MATPOWER

A MATLAB™ Power System Simulation Package

Version 3.0b3
September 20, 2004

User's Manual

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

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 Power Flow 7
3.3 Optimal Power Flow 7
3.4 Unit Decommitment Algorithm 17
3.5 MATPOWER Options 18
3.6 Summary of the Files 22

4 Acknowledgments **23**

5 References **24**

Appendix A: Notes on LP-Solvers for Matlab **25**
Appendix B: Some General Matlab Performance Notes **25**

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[2].

[1] MATPOWER 2.0 and earlier required only version 4 of Matlab.
[2] See http://www.mathworks.com/.

2.2 Installation

Step 1: Go to the *MATPOWER* home page[3] 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
```

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

```
    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 3.5* 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 bus, branch, and gen variables are matrices. Each row in the matrix corresponds to a single bus, branch, or generator, respectively. 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. The format of the data is
    similar to PTI format 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.

    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 = 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

(+) 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
          (or data points for piecewise linear) total cost function
      5 and following, cost data, piecewise linear data as:
                  x0, y0, x1, y1, x2, y2, ...
          and polynomial data as, e.g.:
                  c2, c1, c0
          where the polynomial is c0 + c1*P + c2*P^2
```

3.2 Power Flow

MATPOWER has five power flow solvers. 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. The last method is a direct DC power flow [13].

Currently, none of *MATPOWER*'s power flow solvers include any transformer tap changing or feasibility checking capabilities.

Performance of the power flow solvers 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.

3.3 Optimal Power Flow

MATPOWER includes several solvers for the optimal power flow (OPF) problem. The (chronologically) first 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 offered separately at http://www.pserc.cornell.edu/minopf/ , but with a more restrictive license than that for *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 it's 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. On a Sun Ultra 2200 (dual 200MHz cpu), the LP-based OPF solver using *bpmpd* solves a 30-bus system in under 4 seconds and a 118-bus case in under 25 seconds. 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*.

Traditional OPF Formulation

The OPF 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:

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

such that …

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

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

$$\left|\widetilde{S}_{ij}^{\,f}\right| \le S_{ij}^{\max} \qquad \text{(apparent power flow limit of lines, } \textit{from} \text{ side)}$$

$$\left|\widetilde{S}_{ij}^{\,t}\right| \le S_{ij}^{\max} \qquad \text{(apparent power flow limit of lines, } \textit{to} \text{ side)}$$

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

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

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

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.

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 3.5 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:

$$\min_x f(x_2)$$

such that …

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

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

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.

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

such that …

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

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

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:

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

such that …

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

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

If this is rewritten as:

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

such that …

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

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

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

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:

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

such that …

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

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

$$-\Delta_1 \le A_{11}^{-1}(b_1 - A_{12}\Delta x_2) \le \Delta_1$$
$$-\Delta_2 \le \Delta x_2 \le \Delta_2$$

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.

Generalized Formulation and fmincon.m

The new generalized formulation used by the `fmincon` and *MINOPF* solvers can be written as follows:

$$\min_{P_g, Q_g} \sum f_{1i}(P_{gi}) + f_{2i}(Q_{gi}) + c\begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

such that …

$$g_P(x) = P(V,\theta) - P_{gi} + P_{Li} = 0 \qquad \text{(active power balance equations)}$$
$$g_Q(x) = Q(V,\theta) - Q_{gi} + Q_{Li} = 0 \qquad \text{(reactive power balance equations)}$$
$$\left|\widetilde{S}_{ij}^{\,f}\right| \le S_{ij}^{\max} \qquad \text{(apparent power flow limit of lines, } \textit{from} \text{ side)}$$
$$\left|\widetilde{S}_{ij}^{\,t}\right| \le S_{ij}^{\max} \qquad \text{(apparent power flow limit of lines, } \textit{to} \text{ side)}$$
$$V_i^{\min} \le V_i \le V_i^{\max} \qquad \text{(bus voltage limits)}$$
$$P_{gi}^{\min} \le P_{gi} \le P_{gi}^{\max} \qquad \text{(active power generation limits)}$$
$$Q_{gi}^{\min} \le Q_{gi} \le Q_{gi}^{\max} \qquad \text{(reactive power generation limits)}$$
$$l \le A\begin{pmatrix} x \\ y \\ z \end{pmatrix} \le u \qquad \text{(General linear constraints)}$$

where

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

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 piece wise-linear costs and constant power factor 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 piece wise-linear) in the same problem. Note: at the time of release, fmincon.m seems to be providing slightly inaccurate shadow prices on the constraints. This did not happen with constr.m and it may be a bug in the optimization toolbox.

Problem Data Transformation and General Linear Restrictions

If the user wants to add general linear constraints of his or her own, it is necessary to understand the standard transformations performed on the input data (bus, gen, branch, area and gencost tables) before the problem is actually 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

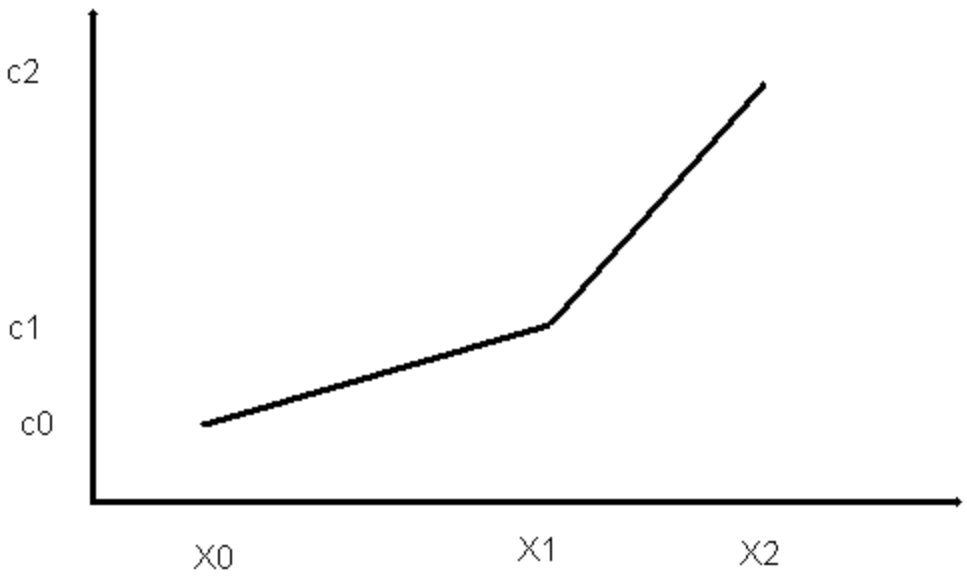
```
g = [  gp ;       % nb real power flow mismatches (p.u.)
       gq;        % nb reactive power flow mismatches (p.u.)
       gsf;       % nl "From" end apparent power injection limits (p.u.)
       gst ];     % nl "To" end apparent power injection limits (p.u.)
```

With this setup, box bounds on the variables are applied as follows: the reference angle is upper and lower bounded with the value that came for it in the original bus() table. The V section of x is lower and upper-bounded with the corresponding values for VMIN and VMAX in the bus() table. The Pg and Qg sections of x are lower and upper-bounded 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: in the standard solution to case9.m, the voltage angle
  for bus 7 lags the voltage angle in bus 2 by 6.09 degrees.
  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 = 0;
    u = 5 * pi/180;
    mpopt = mpoption;
    mpopt(11) = 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 piece wise-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.  Read the next section to understand how this is done.
 If all costs are polynomial, however, no extra variables
 are needed.
```

Piece wise-linear convex cost formulation using constrained cost variables

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



This nondifferentiable cost can be modeled using one helper cost variable for each such cost curve and additional restrictions on this variable and Pg, 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)

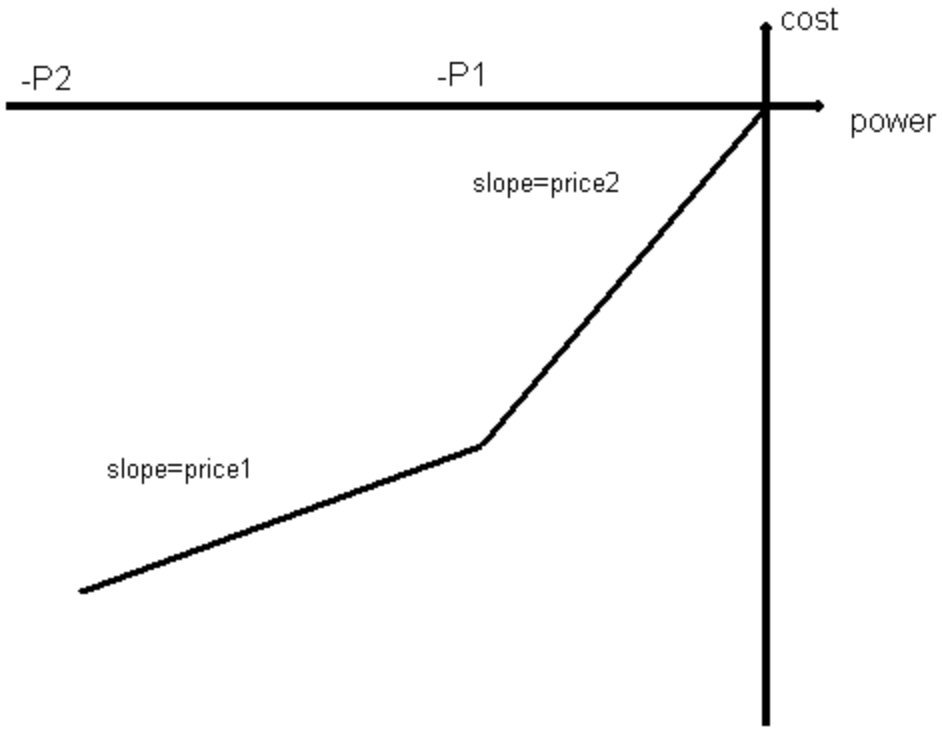
(here, m_1 and m_2 are the slopes of the two segments) and of course, the box restrictions on P_g: $P_{min} \leq P_g \leq P_{max}$. The additive part of the cost contributed by this generator is y.

In the generalized OPF formulation, the capability to accept general linear constraints is used to introduce new variables "y" (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.m . Because a linear cost on the y variables is also required, the last row of the matrix that is actually passed on 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 fmincopf.m (or mopf.m when using *MINOS*) and the users need not worry about this. If the user wants to add linear constraints of his or her own, 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]. In that case it is necessary to know how many piece wise linear cost curves (both active and reactive) there are for the generators that are online. That is equal to the number of "y" variables.

Curtailable and price-sensitive loads

In general, 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 price-sensitive 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. Then, when the load is considered as a negative injection what is wanted is that it is dispatched at zero if the price is greater than price2, at -P1 if the price is higher than price1 but lower than price2, and at -P2 if the price is equal to or lower than price1. This suggests the following piece wise-linear cost curve:



Note that this assumes that the demand blocks can be "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 not inside the scope of MATPOWER at this time.

When there are price-sensitive loads, the issue of reactive dispatch arises. If the QMIN/QMAX generation limits for the generator (really a load) in question are not set to zero, then the algorithm will actually dispatch the reactive injection to the most convenient value. This is not normal load behavior
and therefore in the generalized formulation it is assumed that variable 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 Pg and Qg constant. This ratio is inferred from the initial values of PG and QG in the gen() table; thus, 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).

Additional Information on the Generalized Formulation Structure

The advanced user may need to add linear constraints to the problem and/or linear costs. Sometimes new state variables may need to be defined (in addition to x and y discussed before). And, sometimes the user may also need 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.m and mopf.m when the following factors interact:

a) existence of y variables because of piece wise-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 piece wise-linear costs, then some additional constraints will be constructed to model the cost segments. If there are price-sensitive loads (with either polynomial or piece wise-linear costs) then some more constraints will be added 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], the software takes charge of constructing any linear cost vector needed for modeling of the piece wise-linear costs and it does not expect any user-provided cost row in A.

b) If the number of columns in A is greater than the number of elements in [x;y], then it becomes the user's responsibility to provide

b1) the appropriate linear restrictions on y and the Pg, Qg sections of the x vector to model each of the segments of the piece wise-linear costs as described in section 2; this includes both the coefficient matrix and the left and right hand sides of the restrictions. The function makeAy.m 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 means that if the user wants to add linear costs on just the [x,y] variables, he or she will have to create at least one dummy z variable for the interface to provide the user with the opportunity to specify the linear cost vector.

3.4 Unit Decommitment Algorithm

This section is out-of-date.

(Please see `help uopf` for a brief description of the current 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 a unit decommitment algorithm which allows it to shut down these expensive units. The algorithm is based on a simplified version of the decommitment technique proposed in [7].

The algorithm proceeds as follows:

Step 0: Assume all generators are on-line with all generator limits in place.

Step 1: Solve a normal OPF.

Step 2: If the OPF converged to a feasible solution and the objective function decreased from the previous iteration (or if this is the first iteration), go to step 3, otherwise go to step 4.

Step 3: Compute a decommitment index for each generator i as follows:

$$d_i = f_i(P_i) - \lambda_i \cdot P_i$$

where P_i is generator i's dispatch computed by the OPF, f_i is the cost of operating at P_i, and λ_i is the Lagrange multiplier on the real power equality constraint at the bus where generator i is located. Continue with step 5.

Step 4: Return to the previous commitment and set d_k to zero (to eliminate it from consideration).

Step 5: Find the generator k with the smallest decommitment index. If d_k is negative, shut down generator k and return to step 1. If d_k is positive, stop.

3.5 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 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
    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 'constr'
    18 - CONSTR_TOL_F, 1e-4     termination tol on F for 'constr'
    19 - CONSTR_MAX_IT, 0       max number of iterations for 'constr'
                                [       0 => 2*nb + 150             ]
    20 - LPC_TOL_GRAD, 3e-3     termination tolerance on gradient
                                for 'LPconstr'
    21 - LPC_TOL_X, 1e-4        termination tolerance on x (min
                                step size) for 'LPconstr'
    22 - LPC_MAX_IT, 400        maximum number of iterations for
                                'LPconstr'
    23 - LPC_MAX_RESTART, 5     maximum number of restarts for
                                'LPconstr'
    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  ]
```

```
MINOS OPF 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
    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)
```

3.6 Summary of the Files

This section is out-of-date.

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
```

Input data files:

```
cdf2matp.m  - a stand-alone m-file which reads IEEE CDF formatted
              data and outputs data in MATPOWER's case.m format
case9.m     - a 3 generator, 9 bus case
case30.m    - a 6 generator, 30 bus case
case57.m    - IEEE 57-Bus case
case118.m   - IEEE 118-Bus case
case300.m   - IEEE 300-Bus case
case9Q.m    - case9.m, with costs for reactive generation
case30Q.m   - case30.m, with costs for reactive generation
case30pwl.m - case30.m with a piece-wise linear cost function
```

Source files used by all algorithms:

```
bustypes.m  - lists of buses of type reference, PV or PQ.
dSbus_dV.m  - computes partial derivatives for Jacobian
ext2int.m   - External to internal (successive) bus numbering.
idx_brch.m  - defines column indexes for branch table
idx_bus.m   - defines column indexes for bus table
idx_gen.m   - defines column indexes for gen table
int2ext.m
makeSbus.m
makeYbus.m  - builds Ybus matrix
mpoption.m  - sets MATPOWER options
printpf.m   - prints output
```

Other source files used by PF (Power Flow):

```
fdpf.m      - implements fast decoupled power flow
newtonpf.m  - implements Newton's method power flow
pfsoln.m
runpf.m     - main program for running a power flow
makeB.m
```

Other source files used by OPF (Optimal Power Flow):

```
dAbr_dV.m   - computes partial derivatives of apparent power flows
dSbr_dV.m   - computes partial derivatives of complex power flows
fg_names.m
fun_ccv.m   - computes obj fcn and constraints for CCV formulation
fun_std.m   - computes obj fcn and constraints for standard formulation
grad_ccv.m  - computes gradients for standard formulation
grad_std.m  - computes gradients for standard formulation
idx_area.m
idx_cost.m
opf.m       - implements main OPF routine
copf.m      - front end for constr.m –based solver (algorithms 100 & 200)
opfsoln.m
opf_form.m
opf_slvr.m
poly2pwl.m
pqcost.m
runopf.m    - main program for running an optimal power flow
totcost.m   - computes cost
```

The following are used only by the LP-based OPF algorithms:

```
LPconstr.m
LPeqslvr.m
LPrelax.m
LPsetup.m
```

```
The following are used by the generalized-formulation solvers
      fmincopf.m  - front end for fmincon.m solver (algorithm 520)
      makeAy.m    - compute restrictions and cost vector for CCV cost model
      costfmin.m  - cost and cost gradient for generalized formulation
      consfmin.m  - constraint and gradients of constraints for generalized
                    formulation
```

Other source files used by UOPF (Unit decommitment/OPF):

```
(all files from OPF, except runopf.m)
uopf.m      - implements decommitment heuristic
runuopf.m   - main program for running OPF with decommitment algorithm
```

Files for use with the *bpmpd* LP/QP-solver:

```
bpmpd/lp.m  - replacement for Optimization Toolbox lp.m
bpmpd/qp.m  - replacement for Optimization Toolbox qp.m (used by constr.m)
```

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. Last but not least, we'd like to acknowledge the input of Bob Thomas throughout the of 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

This section is out-of-date.

The *MATPOWER* distribution does not include an LP-solver, however, the Matlab Optimization Toolbox does include and LP-solver, `lp.m`, which is based on it's QP-solver, `qp.m`. For large sparse problems, these routines are ***very*** slow. Fortunately, there are some third party LP and QP-solvers for MATLAB with much better performance.

Several LP and QP-solvers have been 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 (Sun Ultra running Solaris 2.5.1) and others proved to be less than robust in an OPF context.

Here is a list of the solvers we've attempted to use:

- *bpmpd* - QP-solver from http://www.sztaki.hu/~meszaros/bpmpd/ ($100)
  (Matlab MEX interface by Carlos Murillo <cem14@cornell.edu>)
- *lp.m* - LP-solver included with Optimization Toolbox (from MathWorks)
- *lp_solve* - LP-solver from ftp://ftp.ics.ele.tue.nl/pub/lp_solve/ (free)
- *loqo* - LP-solver from http://www.princeton.edu/~rvdb/ (free)
- *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. The distribution includes two files `lp.m` and `qp.m` in the `bpmpd` directory. If *bpmpd* is installed and these two files are included in your Matlab path before the Optimization Toolbox routines, they will be used in place of the `lp.m` and `qp.m` in the Toolbox[4].

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.

Appendix B: Some General Matlab Performance Notes

The performance bottlenecks in Matlab are different for Matlab 4 and Matlab 5. Here are two observations from our testing:

- Matlab 4 is slow at executing `case.m` for large files, Matlab 5 is not.
- Matlab 5 is slow at selecting rows of a large sparse matrix, Matlab 4 is not.
- fmincon.m seems to compute slightly inaccurate shadow prices

[4] Note when using `constr` in Matlab 5, it doesn't seem to find the *bpmpd* replacement for `qp.m`, although this seems to work fine under Matlab 4.