MATPOWER

A MATLAB™ Power System Simulation Package

Version 3.1b1

August 1, 2006

User's Manual

Ray D. Zimmerman *rz10@cornell.edu*
Carlos E. Murillo-Sánchez *carlos_murillo@ieee.org*
Deqiang (David) Gan *dgan@zju.edu.cn*

Table of Contents

1 Introduction

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/

Where did it come from?

MATPOWER was developed by Ray D. Zimmerman, Carlos E. Murillo-Sánchez and Deqiang Gan of PSERC at Cornell University (http://www.pserc.cornell.edu/) under the direction of Robert 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 (see http://www.pserc.cornell.edu/powerweb/).

Who can 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 cite *MATPOWER* http://www.pserc.cornell.edu/matpower/.
- 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.

2 Getting Started

2.1 System Requirements

To use *MATPOWER* you will need:

- MATLAB version 5 or later[1]
- MATLAB Optimization Toolbox (required only for some OPF algorithms)

Both are available from The MathWorks (see http://www.mathworks.com/).

[1] MATPOWER 2.0 and earlier required only version 4 of Matlab.

2.2 Installation

Step 1: Go to the *MATPOWER* home page (http://www.pserc.cornell.edu/matpower/) and follow the download instructions.

Step 2: Unzip the downloaded file.

Step 3: Place the files in a location in your MATLAB path.

2.3 Running a Power Flow

To run a simple Newton power flow on the 9-bus system specified in the file `case9.m`, with the default algorithm options, at the MATLAB prompt, type:

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

2.4 Running an Optimal Power Flow

To run an optimal power flow on the 30-bus system whose data is in `case30.m`, with the default algorithm options, at the MATLAB prompt, type:

```
>> runopf('case30')
```

To run an optimal power flow on the same system, but with the option for *MATPOWER* to shut down (decommit) expensive generators, type:

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

2.5 Getting Help

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. For example, the help for `runopf` looks like:

```
>> help runopf
 RUNOPF  Runs an optimal power flow.

    [baseMVA, bus, gen, gencost, branch, f, success, et] = ...
            runopf(casename, mpopt, fname, solvedcase)

    Runs an optimal power flow and optionally returns the solved values in
    the data matrices, the objective function value, a flag which is true if
    the algorithm was successful in finding a solution, and the elapsed time
    in seconds. All input arguments are optional. If casename is provided it
    specifies the name of the input data file or struct (see also 'help
    caseformat' and 'help loadcase') containing the opf data. The default
    value is 'case9'. If the mpopt is provided it overrides the default
    MATPOWER options vector and can be used to specify the solution
    algorithm and output options among other things (see 'help mpoption' for
    details). If the 3rd argument is given the pretty printed output will be
    appended to the file whose name is given in fname. If solvedcase is
    specified the solved case will be written to a case file in MATPOWER
    format with the specified name. If solvedcase ends with '.mat' it saves
    the case as a MAT-file otherwise it saves it as an M-file.
```

MATPOWER also has many options which control the algorithms and the output. Type:

```
>> help mpoption
```

and see *Section 3.6* for more information on *MATPOWER*'s options.

3 Technical Reference

3.1 Data File Format

The data files used by *MATPOWER* are simply MATLAB M-files or MAT-files which define and return the variables `baseMVA`, `bus`, `branch`, `gen`, `areas`, and `gencost`. The `baseMVA` variable is a scalar and the rest are matrices. Each row in the matrix corresponds to a single bus, branch, or generator. The columns are similar to the columns in the standard IEEE and PTI formats. The details of the specification of the *MATPOWER* case file can be found in the help for `caseformat.m`:

```
>> help caseformat

 CASEFORMAT    Defines the MATPOWER case file format.
    A MATPOWER case file is an M-file or MAT-file which defines the variables
    baseMVA, bus, gen, branch, areas, and gencost. With the exception of
    baseMVA, a scalar, each data variable is a matrix, where a row corresponds
    to a single bus, branch, gen, etc. The format of the data is similar to
    the PTI format described in
        http://www.ee.washington.edu/research/pstca/formats/pti.txt
    except where noted. An item marked with (+) indicates that it is included
    in this data but is not part of the PTI format. An item marked with (-) is
    one that is in the PTI format but is not included here. Those marked with
    (2) were added for version 2 of the case file format. The columns for
    each data matrix are given below.

    MATPOWER Case Version Information:
    A version 1 case file defined the data matrices directly. The last two,
    areas and gencost, were optional since they were not needed for running
    a simple power flow. In version 2, each of the data matrices are stored
    as fields in a struct. It is this struct, rather than the individual
    matrices that is returned by a version 2 M-casefile. Likewise a version 2
    MAT-casefile stores a struct named 'mpc' (for MATPOWER case). The struct
    also contains a 'version' field so MATPOWER knows how to interpret the
    data. Any case file which does not return a struct, or any struct which
    does not have a 'version' field is considered to be in version 1 format.

    See also IDX_BUS, IDX_BRCH, IDX_GEN, IDX_AREA and IDX_COST regarding
    constants which can be used as named column indices for the data matrices.
    Also described in the first three are additional columns that are added
    to the bus, branch and gen matrices by the power flow and OPF solvers.
```

```
Bus Data Format
     1   bus number (1 to 29997)
     2   bus type
             PQ bus          = 1
             PV bus          = 2
             reference bus   = 3
             isolated bus    = 4
     3   Pd, real power demand (MW)
     4   Qd, reactive power demand (MVAr)
     5   Gs, shunt conductance (MW (demanded) at V = 1.0 p.u.)
     6   Bs, shunt susceptance (MVAr (injected) at V = 1.0 p.u.)
     7   area number, 1-100
     8   Vm, voltage magnitude (p.u.)
     9   Va, voltage angle (degrees)
 (-)     (bus name)
     10  baseKV, base voltage (kV)
     11  zone, loss zone (1-999)
 (+) 12  maxVm, maximum voltage magnitude (p.u.)
 (+) 13  minVm, minimum voltage magnitude (p.u.)

 Generator Data Format
     1   bus number
 (-)     (machine identifier, 0-9, A-Z)
     2   Pg, real power output (MW)
     3   Qg, reactive power output (MVAr)
     4   Qmax, maximum reactive power output (MVAr)
     5   Qmin, minimum reactive power output (MVAr)
     6   Vg, voltage magnitude setpoint (p.u.)
 (-)     (remote controlled bus index)
     7   mBase, total MVA base of this machine, defaults to baseMVA
 (-)     (machine impedance, p.u. on mBase)
 (-)     (step up transformer impedance, p.u. on mBase)
 (-)     (step up transformer off nominal turns ratio)
     8   status,  >  0 - machine in service
                 <= 0 - machine out of service
 (-)     (% of total VAr's to come from this gen in order to hold V at
             remote bus controlled by several generators)
     9   Pmax, maximum real power output (MW)
     10  Pmin, minimum real power output (MW)
 (2) 11  Pc1, lower real power output of PQ capability curve (MW)
 (2) 12  Pc2, upper real power output of PQ capability curve (MW)
 (2) 13  Qc1min, minimum reactive power output at Pc1 (MVAr)
 (2) 14  Qc1max, maximum reactive power output at Pc1 (MVAr)
 (2) 15  Qc2min, minimum reactive power output at Pc2 (MVAr)
 (2) 16  Qc2max, maximum reactive power output at Pc2 (MVAr)
 (2) 17  ramp rate for load following/AGC (MW/min)
 (2) 18  ramp rate for 10 minute reserves (MW)
 (2) 19  ramp rate for 30 minute reserves (MW)
 (2) 20  ramp rate for reactive power (2 sec timescale) (MVAr/min)
 (2) 21  APF, area participation factor
```

```
  Branch Data Format
      1   f, from bus number
      2   t, to bus number
  (-)     (circuit identifier)
      3   r, resistance (p.u.)
      4   x, reactance (p.u.)
      5   b, total line charging susceptance (p.u.)
      6   rateA, MVA rating A (long term rating)
      7   rateB, MVA rating B (short term rating)
      8   rateC, MVA rating C (emergency rating)
      9   ratio, transformer off nominal turns ratio ( = 0 for lines )
          (taps at 'from' bus, impedance at 'to' bus, i.e. ratio = Vf / Vt)
      10  angle, transformer phase shift angle (degrees)
  (-)     (Gf, shunt conductance at from bus p.u.)
  (-)     (Bf, shunt susceptance at from bus p.u.)
  (-)     (Gt, shunt conductance at to bus p.u.)
  (-)     (Bt, shunt susceptance at to bus p.u.)
      11  initial branch status, 1 - in service, 0 - out of service
  (2) 12  minimum angle difference, angle(Vf) - angle(Vt) (degrees)
  (2) 13  maximum angle difference, angle(Vf) - angle(Vt) (degrees)

(+) Area Data Format
      1   i, area number
      2   price_ref_bus, reference bus for that area

(+) Generator Cost Data Format
      NOTE: If gen has n rows, then the first n rows of gencost contain
      the cost for active power produced by the corresponding generators.
      If gencost has 2*n rows then rows n+1 to 2*n contain the reactive
      power costs in the same format.
      1   model, 1 - piecewise linear, 2 - polynomial
      2   startup, startup cost in US dollars
      3   shutdown, shutdown cost in US dollars
      4   n, number of cost coefficients to follow for polynomial
          cost function, or number of data points for piecewise linear
      5 and following, cost data defining total cost function
          For polynomial cost:
                  c2, c1, c0
          where the polynomial is c0 + c1*P + c2*P^2
          For piecewise linear cost:
                  x0, y0, x1, y1, x2, y2, ...
          where x0 < x1 < x2 < ... and the points (x0,y0), (x1,y1),
          (x2,y2), ... are the end- and break-points of the cost function.
```

Some columns are added to the `bus`, `branch` and `gen` matrices by the solvers. See the help for `idx_bus`, `idx_brch`, and `idx_gen` for more details.

3.2 Modeling

AC Formulation

Fixed loads are modeled as constant real and reactive power injections, P_d and Q_d specified in columns 3 and 4, respectively, of the `bus` matrix. The shunt admittance of any constant impedance shunt elements at a bus are specified by G_{sh} and B_{sh} in columns 5 and 6, respectively, of the `bus` matrix

$$Y_{sh} = \frac{G_{sh} + jB_{sh}}{\text{baseMVA}}$$

Each branch, whether transmission line, transformer or phase shifter, is modeled as a standard π-model transmission line, with series resistance R and reactance X and total line charging capacitance B_c, in series with an ideal transformer and phase shifter, at the from end, with tap ratio τ and phase shift angle θ_{shift}. The parameters R, X, B_c, τ and θ_{shift}, are found in columns 3, 4, 5, 9 and 10 of the `branch` matrix, respectively. The branch voltages and currents at the *from* and *to* ends of the branch are related by the branch admittance matrix Y_{br} as follows

$$\begin{bmatrix} I_f \\ I_t \end{bmatrix} = \mathbf{Y_{br}} \begin{bmatrix} V_f \\ V_t \end{bmatrix} \tag{1}$$

where $\mathbf{Y_{br}} = \begin{bmatrix} \left(Y_s + j\frac{B_c}{2}\right)\frac{1}{\tau^2} & -Y_s \frac{1}{\tau e^{j\theta_{shift}}} \\ -Y_s \frac{1}{\tau e^{-j\theta_{shift}}} & Y_s + j\frac{B_c}{2} \end{bmatrix}$ and $Y_s = \frac{1}{R + jX}$.

The elements of the individual branch admittance matrices and the bus shunt admittances are combined by *MATPOWER* to form a complex bus admittance matrix $\mathbf{Y_{bus}}$, relating the vector of complex bus voltages $\mathbf{V_{bus}}$ with the vector of complex bus current injections $\mathbf{I_{bus}}$

$$\mathbf{I_{bus}} = \mathbf{Y_{bus}} \mathbf{V_{bus}}$$

Similarly, admittance matrices $\mathbf{Y_f}$ and $\mathbf{Y_t}$, are formed to compute the vector of complex current injections at the from and to ends of each line, given the bus voltages $\mathbf{V_{bus}}$.

$$\mathbf{I_f} = \mathbf{Y_f} \mathbf{V_{bus}}$$
$$\mathbf{I_t} = \mathbf{Y_t} \mathbf{V_{bus}}$$

The vectors of complex bus power injections, and branch power injections can be expressed as

$$\mathbf{S_{bus}} = \text{diag}(\mathbf{V_{bus}})\mathbf{I_{bus}^*}$$
$$\mathbf{S_f} = \text{diag}(\mathbf{V_f})\mathbf{I_f^*}$$
$$\mathbf{S_t} = \text{diag}(\mathbf{V_t})\mathbf{I_t^*}$$

where $\mathbf{V_f}$ and $\mathbf{V_t}$ are vectors of the complex bus voltages at the *from* and *to* ends, respectively, of all branches, and diag() converts a vector into a diagonal matrix with the specified vector on the diagonal.

DC Formulation

For the DC formulation, the same parameters are used, with the exception that the following assumptions are made:

- Branch resistances R and charging capacitances B_c are negligible (i.e. branches are lossless).
- All bus voltage magnitudes are close to 1 p.u.
- Voltage angle differences are small enough that $\sin\theta_{ij} \approx \theta_{ij}$.

Combining these assumptions and equation (1) with the fact that $S = VI^*$, the relationship between the real power flows and voltage angles for an individual branch can be written as

$$\begin{bmatrix} P_f \\ P_t \end{bmatrix} = \mathbf{B_{br}} \begin{bmatrix} \theta_f \\ \theta_t \end{bmatrix} + \begin{bmatrix} P_{f,shift} \\ P_{t,shift} \end{bmatrix} \tag{2}$$

where

$$B_{br} = \frac{1}{X\tau} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \tag{3}$$

$$\begin{bmatrix} P_{f,shift} \\ P_{t,shift} \end{bmatrix} = \frac{\theta_{shift}}{X\tau} \begin{bmatrix} 1 \\ -1 \end{bmatrix}. \tag{4}$$

The elements of the individual branch shift injections and $\mathbf{B_{br}}$ matrices are combined by *MATPOWER* to form a bus $\mathbf{B_{bus}}$ matrix and $\mathbf{P_{bus,shift}}$ shift injection vector, which can be used to compute bus real power injections from bus voltage angles

$$\mathbf{P_{bus}} = \mathbf{B_{bus}} \boldsymbol{\theta}_{\mathbf{bus}} + \mathbf{P_{bus,shift}}$$

Similarly, *MATPOWER* builds the matrix $\mathbf{B_f}$ and the vector $\mathbf{P_{f,shift}}$ which can be used to compute the vector s $\mathbf{P_f}$ and $\mathbf{P_t}$ of branch real power injections

$$\mathbf{P_f} = \mathbf{B_f} \boldsymbol{\theta}_{\mathbf{bus}} + \mathbf{P_{f,shift}}$$
$$\mathbf{P_t} = -\mathbf{P_f}$$

3.3 Power Flow

MATPOWER has five power flow solvers, which can be accessed via the `runpf` function. In addition to printing output to the screen, which it does by default, `runpf` optionally returns the solution in output arguments:

```
>> [baseMVA, bus, gen, branch, success, et] = runpf(casename);
```

The solution values are stored as follows:

`bus(:, VM)`	bus voltage magnitudes
`bus(:, VA)`	bus voltage angles
`gen(:, PG)`	generator real power injections
`gen(:, QG)`	generator reactive power injections
`branch(:, PF)`	real power injected into "from" end of branch
`branch(:, PT)`	real power injected into "to" end of branch
`branch(:, QF)`	reactive power injected into "from" end of branch
`branch(:, QT)`	reactive power injected into "to" end of branch
`success`	1 = solved successfully, 0 = unable to solve
`et`	computation time required for solution

The default power flow solver is based on a standard Newton's method [12] using a full Jacobian, updated at each iteration. This method is described in detail in many textbooks. Algorithms 2 and 3 are variations of the fast-decoupled method [10]. *MATPOWER* implements the XB and BX variations as described in [1]. Algorithm 4 is the standard Gauss-Seidel method from Glimm and Stagg [5], based on code contributed by Alberto Borghetti, from the University of Bologna, Italy. To use one of the power flow solvers other than the default Newton method, the PF_ALG option must be set explicitly. For example, for the XB fast-decoupled method:

```
>> mpopt = mpoption('PF_ALG', 2);
>> runpf(casename, mpopt);
```

The last method is a DC power flow [13], which is obtained by executing `runpf` with the `PF_DC` option set to 1, or equivalently by executing `rundcpf` directly. The DC power flow is obtained by a direct, non-iterative solution of the bus voltage angles from the specified bus real power injections, based on equations (2), (3) and (4).

For the AC power flow solvers, if the `ENFORCE_Q_LIMS` option is set to true (default is false), then if any generator reactive power limit is violated after running the AC power flow, the corresponding bus is converted to a PQ bus, with the reactive output set to the limit, and the case is re-run. The voltage magnitude at the bus will deviate from the specified value in order to satisfy the reactive power limit. If the generator at the reference bus is reaches a reactive power limit and the bus is converted to a PQ bus, the first remaining PV bus will be used as the slack bus for the next iteration. This may result in the real power output at this generator being slightly off from the specified values.

Currently, none of *MATPOWER*'s power flow solvers include any transformer tap changing or handling of disconnected or de-energized sections of the network.

Performance of the power flow solvers, with the exception of Gauss-Seidel, should be excellent even on very large-scale power systems, since the algorithms and implementation take advantage of MATLAB's built-in sparse matrix handling.

3.4 Optimal Power Flow

MATPOWER includes several solvers for the optimal power flow (OPF) problem, which can be accessed via the `runopf` function. In addition to printing output to the screen, which it does by default, `runopf` optionally returns the solution in output arguments:

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

In addition to the values listed for the power flow solvers, the OPF solution also includes the following values:

`bus(:, LAM_P)`	Lagrange multiplier on bus real power mismatch
`bus(:, LAM_Q)`	Lagrange multiplier on bus reactive power mismatch
`bus(:, MU_VMAX)`	Kuhn-Tucker multiplier on upper bus voltage limit
`bus(:, MU_VMIN)`	Kuhn-Tucker multiplier on lower bus voltage limit
`gen(:, MU_PMAX)`	Kuhn-Tucker multiplier on upper generator real power limit
`gen(:, MU_PMIN)`	Kuhn-Tucker multiplier on lower generator real power limit
`gen(:, MU_QMAX)`	Kuhn-Tucker multiplier on upper generator reactive power limit
`gen(:, MU_QMIN)`	Kuhn-Tucker multiplier on lower generator reactive power limit
`branch(:, MU_SF)`	Kuhn-Tucker multiplier on MVA limit at "from" end of branch
`branch(:, MU_ST)`	Kuhn-Tucker multiplier on MVA limit at "to" end of branch
`f`	final objective function value

The (chronologically) first of the OPF solvers in *MATPOWER* is based on the `constr` function included in earlier versions of MATLAB's Optimization Toolbox, which uses a successive quadratic programming technique with a quasi-Newton approximation for the Hessian matrix. The second approach is based on linear programming. It can use the LP solver in the Optimization Toolbox or other MATLAB LP solvers available from third parties. Version 3 of *MATPOWER* has a new generalized OPF formulation that allows general linear constraints on the optimization variables, but requires `fmincon.m` found in MATLAB's Optimization Toolbox 2.0 or later, or the *MINOS* [14] based MEX file available separately as part of the optional *MINOPF* package (see http://www.pserc.cornell.edu/minopf/). *MINOPF* is distributed separately because it has a more restrictive license than *MATPOWER*.

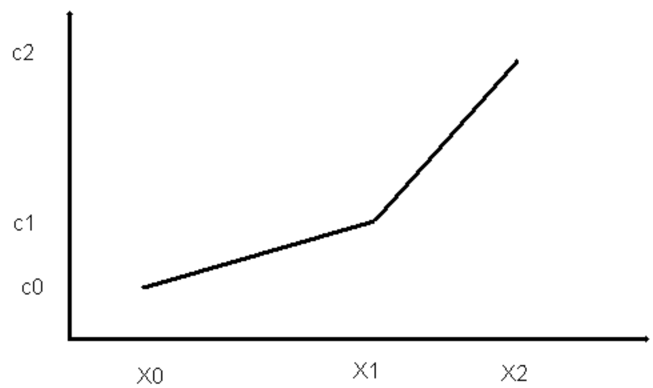
The performance of *MATPOWER*'s OPF solvers depends on several factors. First, the `constr` function uses an algorithm which does not exploit or preserve sparsity, so it is inherently limited to small power systems. The same is still true for the combination of parameters required to be able to employ the newer `fmincon` function. The LP-based algorithm, on the other hand, does preserve sparsity. However, the LP-solver included in the older Optimization Toolbox does not exploit this sparsity. In fact, the LP-based method with the old LP solver performs worse than the `constr`-based method, even on small systems. Fortunately, there are LP-solvers available from third parties which do exploit sparsity. In general, these yield *much* higher performance. One in particular, called *BPMPD* [8] (actually a QP-solver), has proven to be robust and efficient. Even the `constr` or `fmincon`-based methods, when tricked into calling *BPMPD* with full matrix data instead of the older `qp.m`, become much faster.

It should be noted, however, that even with a good LP-solver, *MATPOWER*'s LP-based OPF solver, unlike its power flow solver, is not suitable for very-large scale problems. Substantial improvements in performance may still be possible, though they may require significantly more complicated coding and possibly a custom LP-solver. However, when speed is of the essence, the preferred choice is the *MINOS*-based MEX file solver; assuming that its licensing requirements can be met. It is coded in FORTRAN and evaluates the required Jacobians using an optimized structure that follows the order of evaluation imposed by the compressed-column sparse format which is employed by *MINOS*. In fact, the new generalized OPF formulation included in *MATPOWER* 3.0 and later is inspired by the data format used by *MINOS*.

MATPOWER's OPF implementation is not currently able to handle unconnected or de-energized sections of the network.

Piecewise linear costs using constrained cost variables (CCV)

The OPF formulations in *MATPOWER* allow for the specification of convex piecewise linear cost functions for active or reactive generator output. An example of such a cost curve is shown below.



This non-differentiable cost is modeled using an extra helper cost variable for each such cost curve and additional constraints on this variable and *Pg*, one for each segment of the curve. The constraints build a convex "basin" equivalent to requiring the cost variable to lie in the epigraph of the cost curve. When the cost is minimized, the cost variable will be pushed against this basin. If the helper cost variable is *y*, then the contribution of the generator's cost to the total cost is exactly *y*. In the above case, the two additional required constraints are

1) $y \geq m_1(P_g - x_0) + c_0$ (*y* must lie above the first segment)

2) $y \geq m_2(P_g - x_1) + c_1$ (*y* must lie above the second segment)

where m_1 and m_2 are the slopes of the two segments. Also needed, of course, are the box restrictions on P_g: $P_{min} \leq P_g \leq P_{max}$. The additive part of the cost contributed by this generator is *y*.

This constrained cost variable (CCV) formulation is used by all of the *MATPOWER* OPF solvers for handling piecewise linear cost functions.

3.4.1 Traditional AC OPF Formulation

The AC optimal power flow problem solved by *MATPOWER* is a "smooth" OPF with no discrete variables or controls. The objective function is the total cost of real and/or reactive generation. These costs may be defined as polynomials or as piecewise-linear functions of generator output. The problem is formulated as follows:

$$\min_{\theta,V,P_g,Q_g} \sum_i f_{1i}(P_{gi}) + f_{2i}(Q_{gi})$$

subject to

$$P_i(\theta,V) - P_{gi} + P_{di} = 0 \qquad \text{(active power balance equations)}$$

$$Q_i(\theta,V) - Q_{gi} + Q_{di} = 0 \qquad \text{(reactive power balance equations)}$$

$$\left|S_{ij}^f(\theta,V)\right| \le S_{ij}^{\max} \qquad \text{(apparent power flow limit of lines, \textit{from} end)}$$

$$\left|S_{ij}^t(\theta,V)\right| \le S_{ij}^{\max} \qquad \text{(apparent power flow limit of lines, \textit{to} end)}$$

$$V_i^{\min} \le V_i \le V_i^{\max} \qquad \text{(bus voltage limits)}$$

$$P_{gi}^{\min} \le P_{gi} \le P_{gi}^{\max} \qquad \text{(active power generation limits)}$$

$$Q_{gi}^{\min} \le Q_{gi} \le Q_{gi}^{\max} \qquad \text{(reactive power generation limits)}$$

Here f_{1i} and f_{2i} are the costs of active and reactive power generation, respectively, for generator i at a given dispatch point. Both f_{1i} and f_{2i} are assumed to be polynomial or piecewise-linear functions. By defining the variable x as

$$x = \begin{bmatrix} \theta \\ V \\ P_g \\ Q_g \end{bmatrix}$$

the problem can be expressed compactly as follows:

$$\min_x f(x)$$

subject to

$$g_1(x) = 0 \qquad \text{(power balance equations)}$$

$$g_2(x) \le 0 \qquad \text{(branch flow limits)}$$

$$x_{\min} \le x \le x_{\max} \qquad \text{(variable limits)}$$

Optimization Toolbox Based OPF Solver (`constr`)

The first of the two original OPF solvers in *MATPOWER* is based on the `constr` non-linear constrained optimization function in MATLAB's Optimization Toolbox. The `constr` function and the algorithms it uses are covered in the older Optimization Toolbox manual [6]. *MATPOWER* provides `constr` with two M-files which it uses during for the optimization. One computes the objective function, f, and the constraint violations, g, at a given point, x, and the other computes their gradients $\partial f / \partial x$ and $\partial g / \partial x$.

MATPOWER has two versions of these M-files. One set is used to solve systems with polynomial cost functions. In this formulation, the cost functions are included in a straightforward way into the objective function. The other set is used to solve systems with piecewise-linear costs. Piecewise-linear cost functions are handled by introducing a cost variable for each piecewise-linear cost function. The objective function is simply the sum of these cost variables which are then constrained to lie above each of the linear functions which make up the piecewise-linear cost function. Clearly, this method works only for convex cost functions. In the *MATPOWER* documentation this will be referred to as a constrained cost variable (CCV) formulation.

The algorithm codes 100 and 200, respectively, are used to identify the `constr`-based solver for polynomial and piecewise-linear cost functions. If algorithm 200 is chosen for a system with polynomial cost function, the cost function will be approximated by a piecewise-linear function by evaluating the polynomial at a fixed number of points determined by the options vector (see Section 3.6 for more details on the *MATPOWER* options).

It should be noted that the `constr`-based method can also benefit from a superior QP-solver such as *bpmpd*. See Appendix A for more information on LP and QP-solvers.

LP-Based OPF Solver (`LPconstr`)

Linear programming based OPF methods are in wide use today in the industry. However, the LP-based algorithm included in *MATPOWER* is much simpler than the algorithms used in production-grade software.

The LP-based methods in *MATPOWER* use the same problem formulation as the `constr`-based methods, including the CCV formulation for the case of piecewise-linear costs. The compact form of the OPF problem can be rewritten to partition g into equality and inequality constraints, and to partition the variable x as follows:

$$\min_x f(x_2)$$

subject to

$$g_1(x_1, x_2) = 0 \qquad \text{(equality constraints)}$$

$$g_2(x_1, x_2) \le 0 \qquad \text{(inequality constraints)}$$

where x_1 contains the system voltage magnitudes and angles, and x_2 contains the generator real and reactive power outputs (and corresponding cost variables for the CCV formulation). This is a general nonlinear programming problem, with the additional assumption that the equality constraints can be used to solve for x_1, given a value for x_2.

The LP-based OPF solver is implemented with a function `LPconstr`, which is similar to `constr` in that it uses the same M-files for computing the objective function, constraints, and their respective gradients. In addition, a third M-file (`lpeqslvr.m`) is needed to solve for x_1 from the equality constraints, given a value for x_2. This architecture makes it relatively simple to modify the formulation of the problem and still be able to use both the `constr`-based and LP-based solvers.

The algorithm proceeds as follows, where the superscripts denote iteration number:

Step 0: Set iteration counter $k \leftarrow 0$ and choose an appropriate initial value, call it x_2^0, for x_2.

Step 1: Solve the equality constraint (power flow) equations $g_1(x_1^k, x_2^k) = 0$ for x_1^k.

Step 2: Linearize the problem around x^k, solve the resulting LP for Δx.

$$\min_{\Delta x} \left[\left.\frac{\partial f}{\partial x}\right|_{x=x^k} \right] \cdot \Delta x$$

subject to

$$\left[\left.\frac{\partial g}{\partial x}\right|_{x=x^k} \right] \cdot \Delta x \le -g(x^k)$$

$$-\Delta \le \Delta x \le \Delta$$

Step 3: Set $k \leftarrow k+1$, update current solution $x^k = x^{k-1} + \Delta x$.

Step 4: If x^k meets termination criteria, stop, otherwise go to step 5.

Step 5: Adjust step size limit Δ based on the trust region algorithm in [3], go to step 1.

The termination criteria is outlined below:

$$\frac{\partial L}{\partial x} = \frac{\partial f}{\partial x} + \lambda^T \cdot \frac{\partial g}{\partial x} \le \text{tolerance}_1$$

$$g(x) \le \text{tolerance}_2$$

$$\Delta x \le \text{tolerance}_3$$

Here λ is the vector of Lagrange multipliers of the LP problem. The first condition pertains to the size of the gradient, the second to the violation of constraints, and the third to the step size. More detail can be found in [4].

Quite frequently, the value of x^k given by step 1 is infeasible and could result in an infeasible LP problem. In such cases, a slack variable is added for each violated constraint. These slack variables must be zero at the optimal solution.

The `LPconstr` function implements the following three methods:

- sparse formulation with full set of inequality constraints
- sparse formulation with relaxed constraints (ICS, Iterative Constraint Search)
- dense formulation with relaxed constraints (ICS) [11]

These three methods are specified using algorithm codes 160, 140, and 120, respectively, for systems with polynomial costs, and 260, 240, and 220, respectively, for systems with piecewise-linear costs. As with the *constr*-based method, selecting one of the 2xx algorithms for a system with polynomial cost will cause the cost to be replaced by a piecewise-linear approximation.

In the dense formulation, some of the variables x_1 and the equality constraints g_1 are eliminated from the problem before posing the LP sub-problem. This procedure is outlined below. Suppose the LP sub-problem is given by:

$$\min \; c^T \cdot \Delta x$$

subject to

$$A \cdot \Delta x \le b$$

$$-\Delta \le \Delta x \le \Delta$$

If this is rewritten as:

$$\min \; c_1^T \cdot \Delta x_1 + c_2^T \cdot \Delta x_2$$

subject to

$$A_{11} \cdot \Delta x_1 + A_{12} \cdot \Delta x_2 = b_1$$

$$A_{21} \cdot \Delta x_1 + A_{22} \cdot \Delta x_2 \le b_2$$

$$-\Delta \le \Delta x \le \Delta$$

where A_{11} is a square matrix, Δx_1 can be computed as:

$$\Delta x_1 = A_{11}^{-1}(b_1 - A_{12}\Delta x_2)$$

Substituting back in to the problem, yields a new LP problem:

$$\min \; \left(-c_1^T A_{11}^{-1} A_{12} + c_2^T\right) \cdot \Delta x_2$$

subject to

$$A_{11} \cdot \Delta x_1 + A_{12} \cdot \Delta x_2 = b_1$$

$$A_{21} \cdot A_{11}^{-1}(b_1 - A_{12}\Delta x_2) + A_{22} \cdot \Delta x_2 \le b_2$$

$$-\Delta_1 \le A_{11}^{-1}(b_1 - A_{12}\Delta x_2) \le \Delta_1$$

$$-\Delta_2 \le \Delta x_2 \le \Delta_2$$

This new LP problem is smaller than the original, but it is no longer sparse.

As mentioned above, to realize the full potential of the LP-based OPF solvers, it will be necessary to obtain a good LP-solver, such as *bpmpd*. See Appendix A for more details.

3.4.2 Generalized AC OPF Formulation (`fmincon` and MINOPF)

The generalized AC OPF formulation, used by the `fmincon` and *MINOPF* solvers, offers a number of extra capabilities relative to the traditional formulation used by the classic *MATPOWER* OPF solvers:

- mixed polynomial and piecewise linear costs
- dispatchable loads
- generator P-Q capability curves
- additional user supplied linear constraints
- additional user supplied costs

New in *MATPOWER* 3.1 are the generalized user supplied cost formulation, the generator capability curves and a simplification of the general linear constraint specification used in version 3.0.

The problem is formulated in terms of 3 groups of optimization variables, labeled x, y and z. The x variables are the OPF variables, consisting of the voltage angles θ and magnitudes V at each bus, and real and reactive generator injections P_g and Q_g. The y variables are the helper variables used by the CCV formulation of the piecewise linear generator cost functions. Additional user defined variables are grouped in z.

The optimization problem can be expressed as follows:

$$\min_{x,y,z} \sum_i \left(f_{1i}(P_{gi}) + f_{2i}(Q_{gi}) \right) + \sum_i y_i + \frac{1}{2} w^T H w + C_w^T w$$

subject to

$$g_P(x) = P(\theta,V) - P_g + P_d = 0 \qquad \text{(active power balance equations)}$$

$$g_Q(x) = Q(\theta,V) - Q_g + Q_d = 0 \qquad \text{(reactive power balance equations)}$$

$$g_{S_f}(x) = \left| S_f(\theta,V) \right| - S_{\max} \le 0 \qquad \text{(apparent power flow limit of lines, \textit{from} end)}$$

$$g_{S_t}(x) = \left| S_t(\theta,V) \right| - S_{\max} \le 0 \qquad \text{(apparent power flow limit of lines, \textit{to} end)}$$

$$l \le A \begin{bmatrix} x \\ y \\ z \end{bmatrix} \le u \qquad \text{(general linear constraints)}$$

$$x_{\min} \le x \le x_{\max} \qquad \text{(voltage and generation variable limits)}$$

The most significant additions to the traditional, simple OPF formulation appear in the generalized cost terms containing w and in the general linear constraints involving the matrix A, described in the next two sections. These two frameworks allow tremendous flexibility in customizing the problem formulation, making *MATPOWER* even more useful as a research tool.

Note: In Optimization Toolbox versions 3.0 and earlier, `fmincon` *seems to be providing inaccurate shadow prices on the constraints. This did not happen with* `constr` *and it may be a bug in these versions of the Optimization Toolbox.*

General Linear Constraints

In addition to the standard non-linear equality constraints for nodal power balance and non-linear inequality constraints for line flow limits, this formulation includes a framework for additional linear constraints involving the full set of optimization variables.

$$l \le A \begin{bmatrix} x \\ y \\ z \end{bmatrix} \le u \qquad \text{(general linear constraints)}$$

Some portions of these linear constraints are supplied directly by the user, while others are generated automatically based on the case data. Automatically generated portions include:

- rows for constraints that define generator P-Q capability curves
- rows for constant power factor constraints for dispatchable or price-sensitive loads
- rows and columns for the y variables from the CCV implementation of piecewise linear generator costs and their associated constraints

In addition to these automatically generated constraints, the user can provide a matrix A_u and vectors l_u and u_u to define further linear constraints. These user supplied constraints could be used, for example, to restrict voltage angle differences between specific buses. The matrix A_u must have at least n_x columns where n_x is the number of x variables. If A_u has more than n_x columns, a corresponding z optimization variable is created for each additional column. These z variables also enter into the generalized cost terms described below, so A_u and N must have the same number of columns.

$$l_u \le A_u \begin{bmatrix} x \\ z \end{bmatrix} \le u_u \qquad \text{(user supplied linear constraints)}$$

Change from MATPOWER 3.0: The A_u matrix supplied by the user no longer includes the (all zero) columns corresponding to the y variables for piecewise linear generator costs. This should simplify significantly the creation of the desired A_u matrix.

Generalized Cost Function

The cost function consists of 3 parts. The first two are the polynomial and piecewise linear costs, respectively, of generation. A polynomial or piecewise linear cost is specified for each generator's active output and, optionally, reactive output in the appropriate row(s) of the `gencost` matrix. Any piecewise linear costs are implemented using the CCV formulation described above which introduces corresponding helper y variables. The general formulation allows generator costs of mixed type (polynomial and piecewise linear) in the same problem.

The third part of the cost function provides a general framework for imposing additional costs on the optimization variables, enabling things such as using penalty functions as soft limits on voltages, additional costs on variables involved in constraints handled by Langrangian relaxation, etc.

This general cost term is specified through a set of parameters H, C_w, N and f_{parm}, described below. It consists of a general quadratic function of an $n_w \times 1$ vector w of transformed optimization variables.

$$\frac{1}{2} w^T H w + C_w^T w$$

H is the $n_w \times n_w$ symmetric, sparse matrix of quadratic coefficients and C_w is the $n_w \times 1$ vector of linear coefficients. The sparse N matrix is $n_w \times n_{xz}$, where the number of columns must match that of any user supplied A_u matrix. And f_{parm} is $n_w \times 4$, where the 4 columns are labeled as

$$f_{parm} = \begin{bmatrix} d & \hat{r} & h & m \end{bmatrix}.$$

The vector w is created from the x and z optimization variables by first applying a general linear transformation

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

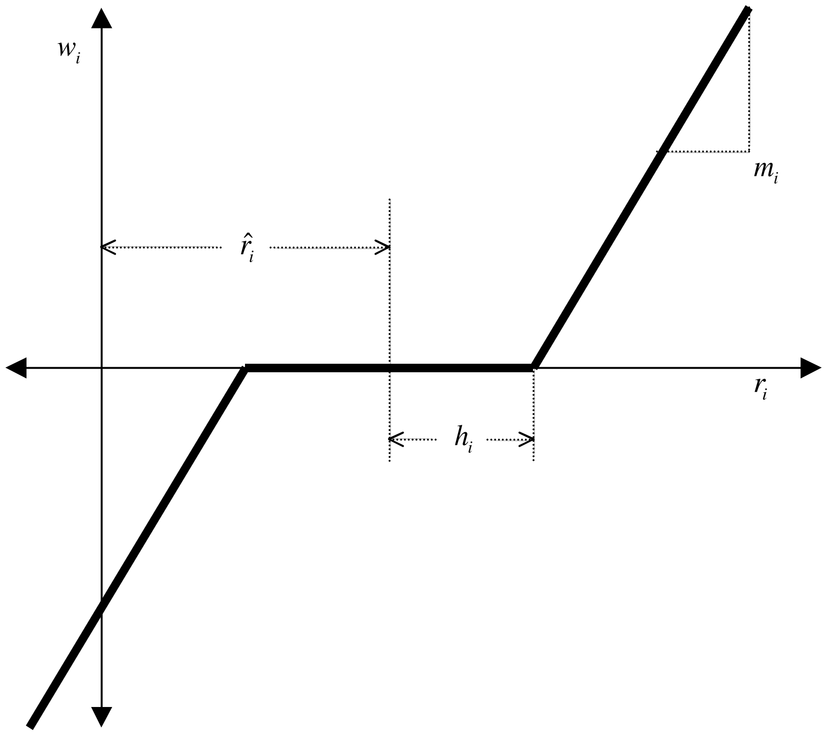
followed by a scaled function with a shifted "dead zone", defined by the remaining elements of f_{parm}. Each element of r is transformed into the corresponding element of w as follows:

$$w_i = \begin{cases} m_i \cdot f_i(r_i - \hat{r}_i + h_i), & r_i - \hat{r}_i < -h_i \\ 0, & -h_i \le r_i - \hat{r}_i \le h_i \\ m_i \cdot f_i(r_i - \hat{r}_i - h_i), & r_i - \hat{r}_i > h_i \end{cases}$$

where the function f_i is a predetermined function selected by the index in d_i. The current implementation includes linear and quadratic options.

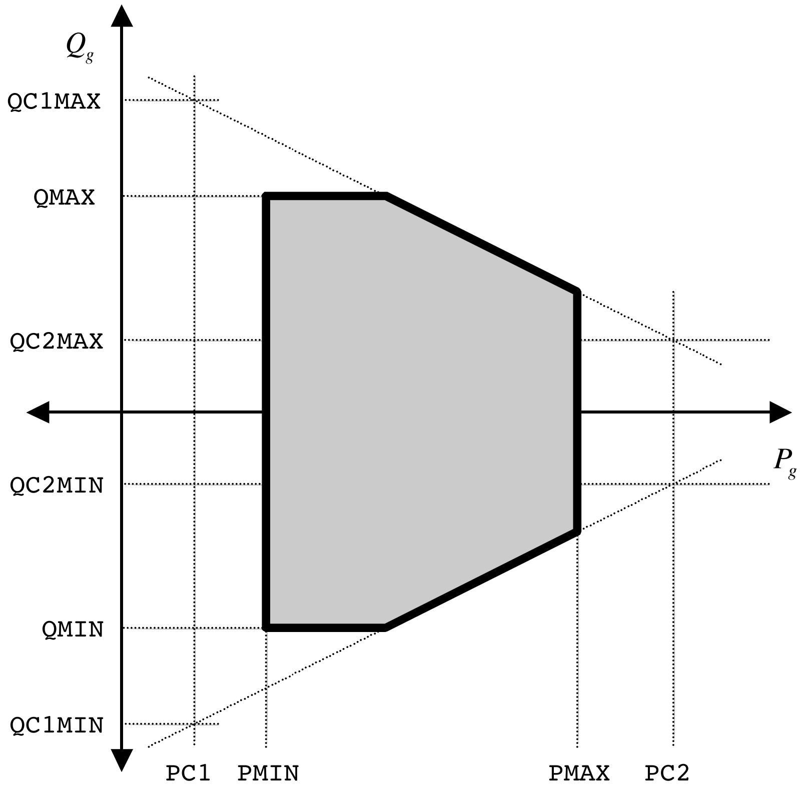
$$f_i(t) = \begin{cases} t, & d_i = 1 \\ t^2, & d_i = 2 \end{cases}$$

The linear case, where $d_i = 1$, is illustrated below, where w_i is found by shifting r_i by $\hat{r}_i$, applying a "dead zone" defined by h_i and then scaling by m_i.



Generator P-Q Capability Curves

The traditional AC OPF formulation models the generator P-Q capability curves as simple box constraints defined by the `PMIN`, `PMAX`, `QMIN` and `QMAX` columns of the `gen` matrix. In *MATPOWER* 3.1, version 2 of the case file format is introduced, which includes 6 new columns in the `gen` matrix for specifying additional sloped upper and lower portions of the capability curves. The new columns are `PC1`, `PC2`, `QC1MIN`, `QC1MAX`, `QC2MIN`, and `QC2MAX`. The feasible region for generator operation with this more general capability curve is illustrated by the shaded region in the figure below.

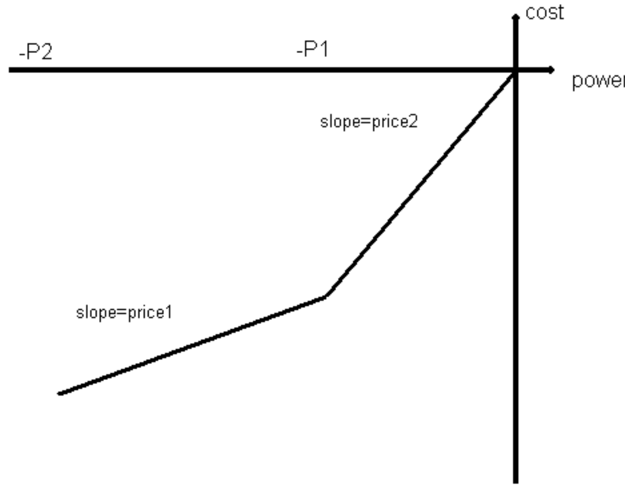


The particular values of `PC1` and `PC2` are not important and may be set equal to `PMIN` and `PMAX` for convenience. The important point is to set the corresponding `QC`*n*`MAX` (`QC`*n*`MIN`) limits such that the two resulting points define the desired line corresponding to the upper (lower) sloped portion of the capability curve.

Dispatchable loads

In general, dispatchable or price-sensitive loads can be modeled as negative real power injections with associated costs. The current test is that if `PMIN` < `PMAX` = 0 for a generator, then it is really a dispatchable load. If a load has a demand curve like the following

so that it will consume zero if the price is higher than *price2*, *P1* if the price is less than *price2* but higher than *price1*, and *P2* if the price is equal or lower than *price1*. Considered as a negative injection, the desired dispatch is zero if the price is greater than *price2*, *-P1* if the price is higher than *price1* but lower than *price2*, and *-P2* if the price is equal to or lower than *price1*. This suggests the following piecewise linear cost curve:



Note that this assumes that the demand blocks can be partially dispatched or "split"; if the price trigger is reached half-way through the block, the load must accept the partial block. Otherwise, accepting or rejecting whole blocks really poses a mixed-integer problem, which is outside the scope of the current *MATPOWER* implementation.

When there are dispatchable loads, the issue of reactive dispatch arises. If the `QMIN`/`QMAX` generation limits for the "negative generator" in question are not set to zero, then the algorithm will dispatch the reactive injection to the most convenient value. Since this is not normal load behavior, in the generalized formulation it is assumed that dispatchable loads maintain a constant power factor. The mechanism for posing additional general linear constraints is employed to automatically include restrictions for these injections to keep the ratio of P_g and Q_g constant. This ratio is inferred from the values of `PMIN` and either `QMIN` (for inductive loads) or `QMAX` (for capacitive loads) in the `gen` table. It is important to set these appropriately, keeping in mind that `PG` is negative and that for normal inductive loads `QG` should also be negative (a positive reactive load is a negative reactive injection). The initial values of the `PG` and `QG` columns of the `gen` matrix must be consistent with the ratio defined by `PMIN` and the appropriate Q limit.

Problem Data Transformation

Defining a user supplied A matrix to add additional linear constraints requires knowledge of the order of the optimization variables in the x vector. This requires an understanding of the standard transformations performed on the input data (`bus`, `gen`, `branch`, `areas` and `gencost` tables) before the problem is solved. All of these transformations are reversed after solving the problem so the output data is correctly placed in the tables.

The first step filters out inactive generators and branches; original tables are saved for data output.

```
comgen   = find(gen(:,GEN_STATUS) > 0);     % find online generators
onbranch = find(branch(:,BR_STATUS) ~= 0);  % find online branches
gen      = gen(comgen, :);
branch   = branch(onbranch, :);
```

The second step is a renumbering of the bus numbers in the `bus` table so that the resulting table contains consecutively-numbered buses starting from 1:

```
[i2e, bus, gen, branch, areas] = ext2int(bus, gen, branch, areas);
```

where `i2e` is saved for inverse reordering at the end. Finally, generators are further reordered by bus number:

```
ng = size(gen,1);                 % number of generators or injections
[tmp, igen] = sort(gen(:, GEN_BUS));
[tmp, inv_gen_ord] = sort(igen);  % save for inverse reordering at the end
gen  = gen(igen, :);
if ng == size(gencost,1)          % This is because gencost might have
  gencost = gencost(igen, :);     % twice as many rows as gen if there
else                              % are reactive injection costs.
  gencost = gencost( [igen; igen+ng], :);
end
```

Having done this, the variables inside the x vector now have the same ordering as in the `bus`, `gen` tables:

```
x = [  Theta ;    % nb bus voltage angles
         V   ;    % nb bus voltage magnitudes
         Pg  ;    % ng active power injections (p.u.) (ascending bus order)
         Qg ];    % ng reactive power injections (p.u.)(ascending bus order)
```

and the nonlinear constraints have the same order as in the `bus`, `branch` tables

```
g = [  gp;        % nb real power flow mismatches (p.u.)
       gq;        % nb reactive power flow mismatches (p.u.)
       gsf;       % nl "from" end apparent power injection limits (p.u.)
       gst ];     % nl "to" end apparent power injection limits (p.u.)
```

With this setup, box bounds on the variables are applied as follows: the reference angle is bounded above and below with the value specified for it in the original `bus` table. The V section of x is bounded above and below with the corresponding values for `VMAX` and `VMIN` in the `bus` table. The Pg and Qg sections of x are bounded above and below with the corresponding values for `PMAX`, `PMIN`, `QMAX` and `QMIN` in the `gen` table. The nonlinear constraints are similarly setup so that `gp` and `gq` are equality constraints (zero RHS) and the limits for `gsf`, `gst` are taken from the `RATE_A` column in the `branch` table.

Example of Additional Linear Constraint

The following example illustrates how an additional general linear constraint can be added to the problem formulation. In the standard solution to `case9.m`, the voltage angle for bus 7 lags the voltage angle in bus 2 by 6.09 degrees. Suppose we want to limit that lag to 5 degrees at the most. A linear restriction of the form

```
Theta(2) – Theta(7) <= 5 degrees
```

would do the trick. We have `nb` = 9 buses, `ng` = 3 generators and `nl` = 9 branches. Therefore the first 9 elements of x are bus voltage angles, elements 10-18 are bus voltage magnitudes, elements 19-21 are active injections corresponding to the generators in buses 1, 2 and 3 (in that order) and elements 22-24 are the corresponding reactive injections. Note that in this case the generators in the original data already appear in ascending bus order, so no permutation with respect to the original data is necessary. Going back to the angle restriction, we see that it can be cast as

```
[ 0 1 0 0 0 0 -1 0 0 zeros(1,nb+ng+ng) ] * x  <= 5 degrees
```

We can set up the problem as follows:

```
A = sparse([1;1], [2;7], [1;-1], 1, 24);
l = -Inf;
```

```
u = 5 * pi/180;
mpopt = mpoption('OPF_ALG', 520); % use fmincon w/generalized formulation
opf('case9', A, l, u, mpopt)
```

which indeed restricts the angular separation to 5 degrees.

3.4.3 DC OPF Formulation

The DC optimal power flow problem solved by *MATPOWER* is similar to the traditional AC OPF formulation described above, but using the DC model of the network, which only includes bus voltage angles and real power injections and flows.

$$\min_{\theta, P_g} \sum_i f_i(P_{gi})$$

subject to

$$B_{bus}\theta = P_g - P_d - P_{bus,shift} - G_{sh} \qquad \text{(active power balance equations)}$$

$$B_f\theta \le P^{\max} - P_{f,shift} \qquad \text{(real power flow limit of lines, \textit{from} end)}$$

$$-B_f\theta \le P^{\max} + P_{f,shift} \qquad \text{(real power flow limit of lines, \textit{to} end)}$$

$$P_{gi}^{\min} \le P_{gi} \le P_{gi}^{\max} \qquad \text{(active power generation limits)}$$

The voltage angle at the reference bus is also constrained to the specified value. Since all constraints are linear, the problem is a simple LP or QP problem depending on the form of the cost function.

The current implementation of the DC OPF does not allow additional user-supplied linear constraints and costs as in the generalized AC OPF formulation described above.

3.5 Unit Decommitment 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 decommitment 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 a `case30`, for example, type:

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

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

The algorithm proceeds 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.

3.6 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 default *MATPOWER* options vector is obtained by calling `mpoption` with no arguments. So, typing:

```
>> runopf('case30', mpoption)
```

is another way to run the OPF solver with the all of the default options.

The *MATPOWER* options vector controls the following:

- power flow algorithm
- power flow termination criterion
- OPF algorithm
- OPF default algorithms for different cost models
- OPF cost conversion parameters
- OPF termination criterion
- verbose level
- printing of results

The details are given below:

```
>> help mpoption
 MPOPTION  Used to set and retrieve a MATPOWER options vector.

    opt = mpoption
        returns the default options vector

    opt = mpoption(name1, value1, name2, value2, ...)
        returns the default options vector with new values for up to 7
        options, name# is the name of an option, and value# is the new
        value. Example: options = mpoption('PF_ALG', 2, 'PF_TOL', 1e-4)

    opt = mpoption(opt, name1, value1, name2, value2, ...)
        same as above except it uses the options vector opt as a base
        instead of the default options vector.

    The currently defined options are as follows:

       idx - NAME, default          description [options]
       ---   -------------          -------------------------------------
    power flow options
        1  - PF_ALG, 1              power flow algorithm
            [    1 - Newton's method                                     ]
            [    2 - Fast-Decoupled (XB version)                         ]
            [    3 - Fast-Decoupled (BX version)                         ]
            [    4 - Gauss Seidel                                        ]
        2  - PF_TOL, 1e-8           termination tolerance on per unit
                                    P & Q mismatch
        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|       [   0 or 1  ]
        10 - PF_DC, 0               use DC power flow formulation, for
                                    power flow and OPF
            [    0 - use AC formulation & corresponding algorithm opts  ]
            [    1 - use DC formulation, ignore AC algorithm options    ]
    OPF options
        11 - OPF_ALG, 0             algorithm to use for OPF
            [    0 - choose best default solver available in the        ]
            [        following order, 500, 520 then 100/200             ]
            [ Otherwise the first digit specifies the problem           ]
            [ formulation and the second specifies the solver,          ]
            [ as follows, (see the User's Manual for more details)      ]
            [  100 - standard formulation (old), constr                 ]
            [  120 - standard formulation (old), dense LP               ]
            [  140 - standard formulation (old), sparse LP (relaxed)    ]
            [  160 - standard formulation (old), sparse LP (full)       ]
            [  200 - CCV formulation (old), constr                      ]
            [  220 - CCV formulation (old), dense LP                    ]
            [  240 - CCV formulation (old), sparse LP (relaxed)         ]
            [  260 - CCV formulation (old), sparse LP (full)            ]
            [  500 - generalized formulation, MINOS                     ]
            [  520 - generalized formulation, fmincon                   ]
            [ See the User's Manual for details on the formulations.    ]
        12 - OPF_ALG_POLY, 100      default OPF algorithm for use with
                                    polynomial cost functions
                                    (used only if no solver available
                                    for generalized formulation)
        13 - OPF_ALG_PWL, 200       default OPF algorithm for use with
                                    piece-wise linear cost functions
                                    (used only if no solver available
                                    for generalized formulation)
        14 - OPF_POLY2PWL_PTS, 10   number of evaluation points to use
                                    when converting from polynomial to
                                    piece-wise linear costs
        16 - OPF_VIOLATION, 5e-6    constraint violation tolerance
        17 - CONSTR_TOL_X, 1e-4     termination tol on x for copf & fmincopf
        18 - CONSTR_TOL_F, 1e-4     termination tol on F for copf & fmincopf
        19 - CONSTR_MAX_IT, 0       max number of iterations for copf & fmincopf
                                    [       0 => 2*nb + 150             ]
        20 - LPC_TOL_GRAD, 3e-3     termination tolerance on gradient for lpopf
        21 - LPC_TOL_X, 1e-4        termination tolerance on x (min step size)
                                    for lpopf
        22 - LPC_MAX_IT, 400        maximum number of iterations for lpopf
        23 - LPC_MAX_RESTART, 5     maximum number of restarts for lpopf
        24 - OPF_P_LINE_LIM, 0      use active power instead of apparent power
                                    for line flow limits    [   0 or 1  ]
        25 - OPF_IGNORE_ANG_LIM, 0  ignore angle difference limits for branches
                                    even if specified       [   0 or 1  ]
```

```
output options
    31 - VERBOSE, 1             amount of progress info printed
        [   0 - print no progress info                            ]
        [   1 - print a little progress info                      ]
        [   2 - print a lot of progress info                      ]
        [   3 - print all progress info                           ]
    32 - OUT_ALL, -1            controls printing of results
        [  -1 - individual flags control what prints              ]
        [   0 - don't print anything                              ]
        [       (overrides individual flags, except OUT_RAW)      ]
        [   1 - print everything                                  ]
        [       (overrides individual flags, except OUT_RAW)      ]
    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        [   0 or 1  ]
    36 - OUT_BRANCH, 1          print branch detail     [   0 or 1  ]
    37 - OUT_GEN, 0             print generator detail  [   0 or 1  ]
                                (OUT_BUS also includes gen info)
    38 - OUT_ALL_LIM, -1        control constraint info output
        [  -1 - individual flags control what constraint info prints]
        [   0 - no constraint info (overrides individual flags)     ]
        [   1 - binding constraint info (overrides individual flags)]
        [   2 - all constraint info (overrides individual flags)    ]
    39 - OUT_V_LIM, 1           control output of voltage limit info
        [   0 - don't print                                        ]
        [   1 - print binding constraints only                     ]
        [   2 - print all constraints                              ]
        [   (same options for OUT_LINE_LIM, OUT_PG_LIM, OUT_QG_LIM) ]
    40 - OUT_LINE_LIM, 1        control output of line limit info
    41 - OUT_PG_LIM, 1          control output of gen P limit info
    42 - OUT_QG_LIM, 1          control output of gen Q limit info
    43 - OUT_RAW, 0             print raw data for Perl database
                                interface code          [   0 or 1  ]
other options
    51 - SPARSE_QP, 1           pass sparse matrices to QP and LP
                                solvers if possible     [   0 or 1  ]
```

```
MINOPF options
    61 - MNS_FEASTOL, 0 (1E-3)  primal feasibility tolerance,
                                set to value of OPF_VIOLATION by default
    62 - MNS_ROWTOL, 0  (1E-3)  row tolerance
                                set to value of OPF_VIOLATION by default
    63 - MNS_XTOL, 0     (1E-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)   iterations limit
    70 - MNS_VERBOSITY, -1
        [  -1 - controlled by VERBOSE flag (0 or 1 below)          ]
        [   0 - print nothing                                      ]
        [   1 - print only termination status message              ]
        [   2 - print termination status and screen progress       ]
        [   3 - print screen progress, report file (usually fort.9) ]
    71 - MNS_CORE, 1200 * nb + 5000
    72 - MNS_SUPBASIC_LIM, 0 (2*ng) superbasics limit
    73 - MNS_MULT_PRICE, 0 (30) multiple price
```

A typical usage of the options vector might be as follows:

Get the default options vector:

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

Use the fast-decoupled method to solve power flow:

```
>> opt = mpoption(opt, 'PF_ALG', 2);
```

Display only system summary and generator info:

```
>> opt = mpoption(opt, 'OUT_BUS', 0, 'OUT_BRANCH', 0, 'OUT_GEN', 1);
```

Show all progress info:

```
>> opt = mpoption(opt, 'VERBOSE', 3);
```

Now, run a bunch of power flows using these settings:

```
>> runpf('case57', opt)
>> runpf('case118', opt)
>> runpf('case300', opt)
```

3.7 Summary of the Files

Documentation files:

`README`	basic intro to *MATPOWER*
`README.txt`	basic intro to *MATPOWER*, with DOS line endings (for Windows)
`docs/CHANGES`	modification history of *MATPOWER*
`docs/CHANGES.txt`	modification history of *MATPOWER*, with DOS line endings
`docs/manual.pdf`	PDF version of the *MATPOWER* User's Manual

(see also `caseformat.m` & `genform.m` below)

Top-level programs:

`cdf2matp.m`	converts data from IEEE CDF to *MATPOWER* format
`runcomp.m`	runs 2 OPFs and compares results
`rundcopf.m`	runs a DC optimal power flow
`rundcpf.m`	runs a DC power flow
`runduopf.m`	runs a DC OPF with unit decommitment
`runopf.m`	runs an optimal power flow
`runpf.m`	runs a power flow
`runuopf.m`	runs an OPF with unit decommitment

(see also `opf.m, copf.m, fmincopf.m, lpopf.m` below which can also be used as top-level programs)

Input data files:

`caseformat.m`	documentation for input data file format
`case_ieee30.m`	IEEE 30 bus system
`case118.m`	IEEE 118 bus system
`case14.m`	IEEE 14 bus system
`case30.m`	modified IEEE 30 bus system
`case300.m`	IEEE 300 bus system
`case30pwl.m`	`case30.m` with piecewise linear costs
`case30Q.m`	`case30.m` with reactive power costs
`case39.m`	39 bus system
`case4gs.m`	4 bus system from Grainger & Stevenson
`case57.m`	IEEE 57 bus system
`case6ww.m`	6 bus system from Wood & Wollenberg
`case9.m`	3 generator, 9 bus system (default case file)
`case9Q.m`	`case9.m` with reactive power costs

Common source files and utility functions used by multiple programs:

`bustypes.m`	creates vectors of bus indices for ref bus, PV buses, PQ buses
`compare.m`	prints summary of differences between 2 solved cases
`dAbr_dV.m`	computes partial derivatives of branch apparent power flows wrt. voltage, used by OPF
`dSbr_dV.m`	computes partial derivatives of branch complex power flows wrt. voltage, used by OPF & state estimator
`dSbus_dV.m`	computes partial derivatives of bus complex power injections wrt. voltage, used by OPF, Newton PF, state estimator
`ext2int.m`	converts data matrices from external to internal bus numbering
`hasPQcap.m`	checks for generator P-Q capability curve constraints
`have_fcn.m`	checks for availability of optional functionality
`idx_area.m`	named column index definitions for `areas` matrix
`idx_brch.m`	named column index definitions for `branch` matrix
`idx_bus.m`	named column index definitions for `bus` matrix
`idx_cost.m`	named column index definitions for `gencost` matrix
`idx_gen.m`	named column index definitions for `gen` matrix
`int2ext.m`	converts data matrices from internal to external bus numbering
`isload.m`	checks if generators are actually dispatchable loads
`loadcase.m`	loads data from a case file or struct into data matrices
`makeB.m`	forms *B* matrix used by fast decoupled power flow
`makeBdc.m`	forms *B* matrix used by DC PF and DC OPF
`makeSbus.m`	forms bus complex power injections from specified generation and load injections
`makeYbus.m`	forms complex bus admittance matrix
`mp_lp.m`	solves an LP problem with best solver available
`mp_qp.m`	solves a QP problem with best solver available
`mpver.m`	prints *MATPOWER* version information
`printpf.m`	pretty prints solved PF or OPF case
`savecase.m`	saves data matrices to a case file
`mpoption.m`	set *MATPOWER* options

Power Flow (PF):

`dcpf.m`	implements DC power flow solver
`fdpf.m`	implements fast decouple power flow solver
`gausspf.m`	implements Gauss-Seidel power flow solver
`newtonpf.m`	implements Newton power flow solver
`pfsoln.m`	updates data matrices with PF solution

Optimal Power Flow (OPF):

common files shared by multiple OPF solvers

`opf_form.m`	returns code for formulation given OPF algorithm code
`opf_slvr.m`	returns code for solver given OPF algorithm code
`opf.m`	top-level OPF solver routine
`poly2pwl.m`	creates piecewise linear approximation to polynomial cost function
`pqcost.m`	splits `gencost` into real and reactive power costs
`totcost.m`	computes total cost for given dispatch

files used only by DC OPF

`dcopf.m`	implements DC optimal power flow

files used only for traditional OPF formulation (`constr`*- and LP-based)*

`fg_names.m`	returns names of function and gradient evaluators for given algorithm
`fun_ccv.m`	computes objective function and constraints for CCV formulation
`fun_std.m`	computes objective function and constraints for standard formulation
`grad_ccv.m`	computes gradients of obj fcn & constraints for CCV formulation
`grad_std.m`	computes gradients of obj fcn & constraints for standard formulation
`opfsoln.m`	updates data matrices with OPF solution

files used only by `constr`*-based OPF*

`copf.m`	implements `constr`-based OPF solver

files used only by LP-based OPF

`lpopf.m`	implements LP-based OPF solver
`LPconstr.m`	solves a non-linear optimization via sequential linear programming
`LPeqslvr.m`	runs Newton power flow
`LPrelax.m`	solves LP problem with constraint relaxation
`LPsetup.m`	solves LP problem using specified method

files used only for generalized OPF formulation (`fmincon`*- and MINOS-based)*

`genform.m`	documentation for generalized OPF formulation
`makeAy.m`	forms *A* matrix and *b* vector for generalized OPF formulation

files used only by `fmincon`*-based OPF*

`consfmin.m`	computes value and gradient of constraints
`costfmin.m`	computes value and gradient of objective function
`fmincopf.m`	implements `fmincon`-based OPF solver

files used only for OPF with unit decommitment

`fairmax.m`	same as MATLAB's built-in `max()`, except breaks ties randomly
`uopf.m`	implements unit decommitment for OPF

Extras: (in `extras` subdirectory)

auction market software (in `smartmarket` *subdirectory)*

`auction.m`	clears set of bids and offers based on pricing rules and OPF result
`case2off.m`	creates set of price/quantity bids/offers given `gen` and `gencost` matrices
`idx_disp.m`	named column index definitions for `dispatch` matrix
`off2case.m`	updates `gen` and `gencost` matrices based on quantity/price bids/offers
`printmkt.m`	prints the market output
`runmkt.m`	top-level program runs an OPF-based auction
`SM_CHANGES`	modification history of the smartmarket software
`smartmkt.m`	implements the smartmarket solver

unfinished state estimator (in `state_estimator` *subdirectory)*

`runse.m`	runs a state estimator
`state_est.m`	implements a state estimator

Tests: (in `t` subdirectory)

`soln9_dcopf.mat`	data used for tests
`soln9_dcpf.mat`	data used for tests
`soln9_opf.mat`	data used for tests
`soln9_opf_extras1.mat`	data used for tests
`soln9_opf_Plim.mat`	data used for tests
`soln9_opf_PQcap.mat`	data used for tests
`soln9_pf.mat`	data used for tests
`t_auction.m`	tests `auction.m` in `extras/smartmarket`
`t_auction_case.m`	test case for `t_auction.m`
`t_auction_fmincopf.m`	`fmincon`-based tests of `auction.m` in `extras/smartmarket`
`t_begin.m`	starts a set of tests
`t_case9_opf.m`	case file (version 1 format) for OPF tests
`t_case9_opfv2.m`	case file (version 2 format) for OPF tests
`t_case9_pf.m`	case file (version 1 format) for power flow tests
`t_case9_pfv2.m`	case file (version 2 format) for power flow tests
`t_end.m`	finishes a set of tests and prints statistics
`t_hasPQcap.m`	tests `hasPQcap.m`
`t_is.m`	tests if two matrices are identical to some tolerance
`t_jacobian.m`	does numerical test of partial derivatives
`t_loadcase.m`	tests `load_case.m`
`t_off2case.m`	tests `off2case.m`
`t_ok.m`	tests if an expression is true
`t_opf.m`	tests OPF solvers
`t_pf.m`	tests PF solvers
`t_run_tests.m`	framework for running a series of tests
`t_runmarket.m`	tests `runmarket.m`
`t_skip.m`	skips a specified number of tests
`test_matpower.m`	runs all available *MATPOWER* tests

4 Acknowledgments

The authors would like to acknowledge contributions from several people. 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 are 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 also to many others who have contributed code, bug reports and suggestions over the years. Last but not least, we would like to acknowledge the input of Bob Thomas throughout the development of *MATPOWER* here at PSERC Cornell.

5 References

1. R. van Amerongen, "A General-Purpose Version of the Fast Decoupled Loadflow", *IEEE Transactions on Power Systems*, Vol. 4, No. 2, May 1989, pp. 760-770.
2. O. Alsac, J. Bright, M. Prais, B. Stott, "Further Developments in LP-based Optimal Power Flow", *IEEE Transactions on Power Systems*, Vol. 5, No. 3, Aug. 1990, pp. 697-711.
3. R. Fletcher, *Practical Methods of Optimization*, 2nd Edition, John Wiley & Sons, p. 96.
4. P. E. Gill, W. Murry, M. H. Wright, *Practical Optimization*, Academic Press, London, 1981.
5. A. F. Glimm and G. W. Stagg, "Automatic calculation of load flows", *AIEE Transactions (Power Apparatus and Systems),* vol. 76, pp. 817-828, Oct. 1957.
6. A. Grace, *Optimization Toolbox*, The MathWorks, Inc., Natick, MA, 1995.
7. C. Li, R. B. Johnson, A. J. Svoboda, "A New Unit Commitment Method", *IEEE Transactions on Power Systems*, Vol. 12, No. 1, Feb. 1997, pp. 113-119.
8. C. Mészáros, "The efficient implementation of interior point methods for linear programming and their applications", *Ph.D. Thesis*, Eötvös Loránd University of Sciences, 1996.
9. B. Stott, "Review of Load-Flow Calculation Methods", *Proceedings of the IEEE*, Vol. 62, No. 7, July 1974, pp. 916-929.
10. B. Stott and O. Alsac, "Fast decoupled load flow", *IEEE Transactions on Power Apparatus and Systems*, Vol. PAS-93, June 1974, pp. 859-869.
11. B. Stott, J. L. Marino, O. Alsac, "Review of Linear Programming Applied to Power System Rescheduling", *1979 PICA*, pp. 142-154.
12. 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, Nov. 1967, pp. 1449-1460.
13. A. J. Wood and B. F. Wollenberg, "Power Generation, Operation, and Control, 2nd Edition, John Wiley & Sons, p. 108-111.
14. B.A Murtagh andM.A. Saunders, "MINOS 5.5 User's Guide", *Stanford University Systems Optimization Laboratory Technical Report SOL83-20R*.

Appendix A: Notes on LP-Solvers for MATLAB

When *MATPOWER* was initially developed the LP and QP solvers available in MATLAB's Optimization Toolbox, `lp.m` and `qp.m`, did not exploit sparsity and were therefore ***very*** slow for the large sparse problems typically encountered in power system simulation. Fortunately, there were some third party LP and QP-solvers for MATLAB with much better performance.

Several LP and QP-solvers were tested for use in the context of an LP-based OPF. Some of them we were unable to get to compile on our architecture of choice and others proved to be less than robust in an OPF context.

Here is a list of the solvers we tested at the time:

- *bpmpd* - QP-solver from http://www.sztaki.hu/~meszaros/bpmpd/
  Please see http://www.pserc.cornell.edu/bpmpd/ for a MATLAB MEX version.
- *lp.m* - LP-solver included with Optimization Toolbox 1.x and 2.x (from MathWorks)
- *lp_solve* - LP-solver from ftp://ftp.ics.ele.tue.nl/pub/lp_solve/
- *loqo* - LP-solver from http://www.princeton.edu/~rvdb/
- *sol_qps.m* - LP-solver developed at U. of Wisconsin (not publicly available)

Of all of the packages tested, the *bpmpd* solver, has been the only one which worked reliably for us. It has proven to be very robust and has exceptional performance.

More information about free optimizers is available in "Decision Tree for Optimization Software" maintained by Mittenlmonn Hans and P. Spellucci at http://plato.la.asu.edu/guide.html.

Since the initial development of *MATPOWER*, more recent versions of the MATLAB Optimization Toolbox have moved to new LP and QP solvers, `linprog.m` and `quadprog.m`. The LP solver, base on LIPSOL, does support sparsity, but is still typically slower than *bpmpd*. The QP solver does not support sparsity in general, only for certain restricted special cases.

Appendix B: Additional Notes

- Some versions of MATLAB 5 were slow at selecting rows of a large sparse matrix, but much faster at transposing and selecting columns.
- `fmincon.m` seems to compute inaccurate shadow prices for Optimization Toolbox 3.0 and earlier.

Appendix C: Auction Code

MATPOWER 3 includes in the `extras/smartmarket` directory code which 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[2] in the context of *POWERWEB*[3] but has not been widely tested in other contexts.

MATPOWER 3.1 includes a new version that supports offers and bids for reactive power and uses a new interface.

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 ≤ `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 "Dispatchable loads" on page 20). 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 the 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 3.5 Unit Decommitment Algorithm).

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

[2] See http://www.pserc.cornell.edu/minopf/

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

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 the table below.

Auction Type	Name	Description
0	discriminative	The 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 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 adding a locational adjustment. For bids and offers at bus *i*, the adjustment is $\lambda_{P,ref} - \lambda_{P,i}$, where $\lambda_{P,ref}$ 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 subtracting the locational adjustment. The appropriate locationally adjusted uniform price is then used for all cleared bids and offers.

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`.

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 equal 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.

Example

Six generators with three blocks of capacity each, offering as follows:

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

One fixed load of 151.64 MW.

Three dispatchable loads, bidding three blocks each as follows:

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

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.

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 generators sold:

Generator	**Quantity Sold** *MW*	**Selling Price** *$/MWh*
1	35.3	$50.00
2	36	$50.24
3	36	$50.34
4	36	$51.02
5	36	$52.17
6	36	$52.98

And the dispatchable loads bought:

Load	**Quantity Bought** *MW*	**Purchase Price** *$/MWh*
1	30.0	$51.82
2	10.0	$54.03
3	20.0	$55.62