MATPOWER

A MATLAB™ Power System Simulation Package

Version 2.0

December 24, 1997

User's Manual

Ray D. Zimmerman
rz10@cornell.edu

Deqiang (David) Gan
deqiang@ee.cornell.edu

Table of Contents

1 Introduction

What is MATPOWER?

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

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

Where did it come from?

MATPOWER was developed by Ray Zimmerman 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. Anyone may modify it for their own use as long as the original copyright notices remain in place. Please don't distribute modified versions of *MATPOWER* without written permission from us.

2 Getting Started

2.1 System Requirements

To use *MATPOWER* you will need a Mac, UNIX machine, or PC with:

- Matlab 4 or higher (available from The MathWorks)
- Matlab Optimization Toolbox (available from The MathWorks[1])

2.2 Installation

Step 1: Go to the *MATPOWER* home page[2] and follow the download instructions.

Step 2: Unpack the archive using the appropriate software for your machine (StuffIt Expander for Mac, gunzip and tar for UNIX, pkzip, WinZip, etc. for PC).

Step 3: Copy all of the m-files in the *MATPOWER* distribution to a location in your Matlab path.

[1] See http://www.mathworks.com/
[2] http://www.pserc.cornell.edu/matpower/matpower.html

2.3 Running a Power Flow

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

```
>> runpf
```

To run a power flow on the 118-bus system whose data is in `case118.m`, type:

```
>> runpf('case118')
```

2.4 Running an Optimal Power Flow

To run an optimal power flow on the default 9-bus system specified in the file `case.m`, with the default algorithm options, at the Matlab prompt, type:

```
>> runopf
```

To run an optimal power flow on the 30-bus system whose data is in `case30.m`, 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)

    Runs an optimal power flow where casename is the name of the m-file
    (without the .m extension) containing the opf data, and mpopt is a
    MATPOWER options vector (see 'help mpoption' for details). Uses default
    options if 2nd parameter is not given, and 'case' if 1st parameter
    is not given. The results may optionally be printed to a file (appended
    if the file exists) whose name is given in fname (in addition to
    printing to STDOUT). Optionally returns the final values of baseMVA,
    bus, gen, gencost, branch, f, success, and et.
```

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 which define and return the variables baseMVA, bus, branch, gen, area, 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 `case.m`:

```
>> help case

 CASE    Defines the power flow data in a format similar to PTI.
    [baseMVA, bus, gen, branch, area, gencost] = case
    The format for 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, 1 - machine in service, 0 - machine out of service
    (-)     (% of total VARS 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 three power flow solvers. The default power flow solver is based on a standard Newton’s method [11] using a full Jacobian, updated at each iteration. This method is described in detail in many textbooks. The other two power flow solvers are variations of the fast-decoupled method [9]. *MATPOWER* implements the XB and BX variations as described in [1]. Currently, *MATPOWER*’s power flow solvers do not 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. On a Sun Ultra 2200, *MATPOWER* solves a 9600-bus test case in about 10 seconds, and a 38400 bus case in about 50 seconds.

3.3 Optimal Power Flow

MATPOWER includes two solvers for the optimal power flow (OPF) problem. The first is based on the `constr` function included in 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.

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 LP-based algorithm, on the other hand, does preserve sparsity. However, the LP-solver included in the Optimization Toolbox does not exploit this sparsity. In fact, the LP-based method with the default 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* [7] (actually a QP-solver), has proven to be robust and efficient.

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, 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.

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 …

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

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

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

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

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

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

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

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 a polynomial or piecewise-linear functions. The problem can be written more compactly in the following form:

$$\min_x f(x)$$

such that …

$$g(x) \le 0$$

where f and g are non-linear functions.

Optimization Toolbox Based OPF Solver (`constr`)

The first of the two 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 Optimization Toolbox manual [5]. *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 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.

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

such that …

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

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

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:

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

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

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

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) [10]

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:

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

such that …

$$A \Delta x \leq b$$
$$-\Delta \leq \Delta x \leq \Delta$$

If this is rewritten as:

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

such that …

$$A_{11} \Delta x_1 + A_{12} \Delta x_2 = b_1$$
$$A_{21} \Delta x_1 + A_{22} \Delta x_2 \leq b_2$$
$$-\Delta \leq \Delta x \leq \Delta$$

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:

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

such that …

$$A_{11} \Delta x_1 + A_{12} \Delta x_2 = b_1$$
$$A_{21} A_{11}^{-1}(b_1 - A_{12}\Delta x_2) + A_{22} \Delta x_2 \leq b_2$$
$$-\Delta_1 \leq A_{11}^{-1}(b_1 - A_{12}\Delta x_2) \leq \Delta_1$$
$$-\Delta_2 \leq \Delta x_2 \leq \Delta_2$$

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

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

3.4 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 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 [6].

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:

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

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)                         ]
        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
    OPF options
        11 - OPF_ALG, 0             algorithm to use for OPF
            [   (see README for more info on formulations/algorithms)   ]
            [   The algorithm code = F * 100 + S * 20, where ...        ]
            [       F specifies one of the following OPF formulations   ]
            [           1 - standard (polynomial cost in obj fcn)       ]
            [           2 - CCV (constrained cost variables)            ]
            [       S specifies one of the following solvers            ]
            [           0 - 'constr' from Optimization Toolbox          ]
            [           1 - Dense LP-based method                       ]
            [           2 - Sparse LP-based method w/relaxed constraints]
            [           3 - Sparse LP-based method w/full constraints   ]
            [   This yields the following 9 codes:                      ]
            [         0 - choose appropriate default from OPF_ALG_POLY  ]
            [             or OPF_ALG_PWL                                ]
            [       100 - standard formulation, constr                  ]
            [       120 - standard formulation, dense LP                ]
            [       140 - standard formulation, sparse LP (relaxed)     ]
            [       160 - standard formulation, sparse LP (full)        ]
            [       200 - CCV formulation, constr                       ]
            [       220 - CCV formulation, dense LP                     ]
            [       240 - CCV formulation, sparse LP (relaxed)          ]
            [       260 - CCV formulation, sparse LP (full)             ]
        12 - OPF_ALG_POLY, 100      default OPF algorithm for use with
                                    polynomial cost functions
        13 - OPF_ALG_PWL, 200       default OPF algorithm for use with
                                    piece-wise linear cost functions
        14 - OPF_POLY2PWL_PTS, 10   number of evaluation points to use
                                    when converting from polynomial to
                                    piece-wise linear costs
        15 - OPF_NEQ, 0             number of equality constraints
                                    (0 => 2*nb, set by program, not a
                                    user option)
        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, 5e-3        termination tolerance on x (min
                                    step size) for 'LPconstr'
        22 - LPC_MAX_IT, 1000       maximum number of iterations for
                                    'LPconstr'
        23 - LPC_MAX_RESTART, 5     maximum number of restarts for
                                    'LPconstr'
    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  ]
```

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

Documentation files:

```
      README      - basic intro to MATPOWER
      CHANGES     - modification history of MATPOWER
      manual.pdf  - PDF version of the MATPOWER User’s Manual
                    (requires Adobe Acrobat Reader)
```

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
      case.m      - same as case9.m
      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
      dSbus_dV.m  - computes partial derivatives for Jacobian
      ext2int.m
      idx_brch.m
      idx_bus.m
      idx_gen.m
      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
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
```

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 Carlos Murillo-Sánchez for suggesting the CCV formulation for handling piecewise linear costs in the OPF, for his help on the decommitment algorithm, and for creating the Matlab MEX interface to the *bpmpd* LP and QP solver. 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. 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. Grace, *Optimization Toolbox*, The MathWorks, Inc., Natick, MA, 1995.
6. 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.
7. C. Mészáros, “The efficient implementation of interior point methods for linear programming and their applications”, *Ph.D. Thesis*, Eötvös Loránd University of Sciences, 1996.
8. B. Stott, “Review of Load-Flow Calculation Methods”, *Proceedings of the IEEE*, Vol. 62, No. 7, July 1974, pp. 916-929.
9. B. Stott and O. Alsac, “Fast decoupled load flow”, *IEEE Transactions on Power Apparatus and Systems,* Vol. PAS-93, June 1974, pp. 859-869.
10. B. Stott, J. L. Marino, O. Alsac, “Review of Linear Programming Applied to Power System Rescheduling”, *1979 PICA*, pp. 142-154.
11. 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.

Appendix A: Notes on LP-Solvers for Matlab

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[3].

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.

[3] 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.