) are general linear restrictions involving all of the optimization variables and are specified via matrix A and lower and upper bound vectors l and u. These parameters can be used to create equality constraints ($l_i = u_i$) or inequality constraints that are bounded below ($u_i = \infty$), bounded above ($l_i = \infty$) or bounded on both sides.

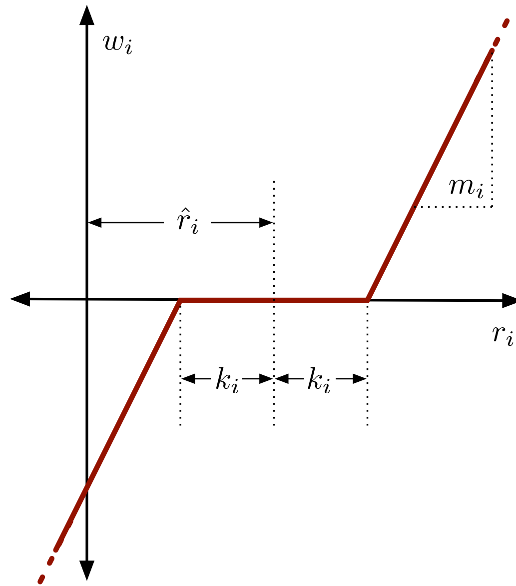

Figure 6-1: Relationship of w_i to r_i for $d_i = 1$ (linear option)

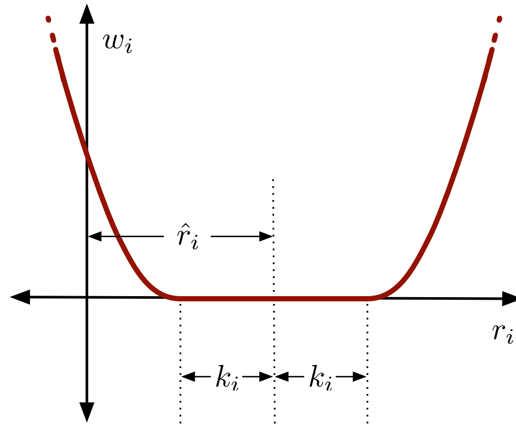

Figure 6-2: Relationship of w_i to r_i for $d_i = 2$ (quadratic option)

6.3.3 User-defined Variables

The creation of additional user-defined z variables is done implicitly based on the difference between the number of columns in A and the dimension of x. The optional vectors $z_{\min}$ and $z_{\max}$ are available to impose lower and upper bounds on z, respectively.

6.4 Standard Extensions

In addition to making this extensible OPF structure available to end users, MATPOWER also takes advantage of it internally to implement several additional capabilities.

6.4.1 Piecewise Linear Costs

The standard OPF formulation in (6.1)–(6.4) does not directly handle the non-smooth piecewise linear cost functions that typically arise from discrete bids and offers in electricity markets. When such cost functions are convex, however, they can be modeled using a constrained cost variable (CCV) method. The piecewise linear cost function $c(x)$ is replaced by a helper variable y and a set of linear constraints that form a convex "basin" requiring the cost variable y to lie in the epigraph of the function $c(x)$.

Figure 6-3 illustrates a convex n-segment piecewise linear cost function

$$c(x) = \left\{ \begin{array}{ll} m_1(x - x_1) + c_1, & x \leq x_1 \\ m_2(x - x_2) + c_2, & x_1 < x \leq x_2 \\ \vdots & \vdots \\ m_n(x - x_n) + c_n, & x_{n-1} < x \end{array} \right. \qquad (6.32)$$

defined by a sequence of points (x_j, c_j), $j = 0 \dots n$, where m_j denotes the slope of the j-th segment

$$m_j = \frac{c_j - c_{j-1}}{x_j - x_{j-1}}, \quad j = 1 \dots n \qquad (6.33)$$

and $x_0 < x_1 < \cdots < x_n$ and $m_1 \leq m_2 \leq \cdots < m_n$.

The "basin" corresponding to this cost function is formed by the following n constraints on the helper cost variable y:

$$y \geq m_j(x - x_j) + c_j, \quad j = 1 \dots n. \qquad (6.34)$$

The cost term added to the objective function in place of $c(x)$ is simply the variable y.

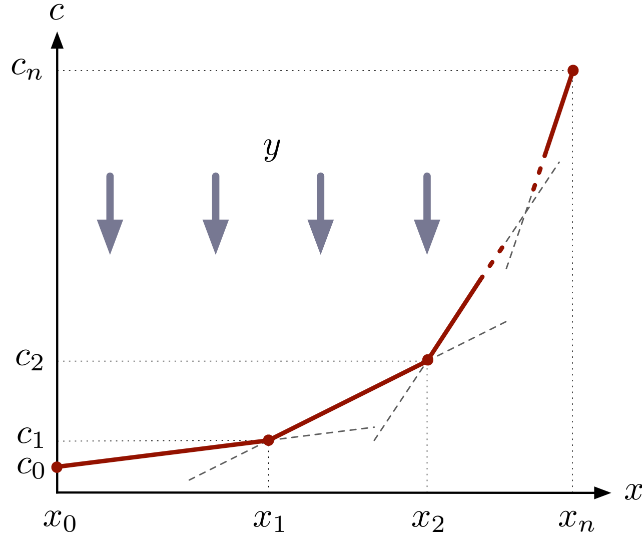

Figure 6-3: Constrained Cost Variable

MATPOWER uses this CCV approach internally to automatically generate the appropriate helper variable, cost term and corresponding set of constraints for any piecewise linear costs on real or reactive generation. All of MATPOWER's OPF solvers, for both AC and DC OPF problems, use the CCV approach with the exception of two that are part of the optional TSPOPF package [16], namely the step-controlled primal/dual interior point method (SCPDIPM) and the trust region based augmented Lagrangian method (TRALM), both of which use a cost smoothing technique instead [17].

6.4.2 Dispatchable Loads

A simple approach to dispatchable or price-sensitive loads is to model them as negative real power injections with associated negative costs. This is done by specifying a generator with a negative output, ranging from a minimum injection equal to the negative of the largest possible load to a maximum injection of zero.

Consider the example of a price-sensitive load whose marginal benefit function is shown in Figure 6-4. The demand p_d of this load will be zero for prices above λ_1, p_1 for prices between λ_1 and λ_2, and $p_1 + p_2$ for prices below λ_2.

This corresponds to a negative generator with the piecewise linear cost curve shown in Figure 6-5. Note that this approach assumes that the demand blocks can be partially dispatched or "split". Requiring blocks to be accepted or rejected in

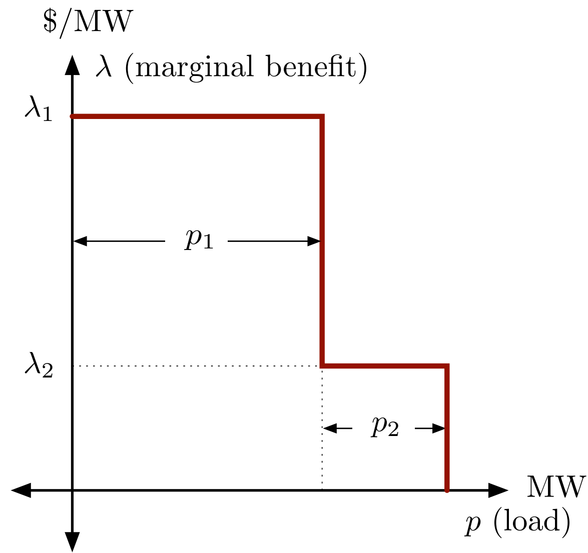

Figure 6-4: Marginal Benefit or Bid Function

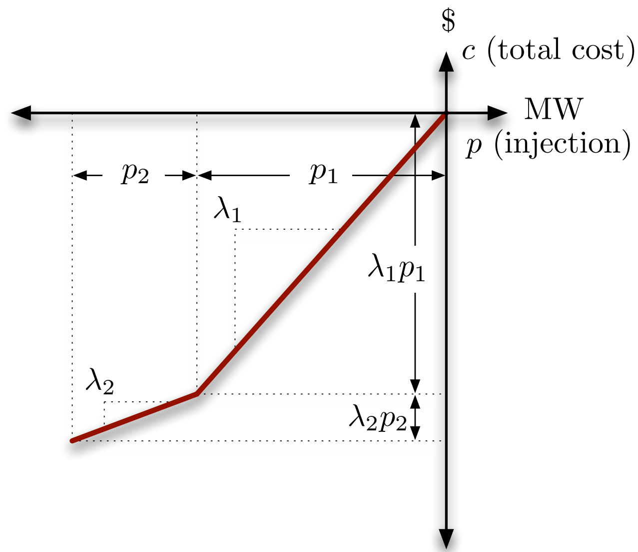

Figure 6-5: Total Cost Function for Negative Injection

their entirety would pose a mixed-integer problem that is beyond the scope of the current MATPOWER implementation.

With an AC network model, there is also the question of reactive dispatch for such loads. Typically the reactive injection for a generator is allowed to take on any value within its defined limits. Since this is not normal load behavior, the model used in MATPOWER assumes that dispatchable loads maintain a constant power factor. When formulating the AC OPF problem, MATPOWER will automatically generate an additional equality constraint to enforce a constant power factor for any "negative generator" being used to model a dispatchable load.

It should be noted that, with this definition of dispatchable loads as negative generators, if the negative cost corresponds to a benefit for consumption, minimizing the cost $f(x)$ of generation is equivalent to maximizing social welfare.

6.4.3 Generator Capability Curves

The typical AC OPF formulation includes box constraints on a generator's real and reactive injections, specified as simple lower and upper bounds on p ($p_{\min}$ and $p_{\max}$) and q ($q_{\min}$ and $q_{\max}$). On the other hand, the true P-Q capability curves of physical generators usually involve some tradeoff between real and reactive capability, so that it is not possible to produce the maximum real output and the maximum (or minimum) reactive output simultaneously. To approximate this tradeoff, MATPOWER includes the ability to add an upper and lower sloped portion to the standard box constraints as illustrated in Figure 6-6, where the shaded portion represents the feasible operating region for the unit.

The two sloped portions are constructed from the lines passing through the two pairs of points defined by the six parameters p_1, $q_1^{\min}$, $q_1^{\max}$, p_2, $q_2^{\min}$, and $q_2^{\max}$. If these six parameters are specified for a given generator in columns `PC1`–`QC2MAX` (11–16), MATPOWER automatically constructs the corresponding additional linear inequality constraints on p and q for that unit.

If one of the sloped portions of the capability constraints is binding for generator k, the corresponding shadow price is decomposed into the corresponding $\mu_{P_{\max}}$ and $\mu_{Q_{\min}}$ or $\mu_{Q_{\max}}$ components and added to the respective column (`MU_PMAX`, `MU_QMIN` or `MU_QMAX`) in the kth row of `gen`.

6.4.4 Branch Angle Difference Limits

The difference between the bus voltage angle θ_f at the *from* end of a branch and the angle θ_t at the *to* end can be bounded above and below to act as a proxy for a transient stability limit, for example. If these limits are provided in columns `ANGMIN`

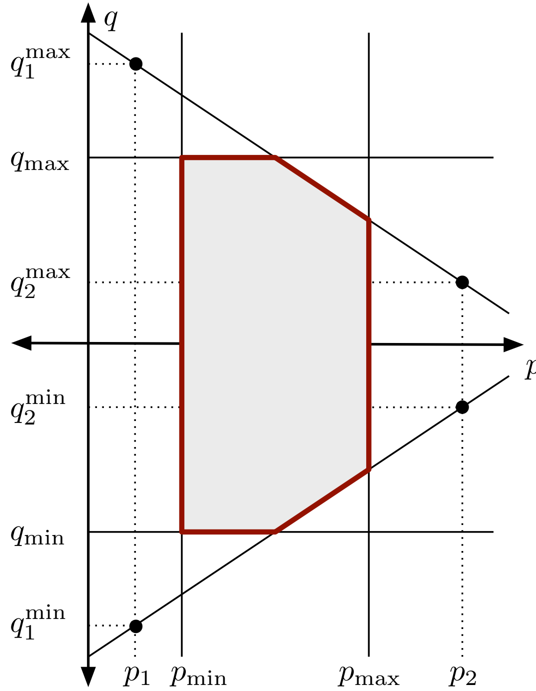

Figure 6-6: Generator P-Q Capability Curve

(12) and `ANGMAX` (13) of the `branch` matrix, MATPOWER creates the corresponding constraints on the voltage angle variables.[10]

6.5 Solvers

Early versions of MATPOWER relied on MATLAB's Optimization Toolbox [18] to provide the NLP and QP solvers needed to solve the AC and DC OPF problems, respectively. While they worked reasonably well for very small systems, they did not scale well to larger networks. Eventually, optional packages with additional solvers were added to improve performance, typically relying on MATLAB extension (MEX) files implemented in Fortran or C and pre-compiled for each machine architecture. Some of these MEX files are distributed as optional packages due to differences in terms of use. For DC optimal power flow, there is a MEX build [19] of the high

[10]The voltage angle difference for branch k is taken to be unbounded below if `branch(k, ANGMIN)` is less than -360 and unbounded above if `branch(k, ANGMAX)` is greater than 360. If both parameters are zero, the voltage angle difference is unconstrained.

performance interior point BPMPD solver [20] for LP/QP problems. For the AC OPF problem, the MINOPF [21] and TSPOPF [16] packages provide solvers suitable for much larger systems. The former is based on MINOS [22] and the latter includes the primal-dual interior point and trust region based augmented Lagrangian methods described in [17]. MATPOWER version 4 and later also includes the option to use the open-source IPOPT solver[11] for solving both AC and DC OPFs, based on the Matlab MEX interface to IPOPT[12]. It also includes the option to use CPLEX[13] or MOSEK[14] for DC OPFs. MATPOWER 4.1 added the option to use KNITRO [29][15] for AC OPFs and the Gurobi Optimizer[16] for DC OPFs and MATPOWER 5 added GLPK. See Appendix G for more details on these optional packages.

Beginnning with version 4, MATPOWER also includes its own primal-dual interior point method implemented in pure-MATLAB code, derived from the MEX implementation of the algorithms described in [17]. This solver is called MIPS (MATLAB Interior Point Solver) and is described in more detail in Appendix A. If no optional packages are installed, MIPS will be used by default for both the AC OPF and as the QP solver used by the DC OPF. The AC OPF solver also employs a unique technique for efficiently forming the required Hessians via a few simple matrix operations [23]. This solver has application to general nonlinear optimization problems outside of MATPOWER and can be called directly as `mips`. There is also a convenience wrapper function called `qps_mips` making it trivial to set up and solve LP and QP problems, with an interface similar to `quadprog` from the MATLAB Optimization Toolbox.

6.6 runopf

In MATPOWER, an optimal power flow is executed by calling `runopf` with a case struct or case file name as the first argument (`casedata`). In addition to printing output to the screen, which it does by default, `runpf` optionally returns the solution in a `results` struct.

```
>> results = runopf(casedata);
```

The `results` struct is a superset of the input MATPOWER case struct `mpc`, with some additional fields as well as additional columns in some of the existing data

[11] Available from https://projects.coin-or.org/Ipopt/.
[12] See https://projects.coin-or.org/Ipopt/wiki/MatlabInterface.
[13] See http://www.ibm.com/software/integration/optimization/cplex-optimizer/.
[14] See http://www.mosek.com/.
[15] See http://www.ziena.com/.
[16] See http://www.gurobi.com/.

fields. In addition to the solution values included in the `results` for a simple power flow, shown in Table 4-1 in Section 4.3, the following additional optimal power flow solution values are stored as shown in Table 6-1.

Table 6-1: Optimal Power Flow Results

name	description
`results.f`	final objective function value
`results.x`	final value of optimization variables (internal order)
`results.om`	OPF model object[†]
`results.bus(:, LAM_P)`	Lagrange multiplier on real power mismatch
`results.bus(:, LAM_Q)`	Lagrange multiplier on reactive power mismatch
`results.bus(:, MU_VMAX)`	Kuhn-Tucker multiplier on upper voltage limit
`results.bus(:, MU_VMIN)`	Kuhn-Tucker multiplier on lower voltage limit
`results.gen(:, MU_PMAX)`	Kuhn-Tucker multiplier on upper P_g limit
`results.gen(:, MU_PMIN)`	Kuhn-Tucker multiplier on lower P_g limit
`results.gen(:, MU_QMAX)`	Kuhn-Tucker multiplier on upper Q_g limit
`results.gen(:, MU_QMIN)`	Kuhn-Tucker multiplier on lower Q_g limit
`results.branch(:, MU_SF)`	Kuhn-Tucker multiplier on flow limit at "from" bus
`results.branch(:, MU_ST)`	Kuhn-Tucker multiplier on flow limit at "to" bus
`results.mu`	shadow prices of constraints[‡]
`results.g`	(optional) constraint values
`results.dg`	(optional) constraint 1st derivatives
`results.raw`	raw solver output in form returned by MINOS, and more[‡]
`results.var.val`	final value of optimization variables, by named subset[‡]
`results.var.mu`	shadow prices on variable bounds, by named subset[‡]
`results.nln`	shadow prices on nonlinear constraints, by named subset[‡]
`results.lin`	shadow prices on linear constraints, by named subset[‡]
`results.cost`	final value of user-defined costs, by named subset[‡]

[†] See help for `opf_model` for more details.
[‡] See help for `opf` for more details.

Additional optional input arguments can be used to set options (`mpopt`) and provide file names for saving the pretty printed output (`fname`) or the solved case data (`solvedcase`).

```
>> results = runopf(casedata, mpopt, fname, solvedcase);
```

Some of the main options that control the optimal power flow simulation are listed in Tables 6-2 and 6-3. There are many other options that can be used to control the termination criteria and other behavior of the individual solvers. See Appendix C or the `mpoption` help for details. As with `runpf` the output printed to the screen can be controlled by the options in Table 4-3, but there are additional output options

for the OPF, related to the display of binding constraints that are listed Table 6-4, along with an option that can be used to force the AC OPF to return information about the constraint values and Jacobian and the objective function gradient and Hessian.

By default, `runopf` solves an AC optimal power flow problem using a primal dual interior point method. To run a DC OPF, the `model` option must be set to `'DC'`. For convenience, Matpower provides a function `rundcopf` which is simply a wrapper that sets the `model` to `'DC'` before calling `runopf`.

Table 6-2: Optimal Power Flow Solver Options

name	default	description

`opf.ac.solver`	`'DEFAULT'`	AC optimal power flow solver:
		`'DEFAULT'` – choose default solver based on availability in the following order: `'PDIPM'`, `'MIPS'`
		`'MIPS'` – MIPS, Matlab Interior Point Solver, primal/dual interior point method[†]
		`'FMINCON'` – Matlab Optimization Toolbox, `fmincon`
		`'IPOPT'` – Ipopt[*]
		`'KNITRO'` – KNITRO[*]
		`'MINOPF'` – MINOPF[*], MINOS-based solver
		`'PDIPM'` – PDIPM[*], primal/dual interior point method[‡]
		`'SDPOPF'` – SDPOPF[*], solver based on semidefinite relaxation
		`'TRALM'` – TRALM[*], trust region based augmented Langrangian method

`opf.dc.solver`	`'DEFAULT'`	DC optimal power flow solver:
		`'DEFAULT'` – choose default solver based on availability in the following order: `'CPLEX'`, `'GUROBI'`, `'MOSEK'`, `'BPMPD'`, `'OT'`, `'GLPK'` *(linear costs only)*, `'MIPS'`
		`'MIPS'` – MIPS, Matlab Interior Point Solver, primal/dual interior point method[†]
		`'BPMPD'` – BPMPD[*]
		`'CPLEX'` – CPLEX[*]
		`'GLPK'` – GLPK[*] *(no quadratic costs)*
		`'GUROBI'` – Gurobi[*]
		`'IPOPT'` – Ipopt[*]
		`'MOSEK'` – MOSEK[*]
		`'OT'` – Matlab Opt Toolbox, `quadprog`, `linprog`

[*] Requires the installation of an optional package. See Appendix G for details on the corresponding package.
[†] For MIPS-sc, the step-controlled version of this solver, the `mips.step_control` option must be set to 1.
[‡] For SC-PDIPM, the step-controlled version of this solver, the `pdipm.step_control` option must be set to 1.

Table 6-3: Other OPF Options

name	default	description
`opf.violation`	5×10^{-6}	constraint violation tolerance
`opf.flow_lim`	`'S'`	quantity to limit for branch flow constraints `'S'` – apparent power flow (limit in MVA) `'P'` – active power flow (limit in MW) `'I'` – current magnitude (limit in MVA at 1 p.u. voltage)
`opf.ignore_ang_lim`	0	ignore angle difference limits for branches 0 – include angle difference limits, if specified 1 – ignore angle difference limits even if specified
`opf.init_from_mpc`	-1	specify whether to use the current state in MATPOWER case to initialize OPF† -1 – MATPOWER decides based on solver/algorithm 0 – ignore current state when initializing OPF 1 – use current state to initialize OPF
`opf.return_raw_der`	0	for AC OPF, return constraint and derivative info in `results.raw` (in fields `g`, `dg`, `df`, `d2f`)

† Currently supported only for IPOPT, KNITRO and MIPS solvers.

Table 6-4: OPF Output Options

name	default	description
`out.lim.all`	-1	controls constraint info output -1 – individual flags control what is printed 0 – do *not* print any constraint info† 1 – print only binding constraint info† 2 – print all constraint info†
`out.lim.v`	1	control output of voltage limit info 0 – do *not* print 1 – print binding constraints only 2 – print all constraints
`out.lim.line`	1	control output of line flow limit info‡
`out.lim.pg`	1	control output of gen active power limit info‡
`out.lim.qg`	1	control output of gen reactive power limit info‡

† Overrides individual flags.
‡ Takes values of 0, 1 or 2 as for `out.lim.v`.

Internally, the `runopf` function does a number of conversions to the problem data before calling the appropriate solver routine for the selected OPF algorithm. This external-to-internal format conversion is performed by the `ext2int` function,

described in more detail in Section 7.2.1, and includes the elimination of out-of-service equipment, the consecutive renumbering of buses and the reordering of generators by increasing bus number. All computations are done using this internal indexing. When the simulation has completed, the data is converted back to external format by `int2ext` before the results are printed and returned. In addition, both `ext2int` and `int2ext` can be customized via user-supplied callback routines to convert data needed by user-supplied variables, constraints or costs into internal indexing.

7 Extending the OPF

The extended OPF formulation described in Section 6.3 allows the user to modify the standard OPF formulation to include additional variables, costs and/or constraints. There are two primary mechanisms available for the user to accomplish this. The first is by directly constructing the full parameters for the addional costs or constraints and supplying them either as fields in the case struct or directly as arguments to the `opf` function. The second, and more powerful, method is via a set of callback functions that customize the OPF at various stages of the execution. MATPOWER includes two examples of using the latter method, one to add a fixed zonal reserve requirement and another to implement interface flow limits.

7.1 Direct Specification

To add costs directly, the parameters H, C, N, $\hat{r}$, k, d and m of (6.27)–(6.31) described in Section 6.3.1 are specified as fields or arguments `H`, `Cw`, `N` and `fparm`, repectively, where `fparm` is the $n_w \times 4$ matrix

$$f_{\text{parm}} = \left[\begin{array}{cccc} d & \hat{r} & k & m \end{array} \right]. \tag{7.1}$$

When specifying additional costs, `N` and `Cw` are required, while `H` and `fparm` are optional. The default value for H is a zero matrix, and the default for f_{parm} is such that d and m are all ones and $\hat{r}$ and k are all zeros, resulting in simple linear cost, with no shift or "dead-zone". `N` and `H` should be specified as sparse matrices.

For additional constraints, the A, l and u parameters of (6.25) are specified as fields or arguments of the same names, `A`, `l` and `u`, respectively, where `A` is sparse.

Additional variables are created implicitly based on the difference between the number of columns in A and the number n_x of standard OPF variables. If A has more columns than x has elements, the extra columns are assumed to correspond to a new z variable. The initial value and lower and upper bounds for z can also be specified in the optional fields or arguments, `z0`, `zl` and `zu`, respectively.

For a simple formulation extension to be used for a small number of OPF cases, this method has the advantage of being direct and straightforward. While MATPOWER does include code to eliminate the columns of A and N corresponding to V_m and Q_g when running a DC OPF[17], as well as code to reorder and eliminate columns appropriately when converting from external to internal data formats, this mechanism still requires the user to take special care in preparing the A and N matrices

[17]Only if they contain all zeros.

to ensure that the columns match the ordering of the elements of the opimization vectors x and z. All extra constraints and variables must be incorporated into a single set of parameters that are constructed before calling the OPF. The bookkeeping needed to access the resulting variables and shadow prices on constraints and variable bounds must be handled manually by the user outside of the OPF, along with any processing of additional input data and processing, printing or saving of the additional result data. Making further modifications to a formulation that already includes user-supplied costs, constraints or variables, requires that both sets be incorporated into a new single consistent set of parameters.

7.2 Callback Functions

The second method, based on defining a set of callback functions, offers several distinct advantages, especially for more complex scenarios or for adding a feature for others to use, such as the zonal reserve requirement or the interface flow limits mentioned previously. This approach makes it possible to:

- define and access variable/constraint sets as individual named blocks
- define constraints, costs only in terms of variables directly involved
- pre-process input data and/or post-process result data
- print and save new result data
- simultaneously use multiple, independently developed extensions (e.g. zonal reserve requirements and interface flow limits)

MATPOWER defines five stages in the execution of a simulation where custom code can be inserted to alter the behavior or data before proceeding to the next stage. This custom code is defined as a set of "callback" functions that are registered via `add_userfcn` for MATPOWER to call automatically at one of the five stages. Each stage has a name and, by convention, the name of a user-defined callback function ends with the name of the corresponding stage. For example, a callback for the `formulation` stage that modifies the OPF problem formulation to add reserve requirements could be registered with the following line of code.

```
mpc = add_userfcn(mpc, 'formulation', @userfcn_reserves_formulation);
```

The sections below will describe each stage and the input and output arguments for the corresponding callback function, which vary depending on the stage. An example that employs additional variables, constraints and costs will be used for illustration.

Consider the problem of jointly optimizing the allocation of both energy and reserves, where the reserve requirements are defined as a set of n_{rz} fixed zonal MW quantities. Let Z_k be the set of generators in zone k and R_k be the MW reserve requirement for zone k. A new set of variables r are introduced representing the reserves provided by each generator. The value r_i, for generator i, must be non-negative and is limited above by a user-provided upper bound $r_i^{\max}$ (e.g. a reserve offer quantity) as well as the physical ramp rate Δ_i.

$$0 \le r_i \le \min(r_i^{\max}, \Delta_i), \quad i = 1 \ldots n_g \tag{7.2}$$

If the vector c contains the marginal cost of reserves for each generator, the user defined cost term from (6.21) is simply

$$f_u(x, z) = c^{\mathsf{T}} r. \tag{7.3}$$

There are two additional sets of constraints needed. The first ensures that, for each generator, the total amount of energy plus reserve provided does not exceed the capacity of the unit.

$$p_g^i + r_i \le p_g^{i,\max}, \quad i = 1 \ldots n_g \tag{7.4}$$

The second requires that the sum of the reserve allocated within each zone k meets the stated requirements.

$$\sum_{i \in Z_k} r_i \ge R_k, \quad k = 1 \ldots n_{rz} \tag{7.5}$$

Table 7-1 describes some of the variables and names that are used in the example callback function listings in the sections below.

7.2.1 `ext2int` Callback

Before doing any simulation of a case, MATPOWER performs some data conversion on the case struct in order to achieve a consistent internal structure, by calling the following.

```
mpc = ext2int(mpc);
```

Table 7-1: Names Used by Implementation of OPF with Reserves

name	description
mpc	MATPOWER case struct
reserves	additional field in mpc containing input parameters for zonal reserves in the following sub-fields:
cost	$n_g \times 1$ vector of reserve costs, c from (7.3)
qty	$n_g \times 1$ vector of reserve quantity upper bounds, i^{th} element is $r_i^{\max}$
zones	$n_{rz} \times n_g$ matrix of reserve zone definitions $\texttt{zones(k,j)} = \begin{cases} 1 & \text{if gen } j \text{ belongs to reserve zone } k \quad (j \in Z_k) \\ 0 & \text{otherwise} \quad (j \notin Z_k) \end{cases}$
req	$n_{rz} \times 1$ vector of zonal reserve requirements, k^{th} element is R_k from (7.5)
om	OPF model object, already includes standard OPF setup
results	OPF results struct, superset of mpc with additional fields for output data
ng	n_g, number of generators
R	name for new reserve variable block, i^{th} element is r_i
Pg_plus_R	name for new capacity limit constraint set (7.4)
Rreq	name for new reserve requirement constraint set (7.5)

All isolated buses, out-of-service generators and branches are removed, along with any generators or branches connected to isolated buses. The buses are renumbered consecutively, beginning at 1, and the in-service generators are sorted by increasing bus number. All of the related indexing information and the original data matrices are stored in an `order` field in the case struct to be used later by `int2ext` to perform the reverse conversions when the simulation is complete.

The first stage callback is invoked from within the `ext2int` function immediately after the case data has been converted. Inputs are a MATPOWER case struct (`mpc`), freshly converted to internal indexing and any (optional) `args` value supplied when the callback was registered via `add_userfcn`. Output is the (presumably updated) `mpc`. This is typically used to reorder any input arguments that may be needed in internal ordering by the formulation stage. The example shows how `e2i_field` can also be used, with a case struct that has already been converted to internal indexing, to convert other data structures by passing in 2 or 3 extra parameters in addition to the case struct. In this case, it automatically converts the input data in the `qty`, `cost` and `zones` fields of `mpc.reserves` to be consistent with the internal generator ordering, where off-line generators have been eliminated and the on-line generators are sorted in order of increasing bus number. Notice that it is the second dimension (columns) of `mpc.reserves.zones` that is being re-ordered. See the on-line help for `e2i_field` and `e2i_data` for more details on what all they can do.

```
function mpc = userfcn_reserves_ext2int(mpc, args)

mpc = e2i_field(mpc, {'reserves', 'qty'}, 'gen');
mpc = e2i_field(mpc, {'reserves', 'cost'}, 'gen');
mpc = e2i_field(mpc, {'reserves', 'zones'}, 'gen', 2);
```

This stage is also a good place to check the consistency of any additional input data required by the extension and throw an error if something is missing or not as expected.

7.2.2 `formulation` Callback

This stage is called from `opf` after the OPF Model (`om`) object has been initialized with the standard OPF formulation, but before calling the solver. This is the ideal place for modifying the problem formulation with additional variables, constraints and costs, using the `add_vars`, `add_constraints` and `add_costs` methods of the OPF Model object. Inputs are the `om` object and any (optional) `args` supplied when the callback was registered via `add_userfcn`. Output is the updated `om` object.

The `om` object contains both the original MATPOWER case data as well as all of the indexing data for the variables and constraints of the standard OPF formulation.[18] See the on-line help for `opf_model` for more details on the OPF model object and the methods available for manipulating and accessing it.

In the example code, a new variable block named `R` with n_g elements and the limits from (7.2) is added to the model via the `add_vars` method. Similarly, two linear constraint blocks named `Pg_plus_R` and `Rreq`, implementing (7.4) and (7.5), respectively, are added via the `add_constraints` method. And finally, the `add_costs` method is used to add to the model a user-defined cost block corresponding to (7.3).

Notice that the last argument to `add_constraints` and `add_costs` allows the constraints and costs to be defined only in terms of the relevant parts of the optimization variable x. For example, the `A` matrix for the `Pg_plus_R` constraint contains only columns corresponding to real power generation (`Pg`) and reserves (`R`) and need not bother with voltages, reactive power injections, etc. As illustrated in Figure 7-1, this allows the same code to be used with both the AC OPF, where x includes V_m and Q_g, and the DC OPF where it does not. This code is also independent of any

[18] It is perfectly legitimate to register more than one callback per stage, such as when enabling multiple independent OPF extensions. In this case, the callbacks are executed in the order they were registered with `add_userfcn`. E.g. when the second and subsequent `formulation` callbacks are invoked, the `om` object will reflect any modifications performed by earlier `formulation` callbacks.

additional variables that may have been added by MATPOWER (e.g. y variables from MATPOWER's CCV handling of piece-wise linear costs) or by the user via previous `formulation` callbacks. MATPOWER will place the constraint matrix blocks in the appropriate place when it constructs the overall `A` matrix at run-time. This is an important feature that enables independently developed MATPOWER OPF extensions to work together.

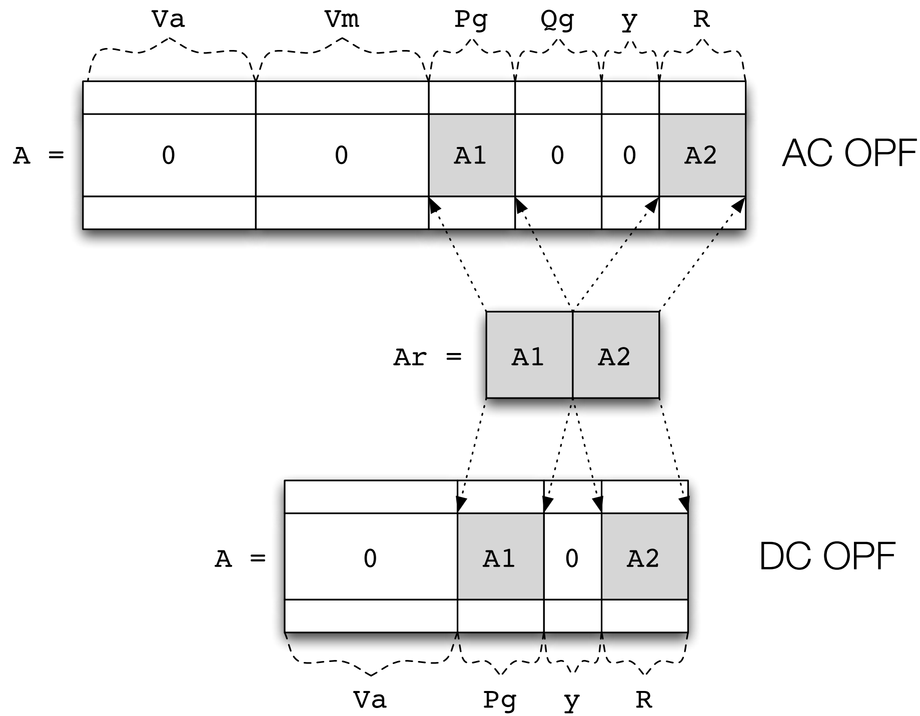

Figure 7-1: Adding Constraints Across Subsets of Variables

```
function om = userfcn_reserves_formulation(om, args)

%% initialize some things
define_constants;
mpc = get_mpc(om);
r = mpc.reserves;
ng  = size(mpc.gen, 1);            %% number of on-line gens

%% variable bounds
Rmin = zeros(ng, 1);               %% bound below by 0
Rmax = r.qty;                      %% bound above by stated max reserve qty ...
k = find(mpc.gen(:, RAMP_10) > 0 & mpc.gen(:, RAMP_10) < Rmax);
Rmax(k) = mpc.gen(k, RAMP_10);     %% ... and ramp rate
Rmax = Rmax / mpc.baseMVA;

%% constraints
I = speye(ng);                     %% identity matrix
Ar = [I I];
Pmax = mpc.gen(:, PMAX) / mpc.baseMVA;
lreq = r.req / mpc.baseMVA;

%% cost
Cw = r.cost * mpc.baseMVA;         %% per unit cost coefficients

%% add them to the model
om = add_vars(om, 'R', ng, [], Rmin, Rmax);
om = add_constraints(om, 'Pg_plus_R', Ar, [], Pmax, {'Pg', 'R'});
om = add_constraints(om, 'Rreq', r.zones, lreq, [], {'R'});
om = add_costs(om, 'Rcost', struct('N', I, 'Cw', Cw), {'R'});
```

7.2.3 int2ext Callback

After the simulation is complete and before the results are printed or saved, MATPOWER converts the case data in the `results` struct back to external indexing by calling the following.

```
results = int2ext(results);
```

This conversion essentially undoes everything that was done by `ext2int`. Generators are restored to their original ordering, buses to their original numbering and all out-of-service or isolated generators, branches and buses are restored.

This callback is invoked from `int2ext` immediately before the resulting case is converted from internal back to external indexing. At this point, the simulation

has been completed and the `results` struct, a superset of the original MATPOWER case struct passed to the OPF, contains all of the results. This `results` struct is passed to the callback, along with any (optional) `args` supplied when the callback was registered via `add_userfcn`. The output of the callback is the updated `results` struct. This is typically used to convert any results to external indexing and populate any corresponding fields in the `results` struct.

The `results` struct contains, in addition to the standard OPF results, solution information related to all of the user-defined variables, constraints and costs. Table 7-2 summarizes where the various data is found. Each of the fields listed in the table is actually a struct whose fields correspond to the named sets created by `add_vars`, `add_constraints` and `add_costs`.

Table 7-2: Results for User-Defined Variables, Constraints and Costs

name	description
`results.var.val`	final value of user-defined variables
`results.var.mu.l`	shadow price on lower limit of user-defined variables
`results.var.mu.u`	shadow price on upper limit of user-defined variables
`results.lin.mu.l`	shadow price on lower (left-hand) limit of linear constraints
`results.lin.mu.u`	shadow price on upper (right-hand) limit of linear constraints
`results.cost`	final value of user-defined costs

In the example code below, the callback function begins by converting the reserves input data in the resulting case (`qty`, `cost` and `zones` fields of `results.reserves`) back to external indexing via calls to `i2e_field`. See the help for `i2e_field` and `i2e_data` for more details on how they can be used.

Then the reserves results of interest are extracted from the appropriate sub-fields of `results.var`, `results.lin` and `results.cost`, converted from per unit to per MW where necessary, and stored with external indexing for the end user in the chosen fields of the `results` struct.

```
function results = userfcn_reserves_int2ext(results, args)

%%-----  convert stuff back to external indexing  -----
%% convert all reserve parameters (zones, costs, qty, rgens)
results = i2e_field(results, {'reserves', 'qty'}, 'gen');
results = i2e_field(results, {'reserves', 'cost'}, 'gen');
results = i2e_field(results, {'reserves', 'zones'}, 'gen', 2);

r = results.reserves;
ng  = size(results.gen, 1);     %% number of on-line gens (internal)
ng0 = size(results.order.ext.gen, 1);   %% number of gens (external)

%%-----  results post-processing  -----
%% get the results (per gen reserves, multipliers) with internal gen indexing
%% and convert from p.u. to per MW units
[R0, Rl, Ru] = getv(results.om, 'R');
R       = results.var.val.R * results.baseMVA;
Rmin    = Rl * results.baseMVA;
Rmax    = Ru * results.baseMVA;
mu_l    = results.var.mu.l.R / results.baseMVA;
mu_u    = results.var.mu.u.R / results.baseMVA;
mu_Pmax = results.lin.mu.u.Pg_plus_R / results.baseMVA;

%% store in results in results struct
z = zeros(ng0, 1);
results.reserves.R       = i2e_data(results, R, z, 'gen');
results.reserves.Rmin    = i2e_data(results, Rmin, z, 'gen');
results.reserves.Rmax    = i2e_data(results, Rmax, z, 'gen');
results.reserves.mu.l    = i2e_data(results, mu_l, z, 'gen');
results.reserves.mu.u    = i2e_data(results, mu_u, z, 'gen');
results.reserves.mu.Pmax = i2e_data(results, mu_Pmax, z, 'gen');
results.reserves.prc     = z;
for k = 1:ng0
    iz = find(r.zones(:, k));
    results.reserves.prc(k) = sum(results.lin.mu.l.Rreq(iz)) / results.baseMVA;
end
results.reserves.totalcost = results.cost.Rcost;
```

7.2.4 `printpf` Callback

The pretty-printing of the standard OPF output is done via a call to `printpf` after the case has been converted back to external indexing. This callback is invoked from within `printpf` after the pretty-printing of the standard OPF output. Inputs are

the `results` struct, the file descriptor to write to, a MATPOWER options struct, and any (optional) `args` supplied via `add_userfcn`. Output is the `results` struct. This is typically used for any additional pretty-printing of results.

In this example, the `out.all` flag in the options struct is checked before printing anything. If it is non-zero, the reserve quantities and prices for each unit are printed first, followed by the per-zone summaries. An additional table with reserve limit shadow prices might also be included.

```
function results = userfcn_reserves_printpf(results, fd, mpopt, args)

%% define named indices into data matrices
[GEN_BUS, PG, QG, QMAX, QMIN, VG, MBASE, GEN_STATUS, PMAX, PMIN, ...
    MU_PMAX, MU_PMIN, MU_QMAX, MU_QMIN, PC1, PC2, QC1MIN, QC1MAX, ...
    QC2MIN, QC2MAX, RAMP_AGC, RAMP_10, RAMP_30, RAMP_Q, APF] = idx_gen;

%%-----  print results  -----
r = results.reserves;
ng = length(r.R);
nrz = size(r.req, 1);
if mpopt.out.all ~= 0
    fprintf(fd, '\n================================================================');
    fprintf(fd, '\n|     Reserves                                                 |');
    fprintf(fd, '\n================================================================');
    fprintf(fd, '\n Gen   Bus   Status  Reserves   Price');
    fprintf(fd, '\n  #     #              (MW)     ($/MW)');
    fprintf(fd, '\n----  -----  ------  --------  --------');
    for k = 1:ng
        fprintf(fd, '\n%3d %6d     %2d ', k, results.gen(k, GEN_BUS), ...
                    results.gen(k, GEN_STATUS));
        if results.gen(k, GEN_STATUS) > 0 && abs(results.reserves.R(k)) > 1e-6
            fprintf(fd, '%10.2f', results.reserves.R(k));
        else
            fprintf(fd, '       -  ');
        end
        fprintf(fd, '%10.2f     ', results.reserves.prc(k));
    end
    fprintf(fd, '\n                     --------');
    fprintf(fd, '\n            Total:%10.2f              Total Cost: $%.2f', ...
        sum(results.reserves.R(r.igr)), results.reserves.totalcost);
    fprintf(fd, '\n');

    fprintf(fd, '\nZone  Reserves   Price  ');
    fprintf(fd, '\n  #     (MW)     ($/MW) ');
    fprintf(fd, '\n----  --------  --------');
    for k = 1:nrz
        iz = find(r.zones(k, :));     %% gens in zone k
        fprintf(fd, '\n%3d%10.2f%10.2f', k, sum(results.reserves.R(iz)), ...
                    results.lin.mu.l.Rreq(k) / results.baseMVA);
    end
    fprintf(fd, '\n');

    %% print binding reserve limit multipliers ...
end
```

7.2.5 `savecase` Callback

The `savecase` is used to save a MATPOWER case struct to an M-file, for example, to save the results of an OPF run. The `savecase` callback is invoked from `savecase` after printing all of the other data to the file. Inputs are the case struct, the file descriptor to write to, the variable prefix (typically `'mpc.'`) and any (optional) `args` supplied via `add_userfcn`. Output is the case struct. The purpose of this callback is to write any non-standard case struct fields to the case file.

In this example, the `zones`, `req`, `cost` and `qty` fields of `mpc.reserves` are written to the M-file. This ensures that a case with reserve data, if it is loaded via `loadcase`, possibly run, then saved via `savecase`, will not lose the data in the `reserves` field. This callback could also include the saving of the output fields if present. The contributed `serialize` function[19] can be very useful for this purpose.

[19] http://www.mathworks.com/matlabcentral/fileexchange/1206

```
function mpc = userfcn_reserves_savecase(mpc, fd, prefix, args)
%
%   mpc = userfcn_reserves_savecase(mpc, fd, mpopt, args)
%
%   This is the 'savecase' stage userfcn callback that prints the M-file
%   code to save the 'reserves' field in the case file. It expects a
%   MATPOWER case struct (mpc), a file descriptor and variable prefix
%   (usually 'mpc.'). The optional args are not currently used.

r = mpc.reserves;

fprintf(fd, '\n%%%%-----  Reserve Data  -----%%%%\n');
fprintf(fd, '%%%% reserve zones, element i, j is 1 iff gen j is in zone i\n');
fprintf(fd, '%sreserves.zones = [\n', prefix);
template = '';
for i = 1:size(r.zones, 2)
    template = [template, '\t%d'];
end
template = [template, ';\n'];
fprintf(fd, template, r.zones.');
fprintf(fd, '];\n');

fprintf(fd, '\n%%%% reserve requirements for each zone in MW\n');
fprintf(fd, '%sreserves.req = [\t%g', prefix, r.req(1));
if length(r.req) > 1
    fprintf(fd, ';\t%g', r.req(2:end));
end
fprintf(fd, '\t];\n');

fprintf(fd, '\n%%%% reserve costs in $/MW for each gen\n');
fprintf(fd, '%sreserves.cost = [\t%g', prefix, r.cost(1));
if length(r.cost) > 1
    fprintf(fd, ';\t%g', r.cost(2:end));
end
fprintf(fd, '\t];\n');

if isfield(r, 'qty')
    fprintf(fd, '\n%%%% max reserve quantities for each gen\n');
    fprintf(fd, '%sreserves.qty = [\t%g', prefix, r.qty(1));
    if length(r.qty) > 1
        fprintf(fd, ';\t%g', r.qty(2:end));
    end
    fprintf(fd, '\t];\n');
end

%% save output fields for solved case ...
```

7.3 Registering the Callbacks

As seen in the fixed zonal reserve example, adding a single extension to the standard OPF formulation is often best accomplished by a set of callback functions. A typical use case might be to run a given case with and without the reserve requirements active, so a simple method for enabling and disabling the whole set of callbacks as a single unit is needed.

The recommended method is to define all of the callbacks in a single file containing a "toggle" function that registers or removes all of the callbacks depending on whether the value of the second argument is `'on'` or `'off'`. The state of the registration of any callbacks is stored directly in the `mpc` struct. In our example, the `toggle_reserves.m` file contains the `toggle_reserves` function as well as the five callback functions.

```
function mpc = toggle_reserves(mpc, on_off)
%TOGGLE_RESERVES Enable, disable or check status of fixed reserve requirements.
%   MPC = TOGGLE_RESERVES(MPC, 'on')
%   MPC = TOGGLE_RESERVES(MPC, 'off')
%   T_F = TOGGLE_RESERVES(MPC, 'status')

if strcmp(upper(on_off), 'ON')
    % <code to check for required 'reserves' fields in mpc>

    %% add callback functions
    mpc = add_userfcn(mpc, 'ext2int', @userfcn_reserves_ext2int);
    mpc = add_userfcn(mpc, 'formulation', @userfcn_reserves_formulation);
    mpc = add_userfcn(mpc, 'int2ext', @userfcn_reserves_int2ext);
    mpc = add_userfcn(mpc, 'printpf', @userfcn_reserves_printpf);
    mpc = add_userfcn(mpc, 'savecase', @userfcn_reserves_savecase);
    mpc.userfcn.status.dcline = 1;
elseif strcmp(upper(on_off), 'OFF')
    mpc = remove_userfcn(mpc, 'savecase', @userfcn_reserves_savecase);
    mpc = remove_userfcn(mpc, 'printpf', @userfcn_reserves_printpf);
    mpc = remove_userfcn(mpc, 'int2ext', @userfcn_reserves_int2ext);
    mpc = remove_userfcn(mpc, 'formulation', @userfcn_reserves_formulation);
    mpc = remove_userfcn(mpc, 'ext2int', @userfcn_reserves_ext2int);
    mpc.userfcn.status.dcline = 0;
elseif strcmp(upper(on_off), 'STATUS')
    if isfield(mpc, 'userfcn') && isfield(mpc.userfcn, 'status') && ...
            isfield(mpc.userfcn.status, 'dcline')
        mpc = mpc.userfcn.status.dcline;
    else
        mpc = 0;
    end
else
    error('toggle_dcline: 2nd argument must be ''on'', ''off'' or ''status''');
end
```

Running a case that includes the fixed reserves requirements is as simple as loading the case, turning on reserves and running it.

```
mpc = loadcase('t_case30_userfcns');
mpc = toggle_reserves(mpc, 'on');
results = runopf(mpc);
```

7.4 Summary

The five callback stages currently defined by MATPOWER are summarized in Table 7-3.

Table 7-3: Callback Functions

name	invoked …	typical use
`ext2int`	… from `ext2int` immediately after case data is converted from external to internal indexing.	Check consistency of input data, convert to internal indexing.
`formulation`	… from `opf` after OPF Model (`om`) object is initialized with standard OPF formulation.	Modify OPF formulation, by adding user-defined variables, constraints, costs.
`int2ext`	… from `int2ext` immediately before case data is converted from internal back to external indexing.	Convert data back to external indexing, populate any additional fields in the `results` struct.
`printpf`	… from `printpf` after pretty-printing the standard OPF output.	Pretty-print any results not included in standard OPF.
`savecase`	… from `savecase` after printing all of the other case data to the file.	Write non-standard case struct fields to the case file.

7.5 Example Extensions

MATPOWER includes three OPF extensions implementing via callbacks, respectively, the co-optimization of energy and reserves, interface flow limits and dispatchable DC transmission lines.

7.5.1 Fixed Zonal Reserves

This extension is a more complete version of the example of fixed zonal reserve requirements used for illustration above in Sections 7.2 and 7.3. The details of the extensions to the standard OPF problem are given in equations (7.2)–(7.5) and a description of the relevant input and output data structures is summarized in Table 7-1.

The code for implementing the callbacks can be found in `toggle_reserves`. A wrapper around `runopf` that turns on this extension before running the OPF is

provided in runopf_w_res, allowing you to run a case with an appropriate reserves field, such as t_case30_userfcns, as follows.

```
results = runopf_w_res('t_case30_userfcns');
```

See help runopf_w_res and help toggle_reserves for more information. Examples of using this extension and a case file defining the necessary input data can be found in t_opf_userfcns and t_case30_userfcns, respectively. Additional tests for runopf_w_res are included in t_runopf_w_res.

7.5.2 Interface Flow Limits

This extension adds interface flow limits based on flows computed from a DC network model. It is implemented in toggle_iflims. A flow interface k is defined as a set $\mathcal{B}_k$ of branch indices i and a direction for each branch. If p_i represents the real power flow ("from" bus $\rightarrow$ "to" bus) in branch i and d_i is equal to 1 or -1 to indicate the direction,[20] then the interface flow f_k for interface k is defined as

$$f_k(\Theta) = \sum_{i \in \mathcal{B}_k} d_i p_i(\Theta), \tag{7.6}$$

where each branch flow p_i is an approximation calculated as a linear function of the bus voltage angles based on the DC power flow model from equation (3.29).

This extension adds to the OPF problem a set of $n_{\rm if}$ doubly-bounded constraints on these flows.

$$F_k^{\min} \leq f_k(\Theta) \leq F_k^{\max} \quad \forall k \in \mathcal{I}_f \tag{7.7}$$

where $F_k^{\min}$ and $F_k^{\max}$ are the specified lower and upper bounds on the interface flow, and $\mathcal{I}_f$ is a the set indices of interfaces whose flow limits are to be enforced.

The data for the problem is specified in an additional if field in the Matpower case struct mpc. This field is itself a struct with two sub-fields, map and lims, used for input data, and two others, P and mu, used for output data. The format of this data is described in detail in Tables 7-4 and 7-5.

See help toggle_iflims for more information. Examples of using this extension and a case file defining the necessary input data for it can be found in t_opf_userfcns and t_case30_userfcns, respectively. Note that, while this extension can be used for AC OPF problems, the actual AC interface flows will not necessarily be limited to the specified values, since it is a DC flow approximation that is used for the constraint.

[20]If $d_i = 1$, the definitions of the positive flow direction for the branch and the interface are the same. If $d_i = -1$, they are opposite.

Table 7-4: Input Data Structures for Interface Flow Limits

name	description
`mpc`	MATPOWER case struct
`if`	additional field in `mpc` containing input parameters for interface flow limits in the following sub-fields:
`map`	$(\sum_k n_k)\times 2$ matrix defining the interfaces, where n_k is the number branches that belong to interface k. The n_k branches of interface k are defined by n_k rows in the matrix, where the first column in each is equal to k and the second is equal to the corresponding branch index i multiplied by d_i to indicate the direction.
`lims`	$n_{\text{if}} \times 3$ matrix of interface limits, where n_{if} is the number of interface limits to be enforced. The first column is the index k of the interface, and the second and third columns are $F_k^{\min}$ and $F_k^{\max}$, the lower and upper limits respectively, on the DC model flow limits (in MW) for the interface.

Table 7-5: Output Data Structures for Interface Flow Limits

name	description
`results`	OPF results struct, superset of `mpc` with additional fields for output data
`if`	additional field in `results` containing output parameters for interface flow limits in the following sub-fields:
`P`	$n_{\text{if}} \times 1$ vector of actual flow in MW across the corresponding interface (as measured at the "from" end of associated branches)
`mu.l`	$n_{\text{if}} \times 1$ vector of shadow prices on lower flow limits (u/MW)†
`mu.u`	$n_{\text{if}} \times 1$ vector of shadow prices on upper flow limits (u/MW)†

† Here we assume the objective function has units u.

Running a case that includes the interface flow limits is as simple as loading the case, turning on the extension and running it. Unlike with the reserves extension, MATPOWER does not currently have a wrapper function to automate this.

```
mpc = loadcase('t_case30_userfcns');
mpc = toggle_iflims(mpc, 'on');
results = runopf(mpc);
```

7.5.3 DC Transmission Lines

Beginning with version 4.1, MATPOWER also includes a simple model for dispatchable DC transmission lines. While the implementation is based on the extensible OPF architecture described above, it can be used for simple power flow problems as well,

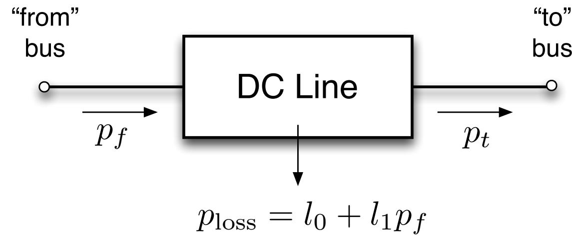

Figure 7-2: DC Line Model

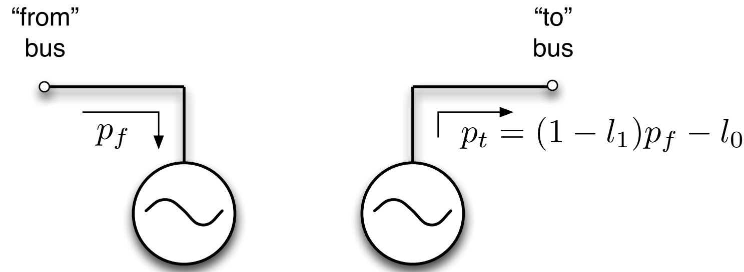

Figure 7-3: Equivalent "Dummy" Generators

in which the case the (OPF only) `formulation` callback is skipped.

A DC line in MATPOWER is modeled as two linked "dummy" generators, as shown in Figures 7-2 and 7-3, one with negative capacity extracting real power from the network at the "from" end of the line and another with positive capacity injecting power into the network at the "to" end. These dummy generators are added by the `ext2int` callback and removed by the `int2ext` callback. The real power flow p_f on the DC line at the "from" end is defined to be equal to the negative of the injection of corresponding dummy generator. The flow at the "to" end p_t is defined to be equal to the injection of the corresponding generator.

MATPOWER links the values of p_f and p_t using the following relationship, which includes a linear approximation of the real power loss in the line.

$$\begin{aligned} p_t &= p_f - p_{\text{loss}} \\ &= p_f - (l_0 + l_1 p_f) \\ &= (1 - l_1)p_f - l_0 \end{aligned} \tag{7.8}$$

Here the linear coefficient l_1 is assumed to be a small ($\ll 1$) positive number. Obviously, this is not applicable for bi-directional lines, where the flow could go either

direction, resulting in decreasing losses for increasing flow in the "to" → "from" direction. There are currently two options for handling bi-directional lines. The first is to use a constant loss model by setting $l_1 = 0$. The second option is to create two separate but identical lines oriented in opposite directions. In this case, it is important that the lower limit on the flow and the constant term of the loss model l_0 be set to zero to ensure that only one of the two lines has non-zero flow at a time.[21]

Upper and lower bounds on the value of the flow can be specified for each DC line, along with an optional operating cost. It is also assumed that the terminals of the line have a range of reactive power capability that can be used to maintain a voltage setpoint. Just as with a normal generator, the voltage setpoint is only used for simple power flow; the OPF dispatches the voltage anywhere between the lower and upper bounds specified for the bus. Similarly, in a simple power flow the input value for p_f and the corresponding value for p_t, computed from (7.8), are used to specify the flow in the line.

Most of the data for DC lines is stored in a `dcline` field in the MATPOWER case struct `mpc`. This field is a matrix similar to the `branch` matrix, where each row corresponds to a particular DC line. The columns of the matrix are defined in Table B-5 and include connection bus indices, line status, flows, terminal reactive injections, voltage setpoints, limits on power flow and VAr injections, and loss parameters. Also, similar to the `branch` or `gen` matrices, some of the columns are used for input values, some for results, and some, such as `PF` can be either input or output, depending on whether the problem is a simple power flow or an optimal power flow. The `idx_dcline` function defines a set of constants for use as named column indices for the `dcline` matrix.

An optional `dclinecost` matrix, in the same form as `gencost`, can be used to specify a cost to be applied to p_f in the OPF. If the `dclinecost` field is not present, the cost is assumed to be zero.

MATPOWER's DC line handling is implemented in `toggle_dcline` and examples of using it can be found in `t_dcline`. The case file `t_case9_dcline` includes some example DC line data. See `help toggle_dcline` for more information.

Running a case that includes DC lines is as simple as loading the case, turning on the extension and running it. Unlike with the reserves extension, MATPOWER does not currently have a wrapper function to automate this.

```
mpc = loadcase('t_case9_dcline');
mpc = toggle_dcline(mpc, 'on');
results = runopf(mpc);
```

[21] A future version may make the handling of this second option automatic.

7.5.4 DC OPF Branch Flow Soft Limits

Beginning with version 5.0, MATPOWER includes an extension that replaces the branch flow limits on specified branches in a DC optimal power flow with soft limits, that is, limits that can be violated with some linear penalty cost. This can be useful in identifying the cause of infeasibility in some DC optimal power flow problems. The extension is implemented in `toggle_softlims`.

A new variable f_{v}^i is defined to represent the flow limit violation on branch i. This variable is constrained to be positive and the sum of f_{v}^i and the flow limit must be greater than both the flow and the negative of the flow on the branch, both of which are expressed as functions of the bus voltage angles, from (3.29). The three constraints on the new flow violation variable can be written as

$$f_{\mathrm{v}}^i \geq 0 \tag{7.9}$$

$$f_{\mathrm{v}}^i + p_f^{i,\max} \geq \quad p_f^i = \quad B_f^i \Theta + p_{f,\mathrm{shift}}^i \tag{7.10}$$

$$f_{\mathrm{v}}^i + p_f^{i,\max} \geq -p_f^i = -B_f^i \Theta - p_{f,\mathrm{shift}}^i, \tag{7.11}$$

where the feasible area is illustrated in Figure 7-4. Furthermore, a simple linear cost coefficient c_{v}^i is applied to each flow violation variable, so that the additional user defined cost term from (6.21) looks like

$$f_u(x, z) = {c_{\mathrm{v}}}^{\mathsf{T}} f_{\mathrm{v}}. \tag{7.12}$$

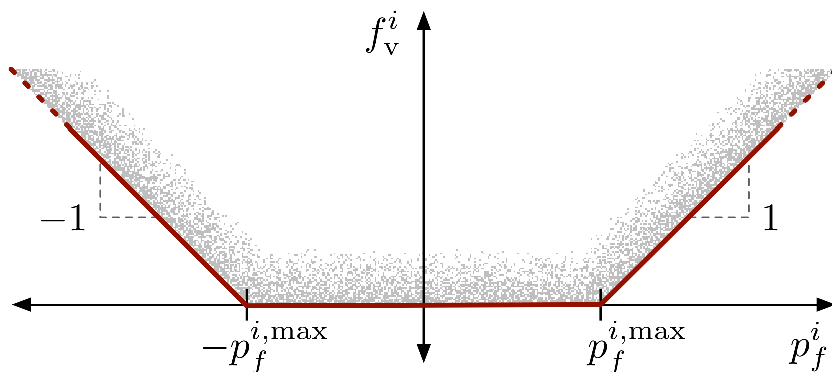

Figure 7-4: Feasible Region for Branch Flow Violation Constraints

The data for the problem is specifed in an additional `softlims` field in the MATPOWER case struct `mpc`. This field is itself a struct with two optional sub-fields,

idx and cost, for input and two others, overload and ovl_cost used for output data. Flow constraint shadow prices are reported in the usual columns of the branch matrix in the results. The format for this data is detailed in Tables 7-6 and 7-7.

Table 7-6: Input Data Structures for DC OPF Branch Flow Soft Limits

name	description
mpc	MATPOWER case struct
softlims	additional field in mpc containing input parameters for DC OPF branch flow soft limits in the following sub-fields:
idx	$n_{\mathrm{sl}} \times 1$ vector of branch indices of the branches whose flow limits are to be converted to soft limits. If empty, it defaults to include all on-line branches with non-zero branch ratings.
cost	$n_{\mathrm{sl}} \times 1$ vector of cost coefficients c_{v}. This is a per MW cost applied to any flow overload in the corresponding branch. Alternatively, the cost can be specified as a scalar, in which case it is used for each soft limit. The default value if not specified is $1000/MW.

Table 7-7: Output Data Structures for DC OPF Branch Flow Soft Limits

name	description
results	OPF results struct, superset of mpc with additional fields for output data
softlims	additional field in results containing output parameters for DC OPF branch flow soft limits in the following sub-fields:
overload	$n_l \times 1$ vector of branch flow overloads in MW
ovl_cost	$n_l \times 1$ vector of branch flow overload penalty costs in u/MW†
branch(:, MU_SF)	for branches whose limits have been replaced with soft limits, these
branch(:, MU_ST)	contain the Kuhn-Tucker multipliers on the soft limit constraints.‡

† Here we assume the objective function has units u.
‡ When there is no violation of the soft limit, this shadow price is the same as it would be for a hard limit. When there is a violation, it is equal to the corresponding user-specified constraint violation cost c_{v}^i.

See help toggle_softlims for more information. Examples of using this extension can be found in t_opf_softlims. Note that, while this extension can be used for AC OPF problems, the flow violations will not be actual AC flow violations, since the computed violations are based on a DC flow model, and it is those values that incur the penalty cost.

Running a case that includes the interface flow limits is as simple as loading the case, turning on the extension and running it. Unlike with the reserves extension, MATPOWER does not currently have a wrapper function to automate this.

```
mpc = loadcase('case2383wp');
mpc = toggle_softlims(mpc, 'on');
results = rundcopf(mpc);
```

8 Unit De-commitment Algorithm

The standard OPF formulation described in the previous section has no mechanism for completely shutting down generators which are very expensive to operate. Instead they are simply dispatched at their minimum generation limits. MATPOWER includes the capability to run an optimal power flow combined with a unit de-commitment for a single time period, which allows it to shut down these expensive units and find a least cost commitment and dispatch. To run this for `case30`, for example, type:

```
>> runuopf('case30')
```

By default, `runuopf` is based on the AC optimal power flow problem. To run a DC OPF, the `model` option must be set to `'DC'`. For convenience, MATPOWER provides a function `runduopf` which is simply a wrapper that sets the `model` option to `'DC'` before calling `runuopf`.

MATPOWER uses an algorithm similar to dynamic programming to handle the de-commitment. It proceeds through a sequence of stages, where stage N has N generators shut down, starting with $N = 0$, as follows:

Step 1: Begin at stage zero ($N = 0$), assuming all generators are on-line with all limits in place.

Step 2: If the sum of the minimum generation limits for all on-line generators is less than the total system demand, then go to Step 3. Otherwise, go to the next stage, $N = N + 1$, shut down the generator whose average per-MW cost of operating at its minimum generation limit is greatest and repeat Step 2.

Step 3: Solve a normal OPF. Save the solution as the current best.

Step 4: Go to the next stage, $N = N + 1$. Using the best solution from the previous stage as the base case for this stage, form a candidate list of generators with minimum generation limits binding. If there are no candidates, skip to Step 6.

Step 5: For each generator on the candidate list, solve an OPF to find the total system cost with this generator shut down. Replace the current best solution with this one if it has a lower cost. If any of the candidate solutions produced an improvement, return to Step 4.

Step 6: Return the current best solution as the final solution.

It should be noted that the method employed here is simply a heuristic. It does not guarantee that the least cost commitment of generators will be found. It is also rather computationally expensive for larger systems and was implemented as a simple way to allow an OPF-based "smart-market", such as described in Appendix F, the option to reject expensive offers while respecting the minimum generation limits on generators.

9 Miscellaneous Matpower Functions

This section describes a number of additional Matpower functions that users may find useful. The descriptions here are simply brief summaries, so please use the Matlab `help` function to get the full details on each function.

9.1 Input/Output Functions

9.1.1 loadcase

```
mpc = loadcase(casefile)
```

The `loadcase` function provides the canonical way of loading a Matpower case from a file or struct. It takes as input either a struct or the name of an M-file or MAT-file in the Matlab path (`casefile`) and returns a standard Matpower case struct (`mpc`). It can also convert from the older version 1 case file format to the current format. This function allows a case to be loaded, and potentially modified, before calling one of the main simulation functions such as `runpf` or `runopf`.

9.1.2 savecase

```
savecase(fname, mpc)
savecase(fname, mpc, version)
savecase(fname, comment, mpc)
savecase(fname, comment, mpc, version)
fname = savecase(fname, ...)
```

The `savecase` function writes out a Matpower case file, given a name for the file to be created or overwritten (`fname`), and a Matpower case struct (`mpc`). If `fname` ends with `'.mat'` it saves the case as a MAT-file, otherwise it saves it as an M-file. Optionally returns the filename, with extension added if necessary. The optional `comment` argument is either string (single line comment) or a cell array of strings which are inserted as comments in the help section of the file. If the optional `version` argument is `'1'` it will modify the data matrices to version 1 format before saving.

9.1.3 cdf2mpc

```
mpc = cdf2mpc(cdf_file_name)
mpc = cdf2mpc(cdf_file_name, verbose)
mpc = cdf2mpc(cdf_file_name, mpc_name)
mpc = cdf2mpc(cdf_file_name, mpc_name, verbose)
[mpc, warnings] = cdf2mpc(cdf_file_name, ...)
```

The `cdf2mpc` function converts an IEEE Common Data Format (CDF) data file into a MATPOWER case struct. Given an optional file name `mpc_name`, it can save the converted case to a MATPOWER case file. Warnings generated during the conversion process can be optionally returned in the `warnings` argument.

Since the IEEE CDF format does not contain all of the data needed to run an optimal power flow, some data, such as voltage limits, generator limits and generator costs are created by `cdf2mpc`. See `help cdf2mpc` for details.

9.1.4 psse2mpc

```
mpc = psse2mpc(rawfile_name)
mpc = psse2mpc(rawfile_name, verbose)
mpc = psse2mpc(rawfile_name, verbose, rev)
mpc = psse2mpc(rawfile_name, mpc_name)
mpc = psse2mpc(rawfile_name, mpc_name, verbose)
mpc = psse2mpc(rawfile_name, mpc_name, verbose, rev)
[mpc, warnings] = psse2mpc(rawfile_name, ...)
```

The `psse2mpc` function converts a PSS/E RAW data file into a MATPOWER case struct. Given an optional file name `mpc_name`, it can save the converted case to a MATPOWER case file. Warnings generated during the conversion process can be optionally returned in the `warnings` argument. By default, `psse2mpc` attempts to determine the revision of the PSS/E RAW file from the contents, but the user can specify an explicit revision number to use in the optional `rev` argument.

9.2 System Information

9.2.1 case_info

```
case_info(mpc)
case_info(mpc, fd)
[groups, isolated] = case_info(mpc)
```

The case_info function prints out detailed information about a MATPOWER case, including connectivity information, summarizing the generation, load and other data by interconnected island. It can optionally print the output to an open file, whose file identifier (as returned by fopen) is specified in the optional second parameter fd. Optional return arguments include groups and isolated buses, as returned by the find_islands function.

9.2.2 compare_case

```
compare_case(mpc1, mpc2)
```

Compares the bus, branch and gen matrices of two MATPOWER cases and prints a summary of the differences. For each column of the matrix it prints the maximum of any non-zero differences.

9.2.3 find_islands

```
groups = find_islands(mpc)
[groups, isolated] = find_islands(mpc)
```

The find_islands function returns the islands in a network. The return value groups is a cell array of vectors of the bus indices for each island. The second and optional return value isolated is a vector of indices of isolated buses that have no connecting branches.

9.2.4 get_losses

```
loss = get_losses(results)
loss = get_losses(baseMVA, bus, branch)

[loss, chg] = get_losses(results)
[loss, fchg, tchg] = get_losses(results)
[loss, fchg, tchg, dloss_dv] = get_losses(results)
[loss, fchg, tchg, dloss_dv, dchg_dvm] = get_losses(results)
```

The get_losses function computes branch series losses, and optionally reactive injections from line charging, as functions of bus voltages and branch parameters, using the following formulae for a branch, as described in Section 3.2, connecting bus f to bus t:

$$\text{loss}_i = \frac{\left|\frac{v_f}{\tau e^{j\theta_{\text{shift}}}} - v_t\right|^2}{r_s - jx_s} \tag{9.1}$$

$$f_{\text{chg}} = \left|\frac{v_f}{\tau e^{j\theta_{\text{shift}}}}\right|^2 \frac{b_c}{2} \tag{9.2}$$

$$t_{\text{chg}} = |v_t|^2 \frac{b_c}{2} \tag{9.3}$$

It can also optionally compute the partial derivatives of the line losses and reactive charging injections with respect to voltage angles and magnitudes.

9.2.5 margcost

```
marginalcost = margcost(gencost, Pg)
```

The `margcost` function computes the marginal cost for generators given a matrix in `gencost` format and a column vector or matrix of generation levels. The return value has the same dimensions as `Pg`. Each row of `gencost` is used to evaluate the cost at the output levels specified in the corresponding row of `Pg`. The rows of `gencost` can specify either polynomial or piecewise linear costs and need not be uniform.

9.2.6 isload

```
TorF = isload(gen)
```

The `isload` function returns a column vector of 1's and 0's. The 1's correspond to rows of the `gen` matrix which represent dispatchable loads. The current test is $P_{\min} < 0$ and $P_{\max} = 0$.

9.2.7 printpf

```
printpf(results, fd, mpopt)
```

The `printpf` function prints power flow and optimal power flow results, as returned to `fd`, a file identifier which defaults to STDOUT (the screen). The details of what gets printed are controlled by an optional MATPOWER options struct `mpopt`.

9.2.8 total_load

```
Pd = total_load(bus)
Pd = total_load(bus, gen, load_zone, opt)
[Pd, Qd] = total_load(...)
```

The total_load function returns a vector of total load in each load zone. The opt argument controls whether it includes fixed loads, dispatchable loads or both, and for dispatchable loads, whether to use the nominal or realized load values. The load_zone argument defines the load zones across which loads will be summed. It uses the BUS_AREA column (7) of the bus matrix by default. The string value 'all' can be used to specify a single zone including the entire system. The reactive demands are also optionally available as an output.

9.2.9 totcost

```
totalcost = totcost(gencost, Pg)
```

The totcost function computes the total cost for generators given a matrix in gencost format and a column vector or matrix of generation levels. The return value has the same dimensions as Pg. Each row of gencost is used to evaluate the cost at the output levels specified in the corresponding row of Pg. The rows of gencost can specify either polynomial or piecewise linear costs and need not be uniform.

9.3 Modifying a Case

9.3.1 extract_islands

```
mpc_array = extract_islands(mpc)
mpc_array = extract_islands(mpc, groups)
mpc_k = extract_islands(mpc, k)
mpc_k = extract_islands(mpc, groups, k)
mpc_k = extract_islands(mpc, k, custom)
mpc_k = extract_islands(mpc, groups, k, custom)
```

The extract_islands function extracts individual islands in a network that is not fully connected. The original network is specified as a MATPOWER case struct (mpc) and the result is returned as a cell array of case structs, or as a single case struct. Supplying the optional group avoids the need to traverse the network again, saving time on large systems. A final optional argument custom is a struct that can be used to indicate custom fields of mpc from which to extract data corresponding to buses generators, branches or DC lines.

9.3.2 load2disp

```
mpc = load2disp(mpc0);
mpc = load2disp(mpc0, fname);
mpc = load2disp(mpc0, fname, idx);
mpc = load2disp(mpc0, fname, idx, voll);
```

The `load2disp` function takes a MATPOWER case `mpc0`, converts fixed loads to dispatchable loads, curtailable at a specific price, and returns the resulting case struct `mpc`. It can optionally save the resulting case to a file (`fname`), convert loads only at specific buses (`idx`), and set the value of lost load (`voll`) to be used as the curtailment price (default is $5,000/MWh).

9.3.3 modcost

```
newgencost = modcost(gencost, alpha)
newgencost = modcost(gencost, alpha, modtype)
```

The `modcost` function can be used to modify generator cost functions by shifting or scaling them, either horizontally or vertically. The `alpha` argument specifies the numerical value of the modification, and `modtype` defines the type of modification as a string that takes one of the following values: `'SCALE_F'` (default), `'SCALE_X'`, `'SHIFT_F'`, or `'SHIFT_X'`.

9.3.4 scale_load

```
bus = scale_load(load, bus);
[bus, gen] = scale_load(load, bus, gen, load_zone, opt)
[bus, gen, gencost] = ...
    scale_load(load, bus, gen, load_zone, opt, gencost)
```

The `scale_load` function is used to scale active (and optionally reactive) loads in each zone by a zone-specific ratio, i.e. $R(k)$ for zone k. The amount of scaling for each zone, either as a direct scale factor or as a target quantity, is specified in `load`. The load zones are defined by `load_zone`, and `opt` specifies the type of scaling (factor or target quantity) and which loads are affected (active, reactive or both and fixed, dispatchable or both). The costs (`gencost`) associated with dispatchable loads can also be optionally scaled with the loads.

9.4 Conversion between External and Internal Numbering

9.4.1 ext2int, int2ext

```
mpc_int = ext2int(mpc_ext)
mpc_ext = int2ext(mpc_int)
```

These functions convert a MATPOWER case struct from external to internal, and from internal to external numbering, respectively. `ext2int` first removes all isolated buses, off-line generators and branches, and any generators or branches connected to isolated buses. Then the buses are renumbered consecutively, beginning at 1, and the generators are sorted by increasing bus number. Any `'ext2int'` callback routines registered in the case are also invoked automatically. All of the related indexing information and the original data matrices are stored in an `'order'` field in the struct to be used later by codeint2ext to perform the reverse conversions. If the case is already using internal numbering it is returned unchanged.

9.4.2 e2i_data, i2e_data

```
val = e2i_data(mpc, val, ordering)
val = e2i_data(mpc, val, ordering, dim)
val = i2e_data(mpc, val, oldval, ordering)
val = i2e_data(mpc, val, oldval, ordering, dim)
```

These functions can be used to convert other data structures from external to internal indexing and vice versa. When given a case struct (`mpc`) that has already been converted to internal indexing, `e2i_data` can be used to convert other data structures as well by passing in 2 or 3 extra parameters in addition to the case struct. If the value passed in the second argument (`val`) is a column vector or cell array, it will be converted according to the `ordering` specified by the third argument (described below). If `val` is an n-dimensional matrix or cell array, then the optional fourth argument (`dim`, default = 1) can be used to specify which dimension to reorder. The return value in this case is the value passed in, converted to internal indexing.

The third argument, `ordering`, is used to indicate whether the data corresponds to bus-, gen- or branch-ordered data. It can be one of the following three strings: `'bus'`, `'gen'` or `'branch'`. For data structures with multiple blocks of data, ordered by bus, gen or branch, they can be converted with a single call by specifying `ordering` as a cell array of strings.

Any extra elements, rows, columns, etc. beyond those indicated in `ordering`, are not disturbed.

The function `i2e_data` performs the opposite conversion, from internal back to external indexing. It also assumes that `mpc` is using internal indexing, and the only difference is that it also includes an `oldval` argument used to initialize the return value before converting `val` to external indexing. In particular, any data corresponding to off-line gens or branches or isolated buses or any connected gens or branches will be taken from `oldval`, with `val` supplying the rest of the returned data.

9.4.3 `e2i_field`, `i2e_field`

```
mpc = e2i_field(mpc, field, ordering)
mpc = e2i_field(mpc, field, ordering, dim)
mpc = i2e_field(mpc, field, ordering)
mpc = i2e_field(mpc, field, ordering, dim)
```

These functions can be used to convert additional fields in `mpc` from external to internal indexing and vice versa. When given a case struct that has already been converted to internal indexing, `e2i_field` can be used to convert other fields as well by passing in 2 or 3 extra parameters in addition to the case struct.

The second argument (`field`) is a string or cell array of strings, specifying a field in the case struct whose value should be converted by a corresponding call to `e2i_data`. The field can contain either a numeric or a cell array. The converted value is stored back in the specified field, the original value is saved for later use and the updated case struct is returned. If `field` is a cell array of strings, they specify nested fields.

The third and optional fourth arguments (`ordering` and `dim`) are simply passed along to the call to `e2i_data`.

Similarly, `i2e_field` performs the opposite conversion, from internal back to external indexing. It also assumes that `mpc` is using internal indexing and utilizes the original data stored by `e2i_field`, calling `i2e_data` to do the conversion work.

9.5 Forming Standard Power Systems Matrices

9.5.1 `makeB`

```
[Bp, Bpp] = makeB(baseMVA, bus, branch, alg)
```

The `makeB` function builds the two matrices B' and B'' used in the fast-decoupled power flow. The `alg`, which can take values `'FDXB'` or `'FDBX'`, determines whether the matrices returned correspond to the XB or BX version of the fast-decoupled power flow.

9.5.2 makeBdc

```
[Bbus, Bf, Pbusinj, Pfinj] = makeBdc(baseMVA, bus, branch)
```

The function builds the B matrices, B_{dc} (`Bbus`) and B_f (`Bf`), and phase shift injections, P_{dc} (`Pbusinj`) and $P_{f,\text{shift}}$ (`Pfinj`), for the DC power flow model as described in (3.29) and (4.7).

9.5.3 makeJac

```
J = makeJac(mpc)
J = makeJac(mpc, fullJac)
[J, Ybus, Yf, Yt] = makejac(mpc)
```

The `makeJac` function forms the power flow Jacobian and, optionally, the system admittance matrices. Bus numbers in the input case must be consecutive beginning at 1 (i.e. internal indexing). If the `fullJac` argument is present and true, it returns the full Jacobian (sensitivities of all bus injections with respect to all voltage angles and magnitudes) as opposed to the reduced version used in the Newton power flow updates.

9.5.4 makeYbus

```
[Ybus, Yf, Yt] = makeYbus(mpc)
[Ybus, Yf, Yt] = makeYbus(baseMVA, bus, branch)
```

The `makeYbus` function builds the bus admittance matrix and branch admittance matrices from (3.11)–(3.13). Bus numbers in the input case must be consecutive beginning at 1 (i.e. internal indexing).

9.6 Miscellaneous

9.6.1 define_constants

```
define_constants
```

The `define_constants` is a convenience script that defines a set of constants to be used as named column indices into the `bus`, `branch`, `gen` and `gencost` data matrices. The purpose is to avoid having to remember column numbers and to allow code to be more robust against potential future changes to the MATPOWER case data format.

9.6.2 have_fcn

```
TorF = have_fcn(tag)
```

The `have_fcn` function provides a unified mechanism for testing for optional functionality, such as the presence of certain solvers, or to detect whether the code is running under MATLAB or Octave. Since its results are cached they allow for a very quick way to check frequently for functionality that may initially be a bit more costly to determine.

9.6.3 mpver

```
mpver
v = mpver
v = mpver('all')
```

The `mpver` function returns the current MATPOWER version number. With the optional `'all'` argument, it returns a struct with the fields `'Name'`, `'Version'`, `'Release'` and `'Date'` (all strings). Calling `mpver` without assigning the return value prints the version and release date of the current installation of MATPOWER, MATLAB (or Octave), the Optimization Toolbox, MIPS and any optional MATPOWER packages.

9.6.4 nested_struct_copy

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

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

10 Acknowledgments

The authors would like to acknowledge contributions from others who have helped make MATPOWER what it is today. First we would like to acknowledge the input and support of Bob Thomas throughout the development of MATPOWER. Thanks to Chris DeMarco, one of our PSERC associates at the University of Wisconsin, for the technique for building the Jacobian matrix. Our appreciation to Bruce Wollenberg for all of his suggestions for improvements to version 1. The enhanced output functionality in version 2.0 is primarily due to his input. Thanks also to Andrew Ward for code which helped us verify and test the ability of the OPF to optimize reactive power costs. Thanks to Alberto Borghetti for contributing code for the Gauss-Seidel power flow solver and to Mu Lin for contributions related to power flow reactive power limits. Real power line limits were suggested by Pan Wei. Thanks to Roman Korab for data for the Polish system. Some state estimation code was contributed by James S. Thorp and Rui Bo contributed additional code for state estimation and continuation power flow. MATPOWER was improved in various ways in response to Doug Mitarotonda's contributions and suggestions.

Thanks also to many others who have contributed code, testing time, bug reports and suggestions over the years. And, last but not least, thanks to all of the many users who, by using MATPOWER in their own work, have helped to extend the contribution of MATPOWER to the field of power systems far beyond what we could do on our own.

Appendix A MIPS – MATLAB Interior Point Solver

Beginning with version 4, MATPOWER includes a new primal-dual interior point solver called MIPS, for MATLAB Interior Point Solver. It is implemented in pure-MATLAB code, derived from the MEX implementation of the algorithms described in [17, 24].

This solver has application outside of MATPOWER to general nonlinear optimization problems of the following form:

$$\min_{x} f(x) \tag{A.1}$$

subject to

$$g(x) = 0 \tag{A.2}$$

$$h(x) \leq 0 \tag{A.3}$$

$$l \leq Ax \leq u \tag{A.4}$$

$$x_{\min} \leq x \leq x_{\max} \tag{A.5}$$

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

The solver is implemented by the `mips` function, which can be called as follows,

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

where the input and output arguments are described in Tables A-1 and A-2, 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] = mips(problem);
```

The calling syntax is nearly identical to that used by `fmincon` from MATLAB's Optimization Toolbox. The primary difference 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. Internally, equality constraints are handled explicitly and determined at run-time based on the values of l and u.

The user-defined functions for evaluating the objective function, constraints and Hessian are identical to those required by `fmincon`, with one exception described

Table A-1: Input Arguments for `mips`[†]

name	description
`f_fcn`	Handle to a function that evaluates the objective function, its gradients and Hessian[‡] for a given value of x. Calling syntax for this function: `[f, df, d2f] = f_fcn(x)`
`x0`	Starting value of optimization vector x.
`A`, `l`, `u`	Define the optional linear constraints $l \leq Ax \leq u$. Default values for the elements of `l` and `u` are `-Inf` and `Inf`, respectively.
`xmin`, `xmax`	Optional lower and upper bounds on the x variables, defaults are `-Inf` and `Inf`, respectively.
`gh_fcn`	Handle to function that evaluates the optional nonlinear constraints and their gradients for a given value of x. Calling syntax for this function is: `[h, g, dh, dg] = gh_fcn(x)`
`hess_fcn`	Handle to function that computes the Hessian[‡]of the Lagrangian for given values of x, λ and μ, where λ and μ are the multipliers on the equality and inequality constraints, g and h, respectively. The calling syntax for this function is: `Lxx = hess_fcn(x, lam, cost_mult)`, where $\lambda =$ `lam.eqnonlin`, $\mu =$ `lam.ineqnonlin` and `cost_mult` is a parameter used to scale the objective function
`opt`	Optional options structure with fields, all of which are also optional, described in Table A-3.
`problem`	Alternative, single argument input struct with fields corresponding to arguments above.

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

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

$$\mathcal{L}(x, \lambda, \mu, \sigma) = \sigma f(x) + \lambda^{\mathsf{T}} g(x) + \mu^{\mathsf{T}} h(x) \tag{A.6}$$

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

Table A-2: Output Arguments for `mips`

name	description	
`x`	solution vector	
`f`	final objective function value	
`exitflag`	exit flag	
	1 – first order optimality conditions satisfied	
	0 – maximum number of iterations reached	
	-1 – numerically failed	
`output`	output struct with fields	
	`iterations`	number of iterations performed
	`hist`	struct array with trajectories of the following: `feascond`, `gradcond`, `compcond`, `costcond`, `gamma`, `stepsize`, `obj`, `alphap`, `alphad`
	`message`	exit message
`lambda`	struct containing the Langrange and Kuhn-Tucker multipliers on the constraints, with fields:	
	`eqnonlin`	nonlinear equality constraints
	`ineqnonlin`	nonlinear inequality constraints
	`mu_l`	lower (left-hand) limit on linear constraints
	`mu_u`	upper (right-hand) limit on linear constraints
	`lower`	lower bound on optimization variables
	`upper`	upper bound on optimization variables

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

A.1 Example 1

The following code shows a simple example of using `mips` to solve a 2-dimensional unconstrained optimization of Rosenbrock's "banana" function[22]

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2. \tag{A.7}$$

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

[22] http://en.wikipedia.org/wiki/Rosenbrock_function

Table A-3: Options for `mips`†

name	default	description
`opt.verbose`	0	controls level of progress output displayed 0 – print no progress info 1 – print a little progress info 2 – print a lot of progress info 3 – print all progress info
`opt.feastol`	10^{-6}	termination tolerance for feasibility condition
`opt.gradtol`	10^{-6}	termination tolerance for gradient condition
`opt.comptol`	10^{-6}	termination tolerance for complementarity condition
`opt.costtol`	10^{-6}	termination tolerance for cost condition
`opt.max_it`	150	maximum number of iterations
`opt.step_control`	0	set to 1 to enable step-size control
`opt.sc.red_it`	20	max number of step-size reductions if step-control is on
`opt.cost_mult`	1	cost multiplier used to scale the objective function for improved conditioning. Note: This value is also passed as the 3rd argument to the Hessian evaluation function so that it can appropriately scale the objective function term in the Hessian of the Lagrangian.
`opt.xi`	0.99995	ξ constant used in α updates in (A.46) and (A.47)
`opt.sigma`	0.1	centering parameter σ used in γ update in (A.52)
`opt.z0`	1	used to initialize elements of slack variable Z
`opt.alpha_min`	10^{-8}	algorithm returns "Numerically Failed" if the α_p or α_d from (A.46) and (A.47) become smaller than this value
`opt.rho_min`	0.95	lower bound on ρ_t corresponding to $1 - \eta$ in Fig. 5 in [17]
`opt.rho_max`	1.05	upper bound on ρ_t corresponding to $1 + \eta$ in Fig. 5 in [17]
`opt.mu_threshold`	10^{-5}	Kuhn-Tucker multipliers smaller than this value for non-binding constraints are forced to zero
`opt.max_stepsize`	10^{10}	algorithm returns "Numerically Failed" if the 2-norm of the Newton step $\begin{bmatrix} \Delta X \\ \Delta \lambda \end{bmatrix}$ from (A.45) exceeds this value

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

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

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

x =

     1
     1


f =

     0
```

A.2 Example 2

The second example[23] solves the following 3-dimensional constrained optimization, printing the details of the solver's progress:

$$\min_x f(x) = -x_1 x_2 - x_2 x_3 \tag{A.8}$$

subject to

$$\begin{aligned} x_1^2 - x_2^2 + x_3^2 - 2 &\leq 0 && \text{(A.9)} \\ x_1^2 + x_2^2 + x_3^2 - 10 &\leq 0. && \text{(A.10)} \end{aligned}$$

First, create a MATLAB function to evaluate the objective function and its gradients,[24]

[23]From http://en.wikipedia.org/wiki/Nonlinear_programming#3-dimensional_example.

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

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

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

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

and another to evaluate the Hessian of the Lagrangian.

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

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

```
function example2
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) ...
);
[x, f, exitflag, output, lambda] = mips(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);
```

```
>> example2
MATLAB Interior Point Solver -- MIPS, Version 1.1, 17-Dec-2014
 it    objective   step size   feascond     gradcond     compcond     costcond
----  ------------ --------- ------------ ------------ ------------ ------------
  0            -1                       0          1.5            5            0
  1    -5.3250167     1.6875            0     0.894235     0.850653      2.16251
  2    -7.4708991    0.97413     0.129183   0.00936418     0.117278     0.339269
  3    -7.0553031    0.10406            0   0.00174933    0.0196518    0.0490616
  4    -7.0686267   0.034574            0   0.00041301    0.0030084   0.00165402
  5    -7.0706104  0.0065191            0  1.53531e-05  0.000337971  0.000245844
  6    -7.0710134 0.00062152            0  1.22094e-07  3.41308e-05  4.99387e-05
  7    -7.0710623 5.7217e-05            0  9.84879e-10  3.41587e-06  6.05875e-06
  8    -7.0710673 5.6761e-06            0  9.73527e-12  3.41615e-07  6.15483e-07
Converged!

f = -7.07107   exitflag = 1

x =
   1.58114
   2.23607
   1.58114

lambda.ineqnonlin =
   0
   0.707107
```

More example problems for `mips` can be found in `t_mips.m`.

A.3 Quadratic Programming Solver

A convenience wrapper function called `qps_mips` is provided to make it trivial to set up and solve linear programming (LP) and quadratic programming (QP) problems of the following form:

$$\min_x \frac{1}{2} x^\mathsf{T} H x + c^\mathsf{T} x \tag{A.11}$$

subject to

$$l \le Ax \le u \tag{A.12}$$

$$x_{\min} \le x \le x_{\max}. \tag{A.13}$$

Instead of a function handle, the objective function is specified in terms of the paramters H and c of quadratic cost coefficients. Internally, `qps_mips` passes `mips`

the handle of a function that uses these paramters to evaluate the objective function, gradients and Hessian.

The calling syntax for qps_mips is similar to that used by quadprog from the MATLAB Optimization Toolbox.

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

Alternatively, the input arguments can be packaged as fields in a problem struct and passed in as a single argument, where all fields except H, c, A and l are optional.

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

Aside from H and c, all input and output arguments correspond exactly to the same arguments for mips as described in Tables A-1 and A-2.

As with mips and fmincon, the primary difference between the calling syntax for qps_mips and quadprog 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.

MATPOWER also includes another wrapper function qps_matpower that provides a consistent interface for all of the QP and LP solvers it has available. This interface is identical to that used by qps_mips with the exception of the structure of the opt input argument. The solver is chosen according to the value of opt.alg. See the help for qps_matpower for details.

Several examples of using qps_matpower to solve LP and QP problems can be found in t_qps_matpower.m.

A.4 Primal-Dual Interior Point Algorithm

This section provides some details on the primal-dual interior point algorithm used by MIPS and described in [17, 24].

A.4.1 Notation

For a scalar function $f\colon \mathbb{R}^n \to \mathbb{R}$ of a real vector $X = \left[\begin{array}{cccc} x_1 & x_2 & \cdots & x_n \end{array}\right]^\mathsf{T}$, we use the following notation for the first derivatives (transpose of the gradient):

$$f_X = \frac{\partial f}{\partial X} = \left[\begin{array}{cccc} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \cdots & \frac{\partial f}{\partial x_n} \end{array}\right]. \tag{A.14}$$

The matrix of second partial derivatives, the Hessian of f, is:

$$f_{XX} = \frac{\partial^2 f}{\partial X^2} = \frac{\partial}{\partial X}\left(\frac{\partial f}{\partial X}\right)^{\mathsf{T}} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}. \tag{A.15}$$

For a vector function $F \colon \mathbb{R}^n \to \mathbb{R}^m$ of a vector X, where

$$F(X) = \begin{bmatrix} f_1(X) & f_2(X) & \cdots & f_m(X) \end{bmatrix}^{\mathsf{T}} \tag{A.16}$$

the first derivatives form the Jacobian matrix, where row i is the transpose of the gradient of f_i

$$F_X = \frac{\partial F}{\partial X} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}. \tag{A.17}$$

In these derivations, the full 3-dimensional set of second partial derivatives of F will not be computed. Instead a matrix of partial derivatives will be formed by computing the Jacobian of the vector function obtained by multiplying the transpose of the Jacobian of F by a vector λ, using the following notation

$$F_{XX}(\lambda) = \frac{\partial}{\partial X}\left(F_X{}^{\mathsf{T}} \lambda\right). \tag{A.18}$$

Please note also that $[A]$ is used to denote a diagonal matrix with vector A on the diagonal and e is a vector of all ones.

A.4.2 Problem Formulation and Lagrangian

The primal-dual interior point method used by MIPS solves a problem of the form:

$$\min_X f(X) \tag{A.19}$$

subject to

$$G(X) = 0 \tag{A.20}$$

$$H(X) \leq 0 \tag{A.21}$$

where the linear constraints and variable bounds from (A.4) and (A.5) have been incorporated into $G(X)$ and $H(X)$. The approach taken involves converting the n_i

inequality constraints into equality constraints using a barrier function and vector of positive slack variables Z.

$$\min_{X} \left[f(X) - \gamma \sum_{m=1}^{n_i} \ln(Z_m) \right] \tag{A.22}$$

subject to

$$G(X) = 0 \tag{A.23}$$
$$H(X) + Z = 0 \tag{A.24}$$
$$Z > 0 \tag{A.25}$$

As the parameter of perturbation γ approaches zero, the solution to this problem approaches that of the original problem.

For a given value of γ, the Lagrangian for this equality constrained problem is

$$\mathcal{L}^{\gamma}(X, Z, \lambda, \mu) = f(X) + \lambda^{\mathsf{T}} G(X) + \mu^{\mathsf{T}}(H(X) + Z) - \gamma \sum_{m=1}^{n_i} \ln(Z_m). \tag{A.26}$$

Taking the partial derivatives with respect to each of the variables yields:

$$\mathcal{L}^{\gamma}_{X}(X, Z, \lambda, \mu) = f_X + \lambda^{\mathsf{T}} G_X + \mu^{\mathsf{T}} H_X \tag{A.27}$$
$$\mathcal{L}^{\gamma}_{Z}(X, Z, \lambda, \mu) = \mu^{\mathsf{T}} - \gamma e^{\mathsf{T}} [Z]^{-1} \tag{A.28}$$
$$\mathcal{L}^{\gamma}_{\lambda}(X, Z, \lambda, \mu) = G^{\mathsf{T}}(X) \tag{A.29}$$
$$\mathcal{L}^{\gamma}_{\mu}(X, Z, \lambda, \mu) = H^{\mathsf{T}}(X) + Z^{\mathsf{T}}. \tag{A.30}$$

And the Hessian of the Lagrangian with respect to X is given by

$$\mathcal{L}^{\gamma}_{XX}(X, Z, \lambda, \mu) = f_{XX} + G_{XX}(\lambda) + H_{XX}(\mu). \tag{A.31}$$

A.4.3 First Order Optimality Conditions

The first order optimality (Karush-Kuhn-Tucker) conditions for this problem are satisfied when the partial derivatives of the Lagrangian above are all set to zero:

$$F(X, Z, \lambda, \mu) = 0 \tag{A.32}$$
$$Z > 0 \tag{A.33}$$
$$\mu > 0 \tag{A.34}$$

where

$$F(X,Z,\lambda,\mu) = \begin{bmatrix} {\mathcal{L}_X^\gamma}^\mathsf{T} \\ [\mu]\,Z - \gamma e \\ G(X) \\ H(X) + Z \end{bmatrix} = \begin{bmatrix} {f_X}^\mathsf{T} + {G_X}^\mathsf{T}\lambda + {H_X}^\mathsf{T}\mu \\ [\mu]\,Z - \gamma e \\ G(X) \\ H(X) + Z \end{bmatrix}. \tag{A.35}$$

A.4.4 Newton Step

The first order optimality conditions are solved using Newton's method. The Newton update step can be written as follows:

$$\begin{bmatrix} F_X & F_Z & F_\lambda & F_\mu \end{bmatrix} \begin{bmatrix} \Delta X \\ \Delta Z \\ \Delta\lambda \\ \Delta\mu \end{bmatrix} = -F(X,Z,\lambda,\mu) \tag{A.36}$$

$$\begin{bmatrix} \mathcal{L}_{XX}^\gamma & 0 & {G_X}^\mathsf{T} & {H_X}^\mathsf{T} \\ 0 & [\mu] & 0 & [Z] \\ G_X & 0 & 0 & 0 \\ H_X & I & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta X \\ \Delta Z \\ \Delta\lambda \\ \Delta\mu \end{bmatrix} = -\begin{bmatrix} {\mathcal{L}_X^\gamma}^\mathsf{T} \\ [\mu]\,Z - \gamma e \\ G(X) \\ H(X) + Z \end{bmatrix}. \tag{A.37}$$

This set of equations can be simplified and reduced to a smaller set of equations by solving explicitly for $\Delta\mu$ in terms of ΔZ and for ΔZ in terms of ΔX. Taking the 2$^{\text{nd}}$ row of (A.37) and solving for $\Delta\mu$ we get

$$\begin{aligned} [\mu]\,\Delta Z + [Z]\,\Delta\mu &= -\,[\mu]\,Z + \gamma e \\ [Z]\,\Delta\mu &= -\,[Z]\,\mu + \gamma e - [\mu]\,\Delta Z \\ \Delta\mu &= -\mu + [Z]^{-1}\,(\gamma e - [\mu]\,\Delta Z). \end{aligned} \tag{A.38}$$

Solving the 4$^{\text{th}}$ row of (A.37) for ΔZ yields

$$\begin{aligned} H_X \Delta X + \Delta Z &= -H(X) - Z \\ \Delta Z &= -H(X) - Z - H_X \Delta X. \end{aligned} \tag{A.39}$$

Then, substituting (A.38) and (A.39) into the 1st row of (A.37) results in

$$\begin{aligned}
&\mathcal{L}^{\gamma}_{XX}\Delta X + {G_X}^{\mathsf{T}}\Delta\lambda + {H_X}^{\mathsf{T}}\Delta\mu && = -{\mathcal{L}^{\gamma}_{X}}^{\mathsf{T}} \\
&\mathcal{L}^{\gamma}_{XX}\Delta X + {G_X}^{\mathsf{T}}\Delta\lambda + {H_X}^{\mathsf{T}}(-\mu + [Z]^{-1}(\gamma e - [\mu]\Delta Z)) && = -{\mathcal{L}^{\gamma}_{X}}^{\mathsf{T}} \\
&\mathcal{L}^{\gamma}_{XX}\Delta X + {G_X}^{\mathsf{T}}\Delta\lambda && \\
&\quad + {H_X}^{\mathsf{T}}(-\mu + [Z]^{-1}(\gamma e - [\mu](-H(X) - Z - H_X\Delta X))) && = -{\mathcal{L}^{\gamma}_{X}}^{\mathsf{T}} \\
&\mathcal{L}^{\gamma}_{XX}\Delta X + {G_X}^{\mathsf{T}}\Delta\lambda - {H_X}^{\mathsf{T}}\mu + {H_X}^{\mathsf{T}}[Z]^{-1}\gamma e && \\
&\quad + {H_X}^{\mathsf{T}}[Z]^{-1}[\mu]H(X) + {H_X}^{\mathsf{T}}[Z]^{-1}[Z]\mu + {H_X}^{\mathsf{T}}[Z]^{-1}[\mu]H_X\Delta X && = -{\mathcal{L}^{\gamma}_{X}}^{\mathsf{T}} \\
&(\mathcal{L}^{\gamma}_{XX} + {H_X}^{\mathsf{T}}[Z]^{-1}[\mu]H_X)\Delta X + {G_X}^{\mathsf{T}}\Delta\lambda && \\
&\quad + {H_X}^{\mathsf{T}}[Z]^{-1}(\gamma e + [\mu]H(X)) && = -{\mathcal{L}^{\gamma}_{X}}^{\mathsf{T}} \\
&M\Delta X + {G_X}^{\mathsf{T}}\Delta\lambda && = -N \qquad \text{(A.40)}
\end{aligned}$$

where

$$M \equiv \mathcal{L}^{\gamma}_{XX} + {H_X}^{\mathsf{T}}[Z]^{-1}[\mu]H_X \tag{A.41}$$

$$= f_{XX} + G_{XX}(\lambda) + H_{XX}(\mu) + {H_X}^{\mathsf{T}}[Z]^{-1}[\mu]H_X \tag{A.42}$$

and

$$N \equiv {\mathcal{L}^{\gamma}_{X}}^{\mathsf{T}} + {H_X}^{\mathsf{T}}[Z]^{-1}(\gamma e + [\mu]H(X)) \tag{A.43}$$

$$= {f_X}^{\mathsf{T}} + {G_X}^{\mathsf{T}}\lambda + {H_X}^{\mathsf{T}}\mu + {H_X}^{\mathsf{T}}[Z]^{-1}(\gamma e + [\mu]H(X)). \tag{A.44}$$

Combining (A.40) and the 3rd row of (A.37) results in a system of equations of reduced size:

$$\begin{bmatrix} M & {G_X}^{\mathsf{T}} \\ G_X & 0 \end{bmatrix} \begin{bmatrix} \Delta X \\ \Delta\lambda \end{bmatrix} = \begin{bmatrix} -N \\ -G(X) \end{bmatrix}. \tag{A.45}$$

The Newton update can then be computed in the following 3 steps:

1. Compute ΔX and $\Delta\lambda$ from (A.45).
2. Compute ΔZ from (A.39).
3. Compute $\Delta\mu$ from (A.38).

In order to maintain strict feasibility of the trial solution, the algorithm truncates the Newton step by scaling the primal and dual variables by α_p and α_d, respectively,

where these scale factors are computed as follows:

$$\alpha_p = \min\left(\xi \min_{\Delta Z_m < 0}\left(-\frac{Z_m}{\Delta Z_m}\right), 1\right) \tag{A.46}$$

$$\alpha_d = \min\left(\xi \min_{\Delta \mu_m < 0}\left(-\frac{\mu_m}{\Delta \mu_m}\right), 1\right) \tag{A.47}$$

resulting in the variable updates below.

$$X \leftarrow X + \alpha_p \Delta X \tag{A.48}$$

$$Z \leftarrow Z + \alpha_p \Delta Z \tag{A.49}$$

$$\lambda \leftarrow \lambda + \alpha_d \Delta \lambda \tag{A.50}$$

$$\mu \leftarrow \mu + \alpha_d \Delta \mu \tag{A.51}$$

The parameter ξ is a constant scalar with a value slightly less than one. In MIPS, ξ is set to 0.99995.

In this method, during the Newton-like iterations, the perturbation parameter γ must converge to zero in order to satisfy the first order optimality conditions of the original problem. MIPS uses the following rule to update γ at each iteration, after updating Z and μ:

$$\gamma \leftarrow \sigma \frac{Z^\mathsf{T} \mu}{n_i} \tag{A.52}$$

where σ is a scalar constant between 0 and 1. In MIPS, σ is set to 0.1.

Appendix B Data File Format

There are two versions of the MATPOWER case file format. MATPOWER versions 3.0.0 and earlier used the version 1 format internally. Subsequent versions of MATPOWER have used the version 2 format described below, though version 1 files are still handled, and converted automatically, by the `loadcase` and `savecase` functions.

In the version 2 format, the input data for MATPOWER are specified in a set of data matrices packaged as the fields of a MATLAB struct, referred to as a "MATPOWER case" struct and conventionally denoted by the variable `mpc`. This struct is typically defined in a case file, either a function M-file whose return value is the `mpc` struct or a MAT-file that defines a variable named `mpc` when loaded. The fields of this struct are `baseMVA`, `bus`, `branch`, `gen` and, optionally, `gencost`. The `baseMVA` field is a scalar and the rest are matrices. Each row in the data matrices corresponds to a single bus, branch, or generator and the columns are similar to the columns in the standard IEEE and PTI formats. The `mpc` struct also has a `version` field whose value is a string set to the current MATPOWER case version, currently `'2'` by default. The version 1 case format defines the data matrices as individual variables rather than fields of a struct, and some do not include all of the columns defined in version 2.

Numerous examples can be found in the case files listed in Table D-16 in Appendix D. The case files created by `savecase` use a tab-delimited format for the data matrices to make it simple to transfer data seamlessly back and forth between a text editor and a spreadsheet via simple copy and paste.

The details of the MATPOWER case format are given in the tables below and can also be accessed by typing `help caseformat` at the MATLAB prompt. First, the `baseMVA` field is a simple scalar value specifying the system MVA base used for converting power into per unit quantities. For convenience and code portability, `idx_bus` defines a set of constants to be used as named indices into the columns of the `bus` matrix. Similarly, `idx_brch`, `idx_gen` and `idx_cost` define names for the columns of `branch`, `gen` and `gencost`, respectively. The script `define_constants` provides a simple way to define all the usual constants at one shot. These are the names that appear in the first column of the tables below.

The MATPOWER case format also allows for additional fields to be included in the structure. The OPF is designed to recognize fields named `A`, `l`, `u`, `H`, `Cw`, `N`, `fparm`, `z0`, `zl` and `zu` as parameters used to directly extend the OPF formulation as described in Section 7.1. Other user-defined fields may also be included, such as the `reserves` field used in the example code throughout Section 7.2. The `loadcase` function will automatically load any extra fields from a case file and, if the appropriate `'savecase'` callback function (see Section 7.2.5) is added via `add_userfcn`, `savecase` will also save them back to a case file.

Table B-1: Bus Data (`mpc.bus`)

name	column	description
BUS_I	1	bus number (positive integer)
BUS_TYPE	2	bus type (1 = PQ, 2 = PV, 3 = ref, 4 = isolated)
PD	3	real power demand (MW)
QD	4	reactive power demand (MVAr)
GS	5	shunt conductance (MW demanded at V = 1.0 p.u.)
BS	6	shunt susceptance (MVAr injected at V = 1.0 p.u.)
BUS_AREA	7	area number (positive integer)
VM	8	voltage magnitude (p.u.)
VA	9	voltage angle (degrees)
BASE_KV	10	base voltage (kV)
ZONE	11	loss zone (positive integer)
VMAX	12	maximum voltage magnitude (p.u.)
VMIN	13	minimum voltage magnitude (p.u.)
LAM_P†	14	Lagrange multiplier on real power mismatch (u/MW)
LAM_Q†	15	Lagrange multiplier on reactive power mismatch (u/MVAr)
MU_VMAX†	16	Kuhn-Tucker multiplier on upper voltage limit (u/p.u.)
MU_VMIN†	17	Kuhn-Tucker multiplier on lower voltage limit (u/p.u.)

† Included in OPF output, typically not included (or ignored) in input matrix. Here we assume the objective function has units u.

Table B-2: Generator Data (mpc.gen)

name	column	description
GEN_BUS	1	bus number
PG	2	real power output (MW)
QG	3	reactive power output (MVAr)
QMAX	4	maximum reactive power output (MVAr)
QMIN	5	minimum reactive power output (MVAr)
VG	6	voltage magnitude setpoint (p.u.)
MBASE	7	total MVA base of machine, defaults to baseMVA
GEN_STATUS	8	machine status, > 0 = machine in-service ≤ 0 = machine out-of-service
PMAX	9	maximum real power output (MW)
PMIN	10	minimum real power output (MW)
PC1*	11	lower real power output of PQ capability curve (MW)
PC2*	12	upper real power output of PQ capability curve (MW)
QC1MIN*	13	minimum reactive power output at PC1 (MVAr)
QC1MAX*	14	maximum reactive power output at PC1 (MVAr)
QC2MIN*	15	minimum reactive power output at PC2 (MVAr)
QC2MAX*	16	maximum reactive power output at PC2 (MVAr)
RAMP_AGC*	17	ramp rate for load following/AGC (MW/min)
RAMP_10*	18	ramp rate for 10 minute reserves (MW)
RAMP_30*	19	ramp rate for 30 minute reserves (MW)
RAMP_Q*	20	ramp rate for reactive power (2 sec timescale) (MVAr/min)
APF*	21	area participation factor
MU_PMAX†	22	Kuhn-Tucker multiplier on upper P_g limit (u/MW)
MU_PMIN†	23	Kuhn-Tucker multiplier on lower P_g limit (u/MW)
MU_QMAX†	24	Kuhn-Tucker multiplier on upper Q_g limit (u/MVAr)
MU_QMIN†	25	Kuhn-Tucker multiplier on lower Q_g limit (u/MVAr)

* Not included in version 1 case format.

† Included in OPF output, typically not included (or ignored) in input matrix. Here we assume the objective function has units u.

Table B-3: Branch Data (`mpc.branch`)

name	column	description
F_BUS	1	“from” bus number
T_BUS	2	“to” bus number
BR_R	3	resistance (p.u.)
BR_X	4	reactance (p.u.)
BR_B	5	total line charging susceptance (p.u.)
RATE_A	6	MVA rating A (long term rating)
RATE_B	7	MVA rating B (short term rating)
RATE_C	8	MVA rating C (emergency rating)
TAP	9	transformer off nominal turns ratio, (taps at “from” bus, impedance at “to” bus, i.e. if $r = x = 0$, $tap = \frac{\lvert V_f \rvert}{\lvert V_t \rvert}$)
SHIFT	10	transformer phase shift angle (degrees), positive $\Rightarrow$ delay
BR_STATUS	11	initial branch status, 1 = in-service, 0 = out-of-service
ANGMIN*	12	minimum angle difference, $\theta_f - \theta_t$ (degrees)
ANGMAX*	13	maximum angle difference, $\theta_f - \theta_t$ (degrees)
PF†	14	real power injected at “from” bus end (MW)
QF†	15	reactive power injected at “from” bus end (MVAr)
PT†	16	real power injected at “to” bus end (MW)
QT†	17	reactive power injected at “to” bus end (MVAr)
MU_SF‡	18	Kuhn-Tucker multiplier on MVA limit at “from” bus (u/MVA)
MU_ST‡	19	Kuhn-Tucker multiplier on MVA limit at “to” bus (u/MVA)
MU_ANGMIN‡	20	Kuhn-Tucker multiplier lower angle difference limit (u/degree)
MU_ANGMAX‡	21	Kuhn-Tucker multiplier upper angle difference limit (u/degree)

* Not included in version 1 case format. The voltage angle difference is taken to be unbounded below if ANGMIN < -360 and unbounded above if ANGMAX > 360. If both parameters are zero, the voltage angle difference is unconstrained.

† Included in power flow and OPF output, ignored on input.

‡ Included in OPF output, typically not included (or ignored) in input matrix. Here we assume the objective function has units u.

Table B-4: Generator Cost Data† (`mpc.gencost`)

name	column	description
`MODEL`	1	cost model, 1 = piecewise linear, 2 = polynomial
`STARTUP`	2	startup cost in US dollars*
`SHUTDOWN`	3	shutdown cost in US dollars*
`NCOST`	4	number of cost coefficients for polynomial cost function, or number of data points for piecewise linear
`COST`	5	parameters defining total cost function $f(p)$ begin in this column, units of f and p are $/hr and MW (or MVAr), respectively (`MODEL` = 1) $\Rightarrow$ $p_0, f_0, p_1, f_1, \ldots, p_n, f_n$ where $p_0 < p_1 < \cdots < p_n$ and the cost $f(p)$ is defined by the coordinates (p_0, f_0), (p_1, f_1), …, (p_n, f_n) of the end/break-points of the piecewise linear cost (`MODEL` = 2) $\Rightarrow$ $c_n, \ldots, c_1, c_0$ $n+1$ coefficients of n-th order polynomial cost, starting with highest order, where cost is $f(p) = c_n p^n + \cdots + c_1 p + c_0$

† If `gen` has n_g rows, then the first n_g rows of `gencost` contain the costs for active power produced by the corresponding generators. If `gencost` has $2n_g$ rows, then rows $n_g + 1$ through $2n_g$ contain the reactive power costs in the same format.

* Not currently used by any MATPOWER functions.

Table B-5: DC Line Data[*] (mpc.dcline)

name	column	description
F_BUS	1	"from" bus number
T_BUS	2	"to" bus number
BR_STATUS	3	initial branch status, 1 = in-service, 0 = out-of-service
PF[†]	4	real power flow at "from" bus end (MW), "from" → "to"
PT[†]	5	real power flow at "to" bus end (MW), "from" → "to"
QF[†]	6	reactive power injected into "from" bus (MVAr)
QT[†]	7	reactive power injected into "to" bus (MVAr)
VF	8	voltage magnitude setpoint at "from" bus (p.u.)
VT	9	voltage magnitude setpoint at "to" bus (p.u.)
PMIN	10	if positive (negative), lower limit on PF (PT)
PMAX	11	if positive (negative), upper limit on PF (PT)
QMINF	12	lower limit on reactive power injection into "from" bus (MVAr)
QMAXF	13	upper limit on reactive power injection into "from" bus (MVAr)
QMINT	14	lower limit on reactive power injection into "to" bus (MVAr)
QMAXT	15	upper limit on reactive power injection into "to" bus (MVAr)
LOSS0	16	coefficient l_0 of constant term of linear loss function (MW)
LOSS1	17	coefficient l_1 of linear term of linear loss function (MW/MW)
		($p_{\text{loss}} = l_0 + l_1 p_f$, where p_f is the flow at the "from" end)
MU_PMIN[‡]	18	Kuhn-Tucker multiplier on lower flow limit at "from" bus (u/MW)
MU_PMAX[‡]	19	Kuhn-Tucker multiplier on upper flow limit at "from" bus (u/MW)
MU_QMINF[‡]	20	Kuhn-Tucker multiplier on lower VAr limit at "from" bus (u/MVAr)
MU_QMAXF[‡]	21	Kuhn-Tucker multiplier on upper VAr limit at "from" bus (u/MVAr)
MU_QMINT[‡]	22	Kuhn-Tucker multiplier on lower VAr limit at "to" bus (u/MVAr)
MU_QMAXT[‡]	23	Kuhn-Tucker multiplier on upper VAr limit at "to" bus (u/MVAr)

[*] Requires explicit use of toggle_dcline.
[†] Output column, value updated by power flow or OPF (except PF in case of simple power flow).
[‡] Included in OPF output, typically not included (or ignored) in input matrix. Here we assume the objective function has units u.

Appendix C MATPOWER Options

Beginning with version 4.2, MATPOWER uses an options struct to control the many options available. Earlier versions used an options vector with named elements. MATPOWER's options are used to control things such as:

- power flow algorithm
- power flow termination criterion
- power flow options (e.g. enforcing of reactive power generation limits)
- continuation power flow options
- OPF algorithm
- OPF termination criterion
- OPF options (e.g. active vs. apparent power vs. current for line limits)
- verbose level
- printing of results
- solver specific options

As with the old-style options vector, the options struct should always be created and modified using the `mpoption` function to ensure compatibility across different versions of MATPOWER. The default MATPOWER options struct is obtained by calling `mpoption` with no arguments.

```
>> mpopt = mpoption;
```

Individual options can be overridden from their default values by calling `mpoption` with a set of name/value pairs as input arguments. For example, the following runs a fast-decoupled power flow of `case30` with very verbose progress output:

```
>> mpopt = mpoption('pf.alg', 'FDXB', 'verbose', 3);
>> runpf('case30', mpopt);
```

For backward compatibility, old-style option names/values can also be used.

```
>> mpopt = mpoption('PF_ALG', 2, 'VERBOSE', 3);
```

Another way to specify option overrides is via a struct. Using the example above, the code would be as follows.

```
>> overrides = struct('pf', struct('alg', 'FDXB'), 'verbose', 3);
>> mpopt = mpoption(overrides);
```

Finally, a string containing the name of a function that returns such a struct, can be passed to `mpoption` instead of the struct itself.

```
>> mpopt = mpoption('verbose_fast_decoupled_pf_opts');
```

where the function `verbose_fast_decoupled_pf_opts` is defined as follows:

```
function ov = verbose_fast_decoupled_pf_opts()
ov = struct('pf', struct('alg', 'FDXB'), 'verbose', 3);
```

To make changes to an existing options struct (as opposed to the default options struct), simply include it as the first argument. For example, to modify the previous run to enforce reactive power limts, suppress the pretty-printing of the output and save the results to a struct instead:

```
>> mpopt = mpoption(mpopt, 'pf.enforce_q_lims', 1, 'out.all', 0);
>> results = runpf('case30', mpopt);
```

This works when specifying the overrides as a struct or function name as well. For backward compatibility, the first argument can be an old-style options vector, followed by old-style option name/value pairs.

The available options and their default values are summarized in the following tables and can also be accessed via the command `help mpoption`. Some of the options require separately installed optional packages available from the MATPOWER website.

Table C-1: Top-Level Options

name	default	description
`model`	`'AC'`	AC vs. DC modeling for power flow and OPF formulation `'AC'` – use AC formulation and corresponding algs/options `'DC'` – use DC formulation and corresponding algs/options
`pf`	*see Table C-2*	power flow options
`cpf`	*see Table C-3*	continuation power flow options
`opf`	*see Tables C-4, C-5*	optimal power flow options
`verbose`	1	amount of progress info printed 0 – print no progress info 1 – print a little progress info 2 – print a lot of progress info 3 – print all progress info
`out`	*see Table C-6*	pretty-printed output options
`mips`	*see Table C-7*	MIPS options
`cplex`	*see Table C-8*	CPLEX options*
`fmincon`	*see Table C-9*	`fmincon` options†
`glpk`	*see Table C-10*	GLPK options*
`gurobi`	*see Table C-11*	Gurobi options*
`ipopt`	*see Table C-12*	IPOPT options*
`knitro`	*see Table C-13*	KNITRO options*
`minopf`	*see Table C-14*	MINOPF options*
`mosek`	*see Table C-15*	MOSEK options*
`pdipm`	*see Table C-16*	PDIPM options*
`tralm`	*see Table C-17*	TRALM options*

* Requires the installation of an optional package. See Appendix G for details on the corresponding package.
† Requires MATLAB's Optimization Toolbox, available from The MathWorks, Inc (http://www.mathworks.com/).

Table C-2: Power Flow Options

name	default	description
`pf.alg`	`'NR'`	AC power flow algorithm: `'NR'` – Newtons's method `'FDXB'` – Fast-Decoupled (XB version) `'FDBX'` – Fast-Decouple (BX version) `'GS'` – Gauss-Seidel
`pf.tol`	10^{-8}	termination tolerance on per unit P and Q dispatch
`pf.nr.max_it`	10	maximum number of iterations for Newton's method
`pf.fd.max_it`	30	maximum number of iterations for fast-decoupled method
`pf.gs.max_it`	1000	maximum number of iterations for Gauss-Seidel method
`pf.enforce_q_lims`	0	enforce gen reactive power limits at expense of V_m 0 – do *not* enforce limits 1 – enforce limits, simultaneous bus type conversion 2 – enforce limits, one-at-a-time bus type conversion

Table C-3: Continuation Power Flow Options

name	default	description
`cpf.parameterization`	3	choice of parameterization 1 — natural 2 — arc length 3 — pseudo arc length
`cpf.stop_at`	`'NOSE'`	determines stopping criterion `'NOSE'` — stop when nose point is reached `'FULL'` — trace full nose curve λ_{stop} — stop upon reaching target λ value λ_{stop}
`cpf.step`	0.05	continuation power flow step size
`cpf.adapt_step`	0	toggle adaptive step size feature 0 — adaptive step size disabled 1 — adaptive step size enabled
`cpf.error_tol`	10^{-3}	tolerance for the adaptive step controller
`cpf.step_min`	10^{-4}	minimum allowed step size
`cpf.step_max`	0.2	maximum allowed step size
`cpf.plot.level`	0	control plotting of nose curve 0 — do not plot nose curve 1 — plot when completed 2 — plot incrementally at each iteration 3 — same as 2, with `pause` at each iteration
`cpf.plot.bus`	*empty*	index of bus whose voltage is to be plotted
`cpf.user_callback`	*empty*	string or cell array of strings with names of user callback functions†
`cpf.user_callback_args`	*empty*	struct passed to user-defined callback functions†

† See `help cpf_default_callback` for details.

Table C-4: OPF Solver Options

name	default	description

`opf.ac.solver`	`'DEFAULT'`	AC optimal power flow solver:
		`'DEFAULT'` – choose default solver based on availability in the following order: `'PDIPM'`, `'MIPS'`
		`'MIPS'` – MIPS, MATLAB Interior Point Solver, primal/dual interior point method[†]
		`'FMINCON'` – MATLAB Optimization Toolbox, `fmincon`
		`'IPOPT'` – IPOPT[*]
		`'KNITRO'` – KNITRO[*]
		`'MINOPF'` – MINOPF,[*] MINOS-based solver
		`'PDIPM'` – PDIPM,[*] primal/dual interior point method[‡]
		`'SDPOPF'` – SDPOPF,[*] solver based on semidefinite relaxation
		`'TRALM'` – TRALM,[*] trust region based augmented Langrangian method

`opf.dc.solver`	`'DEFAULT'`	DC optimal power flow solver:
		`'DEFAULT'` – choose default solver based on availability in the following order: `'CPLEX'`, `'GUROBI'`, `'MOSEK'`, `'BPMPD'`, `'OT'`, `'GLPK'` *(linear costs only)*, `'MIPS'`
		`'MIPS'` – MIPS, MATLAB Interior Point Solver, primal/dual interior point method[†]
		`'BPMPD'` – BPMPD[*]
		`'CPLEX'` – CPLEX[*]
		`'GLPK'` – GLPK[*] *(no quadratic costs)*
		`'GUROBI'` – Gurobi[*]
		`'IPOPT'` – IPOPT[*]
		`'MOSEK'` – MOSEK[*]
		`'OT'` – MATLAB Opt Toolbox, `quadprog`, `linprog`

[*] Requires the installation of an optional package. See Appendix G for details on the corresponding package.
[†] For MIPS-sc, the step-controlled version of this solver, the `mips.step_control` option must be set to 1.
[‡] For SC-PDIPM, the step-controlled version of this solver, the `pdipm.step_control` option must be set to 1.

Table C-5: General OPF Options

name	default	description
`opf.violation`	5×10^{-6}	constraint violation tolerance
`opf.flow_lim`	`'S'`	quantity to limit for branch flow constraints `'S'` – apparent power flow (limit in MVA) `'P'` – active power flow (limit in MW) `'I'` – current magnitude (limit in MVA at 1 p.u. voltage)
`opf.ignore_ang_lim`	0	ignore angle difference limits for branches 0 – include angle difference limits, if specified 1 – ignore angle difference limits even if specified
`opf.init_from_mpc`	-1	specify whether to use the current state in MATPOWER case to initialize OPF[†] -1 – MATPOWER decides based on solver/algorithm 0 – ignore current state when initializing OPF 1 – use current state to initialize OPF
`opf.return_raw_der`	0	for AC OPF, return constraint and derivative info in `results.raw` (in fields `g`, `dg`, `df`, `d2f`)

[†] Currently supported only for IPOPT, KNITRO and MIPS solvers.

Table C-6: Power Flow and OPF Output Options

name	default	description
`verbose`	1	amount of progress info to be printed 0 – print no progress info 1 – print a little progress info 2 – print a lot of progress info 3 – print all progress info
`out.all`	-1	controls pretty-printing of results -1 – individual flags control what is printed 0 – do *not* print anything[†][*] 1 – print everything[†]
`out.sys_sum`	1	print system summary (0 or 1)
`out.area_sum`	0	print area summaries (0 or 1)
`out.bus`	1	print bus detail, includes per bus gen info (0 or 1)
`out.branch`	1	print branch detail (0 or 1)
`out.gen`	0	print generator detail (0 or 1)
`out.lim.all`	-1	controls constraint info output -1 – individual flags control what is printed 0 – do *not* print any constraint info[†] 1 – print only binding constraint info[†] 2 – print all constraint info[†]
`out.lim.v`	1	control output of voltage limit info 0 – do *not* print 1 – print binding constraints only 2 – print all constraints
`out.lim.line`	1	control output of line flow limit info[‡]
`out.lim.pg`	1	control output of gen active power limit info[‡]
`out.lim.qg`	1	control output of gen reactive power limit info[‡]
`out.force`	0	print results even if success flag = 0 (0 or 1)
`out.suppress_detail`	-1	suppress all output but system summary -1 – suppress details for large systems (> 500 buses) 0 – do *not* suppress any output specified by other flags 1 – suppress all output except system summary section[†]

[*] This setting is ignored for pretty-printed output to files specified as `FNAME` argument in calls to `runpf`, `runopf`, etc.
[†] Overrides individual flags, but (in the case of `out.suppress_detail`) not `out.all` = 1.
[‡] Takes values of 0, 1 or 2 as for `out.lim.v`.

Table C-7: OPF Options for MIPS

name	default	description
`mips.feastol`	0	feasibiliy (equality) tolerance set to value of `opf.violation` by default
`mips.gradtol`	10^{-6}	gradient tolerance
`mips.comptol`	10^{-6}	complementarity condition (inequality) tolerance
`mips.costtol`	10^{-6}	optimality tolerance
`mips.max_it`	150	maximum number of iterations
`mips.step_control`	0	set to 1 to enable step-size control
`mips.sc.red_it`‡	20	maximum number of step size reductions per iteration

‡ Only relevant when `mips.step_control` is on.

Table C-8: OPF Options for CPLEX†

name	default	description
`cplex.lpmethod`	0	algorithm used by CPLEX for LP problems 0 – automatic; let CPLEX choose 1 – primal simplex 2 – dual simplex 3 – network simplex 4 – barrier 5 – sifting 6 – concurrent (dual, barrier, and primal)
`cplex.qpmethod`	0	algorithm used by CPLEX for QP problems 0 – automatic; let CPLEX choose 1 – primal simplex 2 – dual simplex 3 – network simplex 4 – barrier
`cplex.opts`	*empty*	struct of native CPLEX options (for `cplexoptimset`) passed to `cplex_options` to override defaults, applied after overrides from `cplex.opt_fname`‡
`cplex.opt_fname`	*empty*	name of user-supplied function passed as `FNAME` argument to `cplex_options` to override defaults‡
`cplex.opt`	0	if `cplex.opt_fname` is empty and `cplex.opt` is non-zero, the value of `cplex.opt_fname` is generated by appending `cplex.opt` to `'cplex_user_options_'` (for backward compatibility with old MATPOWER option `CPLEX_OPT`)

† For `opf.solver.dc` option set to `'CPLEX'` only. Requires the installation of the optional CPLEX package. See Appendix G.2 for details.

‡ For details, see `help cplex_options` and the "Parameters of CPLEX" section of the CPLEX documentation at http://pic.dhe.ibm.com/infocenter/cosinfoc/v12r6/.

Table C-9: OPF Options for `fmincon`[†]

name	default	description
`fmincon.alg`	4	algorithm used by `fmincon` in MATLAB Opt Toolbox ≥ 4 1 – active-set[‡] 2 – interior-point, default "bfgs" Hessian approximation 3 – interior-point, "lbfgs" Hessian approximation 4 – interior-point, exact user-supplied Hessian 5 – interior-point, Hessian via finite-differences 6 – sqp, sequential quadratic programming[‡]
`fmincon.tol_x`	10^{-4}	termination tolerance on x[*]
`fmincon.tol_f`	10^{-4}	termination tolerance on f[*]
`fmincon.max_it`	0	maximum number of iterations[*] 0 ⇒ use solver's default value

[†] Requires MATLAB's Optimization Toolbox, available from The MathWorks, Inc (http://www.mathworks.com/).
[‡] Does not use sparse matrices, so not applicable for large-scale systems.
[*] `Display` is set by `verbose`, `TolCon` by `opf.violation`, `TolX` by `fmincon.tol_x`, `TolFun` by `fmincon.tol_f`, and `MaxIter` and `MaxFunEvals` by `fmincon.max_it`.

Table C-10: OPF Options for GLPK[†]

name	default[§]	description
`glpk.opts`	*empty*	struct of native GLPK options passed to `glpk_options` to override defaults, applied after overrides from `glpk.opt_fname`[‡]
`glpk.opt_fname`	*empty*	name of user-supplied function passed as `FNAME` argument to `glpk_options` to override defaults[‡]

[†] For `opf.solver.dc` option set to `'GLPK'` only. Requires the installation of the optional GLPK package. See Appendix G.3 for details.
[‡] For details, see `help glpk_options` or the "param" section of the GLPK documentation at http://www.gnu.org/software/octave/doc/interpreter/Linear-Programming.html.

Table C-11: OPF Options for Gurobi†

name	default§	description
`gurobi.method`	0	algorithm used by Gurobi for LP/QP problems 0 – primal simplex 1 – dual simplex 2 – barrier 3 – concurrent (LP only) 4 – deterministic concurrent (LP only)
`gurobi.timelimit`	∞	maximum time allowed for solver (secs)
`gurobi.threads`	0 (auto)	maximum number of threads to use
`gurobi.opts`	*empty*	struct of native Gurobi options passed to `gurobi_options` to override defaults, applied after overrides from `gurobi.opt_fname`‡
`gurobi.opt_fname`	*empty*	name of user-supplied function passed as `FNAME` argument to `gurobi_options` to override defaults‡
`gurobi.opt`	0	if `gurobi.opt_fname` is empty and `gurobi.opt` is non-zero, the value of `gurobi.opt_fname` is generated by appending `gurobi.opt` to `'gurobi_user_options_'` (for backward compatibility with old MATPOWER option `GRB_OPT`)

† For `opf.solver.dc` option set to `'GUROBI'` only. Requires the installation of the optional Gurobi package. See Appendix G.4 for details.
§ Default values in parenthesis refer to defaults assigned by Gurobi if called with option equal to 0.
‡ For details, see `help gurobi_options` and the "Parameters" section of the "Gurobi Optimizer Reference Manual" at http://www.gurobi.com/documentation/5.6/reference-manual/parameters.

Table C-12: OPF Options for IPOPT†

name	default	description
`ipopt.opts`	*empty*	struct of native IPOPT options (`options.ipopt` for `ipopt`) passed to `ipopt_options` to override defaults, applied after overrides from `ipopt.opt_fname`‡
`ipopt.opt_fname`	*empty*	name of user-supplied function passed as `FNAME` argument to `ipopt_options` to override defaults‡
`ipopt.opt`	0	if `ipopt.opt_fname` is empty and `ipopt.opt` is non-zero, the value of `ipopt.opt_fname` is generated by appending `ipopt.opt` to `'ipopt_user_options_'` (for backward compatibility with old MATPOWER option `IPOPT_OPT`)

† For `opf.solver.ac` or `opf.solver.dc` option set to `'IPOPT'` only. Requires the installation of the optional IPOPT package [28]. See Appendix G.5 for details.
‡ For details, see `help ipopt_options` and the options reference section of the IPOPT documentation at http://www.coin-or.org/Ipopt/documentation/.

Table C-13: OPF Options for KNITRO†

name	default	description
`knitro.tol_x`	10^{-4}	termination tolerance on x
`knitro.tol_f`	10^{-4}	termination tolerance on f
`knitro.opt_fname`	*empty*	name of user-supplied native KNITRO options file that overrides other options‡
`knitro.opt`	0	if `knitro.opt_fname` is empty and `knitro.opt` is a positive integer n, the value of `knitro.opt_fname` is generated as `'knitro_user_options_`n`.txt'` (for backward compatibility with old MATPOWER option `KNITRO_OPT`)

† For `opf.solver.ac` option set to `'KNITRO'` only. Requires the installation of the optional KNITRO package [29]. See Appendix G.6 for details.
‡ Note that KNITRO uses the `opt_fname` option slightly differently from other optional solvers. Specifically, it is the name of a text file processed directly by KNITRO, not a MATLAB function that returns an options struct passed to the solver.

Table C-14: OPF Options for MINOPF†

name	default‡	description
`minopf.feastol`	0 (10^{-3})	primal feasibility tolerance set to value of `opf.violation` by default
`minopf.rowtol`	0 (10^{-3})	row tolerance set to value of `opf.violation` by default
`minopf.xtol`	0 (10^{-4})	x tolerance
`minopf.majdamp`	0 (0.5)	major damping parameter
`minopf.mindamp`	0 (2.0)	minor damping parameter
`minopf.penalty`	0 (1.0)	penalty parameter
`minopf.major_it`	0 (200)	major iterations
`minopf.minor_it`	0 (2500)	minor iterations
`minopf.max_it`	0 (2500)	iteration limit
`minopf.verbosity`	-1	amount of progress output printed by MEX file -1 – controlled by `verbose` option 0 – do *not* print anything 1 – print only only termination status message 2 – print termination status & screen progress 3 – print screen progress, report file (usually fort.9)
`minopf.core`	0	memory allocation defaults to $1200n_b + 2(n_b + n_g)^2$
`minopf.supbasic_lim`	0	superbasics limit, defaults to $2n_b + 2n_g$
`minopf.mult_price`	0 (30)	multiple price

† For `opf.solver.ac` option set to `'MINOPF'` only. Requires the installation of the optional MINOPF package [21]. See Appendix G.7 for details.
‡ Default values in parenthesis refer to defaults assigned in MEX file if called with option equal to 0.

Table C-15: OPF Options for MOSEK[†]

name	default[§]	description
`mosek.lp_alg`	0	solution algorithm used by MOSEK for continuous LP problems (`MSK_IPAR_OPTIMIZER`) 0 – automatic; let MOSEK choose 1 – interior point 4 – primal simplex 5 – dual simplex 6 – primal dual simplex 7 – automatic simplex (MOSEK chooses which simplex) 10 – concurrent
`mosek.max_it`	0 (400)	interior point maximum iterations (`MSK_IPAR_INTPNT_MAX_ITERATIONS`)
`mosek.gap_tol`	0 (10^{-8})	interior point relative gap tolerance (`MSK_DPAR_INTPNT_TOL_REL_GAP`)
`mosek.max_time`	0 (-1)	maximum time allowed for solver (negative means ∞) (`MSK_DPAR_OPTIMIZER_MAX_TIME`)
`mosek.num_threads`	0 (1)	maximum number of threads to use (`MSK_IPAR_INTPNT_NUM_THREADS`)
`mosek.opts`	*empty*	struct of native MOSEK options (`param` struct normally passed to `mosekopt`) passed to `mosek_options` to override defaults, applied after overrides from `mosek.opt_fname`[‡]
`mosek.opt_fname`	*empty*	name of user-supplied function passed as `FNAME` argument to `mosek_options` to override defaults[‡]
`mosek.opt`	0	if `mosek.opt_fname` is empty and `mosek.opt` is non-zero, the value of `mosek.opt_fname` is generated by appending `mosek.opt` to `'mosek_user_options_'` (for backward compatibility with old MATPOWER option `MOSEK_OPT`)

[†] For `opf.solver.dc` option set to `'MOSEK'` only. Requires the installation of the optional MOSEK package. See Appendix G.8 for details.
[§] Default values in parenthesis refer to defaults assigned by MOSEK if called with option equal to 0.
[‡] For details, see `help mosek_options` and the "Parameters" reference in "The MOSEK optimization toolbox for MATLAB manaul" at http://docs.mosek.com/7.0/toolbox/Parameters.html.

Table C-16: OPF Options for PDIPM†

name	default	description
`pdipm.feastol`	0	feasibiliy (equality) tolerance set to value of `opf.violation` by default
`pdipm.gradtol`	10^{-6}	gradient tolerance
`pdipm.comptol`	10^{-6}	complementarity condition (inequality) tolerance
`pdipm.costtol`	10^{-6}	optimality tolerance
`pdipm.max_it`	150	maximum number of iterations
`pdipm.step_control`	0	set to 1 to enable step-size control
`pdipm.sc.red_it`‡	20	maximum number of step size reductions per iteration
`pdipm.sc.smooth_ratio`‡	0.04	piecewise linear curve smoothing ratio

† Requires installation of the optional TSPOPF package [16]. See Appendix G.10 for details.
‡ Only relevant when `pdipm.step_control` is on.

Table C-17: OPF Options for TRALM†

name	default	description
`tralm.feastol`	0	feasibiliy tolerance set to value of `opf.violation` by default
`tralm.primaltol`	5×10^{-4}	primal variable tolerance
`tralm.dualtol`	5×10^{-4}	dual variable tolerance
`tralm.costtol`	10^{-5}	optimality tolerance
`tralm.major_it`	40	maximum number of major iterations
`tralm.minor_it`	100	maximum number of minor iterations
`tralm.smooth_ratio`	0.04	piecewise linear curve smoothing ratio

† Requires installation of the optional TSPOPF package [16]. See Appendix G.10 for details.

C.1 Mapping of Old-Style Options to New-Style Options

A MATPOWER options struct can be created from an old-style options vector simply by passing it to `mpoption`. The mapping of old-style options into the fields in the option struct are summarized in Table C-18. An old-style options vector can also be created from an options struct by calling `mpoption` with the struct and an empty second argument.

```
mpopt_struct = mpoption(mpopt_vector);
mpopt_vector = mpoption(mpopt_struct, []);
```

Table C-18: Old-Style to New-Style Option Mapping

idx	old option	new option	notes
1	PF_ALG	pf.alg	new option has string values
	1 →	'NR'	
	2 →	'FDXB'	
	3 →	'FDBX'	
	4 →	'GS'	
2	PF_TOL	pf.tol	
3	PF_MAX_IT	pf.nr.max_it	
4	PF_MAX_IT_FD	pf.fd.max_it	
5	PF_MAX_IT_GS	pf.gs.max_it	
6	ENFORCE_Q_LIMS	pf.enforce_q_lims	
10	PF_DC	model	new option has string values
	0 →	'AC'	
	1 →	'DC'	
11	OPF_ALG	opf.ac.solver	new option has string values
	0 →	'DEFAULT'	
	500 →	'MINOPF'	
	520 →	'FMINCON'	
	540 →	'PDIPM' (and pdipm.step_control = 0)	
	545 →	'PDIPM' (and pdipm.step_control = 1)	
	550 →	'TRALM'	
	560 →	'MIPS' (and mips.step_control = 0)	
	565 →	'MIPS' (and mips.step_control = 1)	
	580 →	'IPOPT'	
	600 →	'KNITRO'	
16	OPF_VIOLATION	opf.violation	
17	CONSTR_TOL_X	fmincon.tol_x, knitro.tol_x	support for constr has been removed
18	CONSTR_TOL_F	fmincon.tol_f, knitro.tol_f	support for constr has been removed
19	CONSTR_MAX_IT	fmincon.max_it	support for constr has been removed

continued on next page

Table C-18: Old-Style to New-Style Option Mapping – *continued*

idx	old option	new option	notes
24	OPF_FLOW_LIM	opf.flow_lim	new option has string values
	0 →	'S'	
	1 →	'P'	
	2 →	'I'	
25	OPF_IGNORE_ANG_LIM	opf.ignore_angle_lim	
26	OPF_ALG_DC	opf.dc.solver	new option has string values
	0 →	'DEFAULT'	
	100 →	'BPMPD'	
	200 →	'MIPS' (and mips.step_control = 0)	
	250 →	'MIPS' (and mips.step_control = 1)	
	300 →	'OT'	
	400 →	'IPOPT'	
	500 →	'CPLEX'	
	600 →	'MOSEK'	
	700 →	'GUROBI'	
31	VERBOSE	verbose	
32	OUT_ALL	out.all	
33	OUT_SYS_SUM	out.sys_sum	
34	OUT_AREA_SUM	out.area_sum	
35	OUT_BUS	out.bug	
36	OUT_BRANCH	out.branch	
37	OUT_GEN	out.gen	
38	OUT_ALL_LIM	out.lim.all	
39	OUT_V_LIM	out.lim.v	
40	OUT_LINE_LIM	out.lim.line	
41	OUT_PG_LIM	out.lim.pg	
42	OUT_QG_LIM	out.lim.qg	
44	OUT_FORCE	out.force	
52	RETURN_RAW_DER	opf.return_raw_der	
55	FMC_ALG	fmincon.alg	
58	KNITRO_OPT	knitro.opt	
60	IPOPT_OPT	ipopt.opt	
61	MNS_FEASTOL	minopf.feastol	
62	MNS_ROWTOL	minopf.rowtol	
63	MNS_XTOL	minopf.xtol	
64	MNS_MAJDAMP	minopf.majdamp	
65	MNS_MINDAMP	minopf.mindamp	
66	MNS_PENALTY_PARM	minopf.penalty	
67	MNS_MAJOR_IT	minopf.major_it	

continued on next page

Table C-18: Old-Style to New-Style Option Mapping – *continued*

idx	old option	new option	notes
68	MNS_MINOR_IT	minopf.minor_it	
69	MNS_MAX_IT	minopf.max_it	
70	MNS_VERBOSITY	minopf.verbosity	
71	MNS_CORE	minopf.core	
72	MNS_SUPBASIC_LIM	minopf.supbasic_lim	
73	MNS_MULT_PRICE	minopf.mult_price	
80	FORCE_PC_EQ_P0	sopf.force_Pc_eq_P0	for c3sopf *(not part of* MATPOWER*)*
81	PDIPM_FEASTOL	mips.feastol, pdipm.feastol	
82	PDIPM_GRADTOL	mips.gradtol, pdipm.gradtol	
83	PDIPM_COMPTOL	mips.comptol, pdipm.comptol	
84	PDIPM_COSTTOL	mips.costtol, pdipm.costtol	
85	PDIPM_MAX_IT	mips.max_it, pdipm.max_it	
86	SCPDIPM_RED_IT	mips.sc.red_it, pdipm.sc.red_it	
87	TRALM_FEASTOL	tralm.feastol	
88	TRALM_PRIMETOL	tralm.primaltol	
89	TRALM_DUALTOL	tralm.dualtol	
90	TRALM_COSTTOL	tralm.costtol	
91	TRALM_MAJOR_IT	tralm.major_it	
92	TRALM_MINOR_IT	tralm.minor_it	
93	SMOOTHING_RATIO	tralm.smooth_ratio, pdipm.sc.smooth_ratio	
95	CPLEX_LPMETHOD	cplex.lpmethod	
98	CPLEX_QPMETHOD	cplex.qpmethod	
97	CPLEX_OPT	cplex.opt	
111	MOSEK_LP_ALG	mosek.lp_alg	
112	MOSEK_MAX_IT	mosek.max_it	
113	MOSEK_GAP_TOL	mosek.gap_tol	
114	MOSEK_MAX_TIME	mosek.max_time	
115	MOSEK_NUM_THREADS	mosek.num_threads	
116	MOSEK_OPT	mosek.opt	

continued on next page

Table C-18: Old-Style to New-Style Option Mapping – *continued*

idx	old option	new option	notes
121	GRB_METHOD	gurobi.method	
122	GRB_TIMELIMIT	gurobi.timelimit	
123	GRB_THREADS	gurobi.threads	
124	GRB_OPT	gurobi.opt	

Appendix D MATPOWER Files and Functions

This appendix lists all of the files and functions that MATPOWER provides, with the exception of those in the `extras` directory (see Appendix E). In most cases, the function is found in a MATLAB M-file of the same name in the top-level of the distribution, where the `.m` extension is omitted from this listing. For more information on each, at the MATLAB prompt, simply type `help` followed by the name of the function. For documentation and data files, the filename extensions are included.

D.1 Documentation Files

Table D-1: MATPOWER Documentation Files

name	description
`README`, `README.txt`[†]	basic introduction to MATPOWER
`docs/`	
`CHANGES`, `CHANGES.txt`[†]	MATPOWER change history
`manual.pdf`	MATPOWER *5.0 User's Manual*
`TN1-OPF-Auctions.pdf`	Tech Note 1 *Uniform Price Auctions and Optimal Power Flow*
`TN2-OPF-Derivatives.pdf`	Tech Note 2 *AC Power Flows, Generalized OPF Costs and their Derivatives using Complex Matrix Notation*

[†] For Windows users, text file with Windows-style line endings.

D.2 MATPOWER Functions

Table D-2: Top-Level Simulation Functions

name	description
runpf	power flow†
runcpf	AC continuation power flow
runopf	optimal power flow†
runuopf	optimal power flow with unit-decommitment†
rundcpf	DC power flow‡
rundcopf	DC optimal power flow‡
runduopf	DC optimal power flow with unit-decommitment‡
runopf_w_res	optimal power flow with fixed reserve requirements†

† Uses AC model by default.
‡ Simple wrapper function to set option to use DC model before calling the corresponding general function above.

Table D-3: Input/Output Functions

name	description
cdf2mpc	converts power flow data from IEEE Common Data Format (CDF) to MATPOWER format
loadcase	loads data from a MATPOWER case file or struct into data matrices or a case struct
mpoption	sets and retrieves MATPOWER options
printpf	pretty prints power flow and OPF results
psse2mpc	converts power flow data from PSS/E format to MATPOWER format
savecase	saves case data to a MATPOWER case file

Table D-4: Data Conversion Functions

name	description
ext2int	converts case from external to internal indexing
e2i_data	converts arbitrary data from external to internal indexing
e2i_field	converts fields in mpc from external to internal indexing
int2ext	converts case from internal to external indexing
i2e_data	converts arbitrary data from internal to external indexing
i2e_field	converts fields in mpc from internal to external indexing
get_reorder	returns A with one of its dimensions indexed
set_reorder	assigns B to A with one of the dimensions of A indexed

Table D-5: Power Flow Functions

name	description
dcpf	implementation of DC power flow solver
fdpf	implementation of fast-decoupled power flow solver
gausspf	implementation of Gauss-Seidel power flow solver
newtonpf	implementation of Newton-method power flow solver
pfsoln	computes branch flows, generator reactive power (and real power for slack bus), updates bus, gen, branch matrices with solved values

Table D-6: Continuation Power Flow Functions

name	description
cpf_corrector	computes Newton method corrector steps
cpf_default_callback	callback function that accumulates results from each iteration
cpf_predictor	computes tangent predictor steps
cpf_user_callback_1	example user callback function that plots a nose curve

Table D-7: OPF and Wrapper Functions

name	description
opf†	the main OPF function, called by runopf
dcopf‡	calls opf with options set to solve DC OPF
fmincopf‡	calls opf with options set to use fmincon to solve AC OPF
mopf‡	calls opf with options set to use MINOPF to solve AC OPF§
uopf‡	implements unit-decommitment heuristic, called by runuopf

† Can also be used as a top-level function, run directly from the command line. It provides more calling options than runopf, primarly for backward compatibility with previous versions of mopf from MINOPF, but does not offer the option to save the output or the solved case.
‡ Wrapper with same calling conventions as opf.
§ Requires the installation of an optional package. See Appendix G for details on the corresponding package.

Table D-8: OPF Model Objects

name	description
`@opf_model/`	OPF model object used to encapsulate the OPF problem formulation
`display`	called to display object when statement not terminated by semicolon
`get_mpc`	returns the MATPOWER case struct
`opf_model`	constructor for the `opf_model` class
`@opt_model/`	optimization model object (`@opf_model` base class)
`add_constraints`	adds a named subset of constraints to the model
`add_costs`	adds a named subset of user-defined costs to the model
`add_vars`	adds a named subset of optimization variables to the model
`build_cost_params`	builds and stores the full generalized cost parameters in the model
`compute_cost`	computes a user-defined cost†
`describe_idx`	identifies variable, constraint or cost row indices
`display`	called to display object when statement not terminated by semicolon
`get_cost_params`	returns the cost parameter struct created by `build_cost_params`†
`get_idx`	returns the idx struct for vars, lin/nln constraints, costs
`getN`	returns the number of variables, constraints or cost rows†
`get`	returns the value of a field of the object
`getv`	returns the initial values and bounds for optimimization vector†
`linear_constraints`	builds and returns the full set of linear constraints (A, l, u)
`opt_model`	constructor for the `opt_model` class
`userdata`	saves or returns values of user data stored in the model

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

Table D-9: OPF Solver Functions

name	description
`dcopf_solver`	sets up and solves DC OPF problem
`fmincopf_solver`	sets up and solves OPF problem using `fmincon`, MATLAB Opt Toolbox
`ipoptopf_solver`	sets up and solves OPF problem using IPOPT†
`ktropf_solver`	sets up and solves OPF problem using KNITRO†
`mipsopf_solver`	sets up and solves OPF problem using MIPS
`mopf_solver`	sets up and solves OPF problem using MINOPF†
`tspopf_solver`	sets up and solves OPF problem using PDIPM, SC-PDIPM or TRALM†

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

Table D-10: Other OPF Functions

name	description
`margcost`	computes the marginal cost of generation as a function of generator output
`makeAang`	forms linear constraints for branch angle difference limits
`makeApq`	forms linear constraints for generator PQ capability curves
`makeAvl`	forms linear constraints for dispatchable load constant power factor
`makeAy`	forms linear constraints for piecewise linear generator costs (CCV)
`opf_args`	input argument handling for `opf`
`opf_setup`	constructs an OPF model object from a MATPOWER case
`opf_execute`	executes the OPF specified by an OPF model object
`opf_consfcn`†	evaluates function and gradients for AC OPF nonlinear constraints
`opf_costfcn`†	evaluates function, gradients and Hessian for AC OPF objective function
`opf_hessfcn`†	evaluates the Hessian of the Lagrangian for AC OPF
`totcost`	computes the total cost of generation as a function of generator output
`update_mupq`	updates generator limit prices based on the shadow prices on capability curve constraints

† Used by `fmincon`, MIPS, IPOPT and KNITRO for AC OPF.

Table D-11: OPF User Callback Functions

name	description
`add_userfcn`	appends a user callback function to the list of those to be called for a given case
`remove_userfcn`	removes a user callback function from the list
`run_userfcn`	executes the user callback functions for a given stage
`toggle_dcline`	enable/disable or check the status of the callbacks implementing DC lines
`toggle_iflims`	enable/disable or check the status of the callbacks implementing interface flow limits
`toggle_reserves`	enable/disable or check the status of the callbacks implementing fixed reserve requirements
`toggle_softlims`	enable/disable or check the status of the callbacks implementing DC OPF branch flow soft limits

Table D-12: Power Flow Derivative Functions

name	description†
`dIbr_dV`	evaluates the partial derivatives of $I_{f\|t}$ evaluates the V
`dSbr_dV`	evaluates the partial derivatives of $S_{f\|t}$ evaluates the V
`dSbus_dV`	evaluates the partial derivatives of S_{bus} evaluates the V
`dAbr_dV`	evaluates the partial derivatives of $\|F_{f\|t}\|^2$ with respect to V
`d2Ibr_dV2`	evaluates the 2nd derivatives of $I_{f\|t}$ evaluates the V
`d2Sbr_dV2`	evaluates the 2nd derivatives of $S_{f\|t}$ evaluates the V
`d2AIbr_dV2`	evaluates the 2nd derivatives of $\|I_{f\|t}\|^2$ evaluates the V
`d2ASbr_dV2`	evaluates the 2nd derivatives of $\|S_{f\|t}\|^2$ evaluates the V
`d2Sbus_dV2`	evaluates the 2nd derivatives of S_{bus} evaluates the V

† V represents complex bus voltages, $I_{f|t}$ complex branch current injections, $S_{f|t}$ complex branch power injections, I_{bus} complex bus current injections, S_{bus} complex bus power injections and $F_{f|t}$ refers to branch flows, either $I_{f|t}$ or $S_{f|t}$, depending on the inputs. The second derivatives are all actually partial derivatives of the product of a first derivative matrix and a vector λ.

Table D-13: NLP, LP & QP Solver Functions

name	description
`cplex_options`	default options for CPLEX solver†
`glpk_options`	default options for GLPK solver†
`gurobi_options`	default options for Gurobi solver†
`gurobiver`	prints version information for Gurobi/Gurobi_MEX
`ipopt_options`	default options for Ipopt solver†
`mips`	Matlab Interior Point Solver – primal/dual interior point solver for NLP
`mipsver`	prints version information for MIPS
`mosek_options`	default options for MOSEK solver†
`qps_matpower`	Quadratic Program Solver for Matpower, wrapper function provides a common QP solver interface for various QP/LP solvers
`qps_bpmpd`	common QP/LP solver interface to BPMPD_MEX†
`qps_cplex`	common QP/LP solver interface to CPLEX (`cplexlp` and `cplexqp`)†
`qps_glpk`	common QP/LP solver interface to GLPK†
`qps_gurobi`	common QP/LP solver interface to Gurobi†
`qps_ipopt`	common QP/LP solver interface to Ipopt-based solver†
`qps_mips`	common QP/LP solver interface to MIPS-based solver
`qps_mosek`	common QP/LP solver interface to MOSEK (`mosekopt`)†
`qps_ot`	common QP/LP solver interface to Matlab Opt Toolbox's `quadprog`, `linprog`

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

Table D-14: Matrix Building Functions

name	description
makeB	forms the fast-decoupled power flow matrices, B' and B''
makeBdc	forms the system matrices B_{bus} and B_f and vectors $P_{f,\text{shift}}$ and $P_{\text{bus,shift}}$ for the DC power flow model
makeJac	forms the power flow Jacobian matrix
makeLODF	forms the line outage distribution factor matrix
makePTDF	forms the DC PTDF matrix for a given choice of slack
makeSbus	forms the vector of complex bus power injections
makeYbus	forms the complex bus and branch admittance matrices Y_{bus}, Y_f and Y_t

Table D-15: Utility Functions

name	description
`bustypes`	creates vectors of bus indices for reference bus, PV buses, PQ buses
`case_info`	checks a MATPOWER case for connectivity and prints a system summary
`compare_case`	prints summary of differences between two MATPOWER cases
`connected_components`	returns the connected components of a graph
`define_constants`	convenience script defines constants for named column indices to data matrices (calls `idx_bus`, `idx_brch`, `idx_gen` and `idx_cost`)
`extract_islands`	extracts islands in a network into their own MATPOWER case struct
`fairmax`	same as MATLAB's `max` function, except it breaks ties randomly
`find_islands`	finds islands and isolated buses in a network
`get_losses`	compute branch losses (and derivatives) as functions of bus voltage
`hasPQcap`	checks for generator P-Q capability curve constraints
`have_fcn`	checks for availability of optional functionality
`idx_brch`	named column index definitions for `branch` matrix
`idx_bus`	named column index definitions for `bus` matrix
`idx_cost`	named column index definitions for `gencost` matrix
`idx_dcline`	named column index definitions for `dcline` matrix
`idx_gen`	named column index definitions for `gen` matrix
`isload`	checks if generators are actually dispatchable loads
`load2disp`	converts fixed loads to dispatchable loads
`mpoption_info_cplex`	option information for CPLEX
`mpoption_info_fmincon`	option information for FMINCON
`mpoption_info_glpk`	option information for GLPK
`mpoption_info_gurobi`	option information for Gurobi
`mpoption_info_ipopt`	option information for IPOPT
`mpoption_info_knitro`	option information for KNITRO
`mpoption_info_linprog`	option information for LINPROG
`mpoption_info_mips`	option information for MIPS
`mpoption_info_mosek`	option information for MOSEK
`mpoption_info_quadprog`	option information for QUADPROG
`mpver`	prints version information for MATPOWER and optional packages
`modcost`	modifies `gencost` by horizontal or vertical scaling or shifting
`nested_struct_copy`	copies the contents of nested structs
`poly2pwl`	creates piecewise linear approximation to polynomial cost function
`polycost`	evaluates polynomial generator cost and its derivatives
`pqcost`	splits `gencost` into real and reactive power costs
`psse_convert`	converts data read from PSS/E RAW file to MATPOWER format
`psse_convert_hvdc`	called by `psse_convert` to handle HVDC data
`psse_convert_xfmr`	called by `psse_convert` to handle transformer data
`psse_parse`	parses data from a PSS/E RAW data file
`psse_parse_line`	called by `psse_parse` to parse a single line
`psse_parse_section`	called by `psse_parse` to parse an entire section
`psse_read`	reads data from a PSS/E RAW data file
`scale_load`	scales fixed and/or dispatchable loads by load zone
`total_load`	returns vector of total load in each load zone

D.3 Example MATPOWER Cases

Table D-16: Example Cases

name	description
`caseformat`	help file documenting MATPOWER case format
`case_ieee30`	IEEE 30-bus case
`case24_ieee_rts`	IEEE RTS 24-bus case
`case4gs`	4-bus example case from Grainger & Stevenson
`case5`	modified 5-bus PJM example case from Rui Bo
`case6ww`	6-bus example case from Wood & Wollenberg
`case9`	9-bus example case from Chow
`case9Q`	`case9` with reactive power costs
`case9target`	modified `case9`, target for example continuation power flow
`case14`	IEEE 14-bus case
`case30`	30-bus case, based on IEEE 30-bus case
`case30pwl`	`case30` with piecewise linear costs
`case30Q`	`case30` with reactive power costs
`case39`	39-bus New England case
`case57`	IEEE 57-bus case
`case118`	IEEE 118-bus case
`case300`	IEEE 300-bus case
`case2383wp`	Polish system - winter 1999-2000 peak
`case2736sp`	Polish system - summer 2004 peak
`case2737sop`	Polish system - summer 2004 off-peak
`case2746wop`	Polish system - winter 2003-04 off-peak
`case2746wp`	Polish system - winter 2003-04 evening peak
`case3012wp`	Polish system - winter 2007-08 evening peak
`case3120sp`	Polish system - summer 2008 morning peak
`case3375wp`	Polish system plus - winter 2007-08 evening peak

D.4 Automated Test Suite

Table D-17: Automated Test Utility Functions

name	description
`t/`	
`t_begin`	begin running tests
`t_end`	finish running tests and print statistics
`t_is`	tests if two matrices are identical to with a specified tolerance
`t_ok`	tests if a condition is true
`t_run_tests`	run a series of tests
`t_skip`	skips a number of tests, with explanatory message

Table D-18: Test Data

name	description
t/	
pretty_print_acopf.txt	pretty-printed output of an example AC OPF run
pretty_print_dcopf.txt	pretty-printed output of an example DC OPF run
soln9_dcopf.mat	solution data, DC OPF of t_case9_opf
soln9_dcpf.mat	solution data, DC power flow of t_case9_pf
soln9_opf_ang.mat	solution data, AC OPF of t_case9_opfv2 w/only branch angle difference limits (gen PQ capability constraints removed)
soln9_opf_extras1.mat	solution data, AC OPF of t_case9_opf w/extra cost/constraints
soln9_opf_Plim.mat	solution data, AC OPF of t_case9_opf w/opf.flow_lim = 'P'
soln9_opf_PQcap.mat	solution data, AC OPF of t_case9_opfv2 w/only gen PQ capability constraints (branch angle diff limits removed)
soln9_opf.mat	solution data, AC OPF of t_case9_opf
soln9_pf.mat	solution data, AC power flow of t_case9_pf
t_auction_case.m	case data used to test auction code
t_case_ext.m	case data used to test ext2int and int2ext, external indexing
t_case_info_eg.txt	example output from case_info, used by t_islands
t_case_int.m	case data used to test ext2int and int2ext, internal indexing
t_case9_dcline.m	same as t_case9_opfv2 with additional DC line data
t_case9_opf.m	sample case file with OPF data, version 1 format
t_case9_opfv2.m	sample case file with OPF data, version 2 format, includes additional branch angle diff limits & gen PQ capability constraints
t_case9_pf.m	sample case file with only power flow data, version 1 format
t_case9_pfv2.m	sample case file with only power flow data, version 2 format
t_case30_userfcns.m	sample case file with OPF, reserves and interface flow limit data
t_psse_case.raw	sample PSS/E format RAW file used to test conversion code
t_psse_case2.m	result of converting t_psse_case2.raw to MATPOWER format
t_psse_case2.raw	sample PSS/E format RAW file used to test conversion code
t_psse_case3.m	result of converting t_psse_case3.raw to MATPOWER format
t_psse_case3.raw	sample PSS/E format RAW file used to test conversion code

Table D-19: Miscellaneous MATPOWER Tests

name	description
t/	
test_matpower	runs all MATPOWER tests
t_auction_minopf	runs tests for auction using MINOPF†
t_auction_mips	runs tests for auction using MIPS
t_auction_tspopf_pdipm	runs tests for auction using PDIPM†
t_cpf	runs tests for AC continuation power flow
t_dcline	runs tests for DC line implementation in toggle_dcline
t_ext2int2ext	runs tests for ext2int and int2ext
t_get_losses	runs tests for get_losses
t_hasPQcap	runs tests for hasPQcap
t_hessian	runs tests for 2nd derivative code
t_islands	runs test for find_islands and extract_islands
t_jacobian	runs test for partial derivative code
t_loadcase	runs tests for loadcase
t_makeLODF	runs tests for makeLODF
t_makePTDF	runs tests for makePTDF
t_margcost	runs tests for margcost
t_mips	runs tests for MIPS NLP solver
t_modcost	runs tests for modcost
t_mpoption	runs tests for mpoption
t_nested_struct_copy	runs tests for nested_struct_copy
t_off2case	runs tests for off2case
t_opf_model	runs tests for opf_model and opt_model objects
t_printpf	runs tests for printpf
t_psse	runs tests for psse2mpc and related functions
t_qps_matpower	runs tests for qps_matpower
t_pf	runs tests for AC and DC power flow
t_runmarket	runs tests for runmarket
t_scale_load	runs tests for scale_load
t_total_load	runs tests for total_load
t_totcost	runs tests for totcost

* Deprecated. Will be removed in a subsequent version.
† Requires the installation of an optional package. See Appendix G for details on the corresponding package.

Table D-20: MATPOWER OPF Tests

name	description
`t/`	
`t_opf_dc_bpmpd`	runs tests for DC OPF solver using BPMPD_MEX†
`t_opf_dc_cplex`	runs tests for DC OPF solver using CPLEX†
`t_opf_dc_glpk`	runs tests for DC OPF solver using GLPK†
`t_opf_dc_gurobi`	runs tests for DC OPF solver using Gurobi†
`t_opf_dc_ipopt`	runs tests for DC OPF solver using IPOPT†
`t_opf_dc_mosek`	runs tests for DC OPF solver using MOSEK†
`t_opf_dc_ot`	runs tests for DC OPF solver using MATLAB Opt Toolbox
`t_opf_dc_mips`	runs tests for DC OPF solver using MIPS
`t_opf_dc_mips_sc`	runs tests for DC OPF solver using MIPS-sc
`t_opf_fmincon`	runs tests for AC OPF solver using `fmincon`
`t_opf_ipopt`	runs tests for AC OPF solver using IPOPT†
`t_opf_knitro`	runs tests for AC OPF solver using KNITRO†
`t_opf_minopf`	runs tests for AC OPF solver using MINOPF†
`t_opf_mips`	runs tests for AC OPF solver using MIPS
`t_opf_mips_sc`	runs tests for AC OPF solver using MIPS-sc
`t_opf_softlims`	runs tests for DC OPF with user callback functions for branch flow soft limits
`t_opf_tspopf_pdipm`	runs tests for AC OPF solver using PDIPM†
`t_opf_tspopf_scpdipm`	runs tests for AC OPF solver using SC-PDIPM†
`t_opf_tspopf_tralm`	runs tests for AC OPF solver using TRALM†
`t_opf_userfcns`	runs tests for AC OPF with user callback functions for reserves and interface flow limits
`t_runopf_w_res`	runs tests for AC OPF with fixed reserve requirements

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

Appendix E Extras Directory

For a MATPOWER installation in `$MATPOWER`, the contents of `$MATPOWER/extras` contains additional MATPOWER related code, some contributed by others. Some of these could be moved into the main MATPOWER distribution in the future with a bit of polishing and additional documentation. Please contact the developers if you are interested in helping make this happen.

`misc`	A number of potentially useful functions that are either not yet fully implemented, tested and/or documented. See the help (and the code) in each individual file to understand what it does.
`sdp_pf`	Applications of a semidefinite programming programming relaxation of the power flow equations. Code contributed by Dan Molzahn. See Appendix G.9 and the documenation in the `$MATPOWER/extras/sdp_pf/documentation` directory, especially the file `sdp_pf_documentation.pdf`, for a full description of the functions in this package.
`se`	State-estimation code contributed by Rui Bo. Type `test_se`, `test_se_14bus` or `test_se_14bus_err` to run some examples. See `se_intro.pdf` for a brief introduction to this code.
`smartmarket`	Code that implements a "smart market" auction clearing mechanism based on MATPOWER's optimal power flow solver. See Appendix F for details.
`state_estimator`	Older state estimation example, based on code by James S. Thorp.

Appendix F "Smart Market" Code

Matpower 3 and later includes in the `extras/smartmarket` directory code that implements a "smart market" auction clearing mechanism. The purpose of this code is to take a set of offers to sell and bids to buy and use Matpower's optimal power flow to compute the corresponding allocations and prices. It has been used extensively by the authors with the optional MINOPF package [21] in the context of PowerWeb[25] but has not been widely tested in other contexts.

The smart market algorithm consists of the following basic steps:

1. Convert block offers and bids into corresponding generator capacities and costs.
2. Run an optimal power flow with decommitment option (`uopf`) to find generator allocations and nodal prices (λ_P).
3. Convert generator allocations and nodal prices into set of cleared offers and bids.
4. Print results.

For step 1, the offers and bids are supplied as two structs, `offers` and `bids`, each with fields `P` for real power and `Q` for reactive power (optional). Each of these is also a struct with matrix fields `qty` and `prc`, where the element in the i-th row and j-th column of `qty` and `prc` are the quantity and price, respectively of the j-th block of capacity being offered/bid by the i-th generator. These block offers/bids are converted to the equivalent piecewise linear generator costs and generator capacity limits by the `off2case` function. See `help off2case` for more information.

Offer blocks must be in non-decreasing order of price and the offer must correspond to a generator with $0 \leq$ `PMIN` $<$ `PMAX`. A set of price limits can be specified via the `lim` struct, e.g. and offer price cap on real energy would be stored in `lim.P.max_offer`. Capacity offered above this price is considered to be withheld from the auction and is not included in the cost function produced. Bids must be in non-increasing order of price and correspond to a generator with `PMIN` $<$ `PMAX` ≤ 0 (see Section 6.4.2 on page 47). A lower limit can be set for bids in `lim.P.min_bid`. See `help pricelimits` for more information.

The data specified by a Matpower case file, with the `gen` and `gencost` matrices modified according to step 1, are then used to run an OPF. A decommitment mechanism is used to shut down generators if doing so results in a smaller overall system cost (see Section 8).

[25] See http://www.pserc.cornell.edu/powerweb/.

In step 3 the OPF solution is used to determine for each offer/bid block, how much was cleared and at what price. These values are returned in `co` and `cb`, which have the same structure as `offers` and `bids`. The `mkt` parameter is a struct used to specify a number of things about the market, including the type of auction to use, type of OPF (AC or DC) to use and the price limits.

There are two basic types of pricing options available through `mkt.auction_type`, discriminative pricing and uniform pricing. The various uniform pricing options are best explained in the context of an unconstrained lossless network. In this context, the allocation is identical to what one would get by creating bid and offer stacks and finding the intersection point. The nodal prices (λ_P) computed by the OPF and returned in `bus(:,LAM_P)` are all equal to the price of the marginal block. This is either the last accepted offer (LAO) or the last accepted bid (LAB), depending which is the marginal block (i.e. the one that is split by intersection of the offer and bid stacks). There is often a gap between the last accepted bid and the last accepted offer. Since any price within this range is acceptable to all buyers and sellers, we end up with a number of options for how to set the price, as listed in Table F-1.

Table F-1: Auction Types

auction type	name	description
0	discriminative	price of each cleared offer (bid) is equal to the offered (bid) price
1	LAO	uniform price equal to the last accepted offer
2	FRO	uniform price equal to the first rejected offer
3	LAB	uniform price equal to the last accepted bid
4	FRB	uniform price equal to the first rejected bid
5	first price	uniform price equal to the offer/bid price of the marginal unit
6	second price	uniform price equal to min(FRO, LAB) if the marginal unit is an offer, or max(FRB, LAO) if it is a bid
7	split-the-difference	uniform price equal to the average of the LAO and LAB
8	dual LAOB	uniform price for sellers equal to LAO, for buyers equal to LAB

Generalizing to a network with possible losses and congestion results in nodal prices λ_P which vary according to location. These λ_P values can be used to normalize all bids and offers to a reference location by multiplying by a locational scale factor. For bids and offers at bus i, this scale factor is $\lambda_P^{\text{ref}}/\lambda_P^i$, where λ_P^{ref} is the nodal price at the reference bus. The desired uniform pricing rule can then be applied to the adjusted offers and bids to get the appropriate uniform price at the reference

bus. This uniform price is then adjusted for location by dividing by the locational scale factor. The appropriate locationally adjusted uniform price is then used for all cleared bids and offers.[26] The relationships between the OPF results and the pricing rules of the various uniform price auctions are described in detail in [25].

There are certain circumstances under which the price of a cleared offer determined by the above procedures can be less than the original offer price, such as when a generator is dispatched at its minimum generation limit, or greater than the price cap `lim.P.max_cleared_offer`. For this reason, all cleared offer prices are clipped to be greater than or equal to the offer price but less than or equal to `lim.P.max_cleared_offer`. Likewise, cleared bid prices are less than or equal to the bid price but greater than or equal to `lim.P.min_cleared_bid`.

F.1 Handling Supply Shortfall

In single sided markets, in order to handle situations where the offered capacity is insufficient to meet the demand under all of the other constraints, resulting in an infeasible OPF, we introduce the concept of emergency imports. We model an import as a fixed injection together with an equally sized dispatchable load which is bid in at a high price. Under normal circumstances, the two cancel each other and have no effect on the solution. Under supply shortage situations, the dispatchable load is not fully dispatched, resulting in a net injection at the bus, mimicking an import. When used in conjunction with the LAO pricing rule, the marginal load bid will not set the price if all offered capacity can be used.

F.2 Example

The case file `t/t_auction_case.m`, used for this example, is a modified version of the 30-bus system that has 9 generators, where the last three have negative `PMIN` to model the dispatchable loads.

- Six generators with three blocks of capacity each, offering as shown in Table F-2.
- Fixed load totaling 151.64 MW.
- Three dispatchable loads, bidding three blocks each as shown in Table F-3.

[26]In versions of MATPOWER prior to 4.0, the smart market code incorrectly shifted prices instead of scaling them, resulting in prices that, while falling within the offer/bid "gap" and therefore acceptable to all participants, did not necessarily correspond to the OPF solution.

Table F-2: Generator Offers

Generator	Block 1 MW @ $/MWh	Block 2 MW @ $/MWh	Block 3 MW @ $/MWh
1	12 @ $20	24 @ $50	24 @ $60
2	12 @ $20	24 @ $40	24 @ $70
3	12 @ $20	24 @ $42	24 @ $80
4	12 @ $20	24 @ $44	24 @ $90
5	12 @ $20	24 @ $46	24 @ $75
6	12 @ $20	24 @ $48	24 @ $60

Table F-3: Load Bids

Load	Block 1 MW @ $/MWh	Block 2 MW @ $/MWh	Block 3 MW @ $/MWh
1	10 @ $100	10 @ $70	10 @ $60
2	10 @ $100	10 @ $50	10 @ $20
3	10 @ $100	10 @ $60	10 @ $50

To solve this case using an AC optimal power flow and a last accepted offer (LAO) pricing rule, we use:

```
mkt.OPF = 'AC';
mkt.auction_type = 1;
```

and set up the problem as follows:

```
mpc = loadcase('t_auction_case');
offers.P.qty = [ ...
        12 24 24;
        12 24 24;
        12 24 24;
        12 24 24;
        12 24 24;
        12 24 24 ];

offers.P.prc = [ ...
        20 50 60;
        20 40 70;
        20 42 80;
        20 44 90;
        20 46 75;
        20 48 60 ];

bids.P.qty = [ ...
        10 10 10;
        10 10 10;
        10 10 10 ];

bids.P.prc = [ ...
        100 70 60;
        100 50 20;
        100 60 50 ];

[mpc_out, co, cb, f, dispatch, success, et] = runmarket(mpc, offers, bids, mkt);
```

The resulting cleared offers and bids are:

```
>> co.P.qty

ans =

   12.0000   23.3156         0
   12.0000   24.0000         0
   12.0000   24.0000         0
   12.0000   24.0000         0
   12.0000   24.0000         0
   12.0000   24.0000         0

>> co.P.prc

ans =

   50.0000   50.0000   50.0000
   50.2406   50.2406   50.2406
   50.3368   50.3368   50.3368
   51.0242   51.0242   51.0242
   52.1697   52.1697   52.1697
   52.9832   52.9832   52.9832

>> cb.P.qty

ans =

   10.0000   10.0000   10.0000
   10.0000         0         0
   10.0000   10.0000         0

>> cb.P.prc

ans =

   51.8207   51.8207   51.8207
   54.0312   54.0312   54.0312
   55.6208   55.6208   55.6208
```

In other words, the sales by generators and purchases by loads are as shown summarized in Tables F-4 and Tables F-5, respectively.

Table F-4: Generator Sales

Generator	Quantity Sold *MW*	Selling Price *$/MWh*
1	35.3	$50.00
2	36.0	$50.24
3	36.0	$50.34
4	36.0	$51.02
5	36.0	$52.17
6	36.0	$52.98

Table F-5: Load Purchases

Load	Quantity Bought *MW*	Purchase Price *$/MWh*
1	30.0	$51.82
2	10.0	$54.03
3	20.0	$55.62

F.3 Smartmarket Files and Functions

Table F-6: Smartmarket Files and Functions

name	description
`extras/smartmarket/`	
`auction`	clears set of bids and offers based on pricing rules and OPF results
`case2off`	generates quantity/price offers and bids from `gen` and `gencost`
`idx_disp`	named column index definitions for `dispatch` matrix
`off2case`	updates `gen` and `gencost` based on quantity/price offers and bids
`pricelimits`	fills in a struct with default values for offer and bid limits
`printmkt`	prints the market output
`runmarket`	top-level simulation function, runs the OPF-based smart market
`runmkt`*	top-level simulation function, runs the OPF-based smart market
`smartmkt`	implements the smart market solver
`SM_CHANGES`	change history for the smart market software

* Deprecated. Will be removed in a subsequent version.

Appendix G Optional Packages

There are a number of optional packages, not included in the MATPOWER distribution, that MATPOWER can utilize if they are installed in your MATLAB path. Each of them is based on one or more MEX files pre-compiled for various platforms, some distributed by PSERC, others available from third parties, and each with their own terms of use.

G.1 BPMPD_MEX – MEX interface for BPMPD

BPMPD_MEX [19, 20] is a MATLAB MEX interface to BPMPD, an interior point solver for quadratic programming developed by Csaba Mészáros at the MTA SZTAKI, Computer and Automation Research Institute, Hungarian Academy of Sciences, Budapest, Hungary. It can be used by MATPOWER's DC and LP-based OPF solvers and it improves the robustness of MINOPF. It is also useful outside of MATPOWER as a general-purpose QP/LP solver.

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 selected as the solver for DC OPFs by setting the `opf.solver.dc` option to `'BPMPD'`. It can also be used to solve general LP and QP problems via MATPOWER's common QP solver interface `qps_matpower` with the algorithm option set to `'BPMPD'` (or 100 for backward compatibility), or by calling `qps_bpmpd` directly.

G.2 CPLEX – High-performance LP and QP 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 http://www.ibm.com/software/integration/optimization/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, the CPLEX LP and QP solvers, `cplexlp` and `cplexqp`, can be used to solve DC OPF problems by setting the `opf.solver.dc` option equal to `'CPLEX'`. The solution algorithms can be controlled by MATPOWER's `cplex.lpmethod` and `cplex.qpmethod` options. See Table C-8 for a summary of the CPLEX-related MATPOWER options. A "CPLEX user options" function can also be specified via `cplex.opt_fname` to override the defaults for any of the many CPLEX parameters. See `help cplex_options` and the "Parameters of CPLEX" section of the CPLEX documentation at http://pic.dhe.ibm.com/infocenter/cosinfoc/v12r6/ for details.

It can also be used to solve general LP and QP problems via MATPOWER's common QP solver interface `qps_matpower` with the algorithm option set to `'CPLEX'` (or 500 for backward compatibility), or by calling `qps_cplex` directly.

G.3 GLPK – GNU Linear Programming Kit

The GLPK [26] (**G**NU **L**inear **P**rogramming **K**it) package is intended for solving large-scale linear programming (LP), mixed integer programming (MIP), and other related problems. It is a set of routines written in ANSI C and organized in the form of a callable library.

When GLPK is installed, either as part of Octave or with a MEX interface for MATLAB, it can be used to solve DC OPF problems that do not include any quadratic costs by setting the `opf.solver.dc` option equal to `'GLPK'`. The solution algorithms and other GLPK parameters can be set directly via MATPOWER's `glpk.opts` option. A "GLPK user options" function can also be specified via `glpk.opt_fname` to override the defaults for any of the many GLPK parameters. See `help glpk_options` and the parameters section the GLPK documentation at http://www.gnu.org/software/octave/doc/interpreter/Linear-Programming.html for details. See Table C-10 for a summary of the GLPK-related MATPOWER options.

GLPK can also be used to solve general LP problems via MATPOWER's common QP solver interface `qps_matpower` with the algorithm option set to `'GLPK'`, or by calling `qps_glpk` directly.

G.4 Gurobi – High-performance LP and QP Solvers

Gurobi 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 http://www.gurobi.com/.

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

Beginning with version 5.0.0, Gurobi includes a native MATLAB interface, which is supported in MATPOWER version 4.2 and above.[27]

When Gurobi is installed, it can be used to solve DC OPF problems by setting the `opf.solver.dc` option equal to `'GUROBI'`. The solution algorithms can be controlled by MATPOWER's `gurobi.method` option. See Table C-11 for a summary of the Gurobi-related MATPOWER options. A "Gurobi user options" function can also be specified via `gurobi.opt_fname` to override the defaults for any of the many Gurobi parameters. See `help gurobi_options` and the "Parameters" section of the "Gurobi Optimizer Reference Manual" at http://www.gurobi.com/documentation/5.6/reference-manual/parameters for details.

It can also be used to solve general LP and QP problems via MATPOWER's common QP solver interface `qps_matpower` with the algorithm option set to `'GUROBI'` (or 700 for backward compatibility), or by calling `qps_gurobi` directly.

G.5 IPOPT – Interior Point Optimizer

IPOPT [28] (**I**nterior **P**oint **OPT**imizer, pronounced I-P-Opt) is a software package for large-scale nonlinear optimization. It is is written in C++ and is released as open source code under the Common Public License (CPL). It is available from the COIN-OR initiative at https://projects.coin-or.org/Ipopt/. The code has been written by Carl Laird and Andreas Wächter, who is the COIN project leader for IPOPT.

MATPOWER requires the MATLAB MEX interface to IPOPT, which is included in 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

[27] MATPOWER version 4.1 supported Gurobi version 4.x, which required a free third-party MATLAB MEX interface [27], available at http://www.convexoptimization.com/wikimization/index.php/Gurobi_mex.

lists for help in building and installing 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.

When installed, IPOPT can be used by MATPOWER to solve both AC and DC OPF problems by setting the `opf.solver.ac` or `opf.solver.dc` options, respectively, equal to `'IPOPT'`. See Table C-12 for a summary of the IPOPT-related MATPOWER options. The many algorithm options can be set by creating an "IPOPT user options" function and specifying it via `ipopt.opt_fname` to override the defaults set by `ipopt_options`. See `help ipopt_options` and the options reference section of the IPOPT documentation at http://www.coin-or.org/Ipopt/documentation/ for details.

It can also be used to solve general LP and QP problems via MATPOWER's common QP solver interface `qps_matpower` with the algorithm option set to `'IPOPT'` (or 400 for backward compatibility), or by calling `qps_ipopt` directly.

G.6 KNITRO – Non-Linear Programming Solver

KNITRO [29] is a general purpose optimization solver specializing in nonlinear problems. The MATLAB Optimization Toolbox from The MathWorks includes an interface to the KNITRO libraries called `ktrlink`, but the libraries themselves must be acquired directly from Ziena Optimization, LLC. More information is available at http://www.ziena.com/ and http://www.ziena.com/knitromatlab.htm.

Although 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, `ktrlink`, can be used by MATPOWER to solve AC OPF problems by simply setting the `opf.solver.ac` option to `'KNITRO'`. See Table C-13 for a summary of KNITRO-related MATPOWER options. The `ktrlink` function uses callbacks to MATLAB functions to evaluate the objective function and its gradient, the constraint values and Jacobian, and the Hessian of the Lagrangian.

KNITRO options can be controlled directly by creating a standard KNITRO options file in your working directory and specifying it via the `knitro.opt_fname` (or, for backward compatibility, naming it `knitro_user_options_`n`.txt` and setting MATPOWER's `knitro.opt` option to n, where n is some positive integer value). See the KNITRO user manuals at http://www.ziena.com/documentation.htm for details on the available options.

G.7 MINOPF – AC OPF Solver Based on MINOS

MINOPF [21] is a MINOS-based optimal power flow solver for use with MATPOWER. It is for educational and research use only. MINOS [22] is a legacy Fortran-based software package, developed at the Systems Optimization Laboratory at Stanford University, for solving large-scale optimization problems.

While MINOPF is often MATPOWER's fastest AC OPF solver on small problems, as of MATPOWER 4, it no longer becomes the default AC OPF solver when it is installed. It can be selected manually by setting the `opf.solver.ac` option to `'MINOPF'` (see `help mpoption` for details).

Builds are available for Linux (32-bit), Mac OS X (PPC, Intel 32-bit) and Windows (32-bit) at http://www.pserc.cornell.edu/minopf/.

G.8 MOSEK – High-performance LP and QP 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 http://www.mosek.com/.

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

When the MATLAB interface to MOSEK is installed, the MOSEK LP and QP solvers can be used to solve DC OPF problems by setting the `opf.solver.dc` option equal to `'MOSEK'`. The solution algorithm for LP problems can be controlled by MATPOWER's `mosek.lp_alg` option. See Table C-15 for other MOSEK-related MATPOWER options. A "MOSEK user options" function can also be specified via `mosek.opt_fname` to override the defaults for any of the many MOSEK parameters. For details see `help mosek_options` and the "Parameters" reference in "The MOSEK optimization toolbox for MATLAB manual" at http://docs.mosek.com/7.0/toolbox/Parameters.html.

It can also be used to solve general LP and QP problems via MATPOWER's common QP solver interface `qps_matpower` with the algorithm option set to `'MOSEK'` (or 600 for backward compatibility), or by calling `qps_mosek` directly.

G.9 SDP_PF – Applications of a Semidefinite Programming Relaxation of the Power Flow Equations

This package is contributed by Dan Molzahn and is currently included with MATPOWER in the `$MATPOWER/extras/sdp_pf` directory. Complete documentation is avail-

able in `$MATPOWER/extras/sdp_pf/documentation` directory, and especially in the file `sdp_pf_documentation.pdf`.

G.10 TSPOPF – Three AC OPF Solvers by H. Wang

TSPOPF [16] is a collection of three high performance AC optimal power flow solvers for use with MATPOWER. The three solvers are:

- PDIPM – primal/dual interior point method
- SCPDIPM – step-controlled primal/dual interior point method
- TRALM – trust region based augmented Lagrangian method

The algorithms are described in [17, 24]. The first two are essentially C-language implementations of the algorithms used by MIPS (see Appendix A), with the exception that the step-controlled version in TSPOPF also includes a cost smoothing technique in place of the constrained-cost variable (CCV) approach for handling piece-wise linear costs.

The PDIPM in particular is significantly faster for large systems than any previous MATPOWER AC OPF solver, including MINOPF. When TSPOPF is installed, the PDIPM solver becomes the default optimal power flow solver for MATPOWER. Additional options for TSPOPF can be set using `mpoption` (see `help mpoption` for details).

Builds are available for Linux (32-bit, 64-bit), Mac OS X (PPC, Intel 32-bit, Intel 64-bit) and Windows (32-bit, 64-bit) at http://www.pserc.cornell.edu/tspopf/.

References

[1] GNU General Public License. [Online]. Available: http://www.gnu.org/licenses/. 1.2

[2] 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. 1.3

[3] F. Milano, "An Open Source Power System Analysis Toolbox," *Power Systems, IEEE Transactions on*, vol. 20, no. 3, pp. 1199–1206, Aug. 2005.

[4] W. F. Tinney and C. E. Hart, "Power Flow Solution by Newton's Method," *IEEE Transactions on Power Apparatus and Systems*, vol. PAS-86, no. 11, pp. 1449–1460, November 1967. 4.1

[5] B. Stott and O. Alsaç, "Fast Decoupled Load Flow," *IEEE Transactions on Power Apparatus and Systems*, vol. PAS-93, no. 3, pp. 859–869, May 1974. 4.1

[6] R. A. M. van Amerongen, "A General-Purpose Version of the Fast Decoupled Load Flow," *Power Systems, IEEE Transactions on*, vol. 4, no. 2, pp. 760–770, May 1989. 4.1

[7] A. F. Glimm and G. W. Stagg, "Automatic Calculation of Load Flows," *AIEE Transactions (Power Apparatus and Systems)*, vol. 76, pp. 817–828, October 1957. 4.1

[8] A. J. Wood and B. F. Wollenberg, *Power Generation, Operation, and Control*, 2nd ed. New York: J. Wiley & Sons, 1996. 3.7, 4.2, 4.4

[9] T. Guler, G. Gross, and M. Liu, "Generalized Line Outage Distribution Factors," *Power Systems, IEEE Transactions on*, vol. 22, no. 2, pp. 879–881, May 2007. 4.4

[10] V. Ajjarapu, C. Christy, "The Continuation Power Flow: A Tool for Steady State Voltage Stability Analysis," *Power Systems, IEEE Transacations on*, vol. 7, no. 1, pp. 416–423, Feb. 1992. 5

[11] 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. 5.1

[12] 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. 5.1

[13] A. J. Flueck, "Advances in Numerical Analysis of Nonlinear Dynamical Systems and the Application to Transfer Capability of Power Systems," *Ph. D. Dissertation*, Cornell University, August 1996.

[14] 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. 5.1

[15] R. D. Zimmerman, C. E. Murillo-Sánchez, and R. J. Thomas, "MATPOWER's Extensible Optimal Power Flow Architecture," *Power and Energy Society General Meeting, 2009 IEEE*, pp. 1–7, July 26–30 2009. 6.3

[16] TSPOPF. [Online]. Available: http://www.pserc.cornell.edu/tspopf/. 6.4.1, 6.5, C-16, C-17, G.10

[17] 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. 6.4.1, 6.5, A, A-3, A.4, G.10

[18] *Optimization Toolbox 4 Users's Guide*, The MathWorks, Inc., 2008. [Online]. Available: http://www.mathworks.com/access/helpdesk/help/pdf_doc/optim/optim_tb.pdf. 6.5

[19] BPMPD_MEX. [Online]. Available: http://www.pserc.cornell.edu/bpmpd/. 6.5, G.1

[20] 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. 6.5, G.1

[21] MINOPF. [Online]. Available: http://www.pserc.cornell.edu/minopf/. 6.5, C-14, F, G.7

[22] B. A. Murtagh and M. A. Saunders, *MINOS 5.5 User's Guide*, Stanford University Systems Optimization Laboratory Technical Report SOL83-20R. 6.5, G.7

[23] R. D. Zimmerman, *AC Power Flows, Generalized OPF Costs and their Derivatives using Complex Matrix Notation*, MATPOWER Technical Note 2, February 2010. [Online]. Available: http://www.pserc.cornell.edu/matpower/TN2-OPF-Derivatives.pdf 6.5

[24] H. Wang, *On the Computation and Application of Multi-period Security-constrained Optimal Power Flow for Real-time Electricity Market Operations*, Ph.D. thesis, Electrical and Computer Engineering, Cornell University, May 2007. A, A.4, G.10

[25] R. D. Zimmerman, *Uniform Price Auctions and Optimal Power Flow*, MATPOWER Technical Note 1, February 2010. [Online]. Available: http://www.pserc.cornell.edu/matpower/TN1-OPF-Auctions.pdf F

[26] GLPK. [Online]. Available: http://www.gnu.org/software/glpk/. G.3

[27] Wotao Yin. *Gurobi Mex: A MATLAB interface for Gurobi*, URL: http://convexoptimization.com/wikimization/index.php/gurobi_mex, 2009-2011. 27

[28] 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):2557, 2006. C-12, G.5

[29] 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.
http://www.ziena.com/papers/integratedpackage.pdf 6.5, C-13, G.6