

MATPOWER

A MATLAB™ Power System Simulation Package

Version 3.0b4

January 28, 2005

User's Manual

Ray D. Zimmerman
rz10@cornell.edu

Carlos E. Murillo-Sánchez
carlos_murillo@ieee.org

Deqiang (David) Gan
dgan@zju.edu.cn

© 1997-2005 Power Systems Engineering Research Center (PSERC)
School of Electrical Engineering, Cornell University, Ithaca, NY 14853

Table of Contents

Table of Contents.....	2
1 Introduction.....	3
2 Getting Started	3
2.1 System Requirements.....	3
2.2 Installation.....	4
2.3 Running a Power Flow.....	4
2.4 Running an Optimal Power Flow	4
2.5 Getting Help.....	4
3 Technical Reference	5
3.1 Data File Format	5
3.2 Modeling.....	7
3.3 Power Flow	9
3.4 Optimal Power Flow	10
3.4.1 Traditional AC OPF Formulation.....	11
3.4.2 Generalized AC OPF Formulation (<i>fmincon</i> and MINOPF).....	15
3.4.3 DC OPF Formulation	20
3.5 Unit Decommitment Algorithm.....	20
3.6 MATPOWER Options	21
3.7 Summary of the Files	25
4 Acknowledgments.....	29
5 References.....	29
Appendix A: Notes on LP-Solvers for MATLAB.....	30
Appendix B: Additional Notes.....	30
Appendix C: Auction Code	31

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 later¹
- 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 mption
```

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. The columns for each data matrix are given below.

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)

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 = V_f / V_t)
- 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

(+) 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  $x0 < x1 < x2 < \dots$  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

$$\begin{bmatrix} I_f \\ I_t \end{bmatrix} = \mathbf{Y}_{br} \begin{bmatrix} V_f \\ V_t \end{bmatrix} \quad (1)$$

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

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} .

$$\begin{aligned}\mathbf{I}_f &= \mathbf{Y}_f \mathbf{V}_{bus} \\ \mathbf{I}_t &= \mathbf{Y}_t \mathbf{V}_{bus}\end{aligned}$$

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

$$\begin{aligned}\mathbf{S}_{bus} &= \text{diag}(\mathbf{V}_{bus}) \mathbf{I}_{bus}^* \\ \mathbf{S}_f &= \text{diag}(\mathbf{V}_f) \mathbf{I}_f^* \\ \mathbf{S}_t &= \text{diag}(\mathbf{V}_t) \mathbf{I}_t^*\end{aligned}$$

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 $\text{diag}()$ 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

$$\begin{bmatrix} P_f \\ P_t \end{bmatrix} = \mathbf{B}_{br} \begin{bmatrix} \theta_f \\ \theta_t \end{bmatrix} + \begin{bmatrix} P_{f,shift} \\ P_{t,shift} \end{bmatrix} \quad (2)$$

where

$$\mathbf{B}_{br} = \frac{1}{X\tau} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \quad (3)$$

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

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

$$\mathbf{P}_{bus} = \mathbf{B}_{bus} \boldsymbol{\theta}_{bus} + \mathbf{P}_{bus,shift}$$

Similarly, *MATPOWER* builds the matrix \mathbf{B}_f and the vector $\mathbf{P}_{f,shift}$ which can be used to compute the vectors \mathbf{P}_f and \mathbf{P}_t of branch real power injections

$$\mathbf{P}_f = \mathbf{B}_f \boldsymbol{\theta}_{bus} + \mathbf{P}_{f,shift}$$

$$\mathbf{P}_t = -\mathbf{P}_f$$

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:

<code>bus(:, VM)</code>	bus voltage magnitudes
<code>bus(:, VA)</code>	bus voltage angles
<code>gen(:, PG)</code>	generator real power injections
<code>gen(:, QG)</code>	generator reactive power injections
<code>branch(:, PF)</code>	real power injected into “from” end of branch
<code>branch(:, PT)</code>	real power injected into “to” end of branch
<code>branch(:, QF)</code>	reactive power injected into “from” end of branch
<code>branch(:, QT)</code>	reactive power injected into “to” end of branch
<code>success</code>	1 = solved successfully, 0 = unable to solve
<code>et</code>	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 = mppoption('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 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:

<code>bus(:, LAM_P)</code>	Lagrange multiplier on bus real power mismatch
<code>bus(:, LAM_Q)</code>	Lagrange multiplier on bus reactive power mismatch
<code>bus(:, MU_VMAX)</code>	Kuhn-Tucker multiplier on upper bus voltage limit
<code>bus(:, MU_VMIN)</code>	Kuhn-Tucker multiplier on lower bus voltage limit
<code>gen(:, MU_PMAX)</code>	Kuhn-Tucker multiplier on upper generator real power limit
<code>gen(:, MU_PMIN)</code>	Kuhn-Tucker multiplier on lower generator real power limit
<code>gen(:, MU_QMAX)</code>	Kuhn-Tucker multiplier on upper generator reactive power limit
<code>gen(:, MU_QMIN)</code>	Kuhn-Tucker multiplier on lower generator reactive power limit
<code>branch(:, MU_SF)</code>	Kuhn-Tucker multiplier on MVA limit at "from" end of branch
<code>branch(:, MU_ST)</code>	Kuhn-Tucker multiplier on MVA limit at "to" end of branch
<code>f</code>	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 formulation introduced in this version of *MATPOWER* is inspired by the data format used by *MINOS*.

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

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

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

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

$$|S_{ij}^f(\theta, V)| \leq S_{ij}^{\max} \quad (\text{apparent power flow limit of lines, from end})$$

$$|S_{ij}^t(\theta, V)| \leq S_{ij}^{\max} \quad (\text{apparent power flow limit of lines, to end})$$

$$V_i^{\min} \leq V_i \leq V_i^{\max} \quad (\text{bus voltage limits})$$

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

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

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

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

$$g_2(x) \leq 0 \quad (\text{branch flow limits})$$

$$x_{\min} \leq x \leq x_{\max} \quad (\text{variable limits})$$

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:

$$\begin{aligned} & \min_x f(x_2) \\ & \text{subject to} \\ & \quad g_1(x_1, x_2) = 0 \quad (\text{equality constraints}) \\ & \quad g_2(x_1, x_2) \leq 0 \quad (\text{inequality constraints}) \end{aligned}$$

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 non-linear 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 .

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

subject to

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

$$-\Delta \leq \Delta x \leq \Delta$$

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:

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

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

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

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:

$$\begin{aligned}
& \min \quad c^T \cdot \Delta x \\
& \text{subject to} \\
& \quad A \cdot \Delta x \leq b \\
& \quad -\Delta \leq \Delta x \leq \Delta
\end{aligned}$$

If this is rewritten as:

$$\begin{aligned}
& \min \quad c_1^T \cdot \Delta x_1 + c_2^T \cdot \Delta x_2 \\
& \text{subject to} \\
& \quad A_{11} \cdot \Delta x_1 + A_{12} \cdot \Delta x_2 = b_1 \\
& \quad A_{21} \cdot \Delta x_1 + A_{22} \cdot \Delta x_2 \leq b_2 \\
& \quad -\Delta \leq \Delta x \leq \Delta
\end{aligned}$$

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:

$$\begin{aligned}
& \min \quad (-c_1^T A_{11}^{-1} A_{12} + c_2^T) \cdot \Delta x_2 \\
& \text{subject to} \\
& \quad A_{11} \cdot \Delta x_1 + A_{12} \cdot \Delta x_2 = b_1 \\
& \quad A_{21} \cdot A_{11}^{-1} (b_1 - A_{12} \Delta x_2) + A_{22} \cdot \Delta x_2 \leq b_2 \\
& \quad -\Delta_1 \leq A_{11}^{-1} (b_1 - A_{12} \Delta x_2) \leq \Delta_1 \\
& \quad -\Delta_2 \leq \Delta x_2 \leq \Delta_2
\end{aligned}$$

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 can be written as follows:

$$\min_{x,y,z} \sum_i (f_{1i}(P_{gi}) + f_{2i}(Q_{gi})) + c \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

subject to

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

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

$$g_{S_f}(x) = S_f(\theta, V) \leq S_{\max} \quad (\text{apparent power flow limit of lines, from end})$$

$$g_{S_t}(x) = S_t(\theta, V) \leq S_{\max} \quad (\text{apparent power flow limit of lines, to end})$$

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

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

where

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

is the vector of standard optimization variables for the OPF and (y, z) are other variables to be explained later. The ability to include the general linear constraints and the extra linear cost vector c allow easy modeling for the CCV formulation of piecewise linear costs and constant power factor dispatchable or price-sensitive loads. Furthermore, because the user is given the ability to specify all or part of A , l and u , it is possible to pose additional constraints such as restriction of angle differences or linearly-interrelated injections, making **MATPOWER** even more useful as a research tool. The general formulation also allows generator costs of mixed type (polynomial and piecewise linear) in the same problem. *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.*

Problem Data Transformation and General Linear Restrictions

To add general linear constraints of ones own, it is necessary to understand the standard transformations performed on the input data (bus, gen, branch, areas and gencost tables) before the problem is solved in order to know where the optimization variables end up in the x vector. All of these transformations are reversed after solving the problem so that output data is in the right place 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.

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

$$\text{Theta}(2) - \text{Theta}(7) \leq 5 \text{ degrees}$$

would do the trick. We have *nb* = 9 buses, *ng* = 3 generators and *n1* = 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 ap-

pear 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

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & \text{zeros}(1, \text{nb}+\text{ng}+\text{ng}) \end{bmatrix} * x \leq 5 \text{ degrees}$$

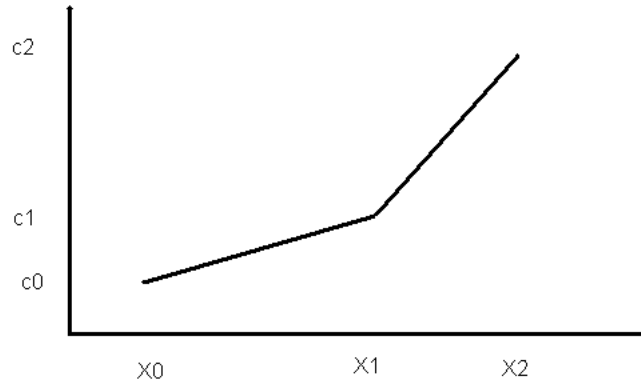
We can set up the problem as follows:

```
A = sparse([1;1], [2;7], [1;-1], 1, 24);
l = -Inf;
u = 5 * pi/180;
mpopt = mpooption('OPF_ALG', 520); % use fmincon with generalized formulation
opf('case9', A, l, u, mpopt)
```

which indeed restricts the angular separation to 5 degrees. NOTE: In this example, the total number of variables is 24, but if there are any piecewise linear cost functions, there may be additional “helper” variables used by the solver and in that case the number of columns in A may need to be larger. The next section describes how this is done. If all costs are polynomial, however, no extra variables are needed.

Piecewise linear convex cost formulation using constrained cost variables

The generalized formulation allows for an easy way to model any piecewise linear costs. Such a cost curve looks like



This non-differentiable cost can be modeled using one helper cost variable for each such cost curve and additional restrictions on this variable and P_g , one restriction for each segment of the curve. The restrictions build a convex “basin” and they are equivalent to saying that the cost variable must 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 generators’ cost to the total cost is exactly y , and in the above case the two restrictions needed 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 .

In the generalized OPF formulation, the capability to accept general linear constraints is used to introduce new y variables (one for each piecewise linear cost in the problem) and constraints (one for each cost segment in the problem). The function that builds the coefficient matrix for the restrictions is `makeAy`.

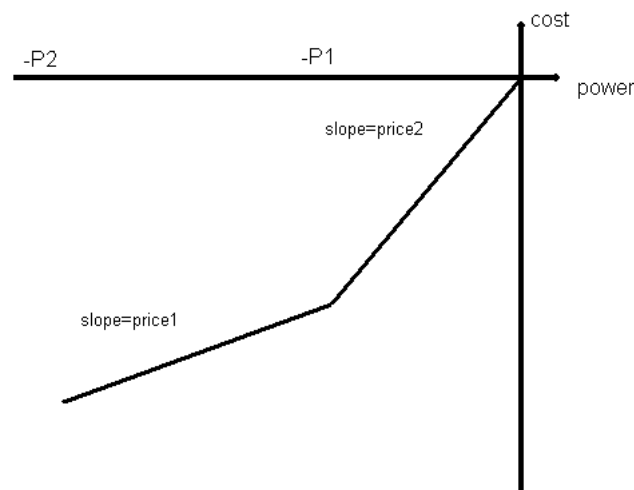
Because a linear cost on the y variables is also required, the last row of the matrix that is passed to the solver is expected to contain not some linear restriction coefficients but a linear cost vector on $[x; y]$. In normal use this is done automatically inside `fminconpf` (or `mopf` when using *MINOPF*) and the user need not worry about this. To incorporate additional linear constraints, however, it is necessary to know in advance how many y variables there are so that the coefficient matrix for the user's constraints have a matching number of columns to multiply $[x; y]$. The number of y variables is equal to the number of piecewise linear cost curves (both active and reactive) there are for the generators that are online.

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 $P_{\min} < P_{\max} = 0$ 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 re-

jecting 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 Q_{MIN}/Q_{MAX} 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 P_{MIN} and either Q_{MIN} (for inductive loads) or Q_{MAX} (for capacitive loads) in the *gen* table. It is important to set these appropriately, keeping in mind that P_g is negative and that for normal inductive loads Q_g should also be negative (a positive reactive load is a negative reactive injection). The initial values of the P_g and Q_g columns of the *gen* matrix must be consistent with the ratio defined by P_{MIN} and the appropriate Q limit.

Additional Information on the Generalized Formulation Structure

The advanced user may want to incorporate additional linear constraints and/or linear costs to the problem. Sometimes new state variables may need to be defined (in addition to x and y discussed above). And, sometimes the user may also want to include additional linear costs on some or all of the variables, but in most cases *fmincopf* (or *mopf*) hide these steps from the user and create certain costs and restrictions automatically. This creates a conflict between ease of use and generality of the software. In this section we explain the behavior adopted by *fmincopf* and *mopf* when the following factors interact:

- a) existence of y variables because of piecewise linear costs
- b) user-provided linear restrictions
 - b1) on whatever the existing variables $[x; y]$ are;
 - b2) with A having more columns than there are elements in $[x; y]$, thus creating extra variables z and an overall optimization vector $[x; y; z]$.

If the user does not provide any additional linear restrictions via the A , l , u parameters, then internally there are only x and y -type variables; if there are y -type variables for modeling piecewise linear costs, then some additional constraints will be constructed to model the cost segments. If there are dispatchable loads (with either polynomial or piecewise linear costs) then additional constraints will be included to maintain a constant power factor.

If a user does provide A , l , u parameters to add general linear constraints, then

- a) if the number of columns in A is the same as the number of variables in $[x; y]$, then any linear cost vector needed for modeling of the piecewise linear costs is constructed automatically and the user is not expected to provide a cost row in A .
- b) if the number of columns in A is greater than the number of elements in $[x; y]$, then the user is responsible to provide
 - b1) the appropriate linear restrictions on y and the P_g , Q_g sections of the x vector to model each of the segments of the piecewise linear costs as described previously; this includes both the coefficient matrix and the left and right hand sides of the restrictions. The function *makeAy* can be used for this, but keep in mind that it will return a coefficient matrix that has only as many columns as elements in $[x; y]$. It must be padded with enough sparse zero columns on the right to make it conformable with $[x; y; z]$ if there are any z variables.
 - b2) the appropriate linear cost in the last row of A and this includes **any necessary cost coefficients on the y section of the cost vector**. Note that l and u must still have the same number of elements as A has rows but the last elements in l and u are meaningless; set them to *(-large, +large)*.

Note that this implies that in order to add linear costs on just the $[x; y]$ variables, at least one dummy z variable must be created for the interface to provide the user with the opportunity to specify the linear cost vector.

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

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

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

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

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

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', mption)
```

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 mption
```

```
MPTION Used to set and retrieve a MATPOWER options vector.
```

```
opt = mption
```

```
returns the default options vector
```

```
opt = mption(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 = mption('PF_ALG', 2, 'PF_TOL', 1e-4)
```

```
opt = mption(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 ]

```

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 <i>MATPOWER</i>
README.txt	basic intro to <i>MATPOWER</i> , with DOS line endings (for Windows)
docs/CHANGES	modification history of <i>MATPOWER</i>
docs/CHANGES.txt	modification history of <i>MATPOWER</i> , with DOS line endings
docs/manual.pdf	PDF version of the <i>MATPOWER</i> User's Manual
(see also <code>caseformat.m</code> & <code>genform.m</code> below)	

Top-level programs:

<code>cdf2matp.m</code>	converts data from IEEE CDF to <i>MATPOWER</i> format
<code>runcomp.m</code>	runs 2 OPFs and compares results
<code>rundcopf.m</code>	runs a DC optimal power flow
<code>rundcpf.m</code>	runs a DC power flow
<code>runduopf.m</code>	runs a DC OPF with unit decommitment
<code>runopf.m</code>	runs an optimal power flow
<code>runpf.m</code>	runs a power flow
<code>runuopf.m</code>	runs an OPF with unit decommitment
(see also <code>opf.m</code> , <code>copf.m</code> , <code>fmincopf.m</code> , <code>lpopf.m</code> below which can also be used as top-level programs)	

Input data files:

<code>caseformat.m</code>	documentation for input data file format
<code>case_ieee30.m</code>	IEEE 30 bus system
<code>case118.m</code>	IEEE 118 bus system
<code>case14.m</code>	IEEE 14 bus system
<code>case30.m</code>	modified IEEE 30 bus system
<code>case300.m</code>	IEEE 300 bus system
<code>case30pwl.m</code>	<code>case30.m</code> with piecewise linear costs
<code>case30Q.m</code>	<code>case30.m</code> with reactive power costs
<code>case39.m</code>	39 bus system
<code>case4gs.m</code>	4 bus system from Grainger & Stevenson
<code>case57.m</code>	IEEE 57 bus system
<code>case6ww.m</code>	6 bus system from Wood & Wollenberg
<code>case9.m</code>	3 generator, 9 bus system (default case file)
<code>case9Q.m</code>	<code>case9.m</code> 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
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 <i>MATPOWER</i> version information
printpf.m	pretty prints solved PF or OPF case
savecase.m	saves data matrices to a case file
mpoption.m	set <i>MATPOWER</i> 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

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

files used only by DC OPF

<code>dcopf.m</code>	implements DC optimal power flow
----------------------	----------------------------------

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

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

files used only by constr-based OPF

<code>copf.m</code>	implements constr-based OPF solver
---------------------	------------------------------------

files used only by LP-based OPF

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

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

<code>genform.m</code>	documentation for generalized OPF formulation
<code>makeAy.m</code>	forms A matrix and b vector for generalized OPF formulation

files used only by fmincon-based OPF

<code>consfmin.m</code>	computes value and gradient of constraints
<code>costfmin.m</code>	computes value and gradient of objective function
<code>fmincopf.m</code>	implements fmincon-based OPF solver

files used only for OPF with unit decommitment

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

Extras: (in extras subdirectory)

auction market software (in smartmarket subdirectory)

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

unfinished state estimator (in state_estimator subdirectory)

<code>runse.m</code>	runs a state estimator
<code>state_est.m</code>	implements a state estimator

Tests: (in t subdirectory)

<code>soln9_dcopf.mat</code>	data used for tests
<code>soln9_dcpf.mat</code>	data used for tests
<code>soln9_opf.mat</code>	data used for tests
<code>soln9_opf_Plim.mat</code>	data used for tests
<code>soln9_pf.mat</code>	data used for tests
<code>t_auction.m</code>	tests <code>auction.m</code> in extras/smartmarket
<code>t_auction_case.m</code>	test case for <code>t_auction.m</code>
<code>t_auction_fmincopf.m</code>	fmincon-based tests of <code>auction.m</code> in extras/smartmarket
<code>t_begin.m</code>	starts a set of tests
<code>t_case9_opf.m</code>	case file for OPF tests
<code>t_case9_pf.m</code>	case file for power flow tests
<code>t_end.m</code>	finishes a set of tests and prints statistics
<code>t_is.m</code>	tests if two matrices are identical to some tolerance
<code>t_jacobian.m</code>	does numerical test of partial derivatives
<code>t_loadcase.m</code>	tests <code>load_case.m</code>
<code>t_ok.m</code>	tests if an expression is true
<code>t_opf.m</code>	tests OPF solvers
<code>t_pf.m</code>	tests PF solvers
<code>t_run_tests.m</code>	framework for running a series of tests
<code>t_skip.m</code>	skips a specified number of tests
<code>test_matpower.m</code>	runs all available <i>MATPOWER</i> 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 and M.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* package² in the context of *POWERWEB*³ but has not been widely tested in other contexts. It has evolved over time and includes a mixture of currently used features and legacy code which we no longer use.

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 matrices, q for offer/bid quantities and p for the corresponding offer/bid prices. The element i -th row and j -th column of q and p 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.

Offer blocks must be in non-decreasing order of price and the offer must correspond to a generator with $0 \leq P_{MIN} < P_{MAX}$. A price cap `max_p` specifies the maximum allowed offer price. 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 $P_{MIN} < P_{MAX} \leq 0$ (see “Dispatchable loads” on page 18). Bids are not affected by `max_p`.

The data specified by a *MATPOWER* case file, with the `gen` and `gencost` matrices modified according to the step 1, are then used to run an OPF. A market code parameter `mkt` is used to specify what type of OPF to use, where a value of 11x0 indicates AC OPF and 12x0 indicates DC 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 Decommittment 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 `cq` and `cp`, which have the same dimensions as q and p . The `mkt` parameter also specifies the rules to use for determining the cleared prices, where the second digit is the rule number described below.

There are two basic types of pricing options, 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 the table below.

² See <http://www.pserc.cornell.edu/minopf/>

³ See <http://www.pserc.cornell.edu/powerweb/>

Rule Number	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 $\min(\text{FRO}, \text{LAB})$ if the marginal unit is an offer, or $\max(\text{FRB}, \text{LAO})$ 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 max_p . 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 max_p . Likewise, cleared bid prices are less than or equal to both the bid price and max_p .

Handling Supply Shortfall

Initial implementations of the auction software did not include dispatchable or price-sensitive loads, so it only handled single-sided auctions. In an attempt to handle situations where the offered capacity was insufficient to meet the demand under all of the other constraints, resulting in an infeasible OPF, imports were introduced. These were simply generators with `GEN_STATUS` set to `-1`. If the auction software failed to find an OPF solution, it would turn on these generators with costs set to \$5/MWh above the highest offer and retry.

This feature still exists but we no longer use it. Instead, for single-sided markets 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 <i>MW @ \$/MWh</i>	Block 2 <i>MW @ \$/MWh</i>	Block 3 <i>MW @ \$/MWh</i>
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 <i>MW @ \$/MWh</i>	Block 2 <i>MW @ \$/MWh</i>	Block 3 <i>MW @ \$/MWh</i>
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 a market code of 1110 and set up the problem as follows:

```
q = [ 12 24 24;
      12 24 24;
      12 24 24;
      12 24 24;
      12 24 24;
      12 24 24;
      10 10 10;
      10 10 10;
      10 10 10  ];
```

```

p = [ 20 50 60;
      20 40 70;
      20 42 80;
      20 44 90;
      20 46 75;
      20 48 60;
      100 70 60;
      100 50 20;
      100 60 50 ];

mpopt = mpoption;

[baseMVA, cq, cp, bus, gen, gencost, branch, f, dispatch, success, et] = ...
    runmkt('my_case', q, p, 1110, [], [], [], mpopt);

```

The resulting cleared offers and bids are:

```
>> cq
```

```
cq =
```

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
10.0000	10.0000	10.0000
10.0000	0	0
10.0000	10.0000	0

```
>> cp
```

```
cp =
```

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
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 <i>MW</i>	Selling Price <i>\$/MWh</i>
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 <i>MW</i>	Purchase Price <i>\$/MWh</i>
1	30.0	\$51.82
2	10.0	\$54.03
3	20.0	\$55.62