MATPOWER 4.0b2
User's Manual

Ray D. Zimmerman

March 19, 2010

Contents

List of Figures

List of Tables

1 Introduction

1.1 What is MATPOWER?

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

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

1.2 Where did it come from?

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

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

1.3 Who may use it?

- MATPOWER is free. Anyone may use it.
- We make no warranties, express or implied. Specifically, we make no guarantees regarding the correctness MATPOWER's code or its fitness for any particular purpose.
- Any publications derived from the use of MATPOWER must acknowledge MATPOWER and cite [9]:[3]

  > 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.

- Anyone may modify MATPOWER for their own use as long as the original copyright notices remain in place.
- MATPOWER may not be redistributed without written permission.
- Modified versions of MATPOWER, or works derived from MATPOWER, may not be distributed without written permission.

[3] Switch to [1] once it is published.

2 Getting Started

2.1 System Requirements

To use MATPOWER 4.0b2 you will need:

- Matlab version 6.5 or later[4], available from The MathWorks[5]

For the hardware requirements, please refer to the system requirements for the version of Matlab you are using[6]. If the Matlab Optimization Toolbox is installed as well, MATPOWER enables an option to use it to solve optimal power flow problems, though this option is not recommended for most applications.

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

[4] Although it is likely that many things work fine in earlier versions of Matlab 6, they are not supported. MATPOWER 3.2 required Matlab 6, MATPOWER 3.0 required Matlab 5 and MATPOWER 2.0 and earlier only required Matlab 4.

[5] http://www.mathworks.com/

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

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

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

```
>> test_matpower
t_loadcase.........ok
t_ext2int2ext.......ok
t_jacobian.........ok
t_hessian..........ok
t_hasPQcap.........ok
t_mips.............ok
t_qps_matpower......ok (35 of 140 skipped)
t_pf...............ok
t_opf_fmincon.......ok
t_opf_mips.........ok
t_opf_mips_sc.......ok
t_opf_dc_ot.........ok
t_opf_dc_mips.......ok
t_opf_dc_mips_sc....ok
t_opf_userfcns......ok
t_runopf_w_res......ok
t_makePTDF.........ok
t_makeLODF.........ok
t_total_load........ok
t_scale_load........ok
All tests successful (1374 passed, 35 skipped of 1409)
Elapsed time 4.82 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[8]. 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-15.

2.3.2 Solving the Case

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

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

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

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

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

[8] 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.

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 5.6.

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 vector. The elements of the vector have names that can be used to set the corresponding value via the `mpoption` function. Calling `mpoption` with no arguments

returns the default options vector, the vector 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', 2, 'VERBOSE', 2, 'OUT_ALL', 0);
>> results = runpf('case300', mpopt);
```

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

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

See Appendix C or type:

```
>> help mpoption
```

for more information on MATPOWER's options.

2.4 Documentation

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

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

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

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

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

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

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

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

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

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

    Example:
        results = runopf('case30');

    See also RUNDCOPF, RUNUOPF.
```

3 Modeling

MATPOWER employs all of the standard steady state models typically used for power flow analysis. The AC models are described first, then the simplified DC models. Internally, the magnitudes of all values are expressed in per unit and angles of complex quantities are expressed in radians. Internally, all off-line generators and branches are removed before forming the models used to solve the power flow or optimal power flow problem. All buses are numbered consecutively, beginning at 1, and generators are reordered by bus number. Conversions to and from this internal indexing is done by the functions `ext2int` and `int2ext`. The notation in this section, as well as Sections 4 and 5, 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, 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 capacitance b_c, in series with an ideal phase shifting transformer. The transformer, whose tap ratio has magnitude τ and 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 3, 4, 5, 9 and 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

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

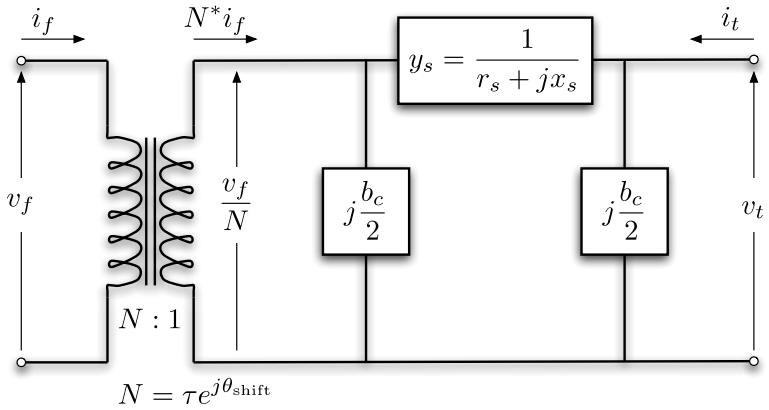

Figure 3-1: Branch Model

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 2 and 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 3 and 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 5 and 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.

$$I_{\text{bus}} = Y_{\text{bus}} V \tag{3.8}$$

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.

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

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.

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

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

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

3.7 DC Modeling

The DC formulation is based on the same parameters, but with the following three additional simplifying assumptions:

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

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

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

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

- Voltage angle differences across branches are small enough that

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

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

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

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

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

The approximate power flow is then derived as follows

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

Finally, extracting the real part and applying the last of the DC modeling assumptions from (3.20) yields

$$p_f \approx \frac{1}{x_s \tau}(\theta_f - \theta_t - \theta_{\text{shift}}) \tag{3.24}$$

As expected, given the lossless assumption, a similar derivation for p_t 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

$$\begin{bmatrix} p_f \\ p_t \end{bmatrix} = B^i_{br} \begin{bmatrix} \theta_f \\ \theta_t \end{bmatrix} + P^i_{\text{shift}} \tag{3.25}$$

where $B^i_{br} = \begin{bmatrix} b_i & -b_i \\ -b_i & b_i \end{bmatrix}$, $P^i_{\text{shift}} = \theta^i_{\text{shift}} \begin{bmatrix} -b_i \\ b_i \end{bmatrix}$ and b_i is defined in terms of the series reactance and tap ratio for that branch as $b_i = \frac{1}{x^i_s \tau^i}$.

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

$$\begin{aligned} s^i_{sh} &= v_i (y^i_{sh} v_i)^* \\ &\approx e^{j\theta_i}(g^i_{sh} - j b^i_{sh}) e^{-j\theta_i} \\ &= g^i_{sh} - j b^i_{sh} \end{aligned} \tag{3.26}$$

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

$$P_{sh} \approx G_{sh} \tag{3.27}$$

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

$$P_{\text{bus}}(\Theta) = B_{\text{bus}}\Theta + P_{\text{bus,shift}} \tag{3.28}$$

where

$$P_{\text{bus,shift}} = (C_f - C_t)^{\mathsf{T}} P_{f,\text{shift}} \tag{3.29}$$

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

$$P_f(\Theta) = B_f \Theta + P_{f,\text{shift}} \tag{3.30}$$

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.

$$B_f = [B_{ff}]\,(C_f - C_t) \tag{3.31}$$

$$B_{\text{bus}} = (C_f - C_t)^{\mathsf{T}} B_f \tag{3.32}$$

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

$$g_P(\Theta, P_g) = B_{\text{bus}}\Theta + P_{\text{bus,shift}} + P_d + G_{sh} - C_g P_g = 0 \tag{3.33}$$

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 classified as PV buses, with the values of voltage magnitude and generator real power injection given. Since the loads P_d and Q_d are also given, all non-generator buses are PQ buses, with real and reactive injections fully specified. 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.

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

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

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

For the AC power flow problem, the function $g(x)$ from (4.1) is formed by taking the left hand side of the real power balance equations (4.2) for all non-slack buses

and the reactive power balance equations (4.3) for all PQ buses and plugging in the reference angle, the loads and the known generator injections and voltage magnitudes.

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

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

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

This yields a system of non-linear 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 [3] 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 [4], specifically, the XB and BX methods described in [5]. 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 [6]. 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 (`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 `QMIN` and `QMAX` parameters for the generator and does not take into account the trapezoidal generator capability curves described in Section 5.4.3.

4.2 DC Power Flow

For the DC power flow problem [7], 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 equation (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 found directly from the non-slack rows of (3.33).

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 (3.30) and the appropriate row in (3.33), respectively.

4.3 `runpf`

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

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

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

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

Table 4-1: Power Flow Results

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

† AC power flow only.

```
>> 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 `PF_DC` option must be set to 1. For convenience, MATPOWER provides a function `rundcpf` which is simply a wrapper that sets `PF_DC` before calling `runpf`.

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 6.2.1, 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 [7]. These $n_l \times n_b$ sensitivity matrices, also called power transfer distribution factors or PTDF's, carry an implicit assumption about

Table 4-2: Power Flow Options

idx	name	default	description
1	PF_ALG	1	AC power flow algorithm: 1 – Newtons's method 2 – Fast-Decoupled (XB version) 3 – Fast-Decouple (BX version) 4 – Gauss-Seidel
2	PF_TOL	10^{-8}	termination tolerance on per unit P and Q dispatch
3	PF_MAX_IT	10	maximum number of iterations for Newton's method
4	PF_MAX_IT_FD	30	maximum number of iterations for fast decoupled method
5	PF_MAX_IT_GS	1000	maximum number of iterations for Gauss-Seidel method
6	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
10	PF_DC	0	DC modeling for power flow and OPF formulation 0 – use AC formulation and corresponding alg options 1 – use DC formulation and corresponding alg options

Table 4-3: Power Flow Output Options

idx	name	default	description
31	VERBOSE	1	amount of progress info to be printed 0 – print no progress info 1 – print a little progress info 2 – print a lot progress info 3 – print all progress info
32	OUT_ALL	-1	controls pretty-printing of results -1 – individual flags control what is printed 0 – do *not* print anything† 1 – print everything†
33	OUT_SYS_SUM	1	print system summary (0 or 1)
34	OUT_AREA_SUM	0	print area summaries (0 or 1)
35	OUT_BUS	1	print bus detail, includes per bus gen info (0 or 1)
36	OUT_BRANCH	1	print branch detail (0 or 1)
37	OUT_GEN	0	print generator detail (0 or 1)

† Overrides individual flags.

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 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 LODF's [8]. 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 its initial flow) for an outage of branch j.

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

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

If h_{ij} is the sensitivity of flow in branch i with respect to flow in branch j, then l_{ij} can be expressed as

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

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

5 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.

$$\min_x f(x) \tag{5.1}$$

subject to

$$g(x) = 0 \tag{5.2}$$
$$h(x) \leq 0 \tag{5.3}$$
$$x_{\min} \leq x \leq x_{\max} \tag{5.4}$$

5.1 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.

$$x = \begin{bmatrix} \Theta \\ V_m \\ P_g \\ Q_g \end{bmatrix} \tag{5.5}$$

The objective function (5.1) 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.

$$\min_{\Theta, V_m, P_g, Q_g} \sum_{i=1}^{n_g} f_P^i(p_g^i) + f_Q^i(q_g^i) \tag{5.6}$$

The equality constraints in (5.2) are simply the full set of $2 \cdot n_b$ non-linear real and reactive power balance equations from (4.2) and (4.3). The inequality constraints (5.3) consist of two sets of n_l branch flow limits as non-linear functions of the bus voltage angles and magnitudes, one for the *from* end and one for the *to* end of each branch.

$$h_f(\Theta, V_m) = |F_f(\Theta, V_m)| - F_{\max} \leq 0 \tag{5.7}$$
$$h_t(\Theta, V_m) = |F_t(\Theta, V_m)| - F_{\max} \leq 0 \tag{5.8}$$

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

$$F_f(\Theta, V_m) = \begin{cases} S_f(\Theta, V_m), & \text{apparent power} \\ P_f(\Theta, V_m), & \text{real power} \\ I_f(\Theta, V_m), & \text{current} \end{cases} \tag{5.9}$$

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. Likewise for $F_t(\Theta, V_m)$.

The variable limits (5.4) include an equality constraint on any reference bus angle and upper and lower limits on all bus voltage magnitudes and real and reactive generator injections.

$$\theta_i^{\text{ref}} \leq \theta_i \leq \theta_i^{\text{ref}}, \qquad i \in \mathcal{I}_{\text{ref}} \tag{5.10}$$
$$v_m^{i,\min} \leq v_m^i \leq v_m^{i,\max}, \qquad i = 1 \ldots n_b \tag{5.11}$$
$$p_g^{i,\min} \leq p_g^i \leq p_g^{i,\max}, \qquad i = 1 \ldots n_g \tag{5.12}$$
$$q_g^{i,\min} \leq q_g^i \leq q_g^{i,\max}, \qquad i = 1 \ldots n_g \tag{5.13}$$

5.2 Standard DC OPF

When using DC network modeling assumptions, 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

$$x = \begin{bmatrix} \Theta \\ P_g \end{bmatrix} \tag{5.14}$$

and the overall problem reduces to the following form.

$$\min_{\Theta, P_g} \sum_{i=1}^{n_g} f_P^i(p_g^i) \tag{5.15}$$

subject to

$$g_P(\Theta, P_g) = B_{\text{bus}}\Theta + P_{\text{bus,shift}} + P_d + G_{sh} - C_g P_g = 0 \tag{5.16}$$
$$h_f(\Theta) = \quad B_f\Theta + P_{f,\text{shift}} - F_{\max} \leq 0 \tag{5.17}$$
$$h_t(\Theta) = -B_f\Theta - P_{f,\text{shift}} - F_{\max} \leq 0 \tag{5.18}$$

$$\theta_i^{\text{ref}} \leq \theta_i \leq \theta_i^{\text{ref}}, \qquad i \in \mathcal{I}_{\text{ref}} \tag{5.19}$$
$$p_g^{i,\min} \leq p_g^i \leq p_g^{i,\max}, \qquad i = 1 \ldots n_g \tag{5.20}$$

5.3 Extended OPF Formulation

MATPOWER employs an extensible OPF structure [9] 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.

$$\min_{x,z} f(x) + f_u(x, z) \tag{5.21}$$

subject to

$$g(x) = 0 \tag{5.22}$$

$$h(x) \leq 0 \tag{5.23}$$

$$x_{\min} \leq x \leq x_{\max} \tag{5.24}$$

$$l \leq A \begin{bmatrix} x \\ z \end{bmatrix} \leq u \tag{5.25}$$

$$z_{\min} \leq z \leq z_{\max} \tag{5.26}$$

Section 6 describes the mechanisms available to the user for taking advantage of the extensible formulation described here.

5.3.1 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

$$f_u(x, z) = \frac{1}{2} w^{\mathsf{T}} H w + C^{\mathsf{T}} w \tag{5.27}$$

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

$$r = N \begin{bmatrix} x \\ z \end{bmatrix}, \tag{5.28}$$

$$u = r - \hat{r}, \tag{5.29}$$

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

$$w_i = \begin{cases} m_i f_{d_i}(u_i + k_i), & u_i < -k_i \\ 0, & -k_i \le u_i \le k_i \\ m_i f_{d_i}(u_i - k_i), & u_i > k_i \end{cases} \tag{5.30}$$

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

$$f_{d_i}(\alpha) = \begin{cases} \alpha, & \text{if } d_i = 1 \\ \alpha^2, & \text{if } d_i = 2 \end{cases} \tag{5.31}$$

as illustrated in Figure 5-1 and Figure 5-2, respectively.

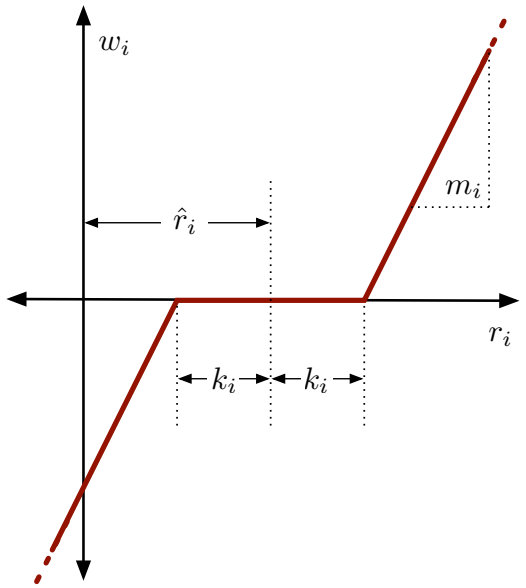

Figure 5-1: Relationship of w_i to r_i for $d_i = 1$ (linear option)

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

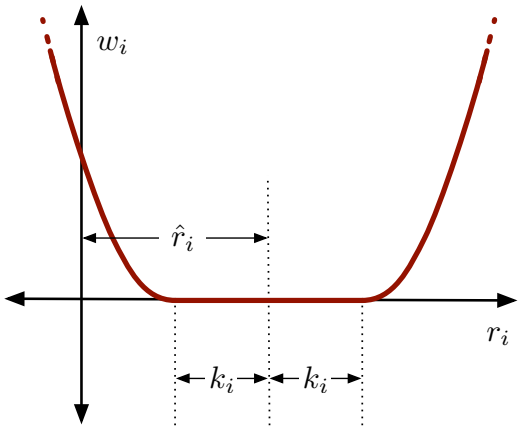

Figure 5-2: Relationship of w_i to r_i for $d_i = 2$ (quadratic option)

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 (5.30) simplifies to $w_i = m_i u_i$.

5.3.2 User-defined Constraints

The user-defined constraints (5.25) 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.

5.3.3 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.

5.4 Standard Extensions

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

5.4.1 Piecewise Linear Costs

The standard OPF formulation in (5.1)–(5.4) 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 5-3 illustrates a convex n-segment piecewise linear cost function

$$c(x) = \begin{cases} m_1(x - x_1) + c_1, & x \le x_1 \\ m_2(x - x_2) + c_2, & x_1 < x \le x_2 \\ \vdots & \vdots \\ m_n(x - x_n) + c_n, & x_{n-1} < x \end{cases} \tag{5.32}$$

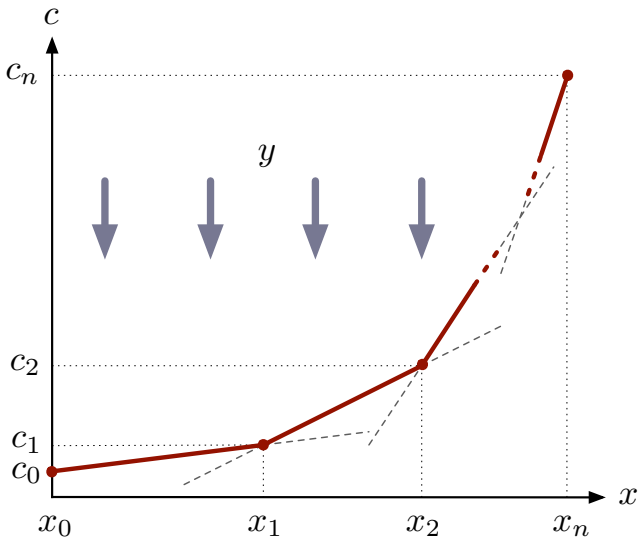

Figure 5-3: Constrained Cost Variable

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,

$$m_j = \frac{c_j - c_{j-1}}{x_j - x_{j-1}}, \quad j = 1 \ldots n \tag{5.33}$$

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

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

$$y \geq m_j(x - x_j) + c_j, \quad j = 1 \ldots n \tag{5.34}$$

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 convert any piecewise linear costs on real or reactive generation into the appropriate helper variable and corresponding set of constraints. All of MATPOWER's OPF solvers use the CCV approach with the exception of two that are part of the optional TSPOPF package [10], 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 [11].

5.4.2 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 5-4. The demand p_d of this load will be zero for prices above λ_1, p_1 for prices between λ_1 and λ_2, and $p_1 + p_2$ for prices below λ_2.

This corresponds to a negative generator with the piecewise linear cost curve shown in Figure 5-5. 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

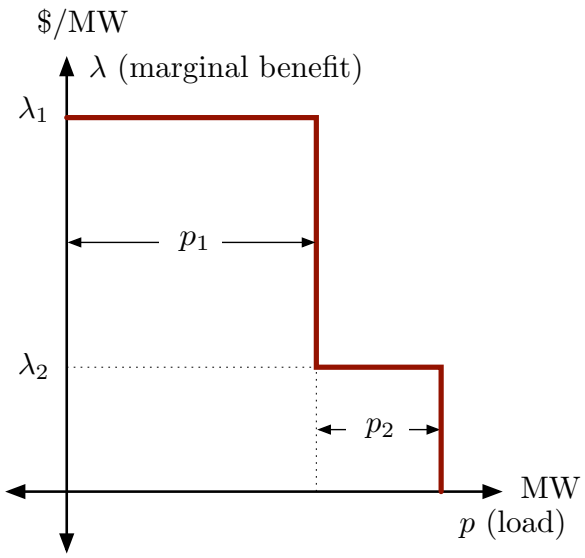

Figure 5-4: Marginal Benefit or Bid Function

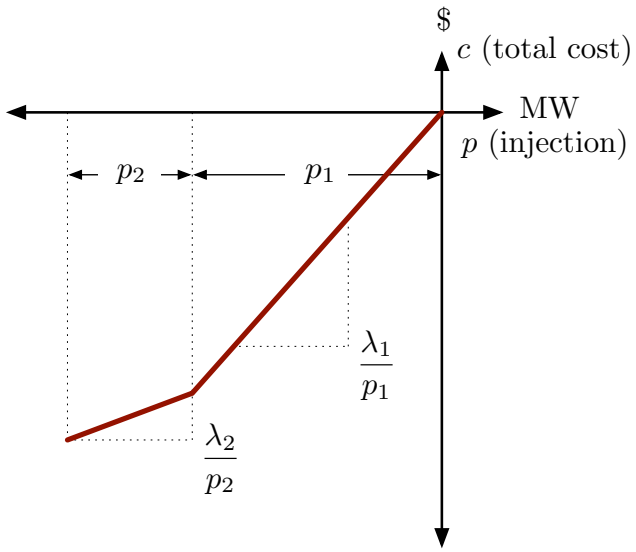

Figure 5-5: Total Cost Function for Negative Injection

an additional equality constraint to enforce a constant power factor for any "negative generator" being used to model a dispatchable load.

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

5.4.3 Generator Capability Curves

The typical AC OPF formulation includes box constraints on a generator's real and reactive injections, specified as simple lower and upper bounds on p ($p_{\min}$ and $p_{\max}$) and q ($q_{\min}$ and $q_{\max}$). On the other hand, the true P-Q capability curves of physical generators usually involve some tradeoff between real and reactive capability, so that it is not possible to produce the maximum real output and the maximum (or minimum) reactive output simultaneously. To approximate this tradeoff, MATPOWER includes the ability to add an upper and lower sloped portion to the standard box constraints as illustrated in Figure 5-6, where the shaded portion represents the

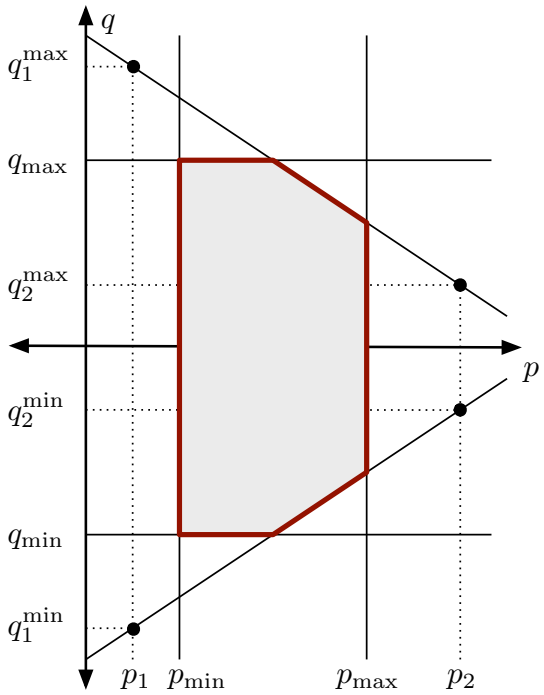

Figure 5-6: Generator P-Q Capability Curve

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, MATPOWER automatically constructs the corresponding additional linear inequality constraints on p and q for that unit.

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

5.4.4 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, MATPOWER creates the corresponding constraints on the voltage angle variables.

5.5 Solvers

Early versions of MATPOWER relied on Matlab's Optimization Toolbox [12] 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. 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 [13] of the high performance interior point BPMPD solver [14] for LP/QP problems. For the AC OPF problem, the MINOPF [15] and TSPOPF [10] packages provide solvers suitable for much larger systems. The former is based on MINOS [16] and the latter includes the primal-dual interior point and trust region based augmented Lagrangian methods described in [11]. 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 [11]. 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 [17]. This solver has application to general non-linear 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.

5.6 runopf

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

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

The `results` struct is a superset of the input MATPOWER case struct `mpc`, with some additional fields as well as additional columns in some of the existing data 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 5-1.

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

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

Some of the main options that control the optimal power flow simulation are listed in Table 5-2. 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 5-3.

By default, `runopf` solves an AC optimal power flow problem using a primal dual interior point method. To run a DC OPF, the `PF_DC` option must be set to 1. For convenience, MATPOWER provides a function `rundcopf` which is simply a wrapper that sets `PF_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,

Table 5-1: 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 non-linear 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.

described in more detail in Section 6.2.1, 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 5-2: Optimal Power Flow Options

idx	name	default	description
11	OPF_ALG	0	AC optimal power flow algorithm: 0 – choose default solver based on availability in the following order: 540, 560 300 – `constr`, Matlab Opt Toolbox 1.x and 2.x 320 – dense successive LP 340 – sparse successive LP (relaxed) 360 – sparse successive LP (full) 500 – MINOPF, MINOS-based solver[†] 520 – `fmincon`, Matlab Opt Toolbox $\geq$ 2.x 540 – PDIPM, primal/dual interior point method[‡] 545 – SC-PDIPM, step-controlled variant of PDIPM[‡] 550 – TRALM, trust region based augmented Lagrangian method[‡] 560 – MIPS, Matlab Interior Point Solver, primal/dual interior point method (pure Matlab) 565 – MIPS-sc, step-controlled variant of MIPS
16	OPF_VIOLATION	5×10^{-6}	constraint violation tolerance
24	OPF_FLOW_LIM	0	quantity to limit for branch flow constraints 0 – apparent power flow (limit in MVA) 1 – active power flow (limit in MW) 2 – current magnitude (limit in MVA at 1 p.u. voltage)
25	OPF_IGNORE_ANG_LIM	0	ignore angle difference limits for branches 0 – include angle difference limits, if specified 1 – ignore angle difference limits even if specified
26	OPF_ALG_DC	0	DC optimal power flow algorithm: 0 – choose default solver based on availability in the following order: 540, 560 100 – BPMPD[§] 200 – MIPS, Matlab Interior Point Solver, primal/dual interior point method (pure Matlab) 250 – MIPS-sc, step-controlled variant of MIPS 300 – Matlab Opt Toolbox, `quadprog`, `linprog`

[†] Requires optional MEX-based MINOPF package, available from http://www.pserc.cornell.edu/minopf/.
[‡] Requires optional MEX-based TSPOPF package, available from http://www.pserc.cornell.edu/tspopf/.
[§] Requires optional MEX-based BPMPD_MEX package, available from http://www.pserc.cornell.edu/bpmpd/.

Table 5-3: OPF Output Options

idx	name	default	description
38	OUT_ALL_LIM	-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†
39	OUT_V_LIM	1	control output of voltage limit info 0 – do *not* print 1 – print binding constraints only 2 – print all constraints
40	OUT_LINE_LIM	1	control output of line flow limit info‡
41	OUT_PG_LIM	1	control output of gen active power limit info‡
42	OUT_QG_LIM	1	control output of gen reactive power limit info‡

† Overrides individual flags.
‡ Takes values of 0, 1 or 2 as for OUT_V_LIM.

6 Extending the OPF

The extended OPF formulation described in Section 5.3 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.

6.1 Direct Specification

To add costs directly, the parameters H, C, N, $\hat{r}$, k, d and m of (5.27)–(5.31) described in Section 5.3.1 are specified as fields or arguments `H`, `Cw`, `N` and `fparm`, repectively, where `fparm` is the $n_w \times 4$ matrix

$$f_{\text{parm}} = \begin{bmatrix} d & \hat{r} & k & m \end{bmatrix}. \tag{6.1}$$

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 (5.25) 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[9], 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

[9] 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.

6.2 Callback Functions

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

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

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

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

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

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

$$0 \le r_i \le \min(r_i^{\max}, \Delta_i), \quad i = 1 \ldots n_g \tag{6.2}$$

If the vector c contains the marginal cost of reserves for each generator, the user defined cost term from (5.21) is simply

$$f_u(x, z) = c^{\mathsf{T}} r. \tag{6.3}$$

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

$$p_g^i + r_i \le p_g^{i,\max}, \quad i = 1 \ldots n_g \tag{6.4}$$

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

$$\sum_{i \in Z_k} r_i \ge R_k, \quad k = 1 \ldots n_{rz} \tag{6.5}$$

Table 6-1 describes some of the variables and names that are used in the example callback function listings in the sections below.

6.2.1 `ext2int` Callback

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

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

Table 6-1: 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 (6.3)
`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 `zones(k,j)` $= \begin{cases} 1 & \text{if gen } j \text{ belongs to reserve zone } k \quad (j \in Z_k) \\ 0 & \text{otherwise} \quad (j \notin Z_k) \end{cases}$
`req`	$n_{rz} \times 1$ vector of zonal reserve requirements, k^{th} element is R_k from (6.5)
`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 (6.4)
`Rreq`	name for new reserve requirement constraint set (6.5)

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 `ext2int` 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 `ext2int` for more details on what all it can do.

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

mpc = ext2int(mpc, {'reserves', 'qty'}, 'gen');
mpc = ext2int(mpc, {'reserves', 'cost'}, 'gen');
mpc = ext2int(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.

6.2.2 `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.[10] 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 (6.2) is added to the model via the `add_vars` method. Similarly, two linear constraint blocks named `Pg_plus_R` and `Rreq`, implementing (6.4) and (6.5), 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 (6.3).

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 6-1, 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

[10] 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.

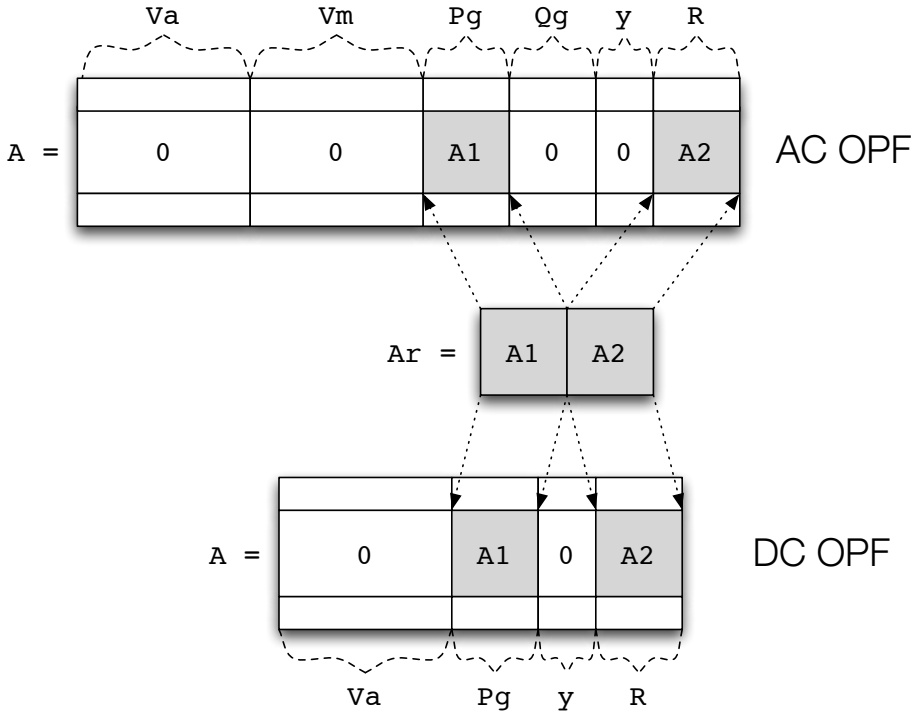

Figure 6-1: Adding Constraints Across Subsets of Variables

```
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'});
```

6.2.3 `int2ext` Callback

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

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

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

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

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

The `results` struct contains, in addition to the standard OPF results, solution information related to all of the user-defined variables, constraints and costs. Table 6-2 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 6-2: 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 `int2ext` with extra arguments. See the help for `int2ext` for more details on how it 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 = int2ext(results, {'reserves', 'qty'}, 'gen');
results = int2ext(results, {'reserves', 'cost'}, 'gen');
results = int2ext(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       = int2ext(results, R, z, 'gen');
results.reserves.Rmin    = int2ext(results, Rmin, z, 'gen');
results.reserves.Rmax    = int2ext(results, Rmax, z, 'gen');
results.reserves.mu.l    = int2ext(results, mu_l, z, 'gen');
results.reserves.mu.u    = int2ext(results, mu_u, z, 'gen');
results.reserves.mu.Pmax = int2ext(results, mu_Pmax, z, 'gen');
results.reserves.prc     = z;
for k = 1:ng0
    iz = find(r.zones(:, k));
    results.reserves.prc(k) = max(results.lin.mu.l.Rreq(iz)) / results.baseMVA;
end
results.reserves.totalcost = results.cost.Rcost;
```

6.2.4 `printpf` Callback

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

the `results` struct, the file descriptor to write to, a MATPOWER options vector, 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 vector 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);
OUT_ALL = mpopt(32);
if 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
```

6.2.5 `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[11] can be very useful for this purpose.

[11] http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=12063&objectType=file

```
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 ...
```

6.3 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 or disable fixed reserve requirements.
%   mpc = toggle_reserves(mpc, 'on')
%   mpc = toggle_reserves(mpc, 'off')

if strcmp(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);
elseif strcmp(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);
else
    error('toggle_reserves: 2nd argument must be either ''on'' or ''off''');
end
```

To run 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);
```

6.4 Summary

The five callback stages currently defined by MATPOWER are summarized in Table 6-3.

Table 6-3: 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.

MATPOWER includes two OPF extensions implemented via callbacks. One is a more complete version of the example of fixed zonal reserve requirements use for illustration above. The code can be found in `toggle_reserves`. A wrapper around `runopf` that turns on this extension before running the OPF is provided in `runopf_w_res`.

A second extension that imposes interface flow limits using DC model based flows is implemented in `toggle_iflims`. Examples of using these extensions and a case file defining the necessary input data for both 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 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 `PF_DC` option must be set to 1. For convenience, MATPOWER provides a function `runduopf` which is simply a wrapper that sets `PF_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: Solve a normal OPF. Save the solution as the current best.

Step 3: 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 5.

Step 4: 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 3.

Step 5: 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.

8 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. 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. Thanks also to many others who have contributed code, 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 [11, 18].

This solver has application outside of MATPOWER to general non-linear 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[12], 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.

[12] For Matlab 6.x use `mips6`.

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 non-linear 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)`, where $\lambda =$ `lam.eqnonlin` and $\mu =$ `lam.ineqnonlin`
`opt`	Optional options structure with the following fields, all of which are also optional (default values shown in parentheses).
	`verbose` (0) — controls level of progress output displayed 0 – print no progress info 1 – print a little progress info 2 – print a lot progress info 3 – print all progress info
	`feastol` (1e-6) — termination tolerance for feasibility condition
	`gradtol` (1e-6) — termination tolerance for gradient condition
	`comptol` (1e-6) — termination tolerance for complementarity condition
	`costtol` (1e-6) — termination tolerance for cost condition
	`max_it` (150) — maximum number of iterations
	`step_control` (0) — set to 1 to enable step-size control
	`max_red` (20) — maximum number of step-size reductions if step-control is on
	`cost_mult` (1) — cost multiplier used to scale the objective function for improved conditioning. Note: The same value must also be passed to the Hessian evaluation function so that it can appropriately scale the objective function term in the Hessian of the Lagrangian.
`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 non-linear constraints, the Hessian information must provided by the `hess_fcn` function and it need not be computed in `f_fcn`.

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`	non-linear equality constraints
`ineqnonlin`	non-linear 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 user-defined functions for evaluating the objective function, constraints and Hessian are identical to those required by `fmincon`. Specifically, the function `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 the `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 non-linear equality constraint violations $g(x)$, the $p \times 1$ vector of non-linear 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 non-linear constraints `hess_fcn` returns the $n \times n$ Hessian $\frac{\partial^2 \mathcal{L}}{\partial x^2}$ of the Lagrangian function

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

for given values of the multipliers λ and μ.

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[13]

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

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`.

```
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
```

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

A.2 Example 2

The second example[14] 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,[15]

```
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)
mu = lam.ineqnonlin;
Lxx = [2*[1 1]*mu -1 0; -1 2*[-1 1]*mu -1; 0 -1 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.

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

[15] Since the problem has non-linear 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 example2
problem = struct( ...
    'f_fcn',    @(x)f2(x), ...
    'gh_fcn',   @(x)gh2(x), ...
    'hess_fcn', @(x, lam)hess2(x, lam), ...
    '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
MIPS --  Matlab Interior Point Solver, Version 1.0, 10-Mar-2010
 it    objective   step size   feascond     gradcond     compcond     costcond
----  ------------ --------- ------------ ------------ ------------ ------------
  0            -1                     0          1.5            5            0
  1    -5.3250167     1.6875            0     0.894235     0.850653      2.16251
  2    -7.4708991    0.97413     0.129183   0.00936418     0.117278     0.339269
  3    -7.0553031    0.10406            0   0.00174933    0.0196518    0.0490616
  4    -7.0686267   0.034574            0   0.00041301    0.0030084   0.00165402
  5    -7.0706104  0.0065191            0  1.53531e-05  0.000337971  0.000245844
  6    -7.0710134 0.00062152            0  1.22094e-07  3.41308e-05  4.99387e-05
  7    -7.0710623 5.7217e-05            0  9.84878e-10  3.41587e-06  6.05875e-06
  8    -7.0710673 5.6761e-06            0  9.73397e-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`[16] is provided to make it trivial to set up and solve linear programming (LP) and quadratic programming (QP) problems of the following form.

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

subject to

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

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

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`.

[16] For Matlab 6.x use `qps_mips6`.

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 [11, 18].

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

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

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

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

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

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

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

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

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

A.4.2 Problem Formulation and Lagrangian

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

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

subject to

$$G(X) = 0 \tag{A.20}$$
$$H(X) \leq 0 \tag{A.21}$$

where the linear constraints and variable bounds from (A.4) and (A.5) have been incorporated into $G(X)$ and $H(X)$. The approach taken involves converting the n_i inequality constraints into equality constraints using a barrier function and vector of positive slack variables Z.

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

subject to

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

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

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

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

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

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

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

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

A.4.3 First Order Optimality Conditions

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

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

$$Z > 0 \tag{A.33}$$

$$\mu > 0 \tag{A.34}$$

where

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

A.4.4 Newton Step

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

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

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

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

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

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

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

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

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

where

$$\begin{aligned}
M &\equiv \mathcal{L}_{XX}^{\gamma} + {H_X}^{\mathsf{T}}[Z]^{-1}[\mu]\,H_X && \text{(A.41)} \\
&= f_{XX} + G_{XX}(\lambda) + H_{XX}(\mu) + {H_X}^{\mathsf{T}}[Z]^{-1}[\mu]\,H_X && \text{(A.42)}
\end{aligned}$$

and

$$\begin{aligned}
N &\equiv {\mathcal{L}_X^{\gamma}}^{\mathsf{T}} + {H_X}^{\mathsf{T}}[Z]^{-1}(\gamma e + [\mu]\,H(X)) && \text{(A.43)} \\
&= {f_X}^{\mathsf{T}} + \lambda^{\mathsf{T}}G_X + \mu^{\mathsf{T}}H_X + {H_X}^{\mathsf{T}}[Z]^{-1}(\gamma e + [\mu]\,H(X)) && \text{(A.44)}
\end{aligned}$$

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

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

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 .

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

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-15 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 6.1. Other user-defined fields may also be included, such as the `reserves` field used in the example code throughout Section 6.2. The `loadcase` function will automatically load any extra fields from a case file and, if the appropriate `'savecase'` callback function (see Section 6.2.5) is added via `add_userfcn`, `savecase` will also save them back to a case file.

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

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

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

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

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

* Not included in version 1 case format.

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

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

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

* Not included in version 1 case format.
† 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 begin in this column (`MODEL` = 1) $\Rightarrow$ $p_0, f_0, p_1, f_1, \ldots, p_n, f_n$ where $p_0 < p_1 < \cdots < p_n$ and the cost $f(p)$ is defined by the coordinates (p_0, f_0), (p_1, f_1), ..., (p_n, f_n) of the end/break-points of the piecewise linear cost (`MODEL` = 2) $\Rightarrow$ $c_n, \ldots, c_1, c_0$ $n + 1$ coefficients of n-th order polynomial cost, starting with highest order, where cost is $f(p) = c_n p^n + \cdots + c_1 p + c_0$

Appendix C MATPOWER Options

MATPOWER uses an options vector to control the many options available. It is similar to the options vector produced by the `foptions` function in early versions of Matlab's Optimization Toolbox. The primary difference is that modifications can be made by option name, as opposed to having to remember the index of each option. The MATPOWER options vector controls the following:

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

The default MATPOWER options vector is obtained by calling `mpoption` with no arguments.

```
>> opt = mpoption;
```

Calling it with a set of name/value pairs as arguments returns an options vector with the named options set to the specified values and all others set to their default values. For example, the following runs a fast-decoupled power flow of `case30` with very verbose progress output:

```
>> opt = mpoption('PF_ALG', 2, 'VERBOSE', 3);
>> runpf('case30', opt);
```

To make changes to an existing options vector, 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:

```
>> opt = mpoption(opt, 'ENFORCE_Q_LIMS', 1, 'OUT_ALL', 0);
>> results = runpf('case30', opt);
```

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: Power Flow Options

idx	name	default	description
1	PF_ALG	1	AC power flow algorithm: 1 – Newtons's method 2 – Fast-Decoupled (XB version) 3 – Fast-Decouple (BX version) 4 – Gauss-Seidel
2	PF_TOL	10^{-8}	termination tolerance on per unit P and Q dispatch
3	PF_MAX_IT	10	maximum number of iterations for Newton's method
4	PF_MAX_IT_FD	30	maximum number of iterations for fast decoupled method
5	PF_MAX_IT_GS	1000	maximum number of iterations for Gauss-Seidel method
6	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
10	PF_DC	0	DC modeling for power flow and OPF formulation 0 – use AC formulation and corresponding alg options 1 – use DC formulation and corresponding alg options

Table C-2: General OPF Options

idx	name	default	description
11	`OPF_ALG`	0	AC optimal power flow algorithm: 0 – choose default solver based on availability in the following order: 540, 560 300 – `constr`, Matlab Opt Toolbox 1.x and 2.x 320 – dense successive LP 340 – sparse successive LP (relaxed) 360 – sparse successive LP (full) 500 – MINOPF, MINOS-based solver[†] 520 – `fmincon`, Matlab Opt Toolbox $\geq$ 2.x 540 – PDIPM, primal/dual interior point method[‡] 545 – SC-PDIPM, step-controlled variant of PDIPM[‡] 550 – TRALM, trust region based augmented Lagrangian method[‡] 560 – MIPS, Matlab Interior Point Solver, primal/dual interior point method (pure Matlab) 565 – MIPS-sc, step-controlled variant of MIPS
16	`OPF_VIOLATION`	5×10^{-6}	constraint violation tolerance
24	`OPF_FLOW_LIM`	0	quantity to limit for branch flow constraints 0 – apparent power flow (limit in MVA) 1 – active power flow (limit in MW) 2 – current magnitude (limit in MVA at 1 p.u. voltage)
25	`OPF_IGNORE_ANG_LIM`	0	ignore angle difference limits for branches 0 – include angle difference limits, if specified 1 – ignore angle difference limits even if specified
26	`OPF_ALG_DC`	0	DC optimal power flow algorithm: 0 – choose default solver based on availability in the following order: 540, 560 100 – BPMPD[§] 200 – MIPS, Matlab Interior Point Solver, primal/dual interior point method (pure Matlab) 250 – MIPS-sc, step-controlled variant of MIPS 300 – Matlab Opt Toolbox, `quadprog`, `linprog`

[†] Requires optional MEX-based MINOPF package, available from http://www.pserc.cornell.edu/minopf/.
[‡] Requires optional MEX-based TSPOPF package, available from http://www.pserc.cornell.edu/tspopf/.
[§] Requires optional MEX-based BPMPD_MEX package, available from http://www.pserc.cornell.edu/bpmpd/.

Table C-3: Power Flow and OPF Output Options

idx	name	default	description
31	VERBOSE	1	amount of progress info to be printed 0 – print no progress info 1 – print a little progress info 2 – print a lot progress info 3 – print all progress info
32	OUT_ALL	-1	controls pretty-printing of results -1 – individual flags control what is printed 0 – do *not* print anything† 1 – print everything†
33	OUT_SYS_SUM	1	print system summary (0 or 1)
34	OUT_AREA_SUM	0	print area summaries (0 or 1)
35	OUT_BUS	1	print bus detail, includes per bus gen info (0 or 1)
36	OUT_BRANCH	1	print branch detail (0 or 1)
37	OUT_GEN	0	print generator detail (0 or 1)
38	OUT_ALL_LIM	-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†
39	OUT_V_LIM	1	control output of voltage limit info 0 – do *not* print 1 – print binding constraints only 2 – print all constraints
40	OUT_LINE_LIM	1	control output of line flow limit info‡
41	OUT_PG_LIM	1	control output of gen active power limit info‡
42	OUT_QG_LIM	1	control output of gen reactive power limit info‡

† Overrides individual flags.
‡ Takes values of 0, 1 or 2 as for OUT_V_LIM.

Table C-4: OPF Options for MIPS and TSPOPF

idx	name	default	description
81	PDIPM_FEASTOL†	0	feasibiliy (equality) tolerance set to value of OPF_VIOLATION by default
82	PDIPM_GRADTOL†	10^{-6}	gradient tolerance
83	PDIPM_COMPTOL†	10^{-6}	complementarity condition (inequality) tolerance
84	PDIPM_COSTTOL†	10^{-6}	optimality tolerance
85	PDIPM_MAX_IT†	150	maximum number of iterations
86	SCPDIPM_RED_IT‡	20	maximum number of step size reductions per iteration
87	TRALM_FEASTOL*	0	feasibiliy tolerance set to value of OPF_VIOLATION by default
88	TRALM_PRIMETOL*	5×10^{-4}	primal variable tolerance
89	TRALM_DUALTOL*	5×10^{-4}	dual variable tolerance
90	TRALM_COSTTOL*	10^{-5}	optimality tolerance
91	TRALM_MAJOR_IT*	40	maximum number of major iterations
92	TRALM_MINOR_IT*	100	maximum number of minor iterations
93	SMOOTHING_RATIO§	0.04	piecewise linear curve smoothing ratio

† For OPF_ALG option set to 540, 545, 560 or 565 and OPF_ALG_DC option set to 200 or 250 (MIPS, PDIPM and SC-PDIPM solvers) only.
‡ For OPF_ALG option set to 545 or 565 and OPF_ALG_DC option set to 250 (step-controlled solvers MIPS-sc or SC-PDIPM) only.
* For OPF_ALG option set to 550 (TRALM solver) only.
§ For OPF_ALG option set to 545 or 550 (SC-PDIPM or TRALM solvers) only.

Table C-5: OPF Options for fmincon, constr and successive LP Solvers

idx	name	default	description
17	CONSTR_TOL_X†	10^{-4}	termination tolerance on x
18	CONSTR_TOL_F†	10^{-4}	termination tolerance on f
19	CONSTR_MAX_IT†	0	maximum number of iterations $0 \Rightarrow 2n_b + 150$, where n_b is number of buses
20	LPC_TOL_GRAD‡	3×10^{-3}	termination tolerance on gradient
21	LPC_TOL_X‡	10^{-4}	termination tolerance on x (min step size)
22	LPC_MAX_IT‡	400	maximum number of iterations
23	LPC_MAX_RESTART‡	5	maximum number of restarts
55	FMC_ALG§	1	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

† For OPF_ALG option set to 300 or 520 (fmincon and constr solvers) only.
‡ For OPF_ALG option set to 320, 340 or 360 (successive LP-based solvers) only.
§ For OPF_ALG option set to 520 (fmincon solver) only.

Table C-6: OPF Options for MINOPF†

idx	name	default‡	description
61	MNS_FEASTOL	0 (10^{-3})	primal feasibility tolerance set to value of OPF_VIOLATION by default
62	MNS_ROWTOL	0 (10^{-3})	row tolerance set to value of OPF_VIOLATION by default
63	MNS_XTOL	0 (10^{-3})	x tolerance set to value of CONSTR_TOL_X by default
64	MNS_MAJDAMP	0 (0.5)	major damping parameter
65	MNS_MINDAMP	0 (2.0)	minor damping parameter
66	MNS_PENALTY_PARM	0 (1.0)	penalty parameter
67	MNS_MAJOR_IT	0 (200)	major iterations
68	MNS_MINOR_IT	0 (2500)	minor iterations
69	MNS_MAX_IT	0 (2500)	iteration limit
70	MNS_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)
71	MNS_CORE	0	memory allocation defaults to $1200n_b + 2(n_b + n_g)^2$
72	MNS_SUPBASIC_LIM	0	superbasics limit, defaults to $2n_b + 2n_g$
73	MNS_MULTI_PRICE	0 (30)	multiple price

† For OPF_ALG option set to 500 (MINOPF) only.
‡ Default values in parenthesis refer to defaults assigned in MEX file if called with option equal to 0.

Appendix D MATPOWER Files and Functions

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

D.1 Documentation Files

Table D-1: MATPOWER Documentation Files

name	description
`README`, `README.txt`†	basic introduction to MATPOWER
`docs/`	
`CHANGES`, `CHANGES.txt`†	MATPOWER change history
`manual.pdf`	MATPOWER *4.0b2 User's Manual*
`TN1-OPF-Auctions.pdf`	Tech Note 1 *Uniform Price Auctions and Optimal Power Flow*
`TN2-OPF-Derivatives.pdf`	Tech Note 2 *AC Power Flows, Generalized OPF Costs and their Derivatives using Complex Matrix Notation*

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

D.2 MATPOWER Functions

Table D-2: Top-Level Simulation Functions

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

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

Table D-3: Input/Output Functions

name	description
`cdf2matp`	converts data from IEEE Common Data Format 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
`savecase`	saves case data to a MATPOWER case file

Table D-4: Data Conversion Functions

name	description
`ext2int`	converts data from external to internal indexing
`int2ext`	converts data 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: OPF and Wrapper Functions

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

† Can also be used as a top-level function, run directly from the command line. It provides more calling options than `runopf`, primarly for backward compatibility with previous versions of `mopf` from MINOPF, but does not offer the option to save the output or the solved case.

‡ Wrapper with same calling conventions as `opf`.

§ Requires optional MEX-based MINOPF package, available from http://www.pserc.cornell.edu/minopf/.

Table D-7: OPF Model Object

name	description
`@opf_model/`	OPF model object used to encapsulate the OPF problem formulation
`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†
`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
`get_lin_N`*	returns the number of linear constraints†
`get_mpc`	returns the MATPOWER case struct
`get_nln_N`*	returns the number of non-linear constraints†
`get_var_N`*	returns the number of variables†
`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)
`opf_model`	constructor for the `opf_model` class
`userdata`	saves or returns values of user data stored in the model

* Deprecated. Will be removed in a subsequent version.
† For all, or alternatively, only for a named subset.

Table D-8: OPF Solver Functions

name	description
`copf_solver`*	sets up and solves OPF problem using `constr`, Matlab Opt Tbx 1.x & 2.x
`dcopf_solver`	sets up and solves DC OPF problem
`fmincopf_solver`	sets up and solves OPF problem using `fmincon`, Matlab Opt Toolbox
`fmincopf6_solver`*	sets up and solves OPF problem using `fmincon`, Matlab Opt Toolbox¶
`lpopf_solver`*	sets up and solves OPF problem using successive LP-based method
`mipsopf_solver`	sets up and solves OPF problem using MIPS
`mips6opf_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‡

* Deprecated. Will be removed in a subsequent version.
¶ For Matlab 6.x, avoids using handles to anonymous functions.
† Requires optional MEX-based MINOPF package, available from http://www.pserc.cornell.edu/minopf/.
‡ Requires optional MEX-based TSPOPF package, available from http://www.pserc.cornell.edu/tspopf/.

Table D-9: Other OPF Functions

name	description
`fun_copf`*†	evaluates AC OPF objective function and non-linear constraints
`grad_copf`*†	evaluates gradients of AC OPF objective function and non-linear constraints
`LPconstr`*	successive LP-based optimizer, calling conventions similar to `constr`
`LPeqslvr`*	runs Newton power flow, used by `lpopf_solver`
`LPrelax`*	solves LP problem with constraint relaxation, used by `lpopf_solver`
`LPsetup`*	solves LP problem using specified method, used by `lpopf_solver`
`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_consfcn`‡	evaluates function and gradients for AC OPF non-linear 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

* Deprecated. Will be removed in a subsequent version.
† Used by `constr` and `LPconstr` for AC OPF.
‡ Used by `fmincon` and MIPS for AC OPF.

Table D-10: OPF User Callback Functions

name	description
`add_userfcn`	appends a userfcn to the list of those to be called for a given case
`remove_userfcn`	appends a userfcn from the list
`run_userfcn`	executes the userfcn callbacks for a given stage
`toggle_iflims`	enable/disable the callbacks implementing interface flow limits
`toggle_reserves`	enable/disable the callbacks implementing fixed reserve requirements

Table D-11: 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 2[nd] derivatives of $I_{f\|t}$ evaluates the V
`d2Sbr_dV2`	evaluates the 2[nd] derivatives of $S_{f\|t}$ evaluates the V
`d2AIbr_dV2`	evaluates the 2[nd] derivatives of $\|I_{f\|t}\|^2$ evaluates the V
`d2ASbr_dV2`	evaluates the 2[nd] derivatives of $\|S_{f\|t}\|^2$ evaluates the V
`d2Sbus_dV2`	evaluates the 2[nd] 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-12: NLP, LP & QP Solver Functions

name	description
`mips`	Matlab Interior Point Solver – primal/dual interior point solver for NLP
`mips6`[*]	Matlab Interior Point Solver – primal/dual interior point solver for NLP[¶]
`mipsver`	prints version information for MIPS
`mp_lp`[*]	old wrapper function for MATPOWER's LP solvers
`mp_qp`[*]	old wrapper function for MATPOWER's QP solvers
`qps_matpower`	Quadratic Program Solver for MATPOWER, wrapper function provides a common QP solver interface for various QP/LP solvers
`qps_bpmpd`[‡]	common QP solver interface BPMPD_MEX
`qps_mips`	common QP solver interface to MIPS-based solver
`qps_mips6`[*]	common QP solver interface to MIPS-based solver[¶]
`qps_ot`	common QP solver interface to Matlab Opt Toolbox's `quadprog`, `linprog`

[*] Deprecated. Will be removed in a subsequent version.
[¶] For Matlab 6.x, avoids using handles to anonymous functions.
[‡] Requires optional MEX-based BPMPD_MEX package, available from http://www.pserc.cornell.edu/bpmpd/.

Table D-13: 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
`makeLODF`	forms the line outage distribution factor matrix
`makePTDF`	forms the DC PTDF matrix for a given choice of slack
`makeSbus`	forms the vector of complex bus power injections
`makeYbus`	forms the complex bus and branch admittance matrices Y_{bus}, Y_f and Y_t

Table D-14: Utility Functions

name	description
`bustypes`	creates vectors of bus indices for reference bus, PV buses, PQ buses
`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`)
`fairmax`	same as Matlab's `max` function, except it breaks ties randomly
`hasPQcap`	checks for generator P-Q capability curve constraints
`have_fcn`	checks for availability of optional functionality
`idx_area`*	named column index definitions for `areas` matrix
`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_gen`	named column index definitions for `gen` matrix
`isload`	checks if generators are actually dispatchable loads
`mpver`	prints version information for MATPOWER and optional packages
`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

* Deprecated. Will be removed in a subsequent version.

D.3 Example MATPOWER Cases

Table D-15: Example Cases

name	description
`caseformat`	help file documenting MATPOWER case format
`case_ieee30`	IEEE 30-bus case
`case24_ieee_rts`	IEEE RTS 24-bus case
`case4gs`	4-bus example case from Grainger & Stevenson
`case6ww`	6-bus example case from Wood & Wollenberg
`case9`	9-bus example case from Chow
`case9Q`	`case9` with reactive power costs
`case14`	IEEE 14-bus case
`case30`	30-bus case, based on IEEE 30-bus case
`case30pwl`	`case30` with piecewise linear costs
`case30Q`	`case30` with reactive power costs
`case39`	39-bus New England case
`case57`	IEEE 57-bus case
`case118`	IEEE 118-bus case
`case300`	IEEE 300-bus case
`case2383wp`	Polish system - winter 1999-2000 peak
`case2736sp`	Polish system - summer 2004 peak
`case2737sop`	Polish system - summer 2004 off-peak
`case2746wop`	Polish system - winter 2003-04 off-peak
`case2746wp`	Polish system - winter 2003-04 evening peak

D.4 Automated Test Suite

Table D-16: 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-17: Test Data

name	description
t/	
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/o gen cap curves
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 = 1
soln9_opf_PQcap.mat	solution data, AC OPF of t_case9_opfv2 w/o branch ang diff lims
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_int.m	case data used to test ext2int and int2ext, internal indexing
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
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

Table D-18: MATPOWER Tests

name	description
t/	
test_matpower	runs all MATPOWER tests
t_auction_minopf	runs tests for auction using MINOPF†
t_auction_mips	runs tests for auction using MIPS
t_auction_tspopf_pdipm	runs tests for auction using PDIPM‡
t_ext2int2ext	runs tests for ext2int and int2ext
t_hasPQcap	runs tests for hasPQcap
t_hessian	runs tests for 2nd derivative code
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_mips	runs tests for MIPS NLP solver
t_mips6*	runs tests for MIPS NLP solver¶
t_off2case	runs tests for off2case
t_opf_constr*	runs tests for AC OPF solver using constr
t_opf_dc_bpmpd	runs tests for DC OPF solver using BPMPD_MEX§
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_lp_den*	runs tests for AC OPF solver using dense successive LP
t_opf_lp_spf*	runs tests for AC OPF solver using sparse successive LP (full)
t_opf_lp_spr*	runs tests for AC OPF solver using sparse successive LP (sparse)
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_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 userfcn callbacks for reserves and interface flow limits
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_runopf_w_res	runs tests for AC OPF with fixed reserve requirements
t_scale_load	runs tests for scale_load
t_total_load	runs tests for total_load

* Deprecated. Will be removed in a subsequent version.
† Requires optional MEX-based MINOPF package, available from http://www.pserc.cornell.edu/minopf/.
‡ Requires optional MEX-based TSPOPF package, available from http://www.pserc.cornell.edu/tspopf/.
§ Requires optional MEX-based BPMPD_MEX package, available from http://www.pserc.cornell.edu/bpmpd/.
¶ For Matlab 6.x, avoids using handles to anonymous functions.

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.

`cpf_se_intro.pdf`	Brief document introducing continuation power flow and state estimation code contributed by Rui Bo.
`cpf`	Continuation power flow code contributed by Rui Bo. Type `test_cpf` to run an example.
`se`	State-estimation code contributed by Rui Bo. Type `test_se` to run an example.
`smartmarket`	Code that implements a "smart market" auction clearing mechanism based on MATPOWER's optimal power flow solver. See Appendix F for details.
`state_estimator`	Older state estimation example, based on code by James S. Thorp.

Appendix F "Smart Market" Code

MATPOWER 3 and later includes in the `extras/smartmarket` directory code that implements a "smart market" auction clearing mechanism. The purpose of this code is to take a set of offers to sell and bids to buy and use MATPOWER's optimal power flow to compute the corresponding allocations and prices. It has been used extensively by the authors with the optional MINOPF package [15] in the context of POWERWEB[17] 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 5.4.2 on page 34). 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 7).

[17] See http://www.pserc.cornell.edu/powerweb/.

In step 3 the OPF solution is used to determine for each offer/bid block, how much was cleared and at what price. These values are returned in `co` and `cb`, which have the same structure as `offers` and `bids`. The `mkt` parameter is a struct used to specify a number of things about the market, including the type of auction to use, type of OPF (AC or DC) to use and the price limits.

There are two basic types of pricing options available through `mkt.auction_type`, discriminative pricing and uniform pricing. The various uniform pricing options are best explained in the context of an unconstrained lossless network. In this context, the allocation is identical to what one would get by creating bid and offer stacks and finding the intersection point. The nodal prices (λ_P) computed by the OPF and returned in `bus(:,LAM_P)` are all equal to the price of the marginal block. This is either the last accepted offer (LAO) or the last accepted bid (LAB), depending which is the marginal block (i.e. the one that is split by intersection of the offer and bid stacks). There is often a gap between the last accepted bid and the last accepted offer. Since any price within this range is acceptable to all buyers and sellers, we end up with a number of options for how to set the price, as listed in Table F-1.

Table F-1: Auction Types

auction type	name	description
0	discriminative	price of each cleared offer (bid) is equal to the offered (bid) price
1	LAO	uniform price equal to the last accepted offer
2	FRO	uniform price equal to the first rejected offer
3	LAB	uniform price equal to the last accepted bid
4	FRB	uniform price equal to the first rejected bid
5	first price	uniform price equal to the offer/bid price of the marginal unit
6	second price	uniform price equal to min(FRO, LAB) if the marginal unit is an offer, or max(FRB, LAO) if it is a bid
7	split-the-difference	uniform price equal to the average of the LAO and LAB
8	dual LAOB	uniform price for sellers equal to LAO, for buyers equal to LAB

Generalizing to a network with possible losses and congestion results in nodal prices λ_P which vary according to location. These λ_P values can be used to normalize all bids and offers to a reference location by multiplying by a locational scale factor. For bids and offers at bus i, this scale factor is $\lambda_P^{\text{ref}}/\lambda_P^i$, where λ_P^{ref} is the nodal price at the reference bus. The desired uniform pricing rule can then be applied to the adjusted offers and bids to get the appropriate uniform price at the reference bus. This uniform price is then adjusted for location by dividing by the locational

scale factor. The appropriate locationally adjusted uniform price is then used for all cleared bids and offers.[18] The relationships between the OPF results and the pricing rules of the various uniform price auctions are described in detail in [19].

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.

[18] 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:

```
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 install in you Matlab path. Each of them is based on one or more MEX files, pre-compiled for various platforms.

G.1 BPMPD_MEX – MEX interface for BPMPD

BPMPD_MEX [13, 14] is a Matlab MEX interface to BPMPD, an interior point solver for quadratic programming developed by Csaba Mészáros at the MTA SZTAKI, Computer and Automation Research Institute, Hungarian Academy of Sciences, Budapest, Hungary. It can be used by MATPOWER's DC and LP-based OPF solvers and it improves the robustness of MINOPF. It is also useful outside of MATPOWER as a general-purpose QP/LP solver.

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

The current version of the MEX interface is based on version 2.21 of the BPMPD solver, implemented in Fortran.

Builds are available for Linux (32-bit), Mac OS X (PPC, Intel 32-bit) and Windows (32-bit) at `http://www.pserc.cornell.edu/bpmpd/`.

G.2 MINOPF – AC OPF Solver Based on MINOS

MINOPF [15] is a MINOS-based optimal power flow solver for use with MATPOWER. It is for educational and research use only. MINOS [16] 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_ALG` option to 500 (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.3 TSPOPF – Three AC OPF Solvers by H. Wang

TSPOPF [10] 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 [11, 18]. The first two are essentially C-language implementations of the algorithms used by MIPS (see Appendix A), with the exception that the step-controlled version in TSPOPF also includes a cost smoothing technique in place of the constrained-cost variable (CCV) approach for handling piece-wise linear costs.

The PDIPM in particular is significantly faster for large systems than any previous MATPOWER AC OPF solver, including MINOPF. When TSPOPF is installed, the PDIPM solver becomes the default optimal power flow solver for MATPOWER. Additional options for TSPOPF can be set using `mpoption` (see `help mpoption` for details).

Builds are available for Linux (32-bit, 64-bit), Mac OS X (PPC, Intel 32-bit, Intel 64-bit) and Windows (32-bit, 64-bit) at `http://www.pserc.cornell.edu/tspopf/`.

References

[1] 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," *submitted to IEEE Transactions on Power Systems*. (update upon publication) 3

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

[3] 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

[4] 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

[5] 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

[6] 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

[7] A. J. Wood and B. F. Wollenberg, *Power Generation, Operation, and Control*, 2nd ed. New York: J. Wiley & Sons, 1996. 4.2, 4.4

[8] 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

[9] 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. 1.3, 5.3

[10] TSPOPF: [Online]. Available: http://www.pserc.cornell.edu/tspopf/ 5.4.1, 5.5, G.3

[11] 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. 5.4.1, 5.5, A, A.4, G.3

[12] *Optimization Toolbox 4 Users's Guide*. The MathWorks, Inc., 2008. [Online]. Available: http://www.mathworks.com/access/helpdesk/help/pdf_doc/optim/optim_tb.pdf 5.5

[13] BPMPD_MEX: [Online]. Available: http://www.pserc.cornell.edu/bpmpd/ 5.5, G.1

[14] C. Mészáros, *The Efficient Implementation of Interior Point Methods for Linear Programming and their Applications*, Ph.D. thesis, Eötvös Loránd University of Sciences, 1996. 5.5, G.1

[15] MINOPF: [Online]. Available: http://www.pserc.cornell.edu/minopf/ 5.5, F, G.2

[16] B. A. Murtagh and M. A. Saunders, *MINOS 5.5 User's Guide*, Stanford University Systems Optimization Laboratory Technical Report SOL83-20R. 5.5, G.2

[17] 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 5.5

[18] 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.3

[19] R. D. Zimmerman, *Uniform Price Auctions and Optimal Power Flow*, MATPOWER Technical Note 1, February 2010. [Online]. Available: http://www.pserc.cornell.edu/matpower/TN1-OPF-Auctions.pdf F