Matpower 6.0b2
User's Manual

Ray D. Zimmerman   Carlos E. Murillo-Sánchez

November 1, 2016

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

MATPOWER [1] 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.

Beginning with version 6, MATPOWER includes a framework for solving generalized steady-state electric power scheduling problems. This framework is known as MOST, for **M**ATPOWER **O**ptimal **S**cheduling **T**ool [2].

MOST can be used to solve problems as simple as a deterministic, single period economic dispatch problem with no transmission constraints or as complex as a stochastic, security-constrained, combined unit-commitment and multiperiod optimal power flow problem with locational contingency and load-following reserves, ramping costs and constraints, deferrable demands, lossy storage resources and uncertain renewable generation.

MOST is documented separately from the main MATPOWER package in its own manual, the MOST User's Manual.

---

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

1.2 License and Terms of Use

Beginning with version 5.1, the code in MATPOWER is distributed under the 3-clause BSD license[3] [3]. The full text of the license can be found in the LICENSE file at the top level of the distribution or at http://www.pserc.cornell.edu/matpower/LICENSE.txt and reads as follows.

```
Copyright (c) 1996-2015, Power System Engineering Research Center
(PSERC) and individual contributors (see AUTHORS file for details).
All rights reserved.

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

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

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

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

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

---

[3]Versions 4.0 through 5.0 of MATPOWER were distributed under version 3.0 of the GNU General Public License (GPL) [4] 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 at http://www.gnu.org/licenses/gpl-3.0.txt. MATPOWER versions prior to version 4 had their own license.

Please note that the Matpower case files distributed with Matpower are not covered by the BSD license. 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 [1].

> R. D. Zimmerman, C. E. Murillo-Sánchez, and R. J. Thomas, "Matpower: Steady-State Operations, Planning and Analysis Tools for Power Systems Research and Education," *Power Systems, IEEE Transactions on*, vol. 26, no. 1, pp. 12–19, Feb. 2011. http://dx.doi.org/10.1109/TPWRS.2010.2051168

Similarly, we request that publications derived from the use of MOST explicitly acknowledge that fact by citing both the main Matpower reference above and reference [2]:

> C. E. Murillo-Sánchez, R. D. Zimmerman, C. L. Anderson, and R. J. Thomas, "Secure Planning and Operations of Systems with Stochastic Sources, Energy Storage and Active Demand," *Smart Grid, IEEE Transactions on*, vol. 4, no. 4, pp. 2220–2229, Dec. 2013, http://dx.doi.org/10.1109/TSG.2013.2281001.

2 Getting Started

2.1 System Requirements

To use MATPOWER 6.0b2 you will need:

- MATLAB® version 7 (R14) or later[4], or

- GNU Octave version 3.4 or later[5]

The PSS/E RAW data import function `psse2mpc` currently requires MATLAB 7.3 or newer or Octave 3.8 or newer. The continuation power flow `runcpf` is not supported in MATLAB 7.0.x (requires 7.1 or newer). See Section 2.1 in the MOST User's Manual for any requirements specific to MOST.

For the hardware requirements, please refer to the system requirements for the version of MATLAB[6] 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.

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, but Octave does typically include GLPK.

---

[4]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.

[5]GNU Octave is free software, available online at http://www.gnu.org/software/octave/. Some parts of MATPOWER 6.0b2 may work on earlier versions of Octave, but it has not been tested on versions prior to 3.4.

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

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[7]. 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.[8] 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.

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

- *<MATPOWER>* – core Matpower functions
- *<MATPOWER>*`/t` – test scripts for Matpower
- *<MATPOWER>*`/most`-- core MOST functions
- *<MATPOWER>*`/most/t` – test scripts for MOST
- (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.[9] The result should resemble the following, possibly including extra tests, depending on the availablility of optional packages, solvers and extras.

---

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

[8]Do *not* place Matpower's files in a directory named `'matlab'` or `'optim'` (both case-insensitive), as these can cause Matlab's built-in `ver` command to behave strangely in ways that affect Matpower.

[9]The MOST test suite is run separately by typing `test_most`. See the MOST User's Manual for details.

```
>> test_matpower
t_test_fcns.............ok
t_nested_struct_copy....ok
t_feval_w_path..........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_mplinsolve............ok (2 of 4 skipped)
t_mips..................ok
t_qps_matpower..........ok (288 of 360 skipped)
t_miqps_matpower........ok (240 of 240 skipped)
t_pf....................ok
t_cpf...................ok
t_islands...............ok
t_opf_model.............ok
t_opf_mips..............ok (125 of 250 skipped)
t_opf_mips_sc...........ok (125 of 250 skipped)
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_vdep_load.............ok
t_total_load............ok
t_scale_load............ok
t_apply_changes.........ok
t_psse..................ok
t_off2case..............ok
t_auction_mips..........ok
t_runmarket.............ok
All tests successful (4830 passed, 780 skipped of 5610)
Elapsed time 17.56 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[10]. 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-17.

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');
```

---

[10]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.

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);
```

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 four 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/MATPOWER-manual.pdf and the latest version is always available at: http://www.pserc.cornell.edu/matpower/MATPOWER-manual.pdf.

18

Secondly, the MOST User's Manual describes MOST and its problem formulation, features, data formats and options. It is located at *<MATPOWER>*/docs/MOST-manual.pdf in your MATPOWER distribution and the latest version is also available online at: http://www.pserc.cornell.edu/matpower/MOST-manual.pdf.

The MATPOWER Online Function Reference is the third source of documentation, allowing you to view not only the help text, but also the code itself for each MATPOWER function directly from a web browser. It is available online at: http://www.pserc.cornell.edu/matpower/docs/ref

Last, but certainly not least, 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.                20
```

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[11], 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

---

[11]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

$$Y_{br} = \begin{bmatrix} \left(y_s + j\frac{b_c}{2}\right)\frac{1}{\tau^2} & -y_s \frac{1}{\tau e^{-j\theta_{\text{shift}}}} \\ -y_s \frac{1}{\tau e^{j\theta_{\text{shift}}}} & y_s + j\frac{b_c}{2} \end{bmatrix}. \tag{3.2}$$
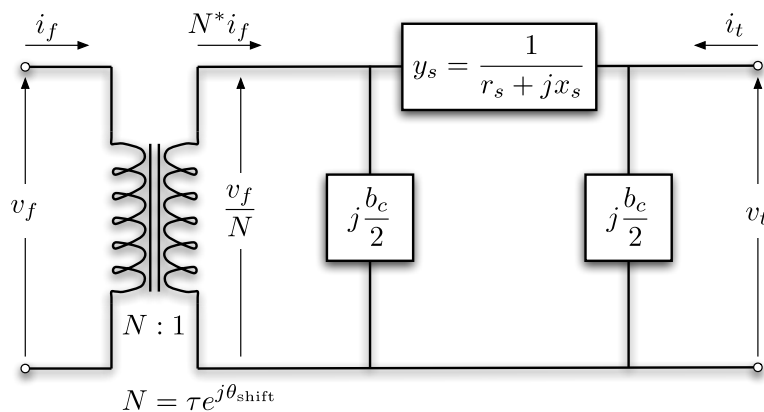
Figure 3-1: Branch Model

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_{bus} that relates the complex nodal current injections I_{bus} to the complex node voltages V:

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

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:

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

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:

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

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

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

3.7   DC Modeling

The DC formulation [10] 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:

$$i_f \approx \frac{1}{jx_s}\left(\frac{1}{\tau^2}e^{j\theta_f} - \frac{1}{\tau e^{-j\theta_{\text{shift}}}}e^{j\theta_t}\right)$$

$$= \frac{1}{jx_s\tau}\left(\frac{1}{\tau}e^{j\theta_f} - e^{j(\theta_t + \theta_{\text{shift}})}\right). \tag{3.22}$$

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).

$$p_f = \Re\{s_f\}$$

$$= \Re\{v_f \cdot i_f^*\}$$

$$\approx \Re\left\{e^{j\theta_f} \cdot \frac{j}{x_s\tau}\left(\frac{1}{\tau}e^{-j\theta_f} - e^{-j(\theta_t + \theta_{\text{shift}})}\right)\right\}$$

$$= \Re\left\{\frac{j}{x_s\tau}\left(\frac{1}{\tau} - e^{j(\theta_f - \theta_t - \theta_{\text{shift}})}\right)\right\}$$

$$= \Re\left\{\frac{1}{x_s\tau}\left[\sin(\theta_f - \theta_t - \theta_{\text{shift}}) + j\left(\frac{1}{\tau} - \cos(\theta_f - \theta_t - \theta_{\text{shift}})\right)\right]\right\}$$

$$\approx \frac{1}{x_s\tau}(\theta_f - \theta_t - \theta_{\text{shift}}) \tag{3.23}$$

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

$$\begin{bmatrix} p_f \\ p_t \end{bmatrix} = B^i_{br} \begin{bmatrix} \theta_f \\ \theta_t \end{bmatrix} + P^i_{\text{shift}} \tag{3.24}$$

where

$$B^i_{br} = b_i \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix},$$

$$P^i_{\text{shift}} = \theta^i_{\text{shift}} b_i \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

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

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

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

$$\begin{aligned}
s^i_{sh} &= v_i (y^i_{sh} v_i)^* \\
&\approx e^{j\theta_i} (g^i_{sh} - jb^i_{sh}) e^{-j\theta_i} \\
&= g^i_{sh} - jb^i_{sh}.
\end{aligned} \tag{3.25}$$

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 $-\theta^i_{\text{shift}} b_i$. 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

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

26

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:

$$g(x) = \left[\begin{array}{c} g_P^{\{i\}}(\Theta, V_m, P_g) \\ g_Q^{\{j\}}(\Theta, V_m, Q_g) \end{array} \right] \qquad \begin{array}{l} \forall i \in \mathcal{I}_{\mathrm{PV}} \cup \mathcal{I}_{\mathrm{PQ}} \\ \forall j \in \mathcal{I}_{\mathrm{PQ}}. \end{array} \qquad (4.4)$$

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:

$$x = \left[\begin{array}{c} \theta_{\{i\}} \\ v_m^{\{j\}} \end{array} \right] \qquad \begin{array}{l} \forall i \notin \mathcal{I}_{\mathrm{ref}} \\ \forall j \in \mathcal{I}_{\mathrm{PQ}}. \end{array} \qquad (4.5)$$

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 [6] 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 [7], specifically, the XB and BX methods described in [8]. 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 [9]. 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). Note also that this option affects generators even if the bus they are attached to is already of type PQ.

4.2 DC Power Flow

For the DC power flow problem [10], 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.

Table 4-1: Power Flow Results

name	description
results.success	success flag, 1 = succeeded, 0 = failed
results.et	computation time required for solution
results.iterations	number of iterations 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

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 = 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 [10]. 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 [11]. 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` and Sections 9.5.5 and 9.5.4, respectively, 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).

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

where, (x^j, λ^j) represents the current solution at step j, $(\hat{x}^{j+1}, \hat{\lambda}^{j+1})$ is the predicted solution for the next step, 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 [12] in what is called a continuation power flow[12]. 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

$$g(x) = \left[\begin{array}{c} P(x) - P^{inj} \\ Q(x) - Q^{inj} \end{array} \right] = 0, \tag{5.2}$$

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

$$b = \left[\begin{array}{c} P^{inj}_{target} - P^{inj}_{base} \\ Q^{inj}_{target} - Q^{inj}_{base} \end{array} \right]. \tag{5.4}$$

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

---

[12]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 [13,14]. 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.

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

- **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.

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

- **Pseudo arc length parameterization** [16] 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

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

where $\bar{z}^j$ is the normalized tangent vector at (x^j, λ^j) and σ^j 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

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

where σ^j 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 size σ is adjusted by a scaling factor α within the limits specified.

$$\sigma^{j+1} = \alpha^j \sigma^j, \qquad \sigma_{\min} \leq \sigma^{j+1} \leq \sigma_{\max} \tag{5.12}$$

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

$$\alpha^j = 1 + \beta_{\text{cpf}} \left(\frac{\epsilon_{\text{cpf}}}{\gamma^j} - 1 \right), \qquad \alpha^j \leq 2, \tag{5.13}$$

where β_{cpf} is a damping factor, ϵ_{cpf} is a specified tolerance, and γ^j is given by

$$\gamma^j = \left\| \left(x^{j+1}, \lambda^{j+1} \right) - \left(\hat{x}^{j+1}, \hat{\lambda}^{j+1} \right) \right\|_\infty. \tag{5.14}$$

5.5 Event Detection and Location

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

- a specified target λ value

- the nose point

- the end of a full trace

- a generator reactive power limit

- a generator active power limit

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

The detection of an event zero, or even an event interval, can be used to trigger further actions. The CPF callback function capability was extended in MATPOWER 6.x to include the ability to handle events by including information about any event intervals or zeros detected. For example, MATPOWER includes a callback that fixes the reactive output of a generator and converts its bus from PV to PQ when the corresponding event function indicates that its reactive power limit has been reached. Another responds to the detection of the nose point by signaling the termination of the continuation. In fact, continuation power flow termination for nose point, target lambda or full trace modes are all based on CPF callback functions in conjunction with event detection.

While MATPOWER does include a mechanism for supplying user defined callback functions, it does not yet have a corresponding mechanism for user specified event functions.

5.6 `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 load-
ing/generation profile while the second contains the target loading/generation pro-
file. 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.

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.

Table 5-1: Continuation Power Flow Results

name	description
results.cpf	CPF output struct whose content depends on any user callback functions, where default contains fields:
done_msg	string with message describing cause of continuation termination
events(eidx)	a structure array of size n_e, where n_e is the number of events located, with fields:
k	continuation step number at which event was located
name	name of event
idx	index(es) of critical elements in corresponding event function, e.g. index of generator reaching VAr limit
msg	descriptive text detailing the event
iterations	n_{steps}, number of continuation steps performed
lam	$1 \times n$ vector of λ values from correction steps[†]
lam_hat	$1 \times n$ vector of λ values from prediction steps[†]
max_lam	maximum value of λ found in results.cpf.lam
steps	$1 \times n$ vector of step sizes for each continuation step performed[†]
V	$n_b \times n$ matrix of complex bus voltages from correction steps[†]
V_hat	$n_b \times n$ matrix of complex bus voltages from prediction steps[†]

[†] n is one more than the number of continuation steps, i.e. $n_{\text{steps}} + 1$, so the first element corresponds to the starting point.

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.enforce_p_lims	0	enforce gen active power limits 0 — do *not* enforce limits 1 — enforce limits
cpf.enforce_q_lims	0	enforce gen reactive power limits at expense of V_m 0 — do *not* enforce limits 1 — enforce limits
cpf.step	0.05	default value for continuation power flow step size σ
cpf.step_min	10^{-4}	minimum allowed step size, σ_{min}
cpf.step_max	0.2	maximum allowed step size, σ_{max}
cpf.adapt_step	0	toggle adaptive step size feature 0 — adaptive step size disabled 1 — adaptive step size enabled
cpf.adapt_step_damping	0.7	damping factor β_{cpf} from (5.13) for adaptive step sizing
cpf.adapt_step_tol	10^{-3}	tolerance ϵ_{cpf} from (5.13) for adaptive step sizing
cpf.target_lam_tol	10^{-5}	tolerance for target lambda detection
cpf.nose_tol	10^{-5}	tolerance for nose point detection (p.u.)
cpf.p_lims_tol	10^{-2}	tolerance for generator active power limit detection (MW)
cpf.q_lims_tol	10^{-2}	tolerance for generator reactive power limit detection (MVAr)
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†

† See `help cpf_default_callback` for details.

5.6.1 CPF Callback Functions

MATPOWER's continuation power flow 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 or to handle a detected event. This callback mechanism is used internally to handle default plotting functionality as well as to handle the standard CPF events. The cpf_default_callback function, for example, is used to collect the λ and V results from each predictor and corrector iteration and optionally plot the PV nose curve.

The prototype for a CPF callback function is

```
function [nx, cx, done, rollback, evnts, cb_data, results] = ...
    cpf_user_callback(k, nx, cx, px, done, rollback, evnts, ...
                            cb_data, cb_args, results)
```

and the input and output arguments are described in Tables 5-3 through 5-5 and in the help for cpf_default_callback. Each registered CPF callback function is called in three different contexts, distinguished by the value of the first argument k as follows:

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

2. *iterations* – called with $k > 0$, without results input/output arguments, at each iteration, after predictor-corrector step

3. *final* – called with $k < 0$, with results input/output arguments, after exiting predictor-corrector loop, inputs identical to last iteration call, except k negated

The user can define their own callback functions which take the same form and are called in the same contexts as cpf_default_callback. User callback functions are specified via the MATPOWER option cpf.user_callback. This option can be a string containing the name of the callback function, or a struct with the following fields, where all but the first are optional:

- fcn - string with name of callback function

- priority - numerical value specifying callback priority,[13] default = 20

- args - arbitrary value (any type) passed to the callback as cb_args each time it is invoked

Multiple user callbacks can be registered by assigning a cell array of such strings and/or structs to the cpf.user_callback option.

---

[13]See cpf_register_callback for details.

Table 5-3: Continuation Power Flow Callback Input Arguments

name	description
k	continuation step iteration count
nx	next CPF state,[*] corresponding to proposed next step
cx	current CPF state,[*] corresponding to most recent successful step
px	previous CPF state,[*] corresponding to last successful step prior to cx
done	struct, with flag to indicate CPF termination and reason, with fields:
.flag	termination flag, $1 \rightarrow$ terminate, $0 \rightarrow$ continue
.msg	string containing reason for termination
rollback	scalar flag to indicate that the current step should be rolled back and retried with a different step size, etc.
evnts	struct array listing any events detected for this step[‡]
cb_data	struct containing static data[§], with the following fields (all based on internal indexing):
.mpc_base	MATPOWER case struct of base case
.mpc_target	MATPOWER case struct of target case
.Sbusb	handle of function returning $n_b \times 1$ vector of complex base case bus injections in p.u. and derivatives w.r.t. $|V|$
.Sbust	handle of function returning $n_b \times 1$ vector of complex target case bus injections in p.u. and derivatives w.r.t. $|V|$
.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
.idx_pmax	vector of gen indices of gens at PMAX
.mpopt	MATPOWER options struct
cb_args	arbitrary data structure containing user-provided callback arguments
results	initial value of output struct to be assigned to cpf field of results struct returned by runcpf

[*] See Table 5-5 for details of the CPF state.
[‡] See cpf_detect_events for details of the evnts field.
[§] Please note that any callback that modifies the underlying problem is responsible to update the contents of cb_data accordingly. E.g. converting a bus from PV to PQ requires updates to mpc_base, mpc_target, Sbusb, Sbust, pv, pq, and possibly ref. So, cb_data should only be thought of as static for a fixed base and target case pair.

Table 5-4: Continuation Power Flow Callback Output Arguments

name	description
All are updated versions of the corresponding input arguments, see Table 5-3 for more details.	
`nx`	next CPF state,[*] user state (`cb` field) should be updated here if `rollback` is false
`cx`	current CPF state,[*] may contain updated `this_step` or `this_parm` values to be used if `rollback` is true
`done`	callback may have requested termination and set the `msg` field
`rollback`	callback can request a rollback step, even if it was not indicated by an event function[†]
`evnts`	`msg` field for a given event may be updated[‡]
`cb_data`	this data should only be modified if the underlying problem has been changed (e.g. generator limit reached), in which case it should always be followed by a step of zero length, i.e. set `nx.this_step` to 0[§]
`results`	updated version of `results` input argument to be assigned to `cpf` field of results struct returned by `runcpf`

[*] See Table 5-5 for details of the CPF state.
[†] In this case, the callback should also modify the step size or parameterization to be used for the re-try, by setting the `this_step` or `this_parm` fields in `cx`.
[‡] See `cpf_detect_events` for details of the `evnts` field.
[§] It is the job of any callback modifying `cb_data` to ensure that all data in `cb_data` is kept consistent.

Table 5-5: Continuation Power Flow State

name	description
`lam_hat`	value of λ after predictor step
`V_hat`	vector of complex bus voltages after predictor step
`lam`	value of λ after corrector step
`V`	vector of complex bus voltages after corrector step
`z`	normalized tangent predictor, $\bar{z}$
`default_step`	default step size
`default_parm`	default parameterization[*]
`this_step`	step size for this step only
`this_parm`	parameterization[*] for this step only
`step`	current step size
`parm`	current parameterization[*]
`events`	event log
`cb`	user state for callbacks[†], a user-defined struct containing any information a callback function would like to pass from one invokation to the next
`ef`	cell array of event function values

[*] Corresponding to the `cpf.parameterization` option in Table 5-2.
[†] Replaces `cb_state` from MATPOWER 5.

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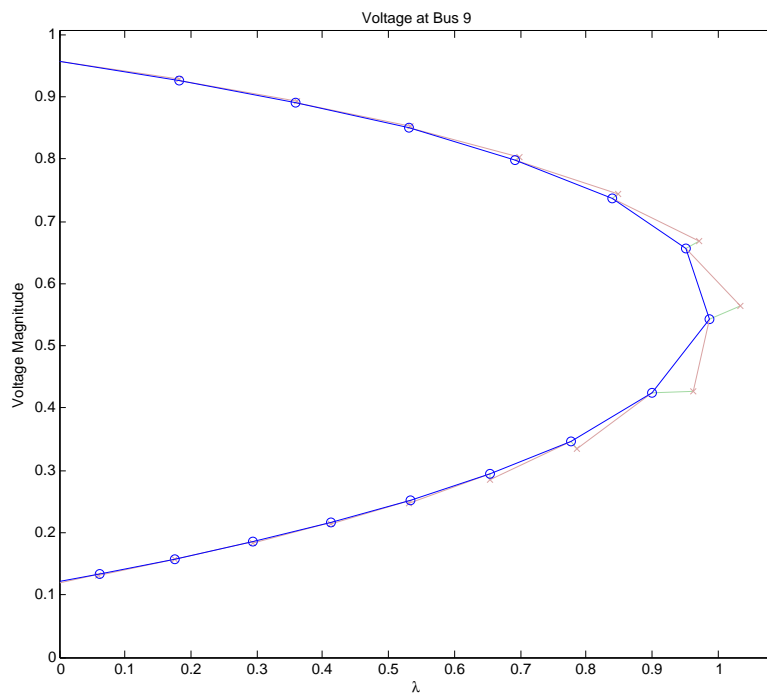
Figure 5-1: Nose Curve of Voltage Magnitude at Bus 9

```
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 6.0b2, 01-Nov-2016 -- AC Continuation Power Flow
step   0  :                         lambda =  0.000,  4 Newton steps
step   1  : stepsize = 0.2          lambda =  0.181   2 corrector Newton steps
step   2  : stepsize = 0.2          lambda =  0.359   2 corrector Newton steps
step   3  : stepsize = 0.2          lambda =  0.530   2 corrector Newton steps
step   4  : stepsize = 0.2          lambda =  0.693   3 corrector Newton steps
step   5  : stepsize = 0.2          lambda =  0.839   3 corrector Newton steps
step   6  : stepsize = 0.2          lambda =  0.952   3 corrector Newton steps
step   7  : stepsize = 0.2          lambda =  0.988   3 corrector Newton steps
step   8  : stepsize = 0.2          lambda =  0.899   3 corrector Newton steps
step   9  : stepsize = 0.2          lambda =  0.776   3 corrector Newton steps
step  10  : stepsize = 0.2          lambda =  0.654   3 corrector Newton steps
step  11  : stepsize = 0.2          lambda =  0.533   2 corrector Newton steps
step  12  : stepsize = 0.2          lambda =  0.413   2 corrector Newton steps
step  13  : stepsize = 0.2          lambda =  0.294   2 corrector Newton steps
step  14  : stepsize = 0.2          lambda =  0.176   2 corrector Newton steps
step  15  : stepsize = 0.2          lambda =  0.060   2 corrector Newton steps
step  16a : stepsize = 0.2          lambda = -0.054   2 corrector Newton steps ^ ROLLBACK
step  16  : stepsize = 0.0604       lambda =  0.000   3 corrector Newton steps
CPF TERMINATION: Traced full continuation curve in 16 continuation steps
```

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

```
>> results.cpf

ans =

         V_hat: [9x17 double]
       lam_hat: [1x17 double]
             V: [9x17 double]
           lam: [1x17 double]
         steps: [1x17 double]
    iterations: 16
       max_lam: 0.9876
        events: [1x1 struct]
      done_msg: 'Traced full continuation curve in 16 continuation steps'
```

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.[14]

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:

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

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

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

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

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

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

---

[14]Setting the RATE_A column (6) of branch to zero is the preferred way to indicate a completely unconstrained line.

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

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

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

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

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

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

6.3 Extended OPF Formulation

MATPOWER employs an extensible OPF structure [17] 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

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

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

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

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

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

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 $w_i = m_i u_i$.
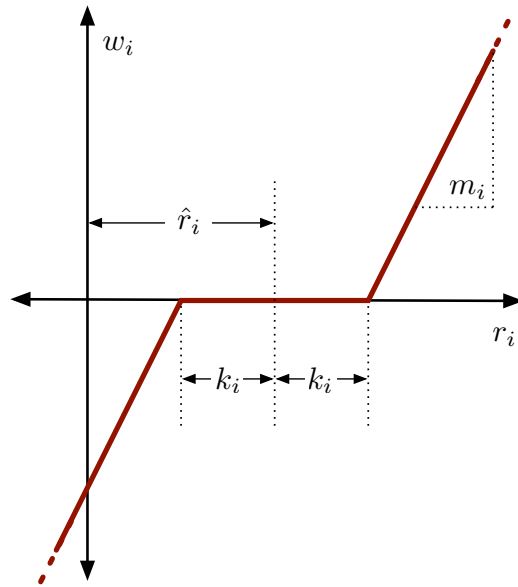
Figure 6-1: Relationship of w_i to r_i for $d_i = 1$ (linear option)
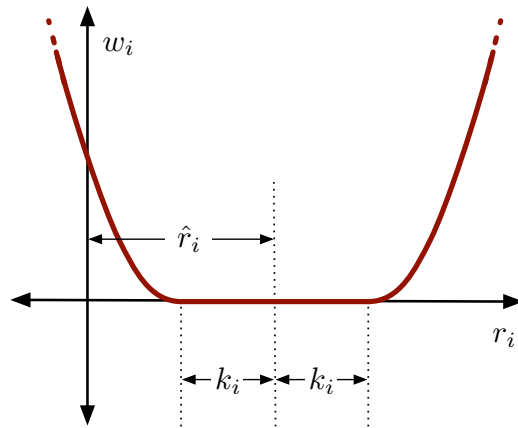
Figure 6-2: Relationship of w_i to r_i for $d_i = 2$ (quadratic option)

6.3.2 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.

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, MAT-POWER 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

$$
c(x) = \begin{cases}
m_1(x - x_1) + c_1, & x \le x_1 \\
m_2(x - x_2) + c_2, & x_1 < x \le x_2 \\
\quad\vdots & \quad\vdots \\
m_n(x - x_n) + c_n, & x_{n-1} < x
\end{cases} \tag{6.32}
$$

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

$$
m_j = \frac{c_j - c_{j-1}}{x_j - x_{j-1}}, \quad j = 1 \ldots n \tag{6.33}
$$

53
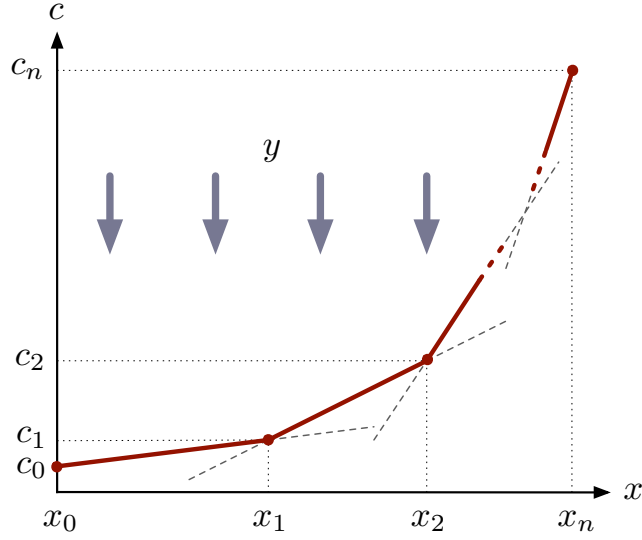
Figure 6-3: Constrained Cost Variable

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

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

$$y \ge m_j(x - x_j) + c_j, \quad j = 1 \ldots n. \tag{6.34}$$

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

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 [18], 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 [19].

Note that MATPOWER (in opf_setup) automatically converts any single-segment piecewise linear costs into polynomial (linear) form to avoid unnecessarily creating extra variables and constraints. It is this modified cost, rather than the original piecewise linear equivalent, that is returned in the gencost field of the results struct.

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.
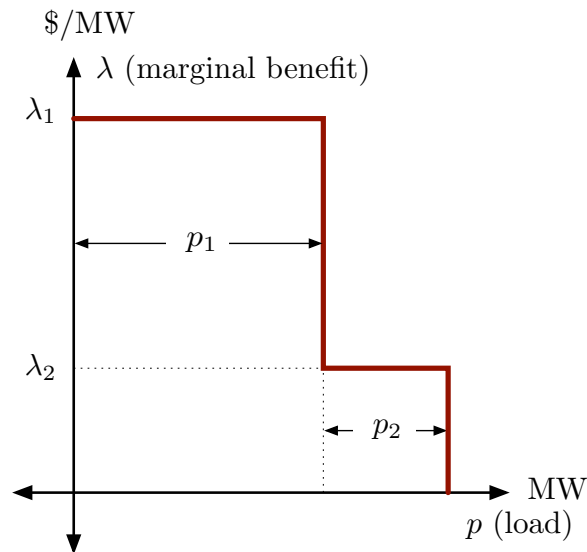
Figure 6-4: Marginal Benefit or Bid Function

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 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
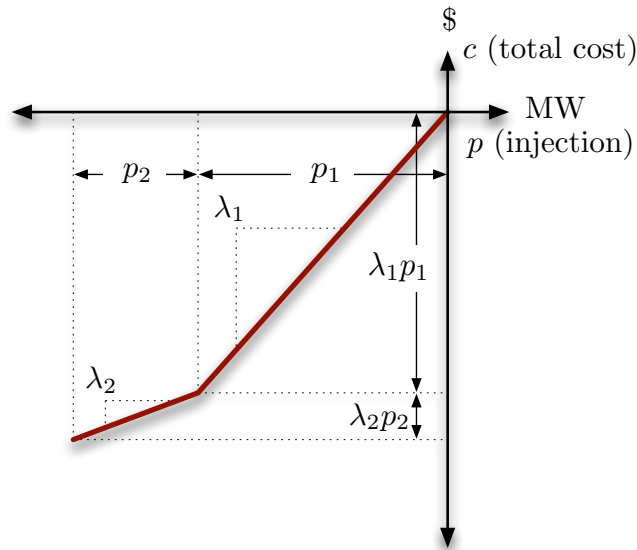
Figure 6-5: Total Cost Function for Negative Injection

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, MAT-POWER 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
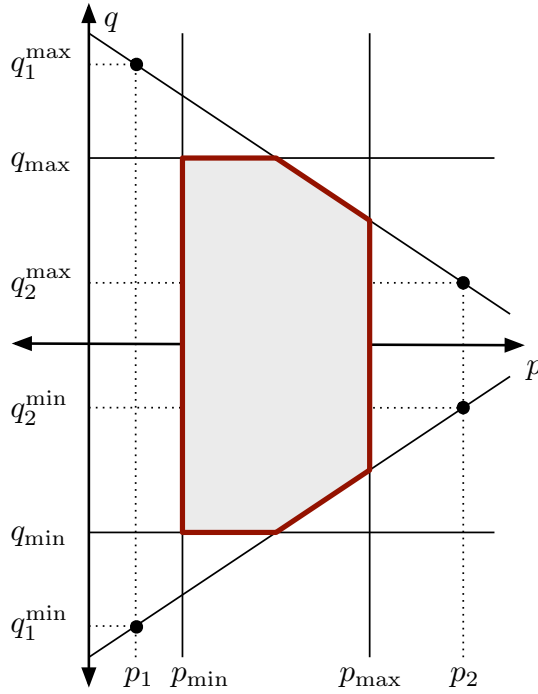
Figure 6-6: Generator P-Q Capability Curve

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 (12) and ANGMAX (13) of the branch matrix, MATPOWER creates the corresponding constraints on the voltage angle variables.[15]

---

[15]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.

6.5 Solvers

Early versions of Matpower relied on Matlab's Optimization Toolbox [20] 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 [22] of the high performance interior point BPMPD solver [23] for LP/QP problems. For the AC OPF problem, the MINOPF [24] and TSPOPF [18] packages provide solvers suitable for much larger systems. The former is based on MINOS [25] and the latter includes the primal-dual interior point and trust region based augmented Lagrangian methods described in [19]. Matpower version 4 and later also includes the option to use the open-source Ipopt solver[16] for solving both AC and DC OPFs, based on the Matlab MEX interface to Ipopt[17]. It also includes the option to use CPLEX[18] or MOSEK[19] for DC OPFs. Matpower 4.1 added the option to use KNITRO [41][20] for AC OPFs and the Gurobi Optimizer[21] for DC OPFs and Matpower 5 added GLPK and 5.1 added CLP. 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 [19]. 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 [26]. 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.

---

[16]Available from http://www.coin-or.org/projects/Ipopt.xml.

[17]See https://projects.coin-or.org/Ipopt/wiki/MatlabInterface.

[18]See http://www.ibm.com/software/integration/optimization/cplex-optimizer/.

[19]See http://www.mosek.com/.

[20]See http://www.ziena.com/.

[21]See http://www.gurobi.com/.

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, `runopf` 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 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.

For OPF problems, the preferred way to eliminate the flow limit on a branch is to set the `RATE_A` column (6) of the corresponding row in the `branch` matrix to zero. This indicates a completely unconstrained flow (as opposed to zero flow).

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`.

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.

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: 'GUROBI', 'CPLEX', 'MOSEK', 'OT', 'GLPK' *(linear costs only)*, 'BPMPD', 'MIPS'
		'MIPS' – MIPS, MATLAB Interior Point Solver, primal/dual interior point method[†]
		'BPMPD' – BPMPD[*]
		'CLP' – CLP[*]
		'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.use_vg	0	respect generator voltage setpoint, 0 or 1[†]
		0 – use voltage magnitude limits specified in `bus`, ignore `VG` in `gen`
		1 – replace voltage magnitude limits specified in `bus` by `VG` in corresponding `gen`
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_angle_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`)

[†] Using a value between 0 and 1 results in the limits being determined by the corresponding weighted average of the 2 options.
[‡] 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.

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[22], 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

---

[22]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.

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

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.

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

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

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

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);
```

66

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 $$\texttt{zones(k,j)} = \begin{cases} 1 & \text{if gen } j \text{ belongs to reserve zone } k \;\; (j \in Z_k) \\ 0 & \text{otherwise} \;\; (j \notin Z_k) \end{cases}$$
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.[23] 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

---

[23]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.
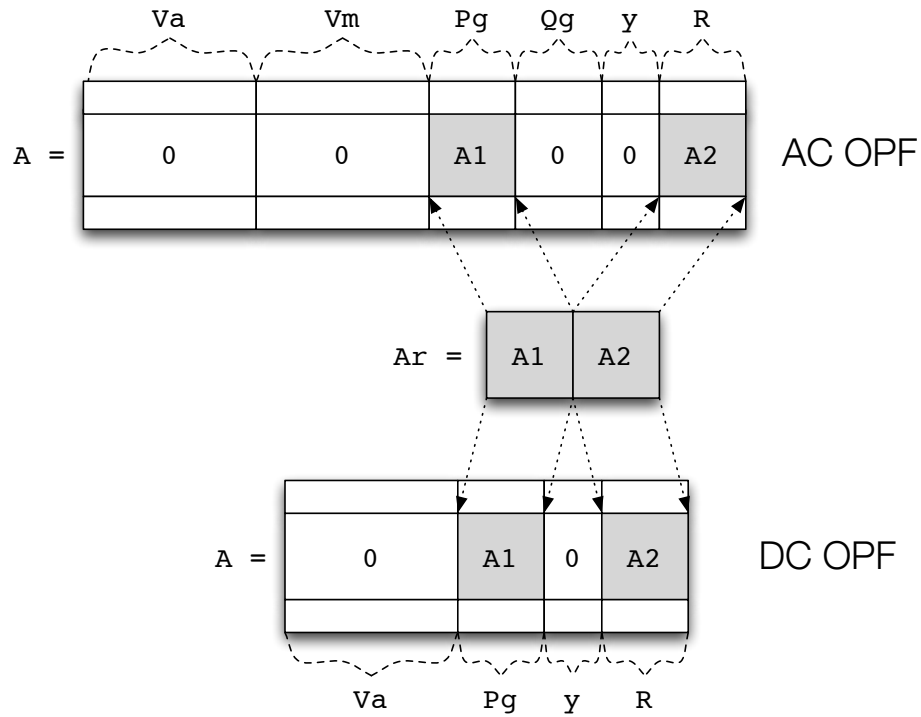
Figure 7-1: Adding Constraints Across Subsets of Variables

```
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, MAT-POWER 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

72

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
```

74

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[24] can be very useful for this purpose.

------

[24]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 Tables 7-4 and 7-5, respectively.

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

Table 7-4: Input Data Structures for Fixed Zonal 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 in \$/MW, c from (7.3)
`qty`	$n_g \times 1$ vector of reserve quantity upper bounds in MW, 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 in MW, k^{th} element is R_k from (7.5)

Table 7-5: Output Data Structures for Fixed Zonal Reserves

name	description
`results`	OPF results struct, superset of `mpc` with additional fields for output data
`reserves`	zonal reserve data, with results in the following sub-fields:[*]
`R`	reserve allocation r_i, in MW
`Rmin`	lower bound of r_i in MW from (7.2), i.e. all zeros
`Rmax`	upper bound of r_i in MW from (7.2), i.e. $\min(r_i^{\max}, \Delta_i)$
`mu`	shadow prices on constraints in \$/MW
`l`	shadow price on lower bound of r_i in (7.2)
`u`	shadow price on upper bound of r_i in (7.2)
`Pmax`	shadow price on capacity constraint (7.4)
`prc`	reserve price in \$/MW, computed for unit i as the sum of the shadow prices of constraints k in (7.5) in which unit i participates $(k \mid i \in Z_k)$

[*] All zonal reserve result sub-fields are $n_g \times 1$ vectors, where the values are set to zero for units not participating in the provision of reserves.

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 → "to" bus) in branch i and d_i is equal to 1 or −1 to indicate the direction,[25] 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_{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 7-6 and 7-7.

Table 7-6: 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.

---

[25]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 7-7: 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	$n_\mathrm{if} \times 1$ vector of actual flow in MW across the corresponding interface (as measured at the "from" end of associated branches)
mu.l	$n_\mathrm{if} \times 1$ vector of shadow prices on lower flow limits $(u/\mathrm{MW})^\dagger$
mu.u	$n_\mathrm{if} \times 1$ vector of shadow prices on upper flow limits $(u/\mathrm{MW})^\dagger$

† Here we assume the objective function has units u.

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.

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, 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.

"from" bus   DC Line   "to" bus

p_f p_t

$$p_{\mathrm{loss}} = l_0 + l_1 p_f$$

Figure 7-2: DC Line Model

"from" bus   "to" bus

p_f $p_t = (1 - l_1)p_f - l_0$
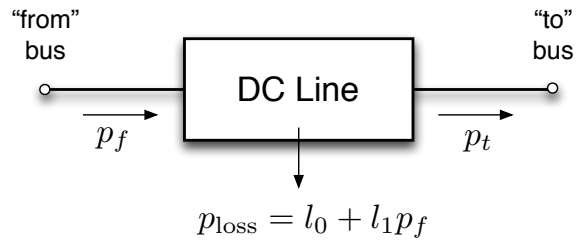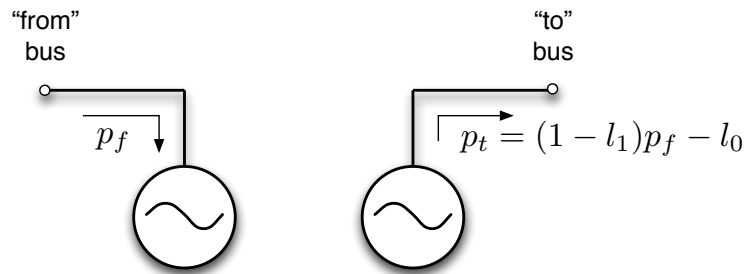
Figure 7-3: Equivalent "Dummy" Generators

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" $\rightarrow$ "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.[26]

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.

---

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

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);
```

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

$$f_v^i \geq 0 \tag{7.9}$$

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

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

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

$$f_u(x, z) = c_v^\mathsf{T} f_v. \tag{7.12}$$

The data for the problem is specfied 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 7-8 and 7-9.

Table 7-8: 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 7-9: 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.
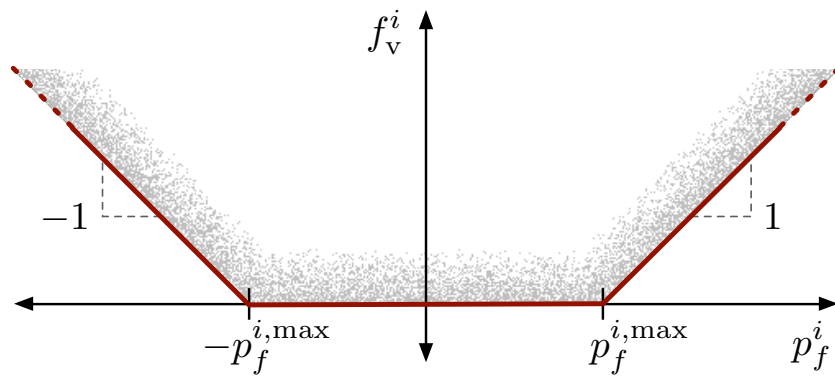
Figure 7-4: Feasible Region for Branch Flow Violation Constraints

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:

93

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

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

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

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 $P_{\text{min}} < 0$ and $P_{\text{max}} = 0$.

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(mpc)
Pd = total_load(mpc, load_zone, opt, mpopt)
Pd = total_load(bus)
Pd = total_load(bus, gen, load_zone, opt, mpopt)
[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`

```
mpc = scale_load(load, mpc)
mpc = scale_load(load, mpc, load_zone)
mpc = scale_load(load, mpc, load_zone, opt)
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.

9.3.5 `apply_changes`

```
mpc_modified = apply_changes(label, mpc_original, chgtab)
```

The `apply_changes` function implements a general mechanism to apply a set of changes to a base MATPOWER case. This can be used, for example, to define and apply a set of contingencies. There are three basic types of changes, those that replace old values with new ones, those that scale old values by some factor, and those that add a constant to existing values.

The *change table* matrix, `chgtab`, specifies modifications to be applied to an existing case. These modifications are grouped into sets, designated *change sets*, that are always applied as a group. A change set consists of one or more changes, each specified in a separate row in the `chgtab`, where the rows share a common `label` (integer ID).

For example, n_c change sets can be used to define n_c contingencies via a single `chgtab` with many rows, but only n_c unique labels. The `chgtab` also optionally specifies a probability π_k of occurance associated with change set k. Table 9-1 summarizes the meaning of the data in each column of `chgtab`. All of the names referenced in Tables 9-1 through 9-4 are defined as constants by the `idx_ct` function. Type `help idx_ct` at the MATLAB prompt for more details. Use of the named constants when constructing a `chgtab` matrix is encouraged to improve readability.

The value in the `CT_TABLE` column of `chgtab` defines which data table is to be modified and the options are given in Table 9-2. With the exception of load and certain generator cost changes, each individual change record specifies modification(s) to a single column of a particular data matrix, either `bus`, `gen`, `branch` or `gencost`. Some are changes to that column for an individual row in the matrix (or all rows, if the row index is set to 0), while others are area-wide changes that modify all rows corresponding to the specified area.[27]

Load changes are special and may modify multiple columns of the `bus` and/or `gen` tables. They offer a more flexible and convenient means of specifying modifications

---

[27] Areas are defined by the `BUS_AREA` column of the `bus` matrix.

Table 9-1: Columns of `chgtab`

name	column	description
CT_LABEL	1	change set label, unique for each change set (integer)
CT_PROB	2	change set probability (number between 0 and 1)[†]
CT_TABLE	3	type of table to be modified (see Table 9-2 for possible values)
CT_ROW	4	row index of data to be modified, 0 means all rows, for area-wide changes this is the area index, rather than row index
CT_COL	5	column index of data to be modified (see Table 9-4 for exceptions)
CT_CHGTYPE	6	type of change, e.g. replace, scale or add (see Table 9-3 for details)
CT_NEWVAL	7	new value used to replace or modify existing data

[†] The change set probability π_k is taken from this column of the *first* row for change set k.

to loads (fixed, dispatchable, real and/or reactive) than directly including individual change specifications for each of the corresponding entries in the `bus` and `gen` matrices. The row indices for load changes refer to bus numbers.

In addition to the normal direct modifications for generator cost parameters, there is also the option to scale or shift an entire cost function, either vertically or horizontally. This is often more convenient than manipulating the individual cost parameters directly, especially when dealing with a mix of polynomial and piecewise linear generator costs.

Table 9-2: Values for `CT_TABLE` Column

name	value	description
CT_TBUS	1	`bus` table
CT_TGEN	2	`gen` table
CT_TBRCH	3	`branch` table
CT_TAREABUS	4	area-wide change in `bus` table
CT_TAREAGEN	5	area-wide change in `gen` table
CT_TAREABRCH	6	area-wide change in `branch` table
CT_TLOAD	7	per bus load change[†]
CT_TAREALOAD	8	area-wide load change[†]
CT_TGENCOST	9	`gencost` table
CT_TAREAGENCOST	10	area-wide change in `gencost` table

[†] Preferred method of modifying load, as opposed to manipulating `bus` and `gen` tables directly.

Normally, the `CT_COL` column contains the column index of the entry or entries in the data tables to be modified. And the `CT_CHGTYPE` and `CT_NEWVAL` columns specify, respectively, the type of change (replacement, scaling or adding) and the correspond-

ing replacement value, scale factor or constant to add, as shown in Table 9-3.

Table 9-3: Values for `CT_CHGTYPE` Column

name	value	description
CT_REP	1	replace old value by new one in CT_NEWVAL column
CT_REL	2	scale old value by factor in CT_NEWVAL column
CT_ADD	3	add value in CT_NEWVAL column to old value

For load changes, the `CT_COL` column is not a column index, but rather a code that defines which loads at the specified bus(es) are to be modified, with the ability to select fixed loads only, dispatchable loads only or both, and for each whether or not to include the reactive load in the change. Similarly, the `CT_COL` column for generator cost modifications can be set to a special code to indicate a scaling or shifting of the entire corresponding cost function(s). The various options for the `CT_COL` column are summarized in Table 9-4.

Table 9-4: Values for `CT_COL` Column

name	value	description
for CT_TABLE *column* = CT_TLOAD *or* CT_TAREALOAD		
CT_LOAD_ALL_PQ	1	modify all (fixed & dispatchable) loads, active and reactive
CT_LOAD_FIX_PQ	2	modify fixed loads, active and reactive
CT_LOAD_DIS_PQ	3	modify dispatchable loads, active and reactive
CT_LOAD_ALL_P	4	modify all (fixed & dispatchable) loads, active power only
CT_LOAD_FIX_P	5	modify fixed loads, active power only
CT_LOAD_DIS_P	6	modify dispatchable loads, active power only
for CT_TABLE *column* = CT_TGENCOST *or* CT_TAREAGENCOST		
CT_MODCOST_F	-1	scales or shifts the cost function vertically[†]
CT_MODCOST_X	-2	scales or shifts the cost function horizontally[†]
otherwise[‡]		
	n	index of column in data matrix to be modified

[†] Use `CT_CHGTYPE` column = `CT_REL` to scale the cost and `CT_ADD` to shift the cost.
[‡] Can also be used for `CT_TGENCOST` or `CT_TAREAGENCOST` in addition to the special codes above.

For example, setting up a `chgtab` matrix for the following four scenarios could be done as shown below.

1. Turn off generator 2 (*10% probability*).

2. Reduce the line rating of all lines to 95% of their nominal values (*0.2% probability*).

3. Scale all loads in area 2 (real & reactive, fixed & dispatchable) up by 10% (*0.1% probability*).

4. Decrease capacity of generator 3 *and* shift its cost function to the left both by 10 MW (*5% probability*).

```
chgtab = [ ...
        1   0.1   CT_TGEN        2   GEN_STATUS      CT_REP    0;
        2   0.002 CT_TBRCH       0   RATE_A          CT_REL    0.95;
        3   0.001 CT_TAREALOAD   2   CT_LOAD_ALL_PQ  CT_REL    1.1;
        4   0.05  CT_TGEN        3   PMAX            CT_ADD   -10;
        4   0.05  CT_TGENCOST    3   CT_MODCOST_X    CT_ADD   -10;
];
```

A change table can be used to easily create modified cases from an existing base case with the `apply_changes` function. Given the `chgtab` from the example above, a new case with all lines derated by 5% can easily be created from an existing case `mpc` with the following line of code.

```
mpc_new = apply_changes(2, mpc, chgtab);
```

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

101

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(mpc, alg)
[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. Bus numbers must be consecutive beginning at 1 (i.e. internal ordering).

9.5.2 `makeBdc`

```
[Bbus, Bf, Pbusinj, Pfinj] = makeBdc(mpc)
[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). Bus numbers must be consecutive beginning at 1 (i.e. internal ordering).

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 `makeLODF`

```
PTDF = makePTDF(mpc)
LODF = makeLODF(mpc.branch, PTDF)
```

The `makeLODF` function forms the DC line outage distribution factor matrix for a given PTDF. The matrix is $n_{br} \times n_{br}$, where n_{br} is the number of branches. See Section 4.4 on Linear Shift Factors for more details.

9.5.5 `makePTDF`

```
H = makePTDF(mpc)
H = makePTDF(mpc, slack)
H = makePTDF(baseMVA, bus, branch)
H = makePTDF(baseMVA, bus, branch, slack)
```

The `makePTDF` function returns the DC PTDF matrix for a given choice of slack. The matrix is $n_{br} \times n_b$, where n_{br} is the number of branches and n_b is the number of buses. The `slack` can be a scalar (single slack bus) or an $n_b \times 1$ column vector of weights specifying the proportion of the slack taken up at each bus. If the `slack` is not specified, the reference bus is used by default. Bus numbers must be consecutive beginning at 1 (i.e. internal ordering). See Section 4.4 on Linear Shift Factors for more details.

9.5.6 `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 script is a convenience script that defines a set of useful constants, mostly 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. It also defines constants for the change tables used by apply_changes.

Specifically, it includes all of the constants defined by idx_bus, idx_brch, idx_gen, idx_cost and idx_ct.

9.6.2 feval_w_path

```
[y1, ..., yn] = feval_w_path(fpath, f, x1, ..., xn)
```

The feval_w_path function is identical to MATLAB's own feval, except that the function f need not be in the MATLAB path if it is defined in a file in the path specified by fpath. Assumes that the current working directory is always first in the MATLAB path.

9.6.3 have_fcn

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

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

- *empty* – 1 if optional functionality is available, 0 if not available

- `'vstr'` – version number as a string (e.g. `'3.11.4'`)

- `'vnum'` – version number as numeric value (e.g. 3.011004)

- `'date'` – release date as a string (e.g. `'20-Jan-2015'`)

- `'all'` – struct with fields named `av` (for "availability"), `vstr`, `vnum` and `date`, and values corresponding to each of the above, respectively.

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

- 0 – turn OFF the optional functionality

- 1 – turn ON the optional functionality (if available)

- −1 – toggle the ON/OFF state of the optional functionality

9.6.4 mpopt2qpopt

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

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

When the solver is set to `'DEFAULT'`, this function also selects the best available solver that is applicable[28] to the specific problem class, based on the following precedence: Gurobi, CPLEX, MOSEK, Optimization Toolbox, GLPK, BPMPD, MIPS.

---

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

9.6.5 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.

9.6.6 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 [19, 29].

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[29]

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

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`.

---

[29]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 [19]
`opt.rho_max`	1.05	upper bound on ρ_t corresponding to $1+\eta$ in Fig. 5 in [19]
`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[30] 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

$$x_1^2 - x_2^2 + x_3^2 - 2 \leq 0 \tag{A.9}$$
$$x_1^2 + x_2^2 + x_3^2 - 10 \leq 0. \tag{A.10}$$

First, create a MATLAB function to evaluate the objective function and its gradients,[31]

---

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

[31]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 1.2.1, 01-Jun-2016
 (using built-in linear solver)
 it    objective   step size   feascond    gradcond    compcond    costcond
 ----  ------------ ---------  ------------ ------------ ------------ ------------
  0           -1                       0         1.5           5            0
  1    -5.3250167    1.6875            0     0.894235    0.850653      2.16251
  2    -7.4708991    0.97413    0.129183   0.00936418    0.117278     0.339269
  3    -7.0553031    0.10406            0   0.00174933   0.0196518    0.0490616
  4    -7.0686267   0.034574            0   0.00041301   0.0030084   0.00165402
  5    -7.0706104  0.0065191            0  1.53531e-05 0.000337971 0.000245844
  6    -7.0710134 0.00062152            0  1.22094e-07 3.41308e-05 4.99387e-05
  7    -7.0710623 5.7217e-05            0  9.84879e-10 3.41587e-06 6.05875e-06
  8    -7.0710673 5.6761e-06            0  9.73527e-12 3.41615e-07 6.15483e-07
Converged!

f = -7.07107    exitflag = 1

x =
   1.58114
   2.23607
   1.58114

lambda.ineqnonlin =
   0
   0.707107
```

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:

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

subject to

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

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

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 [19, 29].

A.4.1   Notation

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

$$f_X = \frac{\partial f}{\partial X} = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \cdots & \frac{\partial f}{\partial x_n} \end{bmatrix}. \tag{A.14}$$

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

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

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

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

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

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

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

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

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:

$$\min_X f(X) \qquad (A.19)$$

subject to

$$G(X) = 0 \qquad (A.20)$$
$$H(X) \le 0 \qquad (A.21)$$

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:

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

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

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

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:

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

117

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 2nd row of (A.37) and solving for $\Delta\mu$ we get

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

Solving the 4th row of (A.37) for ΔZ yields

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

Then, substituting (A.38) and (A.39) into the 1$^{\text{st}}$ row of (A.37) results in

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

where

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

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

and

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

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

Combining (A.40) and the 3$^{\text{rd}}$ 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-17 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, $\begin{matrix} > 0 = \text{machine in-service} \\ \leq 0 = \text{machine out-of-service} \end{matrix}$
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.

[‡] Used to determine voltage setpoint for optimal power flow only if `opf.use_vg` option is non-zero (0 by default). Otherwise generator voltage range is determined by limits set for corresponding bus in `bus` matrix.

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), set to 0 for unlimited
RATE_B	7	MVA rating B (short term rating), set to 0 for unlimited
RATE_C	8	MVA rating C (emergency rating), set to 0 for unlimited
TAP	9	transformer off nominal turns ratio, (taps at "from" bus, impedance at "to" bus, i.e. if $r = x = b = 0$, $tap = \frac{|V_f|}{|V_t|}$)
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 $(\texttt{MODEL} = 1) \Rightarrow \quad p_0, f_0, p_1, f_1, \ldots, p_n, f_n$ 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), $\ldots$, (p_n, f_n) of the end/break-points of the piecewise linear cost $(\texttt{MODEL} = 2) \Rightarrow \quad c_n, \ldots, c_1, c_0$ $n + 1$ 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" $\rightarrow$ "to"
PT[†]	5	real power flow at "to" bus end (MW), "from" $\rightarrow$ "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
clp	*see Table C-8*	CLP options[*]
cplex	*see Table C-9*	CPLEX options[*]
fmincon	*see Table C-10*	fmincon options[†]
glpk	*see Table C-11*	GLPK options[*]
gurobi	*see Table C-12*	Gurobi options[*]
ipopt	*see Table C-13*	IPOPT options[*]
knitro	*see Table C-14*	KNITRO options[*]
minopf	*see Table C-15*	MINOPF options[*]
mosek	*see Table C-16*	MOSEK options[*]
pdipm	*see Table C-17*	PDIPM options[*]
tralm	*see Table C-18*	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.enforce_p_lims	0	enforce gen active power limits 0 — do *not* enforce limits 1 — enforce limits
cpf.enforce_q_lims	0	enforce gen reactive power limits at expense of V_m 0 — do *not* enforce limits 1 — enforce limits
cpf.step	0.05	default value for continuation power flow step size σ
cpf.step_min	10^{-4}	minimum allowed step size, σ_{min}
cpf.step_max	0.2	maximum allowed step size, σ_{max}
cpf.adapt_step	0	toggle adaptive step size feature 0 — adaptive step size disabled 1 — adaptive step size enabled
cpf.adapt_step_damping	0.7	damping factor β_{cpf} from (5.13) for adaptive step sizing
cpf.adapt_step_tol	10^{-3}	tolerance ϵ_{cpf} from (5.13) for adaptive step sizing
cpf.target_lam_tol	10^{-5}	tolerance for target lambda detection
cpf.nose_tol	10^{-5}	tolerance for nose point detection (p.u.)
cpf.p_lims_tol	10^{-2}	tolerance for generator active power limit detection (MW)
cpf.q_lims_tol	10^{-2}	tolerance for generator reactive power limit detection (MVAr)
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[†]

[†] 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: 'GUROBI', 'CPLEX', 'MOSEK', 'OT', 'GLPK' *(linear costs only)*, 'BPMPD', 'MIPS'
		'MIPS' – MIPS, MATLAB Interior Point Solver, primal/dual interior point method[†]
		'BPMPD' – BPMPD[*]
		'CLP' – CLP[*]
		'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.use_vg	0	respect generator voltage setpoint, 0 or 1[†]
		0 – use voltage magnitude limits specified in `bus`, ignore `VG` in `gen`
		1 – replace voltage magnitude limits specified in `bus` by `VG` in corresponding `gen`
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_angle_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`)

[†] Using a value between 0 and 1 results in the limits being determined by the corresponding weighted average of the 2 options.
[‡] 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.linsolver	''	linear system solver for update step '' or '\' – built-in backslash \ operator 'PARDISO' – PARDISO solver[†]
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

[†] Requires installation of the optional PARDISO package. See Appendix G.11 for details.
[‡] Only relevant when mips.step_control is on.

Table C-8: OPF Options for CLP[†]

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

[†] For opf.solver.dc option set to 'CLP' only. Requires the installation of the optional CLP package. See Appendix G.2 for details.
[‡] For details, see help clp_options or help clp.

Table C-9: 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.3 for details.

[‡] For details, see help cplex_options and the "Parameters of CPLEX" section of the CPLEX documentation at http://www.ibm.com/support/knowledgecenter/SSSA5P.

Table C-10: 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[*] $0 \Rightarrow$ 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-11: 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.4 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-12: 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.5 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/6.0/refman/parameters.html.

Table C-13: 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 [38]. See Appendix G.6 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/.

<div align="center">

Table C-14: OPF Options for KNITRO[†]

</div>

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 [41]. See Appendix G.7 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.

<div align="center">

Table C-15: OPF Options for MINOPF[†]

</div>

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 [24]. See Appendix G.8 for details.

[‡] Default values in parenthesis refer to defaults assigned in MEX file if called with option equal to 0.

Table C-16: 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 3 – primal simplex 4 – dual simplex 5 – primal dual simplex 6 – automatic simplex (MOSEK chooses which simplex) 7 – network primal 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.9 for details.

[§] Default values in parenthesis refer to defaults assigned by MOSEK if called with option equal to 0.

[*] The values listed here correspond to those used by MOSEK version 7.x. Version 6.x was different. It is probably safer to write your code using the symbolic constants defined by mosek_symbcon rather than using explicit numerical values.

[‡] For details, see help mosek_options and the "Parameters" reference in "The MOSEK optimization toolbox for MATLAB manual" at http://docs.mosek.com/7.1/toolbox/Parameters.html.

Table C-17: 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 [18]. See Appendix G.13 for details.
[‡] Only relevant when pdipm.step_control is on.

Table C-18: 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 [18]. See Appendix G.13 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-19. 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-19: Old-Style to New-Style Option Mapping

idx	old option			new option	notes
1	PF_ALG			pf.alg	new option has string values
		1	$\rightarrow$	'NR'	
		2	$\rightarrow$	'FDXB'	
		3	$\rightarrow$	'FDBX'	
		4	$\rightarrow$	'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	$\rightarrow$	'AC'	
		1	$\rightarrow$	'DC'	
11	OPF_ALG			opf.ac.solver	new option has string values
		0	$\rightarrow$	'DEFAULT'	
		500	$\rightarrow$	'MINOPF'	
		520	$\rightarrow$	'FMINCON'	
		540	$\rightarrow$	'PDIPM' (and `pdipm.step_control` = 0)	
		545	$\rightarrow$	'PDIPM' (and `pdipm.step_control` = 1)	
		550	$\rightarrow$	'TRALM'	
		560	$\rightarrow$	'MIPS' (and `mips.step_control` = 0)	
		565	$\rightarrow$	'MIPS' (and `mips.step_control` = 1)	
		580	$\rightarrow$	'IPOPT'	
		600	$\rightarrow$	'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

Table C-19: 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	

Table C-19: 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	

Table C-19: 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 distri-
bution, 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
MATPOWER-manual.pdf	MATPOWER 6.0b2 User's Manual
MOST-manual.pdf	MOST 1.0b2 User's Manual
TN1-OPF-Auctions.pdf	Tech Note 1 *Uniform Price Auctions and Optimal Power Flow* [33]
TN2-OPF-Derivatives.pdf	Tech Note 2 *AC Power Flows, Generalized OPF Costs and their Derivatives using Complex Matrix Notation* [26]
caseformat	help file documenting MATPOWER case format (i.e. at the command prompt, type `help caseformat` to see details)

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

146

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[†]
most	MOST, MATPOWER Optimal Scheduling Tool [‡]

[†] Uses AC model by default.
[‡] Simple wrapper function to set option to use DC model before calling the corresponding general function above.
[‡] MOST and its supporting files and functions in the most/ sub-directory are documented in the MOST User's Manual and listed in its Appendix A.

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_current_mpc	construct MATPOWER case struct for current continuation step
cpf_default_callback	callback function that accumulates, and optionally plots, results from each iteration
cpf_detect_events	detects event intervals and zeros given previous and current event function values
cpf_nose_event_cb	callback function to handle 'NOSE' events
cpf_nose_event	event function for 'NOSE' events, to detect nose point of continuation curve
cpf_p_jac	computes partial derivatives of parameterization function
cpf_p	computes value of parameterization function at current solution
cpf_plim_event_cb	callback function to handle 'PLIM' events
cpf_plim_event	event function for 'PLIM' events, to detect generator active power limit
cpf_predictor	computes the predicted solution from the current solution and tangent direction
cpf_qlim_event_cb	callback function to handle 'QLIM' events
cpf_qlim_event	event function for 'QLIM' events, to detect generator reactive power limit
cpf_register_callback	registers a CPF callback function to be called by runcpf
cpf_register_event	registers a CPF event function to be called by runcpf
cpf_tangent	computes the normalized tangent vector at the current solution
cpf_target_lam_event_cb	callback function to handle 'TARGET_LAM' events
cpf_target_lam_event	event function for 'TARGET_LAM' events, to detect end of continuation curve or other target λ value

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, primarily 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
clp_options	default options for CLP solver[†]
cplex_options	default options for CPLEX solver[†]
glpk_options	default options for GLPK solver[†]
gurobi_options	default options for Gurobi solver[†]
gurobiver	prints version information for Gurobi/Gurobi_MEX
ipopt_options	default options for IPOPT solver[†]
mips	MATLAB Interior Point Solver – primal/dual interior point solver for NLP
mipsver	prints version information for MIPS
miqps_matpower	Mixed-Integer Quadratic Program Solver for MATPOWER, wrapper function provides a common MIQP solver interface for various MIQP/MILP solvers
miqps_cplex	common MIQP/MILP solver interface to CPLEX (cplexmiqp and cplexmilp)[†]
miqps_glpk	common MILP solver interface to GLPK[†]
miqps_gurobi	common MIQP/MILP solver interface to Gurobi[†]
miqps_mosek	common MIQP/MILP solver interface to MOSEK (mosekopt)[†]
miqps_ot	common QP/MILP solver interface to MATLAB Opt Toolbox's intlinprog, quadprog, linprog
mosek_options	default options for MOSEK solver[†]
mosek_symbcon	symbolic constants to use for MOSEK solver options[†]
mplinsolve	common linear system solver interface, used by MIPS
mpopt2qpopt	create mi/qps_matpower options struct from MATPOWER options struct
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_clp	common QP/LP solver interface to CLP[†]
qps_cplex	common QP/LP solver interface to CPLEX (cplexqp and cplexlp)[†]
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
makeSdzip	forms a struct w/vectors of nominal values for each component of ZIP load
makeYbus	forms the complex bus and branch admittance matrices Y_{bus}, Y_f and Y_t

Table D-15: Utility Functions

name	description
apply_changes	modifies an existing MATPOWER case by applying changes defined in a *change table*
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
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
feval_w_path	same as MATLAB's feval function, except the function being evaluated need not be in the MATLAB path
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_ct	constant definitions for use with *change tables* and apply_changes
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
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
scale_load	scales fixed and/or dispatchable loads by load zone
total_load	returns vector of total load in each load zone

Table D-16: Other Functions

name	description
connected_components	returns the connected components of a graph
mpoption_info_clp	option information for CLP
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_intlinprog	option information for INTLINPROG
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
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

D.3   Example Matpower Cases

Table D-17: Small Test Cases

name	description
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
case24_ieee_rts	IEEE RTS 24-bus case
case30	30-bus case, based on IEEE 30-bus case
case_ieee30	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
case145	IEEE 145-bus case, 50 generator dynamic test case
case300	IEEE 300-bus case

Table D-18: Polish System Test Cases

name	description
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[*]

[*] Contributed by Roman Korab.

Table D-19: PEGASE European System Test Cases

name	description
case89pegase	89-bus portion of European transmission system from PEGASE project[*]
case1354pegase	1354-bus portion of European transmission system from PEGASE project[*]
case2869pegase	2869-bus portion of European transmission system from PEGASE project[*]
case9241pegase	9241-bus portion of European transmission system from PEGASE project[*]
case13659pegase	13659-bus portion of European transmission system from PEGASE project[*]

[*] Contributed by Cédric Josz and colleagues from the French Transmission System Operator. Please cite references [30, 31] when publishing results based on this data.

Table D-20: RTE French System Test Cases

name	description
case1888rte	1888-bus snapshot of VHV French transmission system from RTE[*]
case1951rte	1951-bus snapshot of VHV French transmission system from RTE[*]
case2848rte	2848-bus snapshot of VHV French transmission system from RTE[*]
case2868rte	2868-bus snapshot of VHV French transmission system from RTE[*]
case6468rte	6468-bus snapshot of VHV and HV French transmission system from RTE[*]
case6470rte	6470-bus snapshot of VHV and HV French transmission system from RTE[*]
case6495rte	6495-bus snapshot of VHV and HV French transmission system from RTE[*]
case6515rte	6515-bus snapshot of VHV and HV French transmission system from RTE[*]

[*] Contributed by Cédric Josz and colleagues from the French Transmission System Operator. Please cite reference [30] when publishing results based on this data.

D.4 Automated Test Suite

Table D-21: 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-22: 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_vg.mat	solution data, AC OPF of t_case9_opf with option opf.use_vg
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-23: Miscellaneous MATPOWER Tests

name	description
t/	
test_matpower	runs full MATPOWER test suite
test_most‡	runs full MOST test suite
t_apply_changes	runs tests for `apply_changes`
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_cpf_cb1	example CPF callback function for `t_cpf`
t_cpf_cb2	example CPF callback function with `cb_args` for `t_cpf`
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_miqps_matpower	runs tests for `miqps_matpower`
t_modcost	runs tests for `modcost`
t_mplinsolve	tests for `mplinsolve`
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_vdep_load	runs PF, CPF and OPF tests for voltage dependent (ZIP) loads
t_totcost	runs tests for `totcost`

† Requires the installation of an optional package. See Appendix G for details on the corresponding package.
‡ While test_most is listed here with other tests, it is actually located in *<MATPOWER>*/most/t, not *<MATPOWER>*/t. MOST and its supporting files and functions in the **most/** sub-directory are documented in the MOST User's Manual and listed in its Appendix A.

Table D-24: MATPOWER OPF Tests

name	description
t/	
t_opf_dc_bpmpd	runs tests for DC OPF solver using BPMPD_MEX[†]
t_opf_dc_clp	runs tests for DC OPF solver using CLP[†]
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.

maxloadlim
: An optimal power flow extension, contributed by Camille Hamon, for computing maximum loadability limits.[32]   Please see extras/maxloadlim/manual/maxloadlim_manual.pdf for details on the formulation and implementation.

misc
: A number of potentially useful functions that are either not yet fully implemented, tested, documented and/or supported. See the help (and the code) in each individual file to understand what it does.[33]

reduction
: A network reduction toolbox that performs a modifed Ward reduction and can be used to produce a smaller approximate equivalent from a larger original system. Code contributed by Yujia Zhu and Daniel Tylavsky. For more details, please see the Network_Reduction_Toolbox.pdf file.

sdp_pf
: Applications of a semidefinite programming programming relaxation of the power flow equations. Code contributed by Dan Molzahn. See Appendix G.12 and the documentation 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.

---

[32]Camille Hamon maintains a GitHub repository of this code at:
https://github.com/CamilleH/Max-Load-Lim-matpower

[33]For more information on qcqp_opf, see [30]. For more information on plot_mpc, see [32].

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 [24] in the context of POWERWEB[34] 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 55). 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).

---

[34]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 $\lambda_P^{\mathrm{ref}}/\lambda_P^i$, 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

166

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.[35] The relationships between the OPF results and the pricing rules of the various uniform price auctions are described in detail in [33].

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.

---

[35]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 [22, 23] is a MATLAB MEX interface to BPMPD, an interior point solver for quadratic programming developed by Csaba Mészáros at the MTA SZ-TAKI, Computer and Automation Research Institute, Hungarian Academy of Sciences, Budapest, Hungary. It can be used by MATPOWER's DC and LP-based OPF solvers and it improves the robustness of MINOPF. It is also useful outside of MAT-POWER 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   CLP – COIN-OR Linear Programming

The CLP [34] (**C**OIN-OR **L**inear **P**rogramming) solver is an open-source linear programming solver written in C++ by John Forrest. It can solve both linear programming (LP) and quadratic programming (QP) problems. It is primarily meant to be used as a callable library, but a basic, stand-alone executable version exists as well. It is available from the COIN-OR initiative at http://www.coin-or.org/projects/Clp.xml.

To use CLP with MATPOWER, a MEX interface is required[36]. For Microsoft Windows users, a pre-compiled MEX version of CLP (and numerous other solvers, such as GLPK and IPOPT) are easily installable as part of the OPTI Toolbox[37] [35]

With the MATLAB interface to CLP installed, it can be used to solve DC OPF problems by setting the `opf.solver.dc` option equal to `'CLP'`. The solution algorithms and other CLP parameters can be set directly via MATPOWER's `clp.opts` option. A "CLP user options" function can also be specified via `clp.opt_fname` to override the defaults for any of the many CLP parameters. See `help clp` for details. See Table C-8 for a summary of the CLP-related MATPOWER options.

CLP 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 `'CLP'`, or by calling `qps_clp` directly.

G.3 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-9 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

---

[36]According to David Gleich at http://web.stanford.edu/~dgleich/notebook/2009/03/coinor_clop_for_matlab.html, there was a MATLAB MEX interface to CLP written by Johan Lofberg and available (at some point in the past) at http://control.ee.ethz.ch/~joloef/mexclp.zip. Unfortunately, at the time of this writing, it seems it is no longer available there, but Davide Barcelli makes some precompiled MEX files for some platforms available here http://www.dii.unisi.it/~barcelli/software.php, and the ZIP file linked as Clp 1.14.3 contains the MEX source as well as a `clp.m` wrapper function with some rudimentary documentation.

[37]The OPTI Toolbox is available from http://www.i2c2.aut.ac.nz/Wiki/OPTI/.

CPLEX" section of the CPLEX documentation at http://www.ibm.com/support/knowledgecenter/SSSA5P for details.

It can also be used to solve general LP and QP problems via MATPOWER's common QP solver interface `qps_matpower`, or MILP and MIQP problems via `miqps_matpower`, with the algorithm option set to `'CPLEX'` (or 500 for backward compatibility), or by calling `qps_cplex` or `miqps_cplex` directly.

G.4   GLPK – GNU Linear Programming Kit

The GLPK [36] (**G**NU **L**inear **P**rogramming **K**it) package is intended for solving large-scale linear programming (LP), mixed integer programming (MIP), and other related problems. It is a set of routines written in ANSI C and organized in the form of a callable library.

To use GLPK with MATPOWER, a MEX interface is required[38]. For Microsoft Windows users, a pre-compiled MEX version of GLPK (and numerous other solvers, such as CLP and IPOPT) are easily installable as part of the OPTI Toolbox[39] [35].

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-11 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`, or MILP problems via `miqps_matpower`, with the algorithm option set to `'GLPK'`, or by calling `qps_glpk` or `miqps_glpk` directly.

G.5   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,

---

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

[39]The OPTI Toolbox is available from http://www.i2c2.aut.ac.nz/Wiki/OPTI/.

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.[40]

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-12 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/6.0/refman/parameters.html for details.

It can also be used to solve general LP and QP problems via Matpower's common QP solver interface `qps_matpower`, or MILP and MIQP problems via `miqps_matpower`, with the algorithm option set to `'GUROBI'` (or 700 for backward compatibility), or by calling `qps_gurobi` or `miqps_gurobi` directly.

G.6   Ipopt – Interior Point Optimizer

Ipopt [38] (Interior Point OPTimizer, 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 http://www.coin-or.org/projects/Ipopt.xml. 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 lists for help in building and installing the Ipopt Matlab interface. This interface uses callbacks to Matlab functions to evaluate the objective function and its gradient, the constraint values and Jacobian, and the Hessian of the Lagrangian.

---

[40]Matpower version 4.1 supported Gurobi version 4.x, which required a free third-party Matlab MEX interface [37], available at http://www.convexoptimization.com/wikimization/index.php/Gurobi_mex.

Precompiled MEX binaries for a high-performance version of IPOPT, using the PARDISO linear solver [39, 40], are available from the PARDISO project[41]. At the time of this writing, these are the highest performing solvers available to MATPOWER for very large scale AC OPF problems. For Microsoft Windows users, a pre-compiled MEX version of IPOPT (and numerous other solvers, such as CLP and GLPK) are easily installable as part of the OPTI Toolbox[42] [35].

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-13 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.7 KNITRO – Non-Linear Programming Solver

KNITRO [41] is a general purpose optimization solver specializing in nonlinear problems, available from Ziena Optimization, LLC. As of version 9, KNITRO includes a native MATLAB interface, `knitromatlab`[43]. 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, `knitromatlab` or `ktrlink`, can be used by MATPOWER to solve AC OPF problems by simply setting the `opf.solver.ac` option to `'KNITRO'`. See Table C-14 for a summary of KNITRO-related MATPOWER options. The `knitromatlab` 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 op-

---

[41]See http://www.pardiso-project.org/ for the download links.

[42]The OPTI Toolbox is available from http://www.i2c2.aut.ac.nz/Wiki/OPTI/.

[43]Earlier versions required the MATLAB Optimization Toolbox from The MathWorks, which includes an interface to the KNITRO libraries called `ktrlink`, but the libraries themselves still had to be acquired directly from Ziena Optimization, LLC.

tions 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 MAT-POWER'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.8 MINOPF – AC OPF Solver Based on MINOS

MINOPF [24] is a MINOS-based optimal power flow solver for use with MATPOWER. It is for educational and research use only. MINOS [25] 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.9 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-16 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/7.1/toolbox/Parameters.html. You may also find it helpful to use the symbolic constants defined by `mosek_symbcon`.

It can also be used to solve general LP and QP problems via Matpower's common QP solver interface `qps_matpower`, or MILP and MIQP problems via `miqps_matpower`, with the algorithm option set to `'MOSEK'` (or 600 for backward compatibility), or by calling `qps_mosek` or `miqps_mosek` directly.

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

Matlab's Optimization Toolbox [20], available from The MathWorks, provides a number of high-performance solvers that Matpower can take advantage of.

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

When available, the Optimization Toolbox solvers can be used to solve AC or DC OPF problems by setting the `opf.solver.ac` or `opf.solver.dc` options, respectively, equal to `'OT'`.

The solution algorithm used by `fmincon` for NLP problems can be controlled by Matpower's `fmincon.alg` option. See Table C-10 for other Matpower options related to `fmincon`. For `linprog`, `quadprog` and `intlingprog`, the corresponding Matpower option can be used to pass in native Optimization Toolbox options directly to the solver. For example, to set the LP solver to use a dual simplex method, simply set Matpower's `'linprog.Algorithm'` option to `'dual-simplex'`. For details on the full set of Optimization Toolbox options, please refer to their documentation.[44]

The Optimization Toolbox can also be used to solve general LP and QP problems via Matpower's common QP solver interface `qps_matpower`, or MILP problems via `miqps_matpower`, with the algorithm option set to `'OT'`, or by calling `qps_ot` or `miqps_ot` directly.

G.11   PARDISO – Parallel Sparse Direct and Multi-Recursive Iterative Linear Solvers

The PARDISO package is a thread-safe, high-performance, robust, memory efficient and easy to use software for solving large sparse symmetric and non-symmetric linear systems of equations on shared-memory and distributed-memory multiprocessor systems [39, 40]. More information is available at http://www.pardiso-project.org.

---

[44]See http://www.mathworks.com/help/optim/ and [21].

When the MATLAB interface to PARDISO is installed, PARDISO's solvers can be used to replace the built-in \ operator for solving for the Newton update step in MATPOWER's default primal-dual interior point solver (MIPS) by setting the `mips.linsolver` option equal to `'PARDISO'`. The `mplinsolve` function can also be called directly to solve $Ax = b$ problems via PARDISO or the built-in solver, depending on the arguments supplied. This interface also gives access to the full range of PARDISO's options. For details, see `help mplinsolve` and the PARDISO User's Manual at http://www.pardiso-project.org/manual/manual.pdf.

When solving very large AC optimal power flow problems with MIPS, selecting PARDISO as the linear solver can often dramtically improve both computation time and memory use.

Also note that precompiled MEX binaries for a high-performance version of IPOPT, using the PARDISO linear solver, are available. Refer to Section G.6 for more details.

G.12 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 available in *<MATPOWER>*/`extras/sdp_pf/documentation` directory, and especially in the file `sdp_pf_documentation.pdf`.

G.13 TSPOPF – Three AC OPF Solvers by H. Wang

TSPOPF [18] 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 [19, 29]. 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.

180

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/.

Appendix H Release History

The full release history can be found in `docs/CHANGES` or `docs/CHANGES.txt` or online at `http://www.pserc.cornell.edu/matpower/CHANGES.txt`.

H.1 Version 6.0b2 – released Nov 1, 2016

The MATPOWER 6.0b2 User's Manual is available online.[45]

New Features

- Redesigned CPF callback mechanism to handle CPF events such as generator limits, nose point detection, etc. Included event log in CPF results.

- Added options `'cpf.enforce_p_lims'` and `'cpf.enforce_q_lims'` to enforce generator active and reactive power limts in the continuation power flow.

- Added OPF option `'opf.use_vg'` to provide a convenient way to have the OPF respect the generator voltage setpoints specified in the `gen` matrix.

Other Improvements

- Added option `'cpf.adapt_step_damping'` to control oscillations in adaptive step size control for continuation power flow.

- Added CPF user options for setting tolerances for target lambda detection and nose point detection, `'cpf.target_lam_tol'` and `'cpf.nose_tol'`, respectively.

- Added support for MATLAB Optimization Toolbox 7.5 (R2016b).

- Added support for MOSEK v8.x.

- Added tests for power flow with `'pf.enforce_q_lims'` option.

- Updated network reduction code to handle cases with radially connected external buses.

- Updated versions of `qcqp_opf()` and `qcqp_opf()` in `extras/misc`, *from Cédric Josz.*

---

[45] `http://www.pserc.cornell.edu/matpower/docs/MATPOWER-manual-6.0b2.pdf`

Bugs Fixed

- Fixed shadow prices on variable bounds for AC OPF for `fmincon`, IPOPT, and KNITRO.

- In `savecase()` single quotes are now escaped properly in bus names.

- Generator capability curve parameters that define a zero-reactive power line no longer cause a fatal error.

- Bad bus numbers no longer cause a fatal error (after reporting the bad bus numbers) in `case_info()`.

Incompatible Changes

- Removed `'cpf.user_callback_args'` option and modified `'cpf.user_callback'`.

- Changed name of `'cpf.error_tol'` option to `'cpf.adapt_step_tol'`.

H.2   Version 6.0b1 – released Jun 1, 2016

The MATPOWER 6.0b1 User's Manual is available online.[46]

New Case Files

- Added 9 new case files, 8 cases ranging from 1888 to 6515 buses representing the French system, and a 13,659-bus case representing parts of the of the European high voltage transmission network, stemming from the Pan European Grid Advanced Simulation and State Estimation (PEGASE) project. *Thanks again to Cédric Josz and colleagues from the French Transmission System Operator.* Please cite reference [30] when publishing results based on these cases.

- Added `case145.m`, IEEE 145 bus, 50 generator dynamic test case from the U of WA Power Systems Test Case Archive[47].

---

[46]http://www.pserc.cornell.edu/matpower/docs/MATPOWER-manual-6.0b1.pdf
[47]http://www.ee.washington.edu/research/pstca/dyn50/pg_tcadd50.htm

New Features

- MATPOWER Optimal Scheduling Tool (MOST) 1.0b1 is a **major new feature**, implementing a full range of optimal power scheduling problems, from a simple as a deterministic, single period economic dispatch problem with no transmission constraints to as complex as a stochastic, security-constrained, combined unit-commitment and multiperiod OPF problem with locational contingency and load-following reserves, ramping costs and constraints, deferrable demands, lossy storage resources and uncertain renewable generation. See `docs/MOST-manual.pdf` for details.

- General mechanism for applying modifications to an existing MATPOWER case. See `apply_changes()` and `idx_ct()`.

- Experimental foundation for handling of ZIP load models in power flow (Newton, fast-decoupled only), continuation power flow, and optimal power flow (MIPS, `fmincon`, KNITRO, IPOPT solvers only). Currently, ZIP loads can only be specified on a system-wide basis using the experimental options `'exp.sys_wide_zip_loads.pw'` and `'exp.sys_wide_zip_loads.qw'`.

- Support for `quadprog()` under GNU Octave.

- New contributed extras:

  - Plot electrically meaningful drawings of a MATPOWER case using `plot_mpc()` in `extras/misc`, contributed by Paul Cuffe.
  - Find the maximum loadability limit of a system via an optimal power flow and dispatchable loads, using `maxloadlim()` in `extras/maxloadlim`, contributed by Camille Hamon.
  - Create a quadratically-constrained quadratic programming (QCQP) representation of the AC power flow problem using using `qcqp_opf()` in `extras/misc`, contributed by Cédric Josz and colleagues.

- New functions:

  - `apply_changes()` and `idx_ct()` provide a general mechanism for applying modifications to an existing MATPOWER case.
  - `feval_w_path()` evaluates a function located at a specified path, outside of the Matlab path.

– `mpopt2qpopt()` provides a common interface for creating options struct for `mi/qps_matpower()` from a MATPOWER options struct.

- New function options:

  – Option to call `makeB()`, `makeBdc()`, `makePTDF()`, `scale_load()`, and `total_load()` with full case struct (`mpc`) instead of individual data matrices (`bus`, `branch`, etc.).

  – `total_load()`, which now computes voltage-dependent load values, accepts the values `'bus'` and `'area'` as valid values for `'load_zone'` argument.

Other Improvements

- Changed default solver order for LP, QP, MILP, MIQP problems to move Gurobi before CPLEX and BPMPD after Optimization Toolbox and GLPK.

- Added some caching to `mpoption()` and made minor changes to `nested_struct_copy()` to greatly decrease the overhead added by `mpoption()` when running many small problems.

- Added "Release History" section to Appendix of manual.

- Many new tests.

Bugs Fixed

- Fixed bug in `toggle_dclines()` that resulted in fatal error when used with OPF with reactive power costs. Thanks to Irina Boiarchuk.

- Fixed fatal bug in `update_mupq()` affecting cases where QMIN is greater than or equal to QC1MIN and QC2MIN (or QMAX is less than or equal to QC1MAX and QC2MAX) for all generators. Thanks Jose Miguel.

- Copying a field containing a struct to a non-struct field with `nested_struct_copy()` now overwrites rather than causing a fatal error.

- Fixed a bug in `psse_convert_xfmr()` where conversion of data for transformers with CZ=3 was done incorrectly. Thanks to Jose Marin and Yujia Zhu.

- Fixed a fatal bug in `psse_convert_xfmr()` affecting transformers with CW and/or CZ equal to 1. Thanks to Matthias Resch.

- Fixed a crash in `have_fcn()` caused by changes in OPTI Toolbox v2.15 (or possibly v2.12)

- Commented out isolated bus 10287 in `case3375wp.m`.

- Added code to DC OPF to return `success = 0` for cases where the matrix is singular (e.g. islanded system without slack).

- Fixed problem in `have_fcn()` where SeDuMi was turning off and leaving off all warnings.

Incompatible Change

- Removed `fairmax()` from the public interface by moving it inside `uopf()`, the only place it was used.

H.3   Version 5.1 – released Mar 20, 2015

The MATPOWER 5.1 User's Manual is available online.[48]

New License

- Switched to the more permissive 3-clause BSD license from the previously used GNU General Public License (GPL) v3.0.

New Case Files

- Added four new case files, ranging from 89 up to 9421 buses, representing parts of the European high voltage transmission network, stemming from the Pan European Grid Advanced Simulation and State Estimation (PEGASE) project. *Thanks to Cédric Josz and colleagues from the French Transmission System Operator.*

New Documentation

- Added an online function reference to the web site at http://www.pserc.cornell.edu/matpower/docs/ref/.

---

[48]http://www.pserc.cornell.edu/matpower/docs/MATPOWER-manual-5.1.pdf

New Features

- Added support for using PARDISO (http://www.pardiso-project.org/) as linear solver for computing interior-point update steps in MIPS, resulting in dramatic improvements in computation time and memory use for very large-scale problems.

- Added support for LP/QP solver CLP (COIN_OR Linear Programming, http://www.coin-or.org/projects/Clp.xml). Use `opf.dc.solver` option `'CLP'` or `qps_clp()`.

- Added support for OPTI Toolbox (http://www.i2c2.aut.ac.nz/Wiki/OPTI/) versions of CLP, GLPK and IPOPT solvers, providing a very simple installation path for some free high-performance solvers on Windows platforms.

- Network reduction toolbox for creating smaller approximate network equivalents from a larger original case file. *Contributed by Yujia Zhu and Daniel Tylavsky.*

- Added unified interface to various solvers for mixed-integer linear and quadratic programming (MILP/MIQP) problems.

- Major update to `have_fcn()`, which now determines and caches version numbers and release dates for optional packages, and includes ability to toggle the availability of optional functionality.

- New and updated support for 3rd party solvers:

  - High-performance IPOPT-PARDISO solver builds from the PARDISO Project http://www.pardiso-project.org (at time of release this is MATPOWER's highest performing solver for very large scale AC OPF problems)
  - OPTI Toolbox versions of CLP, GLPK, IPOPT
  - CLP
  - Gurobi 6.x
  - KNITRO 9.1
  - MOSEK 7.1
  - Optimization Toolbox 7.2
    * dual-simplex algorithm for `linprog()`
    * `intlinprog()` for MILP

- New functions:

  - `mplinsolve()` provides unified interface for linear system solvers, including PARDISO and built-in backslash operator
  - `miqps_matpower()` provides unified interface to multiple MILP/MIQP solvers.
  - `miqps_clex()` provides a unified MILP/MIQP interface to CPLEX.
  - `miqps_glpk()` provides a unified MILP interface to GLPK.
  - `miqps_gurobi()` provides a unified MILP/MIQP interface to Gurobi.
  - `miqps_mosek()` provides a unified MILP/MIQP interface to MOSEK.
  - `miqps_ot()` provides a unified MILP interface to `intlingprog()`.
  - `mosek_symbcon()` defines symbolic constants for setting MOSEK options.

Other Improvements

- Cleaned up and improved consistency of output in `printpf()` for generation and dispatchable load constraints.

- Modified `runcpf()` to gracefully handle the case when the base and target cases are identical (as opposed to getting lost in an infinite loop).

- Optional generator and dispatchable load sections in pretty-printed output now include off-line units.

Bugs Fixed

- Fixed fatal bug in `case_info()` for islands with no generation.

- Fixed fatal bug in `toggle_dcline()` when pretty-printing results. *Thanks to Deep Kiran for reporting.*

- Fixed sign error on multipliers on lower bound on constraints in `qps_clp()` and `qps_glpk()`.

- Fixed bug in handling of interface flow limits, where multipliers on binding interface flow limits were off by a factor of the p.u. MVA base.

- Fixed minor bug with `poly2pwl()`, affecting units with PMAX ≤ 0.

- Fixed error in `qps_mosek()` in printout of selected optimizer when using MOSEK 7.

- Fixed bug in `hasPQcap()` that resulted in ignoring generator capability curves for units whose reactive range increases as real power output increases. *Thanks to Irina Boiarchuk for reporting.*

- Fixed several incompatibilities with MATLAB versions < 7.3.

H.4   Version 5.0 – released Dec 17, 2014

The MATPOWER 5.0 User's Manual is available online.[49]

New Features

- Continuation power flow with tangent predictor and Newton method corrector, based on code contributed by Shrirang Abhyankar and Alex Flueck.

- `SDP_PF`, a set of applications of a semidefinite programming relaxation of the power flow equations, contributed by Dan Molzahn (see `extras/sdp_pf`):

    - Globally optimal AC OPF solver (under certain conditions).
    - Functions to check sufficient conditions for:
        * global optimality of OPF solution
        * insolvability of power flow equations

- PSS/E RAW data conversion to MATPOWER case format (experimental) based on code contributed by Yujia Zhu.

- Brand new extensible MATPOWER options architecture based on options struct instead of options vector.

- Utility routines to check network connectivity and handle islands and isolated buses.

- New extension implementing DC OPF branch flow soft limits. See `help toggle_softlims` for details.

- New and updated support for 3rd party solvers:

    - CPLEX 12.6
    - GLPK

---

[49]http://www.pserc.cornell.edu/matpower/docs/MATPOWER-manual-5.0.pdf

- Gurobi 5.x
- Ipopt 3.11.x
- KNITRO 9.x.x
- Optimization Toolbox 7.1

- Numerous performance enhancements.

- New functions:

  - `runcpf()` for continuation power flow.
  - `case_info()` for summarizing system information, including network connectivity.
  - `extract_islands()` to extract a network island into a separate MATPOWER case.
  - `find_islands()` to detect network islands.
  - `@opt_model/describe_idx()` to identify variable, constraint or cost row indices to aid in debugging.
  - `margcost()` for computing the marginal cost of generation.
  - `psse2mpc()` to convert PSS/E RAW date into MATPOWER case format.
  - `get_losses()` to compute branch series losses and reactive charging injections and derivatives as functions of bus voltages.
  - New experimental functions in `extras/misc` for computing loss factors, checking feasibility of solutions, converting losses to negative bus injections and modifying an OPF problem to make it feasible.

- Added `case5.m`, a 5-bus, 5-generator example case from Rui Bo.

- New options:

  - `scale_load()` can scale corresponding gencost for dispatchable loads.
  - `makeJac()` can return full Jacobian instead of reduced version used in Newton power flow updates.
  - `modcost()` can accept a vector of shift/scale factors.
  - `total_load()` can return actual or nominal values for dispatchable loads.
  - `runpf()`, `runopf()`, etc. can send pretty-printed output to file without also sending it to the screen.

190

- **out.suppress_detail** option suppresses all output except system summary (on by default for large cases).
- **opf.init_from_mpc** option forces some solvers to use user-supplied starting point.
- MIPS 1.1 includes many new user-settable options.

- Reimplementated `@opf_model` class as sub-class of the new `@opt_model` class, which supports indexed named sets of variables, constraints and costs.

- Many new tests in test suite.

Bugs Fixed

- Running a power flow for a case with DC lines but no `gencost` no longer causes an error.

- Fixed a bug in `runpf()` where it was using the wrong initial voltage magnitude for generator buses marked as PQ buses. Power flow of solved case was not converging in zero iterations as expected.

- Fixed fatal bug in MIPS for unconstrained, scalar problems. *Thanks to Han Na Gwon.*

- Fixed a bug in `int2ext()` where converting a case to internal ordering before calling `runpf()` or `runopf()` could result in a fatal error due to mismatched number of columns in internal and external versions of data matrices. *Thanks to Nasiruzzaman and Shiyang Li for reporting and detailing the issue.*

- DC OPF now correctly sets voltage magnitudes to 1 p.u. in `results`.

- Fixed a bug in MIPS where a near-singular matrix could produce an extremely large Newton step, resulting in incorrectly satisfying the relative feasibility criterion for successful termination.

- Improved the starting point created for IPOPT, KNITRO and MIPS for variables that are only bounded on one side.

- Fixed bug in `savecase()` where the function name mistakenly included the path when the `fname` input included a path.

- Fixed bugs in `runpf()` related to enforcing generator reactive power limits when all generators violate limits or when the slack bus is converted to PQ.

- Fixed crash when using KNITRO to solve cases with all lines unconstrained.

- Fixed memory issue resulting from nested `om` fields when repeatedly running an OPF using the results of a previous OPF as input. *Thanks to Carlos Murillo-Sánchez.*

- Fixed fatal error when `uopf()` shuts down all gens attempting to satisfy $P_{\min}$ limits.

- Reactive power output of multiple generators at a PQ bus no longer get re-allocated when running a power flow.

- Fixed a bug in `savecase()` where a `gencost` matrix with extra columns of zeros resulted in a corrupted MATPOWER case file.

- Fixed bug in `runpf()` that caused a crash for cases with `pf.enforce_q_lims` turned on and exactly two Q limit violations, one Qmax and one Qmin. *Thanks to Jose Luis Marin.*

Incompatible Changes

- Optional packages TSPOPF and MINOPF must be updated to latest versions.

- Renamed `cdf2matp()` to `cdf2mpc()` and updated the interface to be consistent with `psse2mpc()`.

- Removed `ot_opts` field, replaced with `linprog_opts` and `quadprog_opts` fields in the `opt` argument to `qps_matpower()` and `qps_ot()`.

- The name of the `mips()` option used to specify the maximum number of step-size reductions with `step_control` on was changed from `max_red` to `sc.red_it` for consistency with other MATPOWER options.

- Removed `max_it` option from `qps_matpower()` (and friends) args. Use algorithm specific options to control iteration limits.

- Changed behavior of branch angle difference limits so that 0 is interpreted as unbounded only if both `ANGMIN` and `ANGMAX` are zero.

- In `results` struct returned by an OPF, the value of `results.raw.output.alg` is now a string, not an old-style numeric alg code.

- Removed:

– Support for MATLAB 6.x.

– Support for `constr()` and successive LP-based OPF solvers.

– Support for Gurobi 4.x/`gurobi_mex()` interface.

– `extras/cpf`, replaced by `runcpf()`.

– `extras/psse2matpower`, replaced by `psse2mpc()`.

H.5 Version 4.1 – released Dec 14, 2011

The MATPOWER 4.1 User's Manual is available online.[50]

New Features

- More new high performance OPF solvers:

  – Support for the KNITRO interior point optimizer for large scale non-linear optimization. Use OPF_ALG = 600 for to select KNITRO to solve the AC OPF. Requires the MATLAB Optimization Toolbox and a license for KNITRO, available from http://www.ziena.com/. See `help mpoption` for more KNITRO options.

  – Support for Gurobi to solve LP and QP problems. Set option OPF_ALG_DC = 700 to use Gurobi to solve the DC OPF. Requires Gurobi (http://www.gurobi.com/) and the Gurobi MEX interface (http://www.convexoptimization.com/wikimization/index.php/Gurobi_mex). See `help mpoption` for more Gurobi options.

  – Updated for compatibility with CPLEX 12.3.

  – Changed options so that `fmincon` uses its interior-point solver by default. Much faster on larger systems.

- Support for basic modeling of DC transmission lines.

- New case files with more recent versions of Polish system.

- Power flow can handle networks with islands.

---

[50]http://www.pserc.cornell.edu/matpower/docs/MATPOWER-manual-4.1.pdf

Bugs Fixed

- Computation of quadratic user-defined costs had a potentially fatal error. *Thanks to Stefanos Delikaraoglou for finding this.*

- Calculation of reserve prices in `toggle_reserves()` had an error.

Incompatible Changes

- Optional packages TSPOPF and MINOPF must be updated to latest versions.

H.6 Version 4.0 – released Feb 7, 2011

The MATPOWER 4.0 User's Manual is available online.[51]

New Features

- Licensed under the GNU General Public License (GPL).

- Added compatibility with GNU Octave, a free, open-source MATLAB clone.

- Extensive OPF enhancements:

  - Generalized, extensible OPF formulation applies to all solvers (AC and DC).
  - Improved method for modifying OPF formulation and output via a new user-defined callback function mechanism.
  - Option to co-optimize reserves based on fixed zonal reserve requirements, implemented using new callback function mechanism.
  - Option to include interface flow limits (based on DC model flows), implemented using new callback function mechanism.

- New high performance OPF solvers:

  - MIPS (MATLAB Interior Point Solver), a new a pure-MATLAB implementation of the primal-dual interior point methods from the optional package TSPOPF. MIPS is suitable for large systems and is used as MATPOWER's default solver for AC and DC OPF problems if no other optional solvers are installed. To select MIPS explicitly, use OPF_ALG = 560/565 and

---

[51]http://www.pserc.cornell.edu/matpower/docs/MATPOWER-manual-4.0.pdf

194

`OPF_ALG_DC` = 200/250 for AC and DC OPF, respectively. MIPS can also be used independently of MATPOWER as a solver for general non-linear constrained optimization problems.

– Support for the IPOPT interior point optimizer for large scale non-linear optimization. Use `OPF_ALG` = 580 and `OPF_ALG_DC` = 400 for AC and DC OPF, respectively. Requires the MATLAB MEX interface for IPOPT, available from https://projects.coin-or.org/Ipopt/.

– Support for CPLEX to solve LP and QP problems. Set option `OPF_ALG_DC` = 500 to use CPLEX to solve the DC OPF. Requires the MATLAB interface to CPLEX, available from http://www.ibm.com/software/integration/optimization/cplex-optimizer/. See `help mpoption` for more CPLEX options.

– Support for MOSEK to solve LP and QP problems. Set option `OPF_ALG_DC` = 600 to use MOSEK to solve the DC OPF. Requires the MATLAB interface to MOSEK, available from http://www.mosek.com/. See `help mpoption` for more MOSEK options.

– Updated support for MathWorks' Optimization Toolbox solvers, `fmincon()`, `linprog()` and `quadprog()`.

• Improved documentation:

– New, rewritten User's Manual (`docs/manual.pdf`).
– Two new Tech Notes, available from MATPOWER home page.
  * Uniform Price Auctions and Optimal Power Flow
  * AC Power Flows, Generalized OPF Costs and their Derivatives using Complex Matrix Notation
– Help text updates to more closely match MathWorks conventions.

• New functions:

– `load2disp()` converts from fixed to dispatchable loads.
– `makeJac()` forms the power flow Jacobian. Optionally returns the system admittance matrices too.
– `makeLODF()` computes line outage distribution factors.
– `modcost()` shifts/scales generator cost functions.

- qps_matpower() provides a consistent, unified interface to all of MAT-POWER's available QP/LP solvers, serving as a single wrapper around qps_bpmpd(), qps_cplex(), qps_ipopt(), qps_mips(), and qps_ot() (Optimization Toolbox, i.e. quadprog(), linprog()).

- scale_load() conveniently modifies multiple loads.

- total_load() retreives total load for the entire system, a specific zone or bus, with options to include fixed loads, dispatchable loads or both.

- Option to return full power flow or OPF solution in a single results struct, which is a superset of the input case struct.

- Ability to read and save generalized OPF user constraints, costs and variable limits as well as other user data in case struct.

- Numerous performance optimizations for large scale systems.

- Perl script psse2matpower for converting PSS/E data files to MATPOWER case format.

- Deprecated areas data matrix (was not being used).

- Many new tests in test suite.

Bugs Fixed

- Auction code in extras/smartmarket in all previous versions contained a design error which has been fixed. Prices are now scaled instead of shifted when modified according to specified pricing rule (e.g. LAO, FRO, LAB, FRB, split-the-difference, etc.). Auctions with both real and reactive offers/bids must be type 0 or 5, type 1 = LAO is no longer allowed.

- Branch power flow limits could be violated when using the option OPF_FLOW_LIM = 1.

Incompatible Changes

- Renamed functions used to compute AC OPF cost, constraints and hessian, since they are used by more than fmincon:

```
costfmin → opf_costfcn
consfmin → opf_consfcn
hessfmin → opf_hessfcn
```

- Input/output arguments to `uopf()` are now consistent with `opf()`.

- `dAbr_dV()` now gives partial derivatives of the **squared** magnitudes of flows w.r.t. V, as opposed to the magnitudes.

H.7   Version 3.2 – released Sep 21, 2007

The MATPOWER 3.2 User's Manual is available online.[52]

New Features

- AC OPF formulation enhancements

  - new generalized cost model
  - piece-wise linear generator PQ capability curves
  - branch angle difference constraints
  - simplified interface for specifying additional linear constraints
  - option to use current magnitude for line flow limits (set `OPF_FLOW_LIM` to 2, `fmincopf` solver only)

- AC OPF solvers

  - support for TSPOPF, a new optional package of three OPF solvers, implemented as C MEX files, suitable for large scale systems
  - ability to specify initial value and bounds on user variables z

- New (v. 2) case file format

  - all data in a single struct
  - generator PQ capability curves
  - generator ramp rates
  - branch angle difference limits

- New function `makePTDF` to build DC PDTF matrix

- Added 5 larger scale (> 2000 bus) cases for Polish system *(thanks to Roman Korab)*.

---

[52]http://www.pserc.cornell.edu/matpower/docs/MATPOWER-manual-3.2.pdf

197

- Improved identification of binding constraints in printout.

- Many new tests in test suite.

Bugs Fixed

- Phase shifters shifted the wrong direction, again (v.2 had it right).

- Fixed bug in `pfsoln` which caused incorrect value for reactive generation when $Q_{\min} = Q_{\max}$ for all generators at a bus in power flow solution.

Incompatible Changes

- User supplied A matrix for general linear constraints in OPF no longer includes columns for helper variables for piecewise linear gen costs, and now requires columns for all x (OPF) variables.

- Changed the sign convention used for phase shifters to be consistent with PTI, PowerWorld, PSAT, etc. E.g. A phase shift of 10 deg now means the voltage at the "to" end is delayed by 10 degrees.

- Name of option 24 in `mpoption` changed from `OPF_P_LINE_LIM` to `OPF_FLOW_LIM`.

H.8 Version 3.0 – released Feb 14, 2005

The MATPOWER 3.0 User's Manual is available online.[53]

New Features

- Compatibility with MATLAB 7 and Optimization Toolbox 3.

- DC power flow and DC OPF solvers added.

- Option to enforce generator reactive power limits in AC power flow solution.

- Gauss-Seidel power flow solver added.

- Support for MINOS-based OPF solver added (separate package, see http://www.pserc.cornell.edu/minopf/ for more details)

- Multiple generators at a single bus.

---

[53]http://www.pserc.cornell.edu/matpower/docs/MATPOWER-manual-3.0.pdf

- Saving of solved cases as M-files or MAT-files.

- Loading of input data from M-files, MAT-files, or structs.

- Improved decommitment algorithm.

- Added a very incomplete test suite.

- Handling of dispatchable loads in OPF, modeled as negative generators with constant power factor constraint.

Bugs Fixed

- Phase shifters shifted the wrong direction.

- Minor fixes to IEEE CDF to MATPOWER format conversion *(reported by D. Devaraj and Venkat)*

- Flows on out-of-service lines were not being zeroed out. *(reported by Ramazan Caglar)*

- Reported total inter-tie flow values and area export values were incorrect.

- Several other bugs in solution printouts.

H.9   Version 2.0 – released Dec 24, 1997

The MATPOWER 2.0 User's Manual is available online.[54]

New Features

- greatly enhanced output options

- fast-decoupled power flow

- optional costs for reactive power generation in OPF

- consolidated parameters in MATPOWER options vector

---

[54]http://www.pserc.cornell.edu/matpower/docs/MATPOWER-manual-2.0.pdf

Other Changes

- optimized building of Y_{bus} and Jacobian matrices (much faster on large systems, especially in MATLAB 5)

- highly improved unit decommitment algorithm

- completely rewritten LP-based OPF solver

- various bug fixes

H.10 Version 1.0.1 – released Sep 19, 1997

Changes

- Bug fixes.

H.11 Version 1.0 – released Sep 17, 1997

The MATPOWER 1.0 User's Manual is available online.[55]

New Features

- Successive LP-based optimal power flow solver, by Deqiang Gan.

- Automatic conversion between possibly non-consecutive external bus numbers and consecutive internal bus numbers.

- IEEE CDF to MATPOWER case conversion script, by Deqiang Gan.

Other Changes

- Top-level scripts converted to functions that take case file as input.

- Updated case file format. Generator costs moved to separate `gencost` table.

H.12 Pre 1.0 – released Jun 25, 1997

Work on MATPOWER begin in 1996, with an optimal power flow based on MATLAB's `constr` function.

---

[55]http://www.pserc.cornell.edu/matpower/docs/MATPOWER-manual-1.pdf

References

[1] R. D. Zimmerman, C. E. Murillo-Sánchez, and R. J. Thomas, "MATPOWER: Steady-State Operations, Planning and Analysis Tools for Power Systems Research and Education," *Power Systems, IEEE Transactions on*, vol. 26, no. 1, pp. 12–19, Feb. 2011. http://dx.doi.org/10.1109/TPWRS.2010.2051168 1.1, 1.3

[2] C. E. Murillo-Sánchez, R. D. Zimmerman, C. L. Anderson, and R. J. Thomas, "Secure Planning and Operations of Systems with Stochastic Sources, Energy Storage and Active Demand," *Smart Grid, IEEE Transactions on*, vol. 4, no. 4, pp. 2220–2229, Dec. 2013, http://dx.doi.org/10.1109/TSG.2013.2281001. 1.1, 1.3

[3] The BSD 3-Clause License. [Online]. Available: http://opensource.org/licenses/BSD-3-Clause. 1.2

[4] GNU General Public License. [Online]. Available: http://www.gnu.org/licenses/. 3

[5] F. Milano, "An Open Source Power System Analysis Toolbox," *Power Systems, IEEE Transactions on*, vol. 20, no. 3, pp. 1199–1206, Aug. 2005.

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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

[15] 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.

[16] 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

[17] 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, http://dx.doi.org/10.1109/PES.2009.5275967. 6.3

[18] TSPOPF. [Online]. Available: http://www.pserc.cornell.edu/tspopf/. 6.4.1, 6.5, C-17, C-18, G.13

[19] 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.13

[20] *Optimization Toolbox*, The MathWorks, Inc. [Online]. Available: http://www.mathworks.com/products/optimization/. 6.5, G.10

[21] *Optimization Toolbox Users's Guide*, The MathWorks, Inc., 2015. [Online]. Available: http://www.mathworks.com/help/releases/R2015b/pdf_doc/optim/optim_tb.pdf. 44

[22] BPMPD_MEX. [Online]. Available: http://www.pserc.cornell.edu/bpmpd/. 6.5, G.1

[23] 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

[24] MINOPF. [Online]. Available: http://www.pserc.cornell.edu/minopf/. 6.5, C-15, F, G.8

[25] 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.8

[26] 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, D-1

[27] A. J. Lamadrid, S. Maneevitjit, T. D. Mount, C. E. Murillo-Sánchez, R. J. Thomas, R. D. Zimmerman, "A 'SuperOPF' Framework", *CERTS Report*, December 2008. [Online]. Available: http://certs.lbl.gov/pdf/superopf-framework.pdf

[28] C. E. Murillo-Sánchez, R. D. Zimmerman, C. L. Anderson, and R. J. Thomas, "A Stochastic, Contingency-Based Security-Constrained Optimal Power Flow for the Procurement of Energy and Distributed Reserve," *Decision Support Systems*, Vol. 56, Dec. 2013, pp. 1-10, http://dx.doi.org/10.1016/j.dss.2013.04.006.

[29] 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.13

[30] C. Josz, S. Fliscounakis, J. Maeght, and P. Panciatici, "AC Power Flow Data in MATPOWER and QCQP Format: iTesla, RTE Snapshots, and PEGASE." Available: http://arxiv.org/abs/1603.01533. D-19, D-20, 33, H.2

[31] S. Fliscounakis, P. Panciatici, F. Capitanescu, and L. Wehenkel, "Contingency Ranking With Respect to Overloads in Very Large Power Systems Taking Into Account Uncertainty, Preventive, and Corrective Actions," *Power Systems, IEEE Transactions on*, vol. 28, no. 4, pp. 4909–4917, Nov. 2013. D-19

[32] P. Cuffe and A. Keane, "Visualizing the Electrical Structure of Power Systems," *IEEE Systems Journal*, Vol. PP, No. 99, pp. 1–12, http://dx.doi.org/10.1109/JSYST.2015.2427994. 33

[33] 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 D-1, F

[34] COIN-OR Linear Programming (CLP) Solver. [Online]. Available: http://www.coin-or.org/projects/Clp.xml. G.2

[35] J. Currie and D. I. Wilson,"OPTI: Lowering the Barrier Between Open Source Optimizers and the Industrial MATLAB User," *Foundations of Computer-Aided Process Operations*, Georgia, USA, 2012. G.2, G.4, G.6

[36] GLPK. [Online]. Available: http://www.gnu.org/software/glpk/. G.4

[37] Wotao Yin. *Gurobi Mex: A MATLAB interface for Gurobi*, URL: http://convexoptimization.com/wikimization/index.php/gurobi_mex, 2009-2011. 40

[38] 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-13, G.6

[39] O. Shenk and K. Gärtner, "Solving unsymmetric sparse systems of linear equations with PARDISO," *Journal of Future Generation Computer Systems*, 20(3):475–487, 2004. G.6, G.11

[40] A. Kuzmin, M. Luisier and O. Shenk, "Fast methods for computing selected elements of the Greens function in massively parallel nanoelectronic device simulations," in F. Wolf, B. Mohr and D. Mey, editors, *Euro-Par 2013 Parallel Processing*, Vol. 8097, *Lecture Notes in Computer Science*, pp. 533-544, Springer Berlin Heidelberg, 2013. G.6, G.11

[41] 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-14, G.7